

Concurrent HOL

Peter Gammie

March 17, 2025

Abstract

This is a simple framework for expressing linear-time properties. It supports the usual programming constructs (including interleaving parallel composition), equational and inequational reasoning about these, compositional assume/guarantee specifications and refinement, and the mixing of specifications and programs, all shallowly embedded in Isabelle/HOL.

Contents

1	Introduction	4
1.1	Road map	4
2	Terminated Aczel sequences	4
2.1	Traces	5
2.2	Combinators on traces	5
2.3	Behaviors	15
2.4	Combinators on behaviors	16
3	Point-free notation	23
4	More lattice	25
4.1	Boolean lattices and implication	28
4.2	Compactness and algebraicity	30
5	Closure operators	34
5.1	Complete lattices and algebraic closures	36
5.2	Closures over powersets	40
5.3	Matroids and antimatroids	42
5.4	Composition	43
5.5	Path independence	44
5.6	Some closures	44
5.6.1	Reflexive, symmetric and transitive closures	44
5.6.2	Relation image	45
5.6.3	Kleene closure	45
6	Galois connections	49
6.1	Some Galois connections	56
7	Heyting algebras	58
7.1	Downwards closure of preorders (downsets)	65
8	Safety logic	68
8.1	Stuttering	68
8.2	The $(\prime a, \prime s, \prime v)$ <i>spec</i> lattice	76
8.3	Irreducible elements	78
8.4	Maps	81
8.5	The idle process	86
8.6	Actions	87
8.7	Operations on return values	94
8.8	Bind	100

8.9	Kleene star	111
8.10	Transition relations	112
8.11	Sequential assertions	124
8.11.1	Preconditions	124
8.11.2	Postconditions	128
8.11.3	Strongest postconditions	132
8.12	Initial steps	134
8.13	Heyting implication	136
8.14	Miscellaneous algebra	139
9	Constructions in the $(\prime a, \prime s, \prime v)$ spec lattice	145
9.1	Constrains-at-most	145
9.2	Abadi and Plotkin's composition principle	152
9.3	Interference closure	154
9.4	The $\prime a$ agent datatype	161
9.5	Parallel composition	163
9.6	Specification Inhabitation	179
10	"Next step" implication ala Abadi and Merz (and Lamport)	184
10.1	Compositionality ala Abadi and Merz (and Lamport)	191
11	Stability	191
12	Refinement	194
12.1	General rules for the $(\prime a, \prime s, \prime v)$ spec lattice	198
12.1.1	Actions	199
12.1.2	Bind	200
12.1.3	Interference	204
12.1.4	Parallel	205
12.2	A relational assume/guarantee program logic for the $(\text{sequential}, \prime s, \prime v)$ spec lattice	206
12.2.1	Stability rules	212
13	A programming language	215
13.1	The $(\prime s, \prime v)$ prog lattice	215
13.2	Morphisms to and from the $(\text{sequential}, \prime s, \prime v)$ spec lattice	215
13.3	Programming language constructs	218
13.3.1	Laws of programming	220
13.4	Refinement for $(\prime s, \prime v)$ prog	234
13.4.1	Introduction rules	234
13.4.2	Galois considerations	234
13.4.3	Rules	235
13.5	A relational assume/guarantee program logic for the $(\prime s, \prime v)$ prog lattice	237
13.5.1	Galois considerations	237
13.5.2	A proof of the parallel rule using Abadi and Plotkin's composition principle	239
13.6	Specification inhabitation	242
14	More combinators	243
15	Structural local state	251
15.1	<i>spec.local</i>	251
15.2	Local state transformations	256
15.2.1	Permuting local actions	263
15.3	<i>spec.localize</i>	268
15.4	<i>spec.local_init</i>	275
15.5	Hoist to $(\prime s, \prime v)$ prog	281
15.6	Refinement rules	285
15.6.1	Data refinement	288
15.7	Assume/guarantee	289
15.8	Specification inhabitation	291

16 A Temporal Logic of Safety (TLS)	293
16.1 Stuttering	293
16.2 The $(\prime a, \prime s, \prime v)$ <i>tls</i> lattice	303
16.3 Irreducible elements	304
16.4 The idle process	306
16.5 Temporal Logic for $(\prime a, \prime s, \prime v)$ <i>tls</i>	307
16.5.1 Leads-to and leads-to-via	325
16.5.2 Fairness	327
16.6 Safety Properties	329
16.7 Maps	340
16.8 Abadi's axioms for TLA	342
16.9 Tweak syntax	344
17 Atomic sections	345
17.1 Inhabitation	348
17.2 Assume/guarantee	349
18 Exceptions	350
19 Assume/Guarantee rule sets	353
19.1 Implicit stabilisation	353
19.1.1 Assume/guarantee rules using implicit stability	355
19.2 Refinement with relational assumes	357
20 Wickerson, Dodds and Parkinson: explicit stabilisation	360
20.1 Assume/Guarantee rules	364
20.2 Examples	365
21 Example: inhabitation	366
22 Example: findP	367
23 Example: data refinement (search)	372
24 Observations about safety closure	380
24.1 Liveness	383
24.2 A Haskell-like <i>Ix</i> class	386
25 A polymorphic heap	389
25.1 References	394
25.2 Arrays	396
25.2.1 Code generation constants: one-dimensional arrays	396
25.2.2 User-facing arrays	397
25.2.3 Stability	402
26 A concurrent variant of Imperative HOL	403
26.1 Code generator setup	405
26.1.1 Haskell	405
26.2 Value-returning parallel	408
27 Total store order (TSO)	408
27.1 References	427
27.2 Inhabitation	428
27.3 Code generator setup for TSO	431
27.3.1 Haskell	432
27.4 A TSO litmus test	433
28 Floyd-Warshall all-pairs shortest paths	435
References	446

1 Introduction

This is a simple framework for expressing linear-time properties. It supports the usual programming constructs (including interleaving parallel composition), equational and inequational reasoning about these, compositional assume/guarantee specifications and refinement, and the mixing of specifications and programs, all shallowly embedded in Isabelle/HOL. The closest extent works to ours are by [Xu and He \(1991, 1994\)](#) and [Dingel \(1996, 2000, 2002\)](#). It is heavily influenced by [Lampert \(1994\)](#).

1.1 Road map

Rather than begin with *a priori* “laws of programming” we take finite and infinite sequences as models of system executions (§16). Also, as transforming realistic concurrent systems while preserving total correctness is too difficult to be usable, we adopt Lampert’s approach to separating liveness and safety properties ([Abadi and Lampert 1991](#)) and do most of our work on safety properties.

The safety model consists of a series of closures (§5) over the powerset lattice of finite, non-empty, terminated “Aczel” sequences (§2), where each transition is ascribed to an agent. The termination marker supports sequential composition. The model of system executions is built similarly.

The *spec* lattice. Firstly and fundamentally we close under prefixes (§7.1), which captures precisely the safety properties (i.e., we identify a safety property with the set of sequences that satisfies it). We also close under stuttering ala Lampert (§8.1) to support refinement and the “laws of programming” (§13.3.1). All properties we consider therefore need to be stuttering invariant which is a mild constraint. We call the set of sets closed in this way the *spec* lattice (§8.2); we can interpret its points as propositions as it is a Heyting algebra. Its chief novelty is that it supports a logical presentation of assume/guarantee reasoning due to Abadi and Plotkin (§13.5.2) where parallel composition (§9.5) is simple (infinitary) conjunction ala [Lampert \(1994\)](#).

This lattice is satisfactory as a logic but deficient as a programming language; see [Zwiers \(1989\)](#) for an extended discussion on this point, and a solution for synchronous message passing. In brief, parallel composition-as-conjunction and the monad laws (§8.8) fail to meet expectations. We therefore look for a stronger closure condition.

The *prog* lattice. We take the view that a concurrent process is a parallel composition of sequential processes where the parallel composition itself yields a sequential process. Abadi and Plotkin’s constrains-at-most (§9.1) closure adds interference to the ends of traces – sufficient to support their circular composition principle (§9.2) – but not their beginnings. Our interference closure (§9.3) makes this symmetric, ensuring that parallel composition conforms to expectations: the monad laws hold as do many of the “laws of programming” (§13.3.1). We define the *prog* type (§13.1) to be the interference-closed specifications. We reason about programs in *prog* using propositions in *spec* via a pair of morphisms that form a Galois connection (§13.2).

Refinement. Abadi and Plotkin’s approach does not support refinement in our setting. We therefore adopt a “next step” implication (§10) and develop a logical account of compositional program refinement (§12). Refinement here is trace inclusion (i.e., the preservation of all safety properties).

Relational assume/guarantee. The definition of relational assume/guarantee in this setting is pleasantly intuitive (§12.2). Its key strength is that program phrases can be abstracted to relational assume/guarantee quadruples that can then be used as program phrases (§13.5). This generalises Morgan’s specification statement to a concurrent setting.

State spaces. As is traditional with shallow embeddings in HOL, we defer state space and value considerations using polymorphism. We develop a mechanism that partially encapsulates local state (§15).

Miscellany. Along the way we assemble some facts about Heyting algebras (§7), and sometimes construct our closures (§5) from Galois connections (§6). We explore the impact of using safety properties and this mix of finite and infinite sequences on TLA (§16).

2 Terminated Aczel sequences

We model a *behavior* of a system as a non-empty finite or infinite sequence of the form $s_0 - a_1 \rightarrow s_1 - a_2 \rightarrow \dots (\rightarrow v)?$ where s_i is a state, a_i an agent and v a return value for finite sequences (see §16). A *trace* is a finite sequence $s_0 - a_1 \rightarrow s_1 - a_2 \rightarrow \dots - a_n \rightarrow s_n \rightarrow v$ for $n \geq 0$ with optional return value v (see §8). States, agents and return values are of arbitrary type.

2.1 Traces

setup $\langle \text{Sign.mandatory-path trace} \rangle$

datatype (aset: 'a , sset: 's , vset: 'v) $t =$
 T (init: 's) ($\text{rest: ('a} \times \text{'s) list}$) (term: 'v option)

for

map: map

pred: pred

rel: rel

declare trace.t.map-id0 [simp]

declare trace.t.map-id0 [unfolded id-def , simp]

declare trace.t.map-sel [simp]

declare trace.t.set-map [simp]

declare trace.t.map-comp [unfolded o-def , simp]

declare trace.t.set [simp del]

instance $\text{trace.t} :: (\text{countable}, \text{countable}, \text{countable}) \text{countable}$ **by** $\text{countable-datatype}$

lemma split-all [no-atp]: — imitate the setup for $'a \times 'b$ without the automation

shows $(\bigwedge x. \text{PROP } P x) \equiv (\bigwedge s \text{ xs } v. \text{PROP } P (\text{trace.T } s \text{ xs } v))$

proof

show $\text{PROP } P (\text{trace.T } s \text{ xs } v)$ **if** $\bigwedge x. \text{PROP } P x$ **for** $s \text{ xs } v$ **by** (rule that)

next

fix x

assume $\bigwedge s \text{ xs } v. \text{PROP } P (\text{trace.T } s \text{ xs } v)$

from $\langle \text{PROP } P (\text{trace.T } (\text{trace.init } x) (\text{trace.rest } x) (\text{trace.term } x)) \rangle$ **show** $\text{PROP } P x$ **by** simp

qed

lemma split-All [no-atp]:

shows $(\forall x. P x) \longleftrightarrow (\forall s \text{ xs } v. P (\text{trace.T } s \text{ xs } v))$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof (intro iffI all)

show $P x$ **if** $?rhs$ **for** x **using that by** ($\text{cases } x$) simp-all

qed simp

lemma split-Ex [no-atp]:

shows $(\exists x. P x) \longleftrightarrow (\exists s \text{ xs } v. P (\text{trace.T } s \text{ xs } v))$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof ($\text{intro iffI all}; \text{elim exE}$)

show $\exists s \text{ xs } v. P (\text{trace.T } s \text{ xs } v)$ **if** $P x$ **for** x **using that by** ($\text{cases } x$) fast

qed auto

2.2 Combinators on traces

definition $\text{final}' :: 's \Rightarrow ('a \times 's) \text{list} \Rightarrow 's$ **where**

$\text{final}' s \text{ xs} = \text{last } (s \# \text{map snd } \text{xs})$

abbreviation (input) $\text{final} :: ('a, 's, 'v) \text{trace.t} \Rightarrow 's$ **where**

$\text{final } \sigma \equiv \text{trace.final}' (\text{trace.init } \sigma) (\text{trace.rest } \sigma)$

definition $continue :: ('a, 's, 'v) trace.t \Rightarrow ('a \times 's) list \times 'v option \Rightarrow ('a, 's, 'v) trace.t$ (**infixl** $\langle @-s \rangle$ 64)
where

$\sigma @-s xsv = (case\ trace.term\ \sigma\ of\ None \Rightarrow trace.T\ (trace.init\ \sigma)\ (trace.rest\ \sigma\ @\ fst\ xsv)\ (snd\ xsv)\ | \ Some\ v \Rightarrow \sigma)$

definition $tl :: ('a, 's, 'v) trace.t \rightarrow ('a, 's, 'v) trace.t$ **where**

$tl\ \sigma = (case\ trace.rest\ \sigma\ of\ [] \Rightarrow None\ | \ x\ \# \ xs \Rightarrow Some\ (trace.T\ (snd\ x)\ xs\ (trace.term\ \sigma)))$

definition $dropn :: nat \Rightarrow ('a, 's, 'v) trace.t \rightarrow ('a, 's, 'v) trace.t$ **where**

$dropn = (\widetilde{\ })\ trace.tl$

definition $take :: nat \Rightarrow ('a, 's, 'v) trace.t \Rightarrow ('a, 's, 'v) trace.t$ **where**

$take\ i\ \sigma = (if\ i \leq length\ (trace.rest\ \sigma)\ then\ trace.T\ (trace.init\ \sigma)\ (List.take\ i\ (trace.rest\ \sigma))\ None\ else\ \sigma)$

type-synonym $('a, 's) transitions = ('a \times 's \times 's) list$

primrec $transitions' :: 's \Rightarrow ('a \times 's) list \Rightarrow ('a, 's) trace.transitions$ **where**

$transitions'\ s\ [] = []$

$| \ transitions'\ s\ (x\ \# \ xs) = (fst\ x,\ s,\ snd\ x)\ \# \ transitions'\ (snd\ x)\ xs$

abbreviation $(input)\ transitions :: ('a, 's, 'v) trace.t \Rightarrow ('a, 's) trace.transitions$ **where**

$transitions\ \sigma \equiv trace.transitions'\ (trace.init\ \sigma)\ (trace.rest\ \sigma)$

setup $\langle Sign.mandatory-path\ final' \rangle$

lemma $simps[simp]$:

shows $trace.final'\ s\ [] = s$

and $trace.final'\ s\ (x\ \# \ xs) = trace.final'\ (snd\ x)\ xs$

and $trace.final'\ s\ (xs\ @ \ ys) = trace.final'\ (trace.final'\ s\ xs)\ ys$

and $idle: \text{snd } ' \text{ set } xs \subseteq \{s\} \implies trace.final'\ s\ xs = s$

and $\text{snd } ' \text{ set } xs \subseteq \{s\} \implies trace.final'\ s\ (xs\ @ \ ys) = trace.final'\ s\ ys$

and $\text{snd } ' \text{ set } ys \subseteq \{trace.final'\ s\ xs\} \implies trace.final'\ s\ (xs\ @ \ ys) = trace.final'\ s\ xs$

by $(simp\text{-all}\ add: trace.final'\text{-def}\ last\text{-map}\ image\text{-subset}\text{-iff}\ split: if\text{-split}\text{-asm})$

lemma map :

shows $trace.final'\ (sf\ s)\ (map\ (map\text{-prod}\ af\ sf)\ xs) = sf\ (trace.final'\ s\ xs)$

by $(simp\ add: trace.final'\text{-def}\ last\text{-map})$

lemma $replicate$:

shows $trace.final'\ s\ (replicate\ i\ as) = (if\ i = 0\ then\ s\ else\ snd\ as)$

by $(simp\ add: trace.final'\text{-def})$

lemma $map\text{-idle}$:

assumes $(\lambda x. sf\ (snd\ x))\ ' \text{ set } xs \subseteq \{sf\ s\}$

shows $sf\ (trace.final'\ s\ xs) = sf\ s$

using $assms$ **by** $(induct\ xs\ arbitrary: s)\ simp\text{-all}$

setup $\langle Sign.parent\text{-path} \rangle$

setup $\langle Sign.mandatory\text{-path}\ tl \rangle$

lemma $simps[simp]$:

shows $trace.tl\ (trace.T\ s\ []\ v) = None$

and $trace.tl\ (trace.T\ s\ (x\ \# \ xs)\ v) = Some\ (trace.T\ (snd\ x)\ xs\ v)$

by $(simp\text{-all}\ add: trace.tl\text{-def})$

setup $\langle Sign.parent\text{-path} \rangle$

lemma *dropn-alt-def*:

shows $\text{trace.dropn } i \ \sigma$

$= (\text{case drop } i \ ((\text{undefined}, \text{trace.init } \sigma) \# \text{trace.rest } \sigma) \text{ of}$
 $\square \Rightarrow \text{None}$
 $| x \# xs \Rightarrow \text{Some } (\text{trace.T } (\text{snd } x) \ xs \ (\text{trace.term } \sigma)))$

proof(*induct i arbitrary: σ*)

case 0 **show** ?case

by (*simp add: trace.dropn-def*)

next

case (*Suc i σ*) **then show** ?case

by (*cases σ ; cases trace.rest σ ; simp add: trace.dropn-def drop-Cons' split: list.split*)

qed

setup $\langle \text{Sign.mandatory-path dropn} \rangle$

lemma *simps[simp]*:

shows 0: $\text{trace.dropn } 0 = \text{Some}$

and *Suc*: $\text{trace.dropn } (\text{Suc } i) \ \sigma = \text{Option.bind } (\text{trace.tl } \sigma) (\text{trace.dropn } i)$

and *dropn*: $\text{Option.bind } (\text{trace.dropn } i \ \sigma) (\text{trace.dropn } j) = \text{trace.dropn } (i + j) \ \sigma$

by (*simp-all add: trace.dropn-def pfunpow-add*)

lemma *Suc-right*:

shows $\text{trace.dropn } (\text{Suc } i) \ \sigma = \text{Option.bind } (\text{trace.dropn } i \ \sigma) \ \text{trace.tl}$

by (*simp add: trace.dropn-def pfunpow-Suc-right del: pfunpow.simps*)

lemma *eq-none-length-conv*:

shows $\text{trace.dropn } i \ \sigma = \text{None} \iff \text{length } (\text{trace.rest } \sigma) < i$

by (*auto simp: trace.dropn-alt-def split: list.split*)

lemma *eq-Some-length-conv*:

shows $(\exists \sigma'. \text{trace.dropn } i \ \sigma = \text{Some } \sigma') \iff i \leq \text{length } (\text{trace.rest } \sigma)$

by (*auto simp: trace.dropn-alt-def dest: drop-eq-Cons-lengthD split: list.split*)

lemma *eq-Some-lengthD*:

assumes $\text{trace.dropn } i \ \sigma = \text{Some } \sigma'$

shows $i \leq \text{length } (\text{trace.rest } \sigma)$

using *assms trace.dropn.eq-Some-length-conv* **by** *blast*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path take} \rangle$

lemma *sel*:

shows $\text{trace.init } (\text{trace.take } i \ \sigma) = \text{trace.init } \sigma$

and $\text{trace.rest } (\text{trace.take } i \ \sigma) = \text{List.take } i \ (\text{trace.rest } \sigma)$

and $\text{trace.term } (\text{trace.take } i \ \sigma) = (\text{if } i \leq \text{length } (\text{trace.rest } \sigma) \text{ then } \text{None} \text{ else } \text{trace.term } \sigma)$

by (*simp-all add: trace.take-def*)

lemma 0:

shows $\text{trace.take } 0 \ \sigma = \text{trace.T } (\text{trace.init } \sigma) \ \square \ \text{None}$

by (*simp add: trace.take-def*)

lemma *Nil*:

shows $\text{trace.take } i \ (\text{trace.T } s \ \square \ \text{None}) = \text{trace.T } s \ \square \ \text{None}$

by (*simp add: trace.take-def*)

lemmas *simps[simp]* =

trace.take.sel

trace.take.0
trace.take.Nil

lemma *map*:

shows $\text{trace.take } i \ (\text{trace.map } af \ sf \ vf \ \sigma) = \text{trace.map } af \ sf \ vf \ (\text{trace.take } i \ \sigma)$

by (*simp add: trace.take-def take-map*)

lemma *append*:

shows $\text{trace.take } i \ (\text{trace.T } s \ (xs \ @ \ ys) \ v) = \text{trace.T } s \ (\text{List.take } i \ (xs \ @ \ ys)) \ (\text{if } \text{length } (xs \ @ \ ys) < i \ \text{then } v \ \text{else } \text{None})$

by (*simp add: trace.take-def*)

lemma *take*:

shows $\text{trace.take } i \ (\text{trace.take } j \ \sigma) = \text{trace.take } (\min \ i \ j) \ \sigma$

by (*simp add: min-le-iff-disj trace.take-def*)

lemma *continue*:

shows $\text{trace.take } i \ (\sigma \ @_{-S} \ xsv) = \text{trace.take } i \ \sigma \ @_{-S} \ (\text{List.take } (i - \text{length } (\text{trace.rest } \sigma)) \ (\text{fst } xsv)),$
if $i \leq \text{length } (\text{trace.rest } \sigma) + \text{length } (\text{fst } xsv)$ *then* *None* *else* *snd xsv*

by (*simp add: trace.continue-def trace.take-def split: option.split*)

lemma *all-iff*:

shows $\text{trace.take } i \ \sigma = \sigma \iff (\text{case } \text{trace.term } \sigma \ \text{of } \text{None} \Rightarrow \text{length } (\text{trace.rest } \sigma) \mid \text{Some } - \Rightarrow \text{Suc } (\text{length } (\text{trace.rest } \sigma))) \leq i \ (\mathbf{is} \ ?thesis1)$

and $\sigma = \text{trace.take } i \ \sigma \iff (\text{case } \text{trace.term } \sigma \ \text{of } \text{None} \Rightarrow \text{length } (\text{trace.rest } \sigma) \mid \text{Some } - \Rightarrow \text{Suc } (\text{length } (\text{trace.rest } \sigma))) \leq i \ (\mathbf{is} \ ?thesis2)$

proof –

show *?thesis1* **by** (*cases* σ) (*simp add: trace.take-def split: option.split*)

then show *?thesis2*

by (*rule eq-commute-conv*)

qed

lemmas *all* = *iffD2[OF trace.take.all-iff(1)]*

lemma *Ex-all*:

shows $\sigma = \text{trace.take } (\text{Suc } (\text{length } (\text{trace.rest } \sigma))) \ \sigma$

by (*simp add: trace.take-def*)

lemma *replicate*:

shows $\text{trace.take } i \ (\text{trace.T } s \ (\text{replicate } j \ as) \ v) = \text{trace.T } s \ (\text{replicate } (\min \ i \ j) \ as) \ (\text{if } i \leq j \ \text{then } \text{None} \ \text{else } v)$

by (*simp add: trace.take-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path continue} \rangle$

lemma *sel[simp]*:

shows $\text{trace.init } (\sigma \ @_{-S} \ xs) = \text{trace.init } \sigma$

and $\text{trace.rest } (\sigma \ @_{-S} \ xsv) = (\text{case } \text{trace.term } \sigma \ \text{of } \text{None} \Rightarrow \text{trace.rest } \sigma \ @ \ \text{fst } xsv \mid \text{Some } v \Rightarrow \text{trace.rest } \sigma)$

and $\text{trace.term } (\sigma \ @_{-S} \ xsv) = (\text{case } \text{trace.term } \sigma \ \text{of } \text{None} \Rightarrow \text{snd } xsv \mid \text{Some } v \Rightarrow \text{trace.term } \sigma)$

by (*simp-all add: trace.continue-def split: option.split*)

lemma *simps[simp]*:

shows $\text{trace.T } s \ xs \ \text{None} \ @_{-S} \ ysv = \text{trace.T } s \ (xs \ @ \ \text{fst } ysv) \ (\text{snd } ysv)$

and $\text{trace.T } s \ xs \ (\text{Some } v) \ @_{-S} \ ysv = \text{trace.T } s \ xs \ (\text{Some } v)$

and $\sigma \ @_{-S} \ ([], \ \text{None}) = \sigma$

by (simp-all add: trace.continue-def trace.t.expand split: option.split)

lemma Nil:

shows $\sigma @_{-S} ([], \text{trace.term } \sigma) = \sigma$

and $\text{trace.T } (\text{trace.init } \sigma) [] \text{ None } @_{-S} (\text{trace.rest } \sigma, \text{trace.term } \sigma) = \sigma$

by (cases σ) (simp-all add: trace.continue-def split: option.split)

lemma map:

shows $\text{trace.map af sf vf } (\sigma @_{-S} xsv) = \text{trace.map af sf vf } \sigma @_{-S} \text{map-prod } (\text{map } (\text{map-prod af sf})) (\text{map-option vf}) xsv$

by (simp add: trace.continue-def split: option.split)

lemma eq-trace-conv:

shows $\sigma @_{-S} xsv = \text{trace.T } s \text{ xs } v \longleftrightarrow \text{trace.init } \sigma = s \wedge (\text{case trace.term } \sigma \text{ of None } \Rightarrow \text{trace.rest } \sigma @ \text{fst } xsv = xs \wedge v = \text{snd } xsv \mid \text{Some } v' \Rightarrow \text{trace.rest } \sigma = xs \wedge v = \text{Some } v')$

and $\text{trace.T } s \text{ xs } v = \sigma @_{-S} xsv \longleftrightarrow \text{trace.init } \sigma = s \wedge (\text{case trace.term } \sigma \text{ of None } \Rightarrow \text{trace.rest } \sigma @ \text{fst } xsv = xs \wedge v = \text{snd } xsv \mid \text{Some } v' \Rightarrow \text{trace.rest } \sigma = xs \wedge v = \text{Some } v')$

by (case-tac $[\!]$ σ) (auto simp: trace.continue-def split: option.split)

lemma self-conv:

shows $(\sigma = \sigma @_{-S} xsv) \longleftrightarrow (\text{case trace.term } \sigma \text{ of None } \Rightarrow xsv = ([], \text{None}) \mid \text{Some } - \Rightarrow \text{True})$

and $(\sigma @_{-S} xsv = \sigma) \longleftrightarrow (\text{case trace.term } \sigma \text{ of None } \Rightarrow xsv = ([], \text{None}) \mid \text{Some } - \Rightarrow \text{True})$

by (cases σ ; cases xsv ; fastforce split: option.splits)+

lemma same-eq:

shows $(\sigma @_{-S} xsv = \sigma @_{-S} ysv) \longleftrightarrow (\text{case trace.term } \sigma \text{ of None } \Rightarrow xsv = ysv \mid \text{Some } - \Rightarrow \text{True})$

by (fastforce simp: trace.continue-def prod.expand split: option.split)

lemma continue:

shows $\sigma @_{-S} xsv @_{-S} ysv = \sigma @_{-S} (\text{case snd } xsv \text{ of None } \Rightarrow (\text{fst } xsv @ \text{fst } ysv, \text{snd } ysv) \mid \text{Some } - \Rightarrow xsv)$

by (simp add: trace.continue-def split: option.split)

lemma take-drop-id:

shows $\text{trace.take } i \sigma @_{-S} \text{case-option } ([], \text{None}) (\lambda\sigma'. (\text{trace.rest } \sigma', \text{trace.term } \sigma')) (\text{trace.dropn } i \sigma) = \sigma$

by (cases σ)

(clarsimp simp: trace.take-def trace.dropn-alt-def split: list.split;

metis append-take-drop-id list.sel(3) tl-drop)

setup $\langle \text{Sign.parent-path} \rangle$

Prefix ordering instantiation $\text{trace.t} :: (\text{type}, \text{type}, \text{type}) \text{ order}$

begin

definition less-eq-t :: $(\text{'a}, \text{'s}, \text{'v}) \text{ trace.t relp}$ **where**

$\text{less-eq-t } \sigma_1 \sigma_2 \longleftrightarrow (\exists xsv. \sigma_2 = \sigma_1 @_{-S} xsv)$

definition less-t :: $(\text{'a}, \text{'s}, \text{'v}) \text{ trace.t relp}$ **where**

$\text{less-t } \sigma_1 \sigma_2 \longleftrightarrow \sigma_1 \leq \sigma_2 \wedge \sigma_1 \neq \sigma_2$

instance

by standard

(auto simp: less-eq-t-def less-t-def trace.continue.self-conv trace.continue.continue trace.continue.same-eq split: option.splits)

end

lemma less-eqE[consumes 1, case-names prefix maximal]:

assumes $\sigma_1 \leq \sigma_2$

assumes $\llbracket \text{trace.term } \sigma_1 = \text{None}; \text{trace.init } \sigma_1 = \text{trace.init } \sigma_2; \text{prefix } (\text{trace.rest } \sigma_1) (\text{trace.rest } \sigma_2) \rrbracket \implies P$
assumes $\bigwedge v. \llbracket \text{trace.term } \sigma_1 = \text{Some } v; \sigma_1 = \sigma_2 \rrbracket \implies P$
shows P
using *assms* **by** (*cases trace.term* σ_1) (*auto simp: trace.less-eq-t-def trace.continue.self-conv*)

lemmas *less-eq-extE*[*consumes 1, case-names prefix maximal*]
 $= \text{trace.less-eqE}$ [*of trace.T s₁ xs₁ v₁ trace.T s₂ xs₂ v₂, simplified, simplified conj-explode*]
for $s_1 \ xs_1 \ v_1 \ s_2 \ xs_2 \ v_2$

lemma *less-eq-self-continue*:
shows $\sigma \leq \sigma @_s xs$
using *trace.less-eq-t-def* **by** *blast*

lemma *less-eq-same-append-conv*:
shows $\text{trace.T } s \ xs \ v \leq \text{trace.T } s' (xs @ ys) \ v' \longleftrightarrow s = s' \wedge (\forall v''. v = \text{Some } v'' \longrightarrow ys = [] \wedge v = v')$
by (*auto simp: trace.less-eq-t-def trace.continue.eq-trace-conv split: option.split*)

lemma *less-same-append-conv*:
shows $\text{trace.T } s \ xs \ v < \text{trace.T } s' (xs @ ys) \ v' \longleftrightarrow s = s' \wedge v = \text{None} \wedge (ys \neq [] \vee (\exists v''. v' = \text{Some } v''))$
by (*cases v*) (*auto simp: trace.less-t-def trace.less-eq-same-append-conv*)

lemma *less-eq-Some[simp]*:
shows $\text{trace.T } s \ xs \ (\text{Some } v) \leq \sigma \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{trace.rest } \sigma = xs \wedge \text{trace.term } \sigma = \text{Some } v$
by (*cases* σ) (*simp add: trace.less-eq-t-def*)

lemma *less-eq-None*:
shows $\sigma \leq \text{trace.T } s \ xs \ \text{None} \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{prefix } (\text{trace.rest } \sigma) \ xs \wedge \text{trace.term } \sigma = \text{None}$
and $\text{trace.T } s \ xs \ \text{None} \leq \sigma \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{prefix } xs \ (\text{trace.rest } \sigma)$
by (*case-tac* $[\!| \sigma]$) (*auto simp: trace.less-eq-same-append-conv elim!: trace.less-eqE prefixE*)

lemma *less*:
shows $\text{trace.T } s \ xs \ v < \sigma \longleftrightarrow \text{trace.init } \sigma = s \wedge (\exists ys. \text{trace.rest } \sigma = xs @ ys \wedge (\text{trace.term } \sigma = \text{None} \longrightarrow ys \neq [])) \wedge v = \text{None}$
and $\sigma < \text{trace.T } s \ xs \ v \longleftrightarrow \text{trace.init } \sigma = s \wedge (\exists ys. xs = \text{trace.rest } \sigma @ ys \wedge (v = \text{None} \longrightarrow ys \neq [])) \wedge \text{trace.term } \sigma = \text{None}$
by (*case-tac* $[\!| \sigma]$)
(auto simp: trace.less-t-def trace.less-eq-t-def trace.continue.eq-trace-conv split: option.split-asm)

lemma *less-eq-take[iff]*:
shows $\text{trace.take } i \ \sigma \leq \sigma$
by (*simp add: trace.take-def take-is-prefix trace.less-eq-None*)

lemma *less-eq-takeE*:
assumes $\sigma_1 \leq \sigma_2$
obtains i **where** $\sigma_1 = \text{trace.take } i \ \sigma_2$
using *assms*
by (*cases* σ_1)
(auto simp: trace.take-def
elim!: trace.less-eqE prefixE
dest: meta-spec[where x=length (trace.rest σ_1)]
meta-spec[where x=Suc (length (trace.rest σ_1))])

lemma *less-eq-take-def*:
shows $\sigma_1 \leq \sigma_2 \longleftrightarrow (\exists i. \sigma_1 = \text{trace.take } i \ \sigma_2)$
by (*blast elim: trace.less-eq-takeE*)

lemma *less-take-less-eq*:
assumes $\sigma < \text{trace.take } (\text{Suc } i) \ \sigma'$

shows $\sigma \leq \text{trace.take } i \sigma'$
using *assms*
by (*clarsimp simp: trace.less-t-def trace.less-eq-take-def trace.take.take*) (*metis le-SucE min-def*)

lemma *wfP-less*:

shows *wfP* ($(<) :: (-, -, -) \text{ trace.t relp}$)
unfolding *wfp-def*
proof(*rule wf-subset[rotated]*)
let $?r = \text{inv-image } (\{(None, Some v) \mid v. True\} <*lex*> \{(x, y). \text{strict-prefix } x \ y\}) (\lambda\sigma. (\text{trace.term } \sigma, \text{trace.rest } \sigma))$
show *wf* $?r$
using *wfp-def wfp-strict-prefix wf-def* **by** *fastforce*
show $\{(x, y). x < y\} \subseteq ?r$
by (*auto simp: trace.less-t-def trace.less-eq-take-def trace.take.all-iff split: option.splits*)
qed

lemma *less-eq-same-cases*:

fixes *ys* :: $(-, -, -) \text{ trace.t}$
assumes $xs_1 \leq ys$
assumes $xs_2 \leq ys$
shows $xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
using *assms*
by (*clarsimp simp: trace.less-eq-take-def trace.take.take*) (*metis min.absorb-iff1 nle-le*)

setup $\langle \text{Sign.mandatory-path take} \rangle$

lemma *mono*:

assumes $\sigma_1 \leq \sigma_2$
assumes $i \leq j$
shows $\text{trace.take } i \sigma_1 \leq \text{trace.take } j \sigma_2$
using *assms*
by (*clarsimp simp: trace.less-eq-take-def trace.take.take*) (*metis min.assoc min.commute min-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path map} \rangle$

lemmas *map* = *trace.t.map-comp[unfolded comp-def]*

lemma *monotone*:

shows *mono* (*trace.map af sf vf*)
by (*rule monoI*) (*fastforce simp: trace.less-eq-take-def trace.take.map*)

lemmas *strengthen[strg]* = *st-monotone[OF trace.map.monotone]*

lemmas *mono* = *monoD[OF trace.map.monotone]*

lemma *monotone-less*:

shows *monotone* ($<$) ($<$) (*trace.map af sf vf*)
by (*rule monotoneI*)
(auto simp: trace.less-t-def trace.map.mono[OF order.strict-implies-order] trace.split-all elim!: trace.less-eqE prefixE)

lemma *less-eqR*:

assumes $\sigma_1 \leq \text{trace.map } af \ sf \ vf \ \sigma_2$
obtains σ_2' **where** $\sigma_2' \leq \sigma_2$ **and** $\sigma_1 = \text{trace.map } af \ sf \ vf \ \sigma_2'$
using *assms* **by** (*fastforce simp: trace.less-eq-take-def trace.take.map*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

lemmas $eq = \text{trace.t.rel-eq}$

lemmas $mono = \text{trace.t.rel-mono-strong}[\text{of } ar \ sr \ vr \ \sigma_1 \ \sigma_2 \ ar' \ sr' \ vr']$ **for** $ar \ sr \ vr \ \sigma_1 \ \sigma_2 \ ar' \ sr' \ vr'$

lemma $\text{strengthen}[\text{strg}]$:

assumes $st\text{-ord } F \ ar \ ar'$

assumes $st\text{-ord } F \ sr \ sr'$

assumes $st\text{-ord } F \ vr \ vr'$

shows $st\text{-ord } F \ (\text{trace.rel } ar \ sr \ vr \ \sigma_1 \ \sigma_2) \ (\text{trace.rel } ar' \ sr' \ vr' \ \sigma_1 \ \sigma_2)$

using assms **by** $(\text{cases } F) \ (\text{auto intro!}:\ \text{le-boolI} \ \text{elim}:\ \text{trace.rel.mono})$

lemma length-rest :

assumes $\text{trace.rel } ar \ sr \ vr \ \sigma_1 \ \sigma_2$

shows $\text{length} \ (\text{trace.rest } \sigma_1)$

$= \text{length} \ (\text{trace.rest } \sigma_2) \wedge (\forall i < \text{length} \ (\text{trace.rest } \sigma_1). \ \text{rel-prod } ar \ sr \ (\text{trace.rest } \sigma_1 \ ! \ i) \ (\text{trace.rest } \sigma_2 \ ! \ i))$

by $(\text{rule } \text{rel-funE}[\text{OF } \text{trace.t.sel-transfer}(2) \ \text{assms}]) \ (\text{simp add}:\ \text{list-all2-conv-all-nth})$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path take} \rangle$

lemma rel :

assumes $\text{trace.rel } ar \ sr \ vr \ \sigma_1 \ \sigma_2$

shows $\text{trace.rel } ar \ sr \ vr \ (\text{trace.take } i \ \sigma_1) \ (\text{trace.take } i \ \sigma_2)$

using assms

by $(\text{auto simp}:\ \text{trace.take-def } \text{trace.t.rel-sel } \text{trace.rel.length-rest} \ \text{elim}:\ \text{rel-funE}[\text{OF } \text{trace.t.sel-transfer}(1)])$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path transitions}' \rangle$

lemma prefix-conv :

shows $\text{prefix} \ (\text{trace.transitions}' \ s \ xs) \ (\text{trace.transitions}' \ s \ ys) \longleftrightarrow \text{prefix } xs \ ys$

proof $(\text{induct } xs \ \text{arbitrary}:\ s \ ys)$

case $(\text{Cons } x \ xs \ s \ ys)$ **then show** $?case$ **by** $(\text{cases } ys) \ \text{auto}$
qed simp

lemma monotone :

shows $\text{monotone prefix prefix} \ (\text{trace.transitions}' \ s)$

by $(\text{rule } \text{monotoneI}) \ (\text{simp add}:\ \text{trace.transitions}'.\text{prefix-conv})$

lemma append :

shows $\text{trace.transitions}' \ s \ (xs \ @ \ ys) = \text{trace.transitions}' \ s \ xs \ @ \ \text{trace.transitions}' \ (\text{trace.final}' \ s \ xs) \ ys$

by $(\text{induct } xs \ \text{arbitrary}:\ s \ ys) \ \text{simp-all}$

lemma eq-Nil-conv :

shows $\text{trace.transitions}' \ s \ xs = [] \longleftrightarrow xs = []$

and $[] = \text{trace.transitions}' \ s \ xs \longleftrightarrow xs = []$

by $(\text{case-tac } [!]\ xs) \ \text{simp-all}$

lemma eq-Cons-conv :

shows $\text{trace.transitions}' \ s \ xs = y \ \# \ ys \longleftrightarrow (\exists a \ s' \ xs'. \ xs = (a, s') \ \# \ xs' \wedge y = (a, s, s') \wedge ys = \text{trace.transitions}' \ s' \ xs')$

and $y \# ys = \text{trace.transitions}' s xs \longleftrightarrow (\exists a s' xs'. xs = (a, s') \# xs' \wedge y = (a, s, s') \wedge ys = \text{trace.transitions}' s' xs')$

by (*case-tac* [!] *xs*) *auto*

lemma *inj-conv*:

shows $\text{trace.transitions}' s xs = \text{trace.transitions}' s ys \longleftrightarrow xs = ys$

by (*induct xs arbitrary: s ys*) (*auto simp: trace.transitions'.eq-Nil-conv trace.transitions'.eq-Cons-conv*)

lemma *continue*:

shows $\text{trace.transitions} (\sigma @_{-S} xsv)$

$= \text{trace.transitions} \sigma @ (\text{case trace.term } \sigma \text{ of None } \Rightarrow \text{trace.transitions}' (\text{trace.final } \sigma) (\text{fst } xsv) \mid \text{Some } v \Rightarrow$

$\square)$

by (*simp add: trace.transitions'.append last-map trace.final'-def split: option.splits*)

lemma *idle-conv*:

shows $\text{set} (\text{trace.transitions}' s xs) \subseteq \text{UNIV} \times \text{Id} \longleftrightarrow \text{snd} \text{ ` set } xs \subseteq \{s\}$

by (*induct xs arbitrary: s*) (*simp; fast*) $+$

lemma *map*:

shows $\text{trace.transitions}' (sf s) (\text{map} (\text{map-prod } af \text{ } sf) xs)$

$= \text{map} (\text{map-prod } af (\text{map-prod } sf \text{ } sf)) (\text{trace.transitions}' s xs)$

by (*induct xs arbitrary: s*) *simp-all*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path transitions} \rangle$

lemma *monotone*:

shows *monotone* (\leq) *prefix trace.transitions*

by (*rule monotoneI*) (*metis prefix-order.eq-iff trace.less-eqE trace.transitions'.prefix-conv*)

lemmas *mono = monotoneD*[*OF trace.transitions.monotone*]

lemma *subseq*:

assumes $\sigma \leq \sigma'$

shows *subseq* ($\text{trace.transitions} \sigma$) ($\text{trace.transitions} \sigma'$)

by (*rule prefix-imp-subseq*[*OF trace.transitions.mono*[*OF assms*]])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

type-synonym $('a, 's) \text{ steps} = ('a \times 's \times 's) \text{ set}$

setup $\langle \text{Sign.mandatory-path trace} \rangle$

definition *steps'* $:: 's \Rightarrow ('a \times 's) \text{ list} \Rightarrow ('a, 's) \text{ steps}$ **where**

$\text{steps}' s xs = \text{set} (\text{trace.transitions}' s xs) - \text{UNIV} \times \text{Id}$

abbreviation (*input*) *steps* $:: ('a, 's, 'v) \text{ trace.t} \Rightarrow ('a, 's) \text{ steps}$ **where**

$\text{steps} \sigma \equiv \text{trace.steps}' (\text{trace.init } \sigma) (\text{trace.rest } \sigma)$

setup $\langle \text{Sign.mandatory-path steps}' \rangle$

lemma *simps*[*simp*]:

shows $\text{trace.steps}' s [] = \{\}$

and $\text{trace.steps}' s ((a, s) \# xs) = \text{trace.steps}' s xs$

and $s \neq \text{snd } x \Longrightarrow \text{trace.steps}' s (x \# xs) = \text{insert} (\text{fst } x, s, \text{snd } x) (\text{trace.steps}' (\text{snd } x) xs)$

and $(a, s', s') \notin \text{trace.steps}' s xs$
and $\text{snd}' \text{ set } xs \subseteq \{s\} \implies \text{trace.steps}' s xs = \{\}$
and $\text{trace.steps}' s [x] = (\text{if } s = \text{snd } x \text{ then } \{\} \text{ else } \{(\text{fst } x, s, \text{snd } x)\})$
by (*simp-all add: trace.steps'-def insert-Diff-if trace.transitions'.idle-conv*)

lemma *Cons-eq-if*:

shows $\text{trace.steps}' s (x \# xs)$
 $= (\text{if } s = \text{snd } x \text{ then } \text{trace.steps}' s xs \text{ else } \text{insert } (\text{fst } x, s, \text{snd } x) (\text{trace.steps}' (\text{snd } x) xs))$
by (*auto simp: trace.steps'-def*)

lemma *stuttering*:

shows $\text{trace.steps}' s xs \subseteq r \cup A \times \text{Id} \iff \text{trace.steps}' s xs \subseteq r$
and $\text{trace.steps}' s xs \subseteq A \times \text{Id} \cup r \iff \text{trace.steps}' s xs \subseteq r$
by (*auto simp: trace.steps'-def*)

lemma *empty-conv*[*simp*]:

shows $\text{trace.steps}' s xs = \{\} \iff \text{snd}' \text{ set } xs \subseteq \{s\}$ (**is** *?thesis1*)
and $\{\} = \text{trace.steps}' s xs \iff \text{snd}' \text{ set } xs \subseteq \{s\}$ (**is** *?thesis2*)

proof –

show *?thesis1* **by** (*simp add: trace.steps'-def trace.transitions'.idle-conv*)
then show *?thesis2*
by (*rule eq-commute-conv*)
qed

lemma *append*:

shows $\text{trace.steps}' s (xs @ ys)$
 $= \text{trace.steps}' s xs \cup \text{trace.steps}' (\text{trace.final}' s xs) ys$
by (*simp add: trace.steps'-def trace.transitions'.append Un-Diff*)

lemma *map*:

shows $\text{trace.steps}' (sf s) (\text{map } (\text{map-prod } af \text{ sf}) xs) = \text{map-prod } af (\text{map-prod } sf sf) \text{ ' trace.steps}' s xs - \text{UNIV} \times \text{Id}$
and $\text{trace.steps}' s (\text{map } (\text{map-prod } af \text{ id}) xs) = \text{map-prod } af \text{ id ' trace.steps}' s xs - \text{UNIV} \times \text{Id}$
by (*force simp: trace.steps'-def trace.transitions'.map trace.transitions'.map[where sf=id, simplified]*)+

lemma *memberD*:

assumes $(a, s, s') \in \text{trace.steps}' s_0 xs$
shows $(a, s') \in \text{set } xs$
using *assms* **by** (*induct xs arbitrary: s_0*) (*auto simp: trace.steps'.Cons-eq-if split: if-split-asm*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path steps} \rangle$

lemma *monotone*:

shows *mono trace.steps*
by (*simp add: monoI trace.steps'-def Diff-mono set-subseq[OF trace.transitions.subseq]*)

lemmas *mono = monoD*[*OF trace.steps.monotone*]

lemmas *strengthen*[*strg*] = *st-monotone*[*OF trace.steps.monotone*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path aset} \rangle$

lemma *simps*:

shows $\text{trace.aset } (\text{trace.T } s xs v) = \text{fst}' \text{ set } xs$
by (*force simp: trace.t.set*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path sset} \rangle$

lemma *simps*:

shows $\text{trace.sset} (\text{trace.T } s \text{ } xs \text{ } v) = \text{insert } s (\text{snd } \text{'set } xs)$

by (*fastforce simp: trace.t.set image-iff*)

lemma *dropn-le*:

assumes $\text{trace.dropn } i \text{ } \sigma = \text{Some } \sigma'$

shows $\text{trace.sset } \sigma' \subseteq \text{trace.sset } \sigma$

using *assms*

by (*cases* σ ; *cases* σ')

(*fastforce simp: trace.dropn-alt-def trace.sset.simps image-iff*

split: list.split-asm

dest: arg-cong[where f=set] in-set-dropD)

lemma *take-le*:

shows $\text{trace.sset} (\text{trace.take } i \text{ } \sigma) \subseteq \text{trace.sset } \sigma$

by (*cases* σ) (*auto simp: trace.take-def trace.sset.simps dest: in-set-takeD*)

lemma *mono*:

shows *mono* trace.sset

by (*rule* *monoI*, *unfold* $\text{trace.less-eq-take-def}$)

(*blast dest: subsetD[OF trace.sset.take-le]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

2.3 Behaviors

setup $\langle \text{Sign.mandatory-path behavior} \rangle$

datatype (*aset*: 'a, *sset*: 's, *vset*: 'v) *t* =

B (*init*: 's) (*rest*: ('a \times 's, 'v) *tlist*)

for

map: *map*

definition *term* :: ('a, 's, 'v) *behavior.t* \Rightarrow 'v *option* **where**

term $\omega = (\text{if } \text{tfinite} (\text{behavior.rest } \omega) \text{ then } \text{Some} (\text{terminal} (\text{behavior.rest } \omega)) \text{ else } \text{None})$

declare *behavior.t.map-id0*[*simp*]

declare *behavior.t.map-id0*[*unfolded id-def, simp*]

declare *behavior.t.map-sel*[*simp*]

declare *behavior.t.set-map*[*simp*]

declare *behavior.t.map-comp*[*unfolded o-def, simp*]

declare *behavior.t.set*[*simp del*]

lemma *split-all*[*no-atp*]: — imitate the setup for 'a \times 'b without the automation

shows $(\bigwedge x. \text{PROP } P \text{ } x) \equiv (\bigwedge s \text{ } xs. \text{PROP } P (\text{behavior.B } s \text{ } xs))$

proof

show $\text{PROP } P (\text{behavior.B } s \text{ } xs)$ **if** $\bigwedge x. \text{PROP } P \text{ } x$ **for** $s \text{ } xs$ **by** (*rule that*)

next

fix *x*

assume $\bigwedge s \text{ } xs. \text{PROP } P (\text{behavior.B } s \text{ } xs)$

from $\langle \text{PROP } P (\text{behavior.B } (\text{behavior.init } x) (\text{behavior.rest } x)) \rangle$ **show** $\text{PROP } P \text{ } x$ **by** *simp*

qed

lemma *split-All[no-atp]*:

shows $(\forall x. P x) \longleftrightarrow (\forall s xs. P (\text{behavior}.B s xs))$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof (*intro iffI allI*)

fix x **assume** $?rhs$ **then show** $P x$ **by** (*cases x simp-all*)

qed *simp*

lemma *split-Ex[no-atp]*:

shows $(\exists x. P x) \longleftrightarrow (\exists s xs. P (\text{behavior}.B s xs))$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof (*intro iffI allI; elim exE*)

fix x **assume** $P x$ **then show** $\exists s xs. P (\text{behavior}.B s xs)$ **by** (*cases x fast*)

qed *auto*

2.4 Combinators on behaviors

definition *continue* :: $(\text{'a}, \text{'s}, \text{'v}) \text{ trace}.t \Rightarrow (\text{'a} \times \text{'s}, \text{'v}) \text{ tllist} \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ behavior}.t$ (**infix** $\langle @_{-B} \rangle$ 64) **where**
 $\sigma @_{-B} xs = \text{behavior}.B (\text{trace}.init \sigma) (\text{tshift2} (\text{trace}.rest \sigma, \text{trace}.term \sigma) xs)$

definition *tl* :: $(\text{'a}, \text{'s}, \text{'v}) \text{ behavior}.t \rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ behavior}.t$ **where**

$tl \omega = (\text{case } \text{behavior}.rest \omega \text{ of } TNil v \Rightarrow None \mid TCons x xs \Rightarrow Some (\text{behavior}.B (snd x) xs))$

definition *dropn* :: $nat \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ behavior}.t \rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ behavior}.t$ **where**

$dropn = (\widetilde{\quad}) \text{ behavior}.tl$

definition *take* :: $nat \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ behavior}.t \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ trace}.t$ **where**

$take i \omega = \text{uncurry} (\text{trace}.T (\text{behavior}.init \omega)) (\text{take } i (\text{behavior}.rest \omega))$

setup $\langle \text{Sign}.mandatory\text{-path } \text{continue} \rangle$

lemma *simps*:

shows $\text{trace}.T s xs None @_{-B} ys = \text{behavior}.B s (\text{tshift } xs ys)$

and $\text{trace}.T s xs (Some v) @_{-B} ys = \text{behavior}.B s (\text{tshift } xs (TNil v))$

and $\text{trace}.T s (x \# xs) w @_{-B} ys = \text{behavior}.B s (TCons x (\text{tshift2 } (xs, w) ys))$

by (*simp-all add: behavior.continue-def*)

lemma *sel[simp]*:

shows *init*: $\text{behavior}.init (\sigma @_{-B} xs) = \text{trace}.init \sigma$

and *rest*: $\text{behavior}.rest (\sigma @_{-B} xs) = \text{tshift2} (\text{trace}.rest \sigma, \text{trace}.term \sigma) xs$

by (*simp-all add: behavior.continue-def*)

lemma *term-None*:

assumes $\text{trace}.term \sigma = None$

shows $\sigma @_{-B} xs = \text{behavior}.B (\text{trace}.init \sigma) (\text{tshift} (\text{trace}.rest \sigma) xs)$

by (*simp add: assms behavior.continue-def*)

lemma *term-Some*:

assumes $\text{trace}.term \sigma = Some v$

shows $\sigma @_{-B} xs = \text{behavior}.B (\text{trace}.init \sigma) (\text{tshift} (\text{trace}.rest \sigma) (TNil v))$

by (*simp add: assms behavior.continue-def*)

lemma *tshift2*:

shows $\sigma @_{-B} \text{tshift2 } xsv ys = (\sigma @_{-S} xsv) @_{-B} ys$

by (*simp add: behavior.continue-def tshift2-def tshift-append split: option.split*)

setup $\langle \text{Sign}.parent\text{-path} \rangle$

setup $\langle \text{Sign}.mandatory\text{-path } tl \rangle$

lemma *TNil*:

shows $\text{behavior.tl } (\text{behavior.B } s \text{ (TNil } v)) = \text{None}$
by (*simp add: behavior.tl-def*)

lemma *TCons*:

shows $\text{behavior.tl } (\text{behavior.B } s \text{ (TCons } x \text{ } xs)) = \text{Some } (\text{behavior.B } (\text{snd } x) \text{ } xs)$
by (*simp add: behavior.tl-def*)

lemma *eq-None-conv*:

shows $\text{behavior.tl } \omega = \text{None} \iff \text{is-TNil } (\text{behavior.rest } \omega)$
by (*simp add: behavior.tl-def split: tllist.split*)

lemma *continue-Cons*:

shows $\text{behavior.tl } (\text{trace.T } s \text{ (} x \text{ \# } xs) \text{ } v \text{ @-}_B \text{ } ys) = \text{Some } (\text{trace.T } (\text{snd } x) \text{ } xs \text{ } v \text{ @-}_B \text{ } ys)$
by (*simp add: behavior.tl-def behavior.continue-def*)

lemmas *simps[simp]* =

behavior.tl.TNil
behavior.tl.TCons
behavior.tl.eq-None-conv
behavior.tl.continue-Cons

lemma *tfiniteD*:

assumes $\text{behavior.tl } \omega = \text{Some } \omega'$
shows $t\text{finite } (\text{behavior.rest } \omega') \iff t\text{finite } (\text{behavior.rest } \omega)$
using *assms* **by** (*auto simp: behavior.tl-def split: tllist.splits*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *dropn-alt-def*:

shows $\text{behavior.dropn } i \text{ } \omega$
 $= (\text{case } t\text{dropn } i \text{ (TCons (undefined, behavior.init } \omega) \text{ (behavior.rest } \omega)) \text{ of}$
 $\text{TNil } - \Rightarrow \text{None}$
 $| \text{TCons } x \text{ } xs \Rightarrow \text{Some } (\text{behavior.B } (\text{snd } x) \text{ } xs))$

proof(*induct i arbitrary: ω*)

case 0 **show** ?*case* **by** (*simp add: behavior.dropn-def*)

next

case (*Suc i ω*) **then show** ?*case*
by (*cases ω ; cases behavior.rest ω ; cases i;*
simp add: behavior.dropn-def tdropn-eq-TNil-conv tdropn-tlength split: tllist.splits)

qed

setup $\langle \text{Sign.mandatory-path dropn} \rangle$

lemma *simps[simp]*:

shows 0: $\text{behavior.dropn } 0 \text{ } \omega = \text{Some } \omega$
and *TNil*: $\text{behavior.dropn } i \text{ (behavior.B } s \text{ (TNil } v)) = (\text{case } i \text{ of } 0 \Rightarrow \text{Some } (\text{behavior.B } s \text{ (TNil } v)) \mid - \Rightarrow \text{None})$
by (*simp-all add: behavior.dropn-def split: nat.splits*)

lemma *TCons*:

shows $\text{behavior.dropn } i \text{ (behavior.B } s \text{ (TCons } x \text{ } xs))$
 $= (\text{case } i \text{ of } 0 \Rightarrow \text{Some } (\text{behavior.B } s \text{ (TCons } x \text{ } xs)) \mid \text{Suc } j \Rightarrow \text{behavior.dropn } j \text{ (behavior.B } (\text{snd } x) \text{ } xs))$
by (*simp add: behavior.dropn-def split: nat.splits*)

lemma *Suc*:

shows $\text{behavior.dropn } (\text{Suc } i) \text{ } \omega = \text{Option.bind } (\text{behavior.tl } \omega) \text{ (behavior.dropn } i)$

by (simp-all add: behavior.dropn-def)

lemma bind-tl-commute:

shows behavior.tl $\omega \gg=$ behavior.dropn $i =$ behavior.dropn $i \omega \gg=$ behavior.tl

by (simp add: behavior.dropn-def pfunpow-swap1)

lemma Suc-right:

shows behavior.dropn (Suc i) $\omega =$ Option.bind (behavior.dropn $i \omega$) behavior.tl

by (simp add: behavior.dropn-def pfunpow-Suc-right del: pfunpow.simps)

lemma dropn:

shows Option.bind (behavior.dropn $i \omega$) (behavior.dropn j) = behavior.dropn ($i + j$) ω

by (simp add: behavior.dropn-def pfunpow-add)

lemma add:

shows behavior.dropn ($i + j$) = ($\lambda\omega$. Option.bind (behavior.dropn $i \omega$) (behavior.dropn j))

by (simp add: fun-eq-iff behavior.dropn.dropn)

lemma tfiniteD:

assumes behavior.dropn $i \omega =$ Some ω'

shows tfinite (behavior.rest ω') \longleftrightarrow tfinite (behavior.rest ω)

using assms

by (induct i arbitrary: ω')

(auto simp: behavior.dropn.Suc-right behavior.tl.tfiniteD split: bind-split-asm)

lemma shorterD:

assumes behavior.dropn $i \omega =$ Some ω'

assumes $j \leq i$

shows $\exists \omega''$. behavior.dropn $j \omega =$ Some ω''

using assms(1) le-Suc-ex[OF assms(2)]

by (clarsimp simp flip: behavior.dropn.dropn split: Option.bind-split-asm)

lemma eq-None-tlength-conv:

shows behavior.dropn $i \omega =$ None \longleftrightarrow tlength (behavior.rest ω) < enat i

proof(induct i arbitrary: ω)

case 0 show ?case by (simp add: enat-0)

next

case (Suc i) then show ?case

by (cases ω ; cases behavior.rest ω ; simp add: behavior.dropn.Suc enat-0-iff flip: eSuc-enat)

qed

lemma eq-Some-tlength-conv:

shows ($\exists \omega'$. behavior.dropn $i \omega =$ Some ω') \longleftrightarrow enat $i \leq$ tlength (behavior.rest ω)

by (metis behavior.dropn.eq-None-tlength-conv leD leI not-None-eq2)

lemma eq-Some-tlengthD:

assumes behavior.dropn $i \omega =$ Some ω'

shows enat $i \leq$ tlength (behavior.rest ω)

using assms behavior.dropn.eq-Some-tlength-conv by blast

lemma tlength-eq-SomeD:

assumes enat $i \leq$ tlength (behavior.rest ω)

shows $\exists \omega'$. behavior.dropn $i \omega =$ Some ω'

using assms behavior.dropn.eq-Some-tlength-conv by blast

lemma eq-Some-tdropnD:

assumes behavior.dropn $i \omega =$ Some ω'

shows tdropn i (behavior.rest ω) = behavior.rest ω'

using *assms*
proof(*induct i arbitrary: ω*)
 case (*Suc i*) **then show** *?case*
 by (*cases ω; cases behavior.rest ω; fastforce simp: behavior.dropn.Suc*)
qed *simp*

lemma *continue-shorter*:
 assumes $i \leq \text{length } (\text{trace.rest } \sigma)$
 shows $\text{behavior.dropn } i (\sigma @_{-B} xs) = \text{Option.bind } (\text{trace.dropn } i \sigma) (\lambda\sigma'. \text{Some } (\sigma' @_{-B} xs))$
using *assms*
proof(*induct i arbitrary: σ*)
 case (*Suc i σ*) **from** *Suc.prem*s **show** *?case*
 by (*cases σ; cases trace.rest σ*)
 (*simp-all add: behavior.dropn.Suc flip: Suc.hyps*)
qed *simp*

lemma *continue-Some*:
 assumes $\text{length } (\text{trace.rest } \sigma) < i$
 assumes $\text{trace.term } \sigma = \text{Some } v$
 shows $\text{behavior.dropn } i (\sigma @_{-B} xs) = \text{None}$
using *assms* **by** (*simp add: behavior.dropn.eq-None-tlength-conv tlength-tshift2 tlength-tshift*)

lemma *continue-None*:
 assumes $\text{length } (\text{trace.rest } \sigma) < i$
 assumes $\text{trace.term } \sigma = \text{None}$
 shows $\text{behavior.dropn } i (\sigma @_{-B} xs)$
 $= (\text{case } \text{tdropn } (i - \text{Suc } (\text{length } (\text{trace.rest } \sigma)))) \text{ xs of}$
 $\text{TNil} - \Rightarrow \text{None}$
 $| \text{TCons } y \text{ ys} \Rightarrow \text{Some } (\text{behavior.B } (\text{snd } y) \text{ ys})$
using *assms* **by** (*cases i*) (*auto simp: behavior.dropn-alt-def tdropn-tshift*)

lemma *continue*:
 shows $\text{behavior.dropn } i (\sigma @_{-B} xs)$
 $= (\text{if } i \leq \text{length } (\text{trace.rest } \sigma)$
 $\text{then } \text{Option.bind } (\text{trace.dropn } i \sigma) (\lambda\sigma'. \text{Some } (\sigma' @_{-B} xs))$
 $\text{else if } \text{trace.term } \sigma = \text{None}$
 $\text{then case } \text{tdropn } (i - \text{Suc } (\text{length } (\text{trace.rest } \sigma))) \text{ xs of}$
 $\text{TNil} - \Rightarrow \text{None}$
 $| \text{TCons } y \text{ ys} \Rightarrow \text{Some } (\text{behavior.B } (\text{snd } y) \text{ ys})$
 $\text{else None})$
by (*clarsimp simp: behavior.dropn.continue-None behavior.dropn.continue-Some*
 behavior.dropn.continue-shorter)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path trace.take.behavior} \rangle$

lemma *take*:
 shows $\text{trace.take } i (\text{behavior.take } j \omega) = \text{behavior.take } (\text{min } i \text{ } j) \omega$
by (*simp add: behavior.take-def trace.take-def split-def*
 ttake-eq-None-conv ttake-flat length-ttake take-fst-ttake
 split: enat.split split-min)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path behavior} \rangle$

setup $\langle \text{Sign.mandatory-path take} \rangle$

lemma *simps[simp]*:

shows $0: \text{behavior.take } 0 \ \omega = \text{trace.T } (\text{behavior.init } \omega) \ [] \ \text{None}$

and *Suc-TNil*: $\text{behavior.take } (\text{Suc } i) (\text{behavior.B } s (\text{TNil } v)) = \text{trace.T } s \ [] \ (\text{Some } v)$

by (*simp-all add: behavior.take-def*)

lemma *sel[simp]*:

shows $\text{trace.init } (\text{behavior.take } i \ \omega) = \text{behavior.init } \omega$

and $\text{trace.rest } (\text{behavior.take } i \ \omega) = \text{fst } (\text{ttake } i (\text{behavior.rest } \omega))$

and $\text{trace.term } (\text{behavior.take } i \ \omega) = \text{snd } (\text{ttake } i (\text{behavior.rest } \omega))$

by (*simp-all add: behavior.take-def split-def*)

lemma *monotone*:

shows *mono* $(\lambda i. \text{behavior.take } i \ \omega)$

by (*rule monoI*) (*fastforce simp: trace.less-eq-take-def trace.take.behavior.take min-def*)

lemmas *mono = monoD[OF behavior.take.monotone]*

lemma *map*:

shows $\text{behavior.take } i (\text{behavior.map } af \ sf \ vf \ \omega) = \text{trace.map } af \ sf \ vf (\text{behavior.take } i \ \omega)$

by (*induct i arbitrary: ω*) (*simp-all add: behavior.take-def split-def ttake-tmap split: tllist.split*)

lemma *continue*:

shows $\text{behavior.take } i (\sigma \ @_{-B} \ \omega) = \text{trace.take } i \ \sigma \ @_{-S} \ \text{ttake } (i - \text{length } (\text{trace.rest } \sigma)) \ \omega$

by (*cases σ*)

(*auto simp: behavior.take-def split-def trace.take-def ttake-tshift2 ttake-TNil*

split: option.split nat.split)

lemma *all-continue*:

assumes $\text{tlength } (\text{behavior.rest } \omega) < \text{enat } i$

shows $\text{behavior.take } i \ \omega \ @_{-S} \ xsv = \text{behavior.take } i \ \omega$

using *assms*

by (*auto simp: behavior.take-def split-def trace.continue-def ttake-eq-None-conv split: option.split*)

lemma *continue-same*:

shows $\text{behavior.take } i (\text{behavior.take } i \ \omega \ @_{-B} \ xsv) = \text{behavior.take } i \ \omega$

by (*auto simp: behavior.take-def split-def ttake-tshift2 length-ttake*

ttake-eq-None-conv ttake-eq-Nil-conv ttake-eq-Some-conv ttake-TNil

split: enat.split nat.split option.split)

lemma *trePLICATE*:

shows $\text{behavior.take } i (\text{behavior.B } s (\text{trePLICATE } j \ as \ v))$

$= \text{trace.T } s (\text{List.replicate } (\text{min } i \ j) \ as) \ (\text{if } j < i \ \text{then } \text{Some } v \ \text{else } \text{None})$

by (*simp add: behavior.take-def*)

lemma *trepeat*:

shows $\text{behavior.take } i (\text{behavior.B } s (\text{trepeat } as)) = \text{trace.T } s (\text{List.replicate } i \ as) \ \text{None}$

by (*simp add: behavior.take-def ttake-trepeat*)

lemma *tshift*:

shows $\text{behavior.take } i (\text{behavior.B } s (\text{tshift } xs \ ys)) = \text{trace.take } i (\text{trace.T } s \ xs \ \text{None}) \ @_{-S} \ \text{ttake } (i - \text{length } xs)$

by (*simp add: behavior.take-def trace.take-def ttake-tshift split-def*)

lemma *length*:

shows $length (trace.rest (behavior.take\ j\ \omega))$
 $= (case\ tlength\ (behavior.rest\ \omega)\ of\ enat\ i\ \Rightarrow\ min\ i\ j\ | \infty\ \Rightarrow\ j)$
by (*auto simp: length-ttake split: enat.split*)

lemma add:

shows $behavior.take\ (i + j)\ \omega$
 $= behavior.take\ i\ \omega\ @_{-S}\ (case\ behavior.dropn\ i\ \omega\ of\ Some\ \omega'\ \Rightarrow\ ttake\ j\ (behavior.rest\ \omega'))$
by (*auto simp: behavior.take-def split-def ttake-add Let-def behavior.dropn.eq-Some-tdropnD*
 $behavior.dropn.eq-None-tlength-conv$
 $dest: iffD1[OF\ ttake-eq-None-conv(1)]$
 $split: option.split$)

lemma term-Some-conv:

shows $trace.term\ (behavior.take\ j\ \omega) = Some\ v$
 $\iff (tlength\ (behavior.rest\ \omega) < enat\ j \wedge Some\ v = behavior.term\ \omega)$
by (*auto simp: behavior.term-def ttake-eq-Some-conv tfinite-tlength-conv*)

lemma dropn:

assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
shows $behavior.take\ j\ \omega' = the\ (trace.dropn\ i\ (behavior.take\ (i + j)\ \omega))$
using *assms*
proof(*induct i arbitrary: j \omega \omega'*)
case (*Suc i j \omega*) **then show** ?*case*
by (*cases \omega; cases behavior.rest \omega; cases i*)
(*auto simp: behavior.dropn.Suc behavior.take-def split-def*)
qed *simp*

lemma continue-id:

assumes $tlength\ (behavior.rest\ \omega) < enat\ i$
shows $behavior.take\ i\ \omega\ @_{-B}\ xs = \omega$
using *assms* **by** (*simp add: behavior.continue-def tshift2-ttake-shorter*)

lemma flat:

assumes $tlength\ (behavior.rest\ \omega) < enat\ i$
assumes $i \leq j$
shows $behavior.take\ i\ \omega = behavior.take\ j\ \omega$
using *assms* **by** (*simp add: behavior.take-def ttake-flat*)

lemma eqI:

assumes $\bigwedge i. behavior.take\ i\ \omega_1 = behavior.take\ i\ \omega_2$
shows $\omega_1 = \omega_2$
using *assms*
by (*cases \omega_1; cases \omega_2; simp add: behavior.take-def case-prod-beta prod-eq-iff ttake-eq-imp-eq*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ continue \rangle$

lemma take-drop-shorter:

assumes $i \leq j$
shows $behavior.take\ i\ \omega\ @_{-S}\ apfst\ (drop\ i)\ (ttake\ j\ (behavior.rest\ \omega)) = behavior.take\ j\ \omega$
using *assms*
by (*simp add: trace.continue-def behavior.take.flat ttake-eq-None-conv ttake-eq-Some-conv trace.t.expand*
 $flip: take-fst-ttake[\mathbf{where\ } i=i\ \mathbf{and\ } j=j, simplified\ min-absorb1[OF\ assms]]$
 $split: option.split$)

lemma take-drop-id:

shows $behavior.take\ i\ \omega\ @_{-B}\ behavior.rest\ (the\ (behavior.dropn\ i\ \omega)) = \omega$

by (*cases* *tlength* (*behavior.rest* ω) < *enat* *i*)
 (*simp add*: *behavior.take.continue-id*,
simp add: *behavior.continue-def tshift2-ttake-tdropn-id behavior.dropn.tlength-eq-SomeD*
flip: *behavior.dropn.eq-Some-tdropnD*)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path aset* \rangle

lemma *simps*:

shows *behavior.aset* (*behavior.B s xs*) = *fst* ‘ *tset xs*
by (*force simp*: *behavior.t.set*)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path sset* \rangle

lemma *simps*:

shows *behavior.sset* (*behavior.B s xs*) = *insert s* (*snd* ‘ *tset xs*)
by (*fastforce simp*: *behavior.t.set image-iff*)

lemma *dropn-le*:

assumes *behavior.dropn i* ω = *Some* ω'
shows *behavior.sset* $\omega' \subseteq$ *behavior.sset* ω
using *assms*
by (*cases* ω ; *cases* ω')
 (*fastforce simp*: *behavior.dropn-alt-def behavior.sset.simps image-iff*
split: *tllist.split-asm*
dest: *arg-cong*[**where** *f=tset*] *in-tset-tdropnD*)

lemma *take-le*:

shows *trace.sset* (*behavior.take i* ω) \subseteq *behavior.sset* ω
by (*cases* ω)
 (*auto simp*: *behavior.take-def behavior.sset.simps trace.sset.simps split-def*
dest: *in-set-ttakeD*)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path trace.dropn.behavior* \rangle

lemma *take*:

shows *trace.dropn i* (*behavior.take j* ω)
 = (*if* $i \leq j$ *then* *Option.bind* (*behavior.dropn i* ω) ($\lambda\omega'$. *Some* (*behavior.take* ($j - i$) ω'))
else *None*)

proof(*cases* $i \leq j$)

case *False* **then show** *?thesis*

by (*clarsimp simp*: *trace.dropn.eq-none-length-conv length-ttake split*: *enat.split*) *linarith*
next

case *True* **then show** *?thesis*

proof(*induct j arbitrary*: *i* ω)

case (*Suc j i* ω) **then show** *?case*

by (*cases* ω ; *cases* *i*)

(*auto simp*: *behavior.dropn.Suc behavior.take-def split-def behavior.split-all*
split: *tllist.splits*)

qed *simp*

qed

setup $\langle \text{Sign.parent-path} \rangle$

3 Point-free notation

Typically we define predicates as functions of a state. The following provide a somewhat comfortable point-free imitation of Isabelle/HOL's operators.

type-synonym $'s \text{ pred} = 's \Rightarrow \text{bool}$

abbreviation (*input*)

$\text{pred-}K :: 'b \Rightarrow 'a \Rightarrow 'b \langle \langle - \rangle \rangle$ **where**
 $\langle f \rangle \equiv \lambda s. f$

abbreviation (*input*)

$\text{pred-not} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \langle \langle \neg \rightarrow [40] 40 \rangle \rangle$ **where**
 $\neg a \equiv \lambda s. \neg a s$

abbreviation (*input*)

$\text{pred-conj} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \langle \langle \wedge \rangle 35 \rangle$ **where**
 $a \wedge b \equiv \lambda s. a s \wedge b s$

abbreviation (*input*)

$\text{pred-disj} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \langle \langle \vee \rangle 30 \rangle$ **where**
 $a \vee b \equiv \lambda s. a s \vee b s$

abbreviation (*input*)

$\text{pred-implies} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \langle \langle \longrightarrow \rangle 25 \rangle$ **where**
 $a \longrightarrow b \equiv \lambda s. a s \longrightarrow b s$

abbreviation (*input*)

$\text{pred-iff} :: 'a \text{ pred} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred} \langle \langle \longleftrightarrow \rangle 25 \rangle$ **where**
 $a \longleftrightarrow b \equiv \lambda s. a s \longleftrightarrow b s$

abbreviation (*input*)

$\text{pred-eq} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \langle \langle \Rightarrow \rangle 40 \rangle$ **where**
 $a = b \equiv \lambda s. a s = b s$

abbreviation (*input*)

$\text{pred-neq} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \langle \langle \neq \rangle 40 \rangle$ **where**
 $a \neq b \equiv \lambda s. a s \neq b s$

abbreviation (*input*)

$\text{pred-If} :: 'a \text{ pred} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \langle \langle \text{If } (-) / \text{Then } (-) / \text{Else } (-) \rangle [0, 0, 10] 10 \rangle$
where $\text{If } P \text{ Then } x \text{ Else } y \equiv \lambda s. \text{if } P s \text{ then } x s \text{ else } y s$

abbreviation (*input*)

$\text{pred-less} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \langle \langle < \rangle 40 \rangle$ **where**
 $a < b \equiv \lambda s. a s < b s$

abbreviation (*input*)

$\text{pred-less-eq} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \langle \langle \leq \rangle 40 \rangle$ **where**
 $a \leq b \equiv \lambda s. a s \leq b s$

abbreviation (*input*)

$\text{pred-greater} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ pred} \langle \langle > \rangle 40 \rangle$ **where**
 $a > b \equiv \lambda s. a s > b s$

abbreviation (*input*)

pred-greater-eq :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ 'b) ⇒ 'a *pred* (**infix** <≥> 40) **where**
 $a \geq b \equiv \lambda s. a\ s \geq b\ s$

abbreviation (*input*)

pred-plus :: ('a ⇒ 'b::plus) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b (**infixl** <+> 65) **where**
 $a + b \equiv \lambda s. a\ s + b\ s$

abbreviation (*input*)

pred-minus :: ('a ⇒ 'b::minus) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b (**infixl** <-> 65) **where**
 $a - b \equiv \lambda s. a\ s - b\ s$

abbreviation (*input*)

pred-times :: ('a ⇒ 'b::times) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b (**infixl** <*> 65) **where**
 $a * b \equiv \lambda s. a\ s * b\ s$

abbreviation (*input*)

pred-all :: ('b ⇒ 'a *pred*) ⇒ 'a *pred* (**binder** <∀> 10) **where**
 $\forall x. P\ x \equiv \lambda s. \forall x. P\ x\ s$

abbreviation (*input*)

pred-ex :: ('b ⇒ 'a *pred*) ⇒ 'a *pred* (**binder** <∃> 10) **where**
 $\exists x. P\ x \equiv \lambda s. \exists x. P\ x\ s$

abbreviation (*input*)

pred-app :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c (**infixl** <\$> 100) **where**
 $f\ \$\ g \equiv \lambda s. f\ s\ (g\ s)$

abbreviation (*input*)

pred-app' :: ('b ⇒ 'a ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c (**infixl** <\$ \$> 100) **where**
 $f\ \$ \$\ g \equiv \lambda s. f\ (g\ s)\ s$

abbreviation (*input*)

pred-member :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b *set*) ⇒ 'a *pred* (**infix** <∈> 40) **where**
 $a \in b \equiv \lambda s. a\ s \in b\ s$

abbreviation (*input*)

pred-subseteq :: ('a ⇒ 'b *set*) ⇒ ('a ⇒ 'b *set*) ⇒ 'a *pred* (**infix** <⊆> 50) **where**
 $A \subseteq B \equiv \lambda s. A\ s \subseteq B\ s$

abbreviation (*input*)

pred-union :: ('a ⇒ 'b *set*) ⇒ ('a ⇒ 'b *set*) ⇒ 'a ⇒ 'b *set* (**infixl** <∪> 65) **where**
 $a \cup b \equiv \lambda s. a\ s \cup b\ s$

abbreviation (*input*)

pred-inter :: ('a ⇒ 'b *set*) ⇒ ('a ⇒ 'b *set*) ⇒ 'a ⇒ 'b *set* (**infixl** <∩> 65) **where**
 $a \cap b \equiv \lambda s. a\ s \cap b\ s$

abbreviation (*input*)

pred-conjoin :: 'a *pred list* ⇒ 'a *pred* **where**
 $\text{pred-conjoin } xs \equiv \text{foldr } (\wedge) \ xs \ \langle \text{True} \rangle$

abbreviation (*input*)

pred-disjoin :: 'a *pred list* ⇒ 'a *pred* **where**
 $\text{pred-disjoin } xs \equiv \text{foldr } (\vee) \ xs \ \langle \text{False} \rangle$

abbreviation (*input*)

$\text{pred-min} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
 $\text{pred-min } x \ y \equiv \lambda s. \text{min } (x \ s) \ (y \ s)$

abbreviation (*input*)

$\text{pred-max} :: ('a \Rightarrow 'b::\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
 $\text{pred-max } x \ y \equiv \lambda s. \text{max } (x \ s) \ (y \ s)$

abbreviation (*input*)

$\text{NULL} :: ('a \Rightarrow 'b \text{ option}) \Rightarrow 'a \text{ pred}$ **where**
 $\text{NULL } a \equiv \lambda s. a \ s = \text{None}$

abbreviation (*input*)

$\text{EMPTY} :: ('a \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ pred}$ **where**
 $\text{EMPTY } a \equiv \lambda s. a \ s = \{\}$

abbreviation (*input*)

$\text{LIST-NULL} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \text{ pred}$ **where**
 $\text{LIST-NULL } a \equiv \lambda s. a \ s = []$

abbreviation (*input*)

$\text{SIZE} :: ('a \Rightarrow 'b::\text{size}) \Rightarrow 'a \Rightarrow \text{nat}$ **where**
 $\text{SIZE } a \equiv \lambda s. \text{size } (a \ s)$

abbreviation (*input*)

$\text{SET} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \Rightarrow 'b \text{ set}$ **where**
 $\text{SET } a \equiv \lambda s. \text{set } (a \ s)$

abbreviation (*input*)

$\text{pred-singleton} :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \text{ set}$ **where**
 $\text{pred-singleton } x \equiv \lambda s. \{x \ s\}$

abbreviation (*input*)

$\text{pred-list-nth} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow ('a \Rightarrow \text{nat}) \Rightarrow 'a \Rightarrow 'b$ (**infixl** $\langle ! \rangle$ 150) **where**
 $xs \ ! \ i \equiv \lambda s. xs \ s \ ! \ i \ s$

abbreviation (*input*)

$\text{pred-list-append} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \Rightarrow 'b \text{ list}$ (**infixr** $\langle @ \rangle$ 65) **where**
 $xs \ @ \ ys \equiv \lambda s. xs \ s \ @ \ ys \ s$

abbreviation (*input*)

$\text{FST} :: 'a \text{ pred} \Rightarrow ('a \times 'b) \text{ pred}$ **where**
 $\text{FST } P \equiv \lambda s. P \ (\text{fst } s)$

abbreviation (*input*)

$\text{SND} :: 'b \text{ pred} \Rightarrow ('a \times 'b) \text{ pred}$ **where**
 $\text{SND } P \equiv \lambda s. P \ (\text{snd } s)$

abbreviation (*input*)

$\text{pred-pair} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infixr** $\langle \otimes \rangle$ 60) **where**
 $a \ \otimes \ b \equiv \lambda s. (a \ s, b \ s)$

4 More lattice

lemma (**in** *semilattice-sup*) *sup-iff-le*:

shows $x \sqcup y = y \iff x \leq y$
and $y \sqcup x = y \iff x \leq y$

by (*simp-all add: le-iff-sup ac-simps*)

lemma (*in semilattice-inf*) *inf-iff-le*:

shows $x \sqcap y = x \iff x \leq y$

and $y \sqcap x = x \iff x \leq y$

by (*simp-all add: le-iff-inf ac-simps*)

lemma *if-sup-distr*:

fixes $t e :: \text{semilattice-sup}$

shows *if-sup-distrL*: $(\text{if } b \text{ then } t_1 \sqcup t_2 \text{ else } e) = (\text{if } b \text{ then } t_1 \text{ else } e) \sqcup (\text{if } b \text{ then } t_2 \text{ else } e)$

and *if-sup-distrR*: $(\text{if } b \text{ then } t \text{ else } e_1 \sqcup e_2) = (\text{if } b \text{ then } t \text{ else } e_1) \sqcup (\text{if } b \text{ then } t \text{ else } e_2)$

by (*simp-all split: if-splits*)

lemma *INF-bot*:

assumes $F i = (\perp :: \text{complete-lattice})$

assumes $i \in X$

shows $(\prod_{i \in X}. F i) = \perp$

using *assms* by (*metis INF-lower bot.extremum-uniqueI*)

lemma *mcont-fun-app-const[cont-intro]*:

shows $mcont \text{ Sup } (\leq) \text{ Sup } (\leq) (\lambda f. f c)$

by (*fastforce intro!: mcontI monotoneI contI simp: le-fun-def*)

declare *mcont-applyI[cont-intro]*

lemma *INF-rename-bij*:

assumes *bij-betw* $\pi X Y$

shows $(\prod_{y \in Y}. F Y y) = (\prod_{x \in X}. F (\pi ' X) (\pi x))$

using *assms* by (*metis bij-betw-imp-surj-on image-image*)

lemma *Inf-rename-surj*:

assumes *surj* π

shows $(\prod x. F x) = (\prod x. F (\pi x))$

using *assms* by (*metis image-image*)

lemma *INF-unwind-index*:

fixes $A :: \text{complete-lattice}$

assumes $i \in I$

shows $(\prod_{x \in I}. A x) = A i \sqcap (\prod_{x \in I - \{i\}}. A x)$

by (*metis INF-insert assms insert-Diff*)

lemma *Sup-fst*:

shows $(\bigsqcup_{x \in X}. P (\text{fst } x)) = (\bigsqcup_{x \in \text{fst } ' X}. P x)$

by (*simp add: image-image*)

setup $\langle \text{Sign.mandatory-path order} \rangle$

lemma *assms-cong*: — simplify assumptions only

assumes $x = x'$

shows $x \leq y \iff x' \leq y$

using *assms* by *simp*

lemma *concl-cong*: — simplify conclusions only

assumes $y = y'$

shows $x \leq y \iff x \leq y'$

using *assms* by *simp*

lemma *subgoal*: — cut for lattice logics

fixes $P :: \text{semilattice-inf}$
assumes $P \leq Q$
assumes $P \sqcap Q \leq R$
shows $P \leq R$
using *assms* **by** (*simp add: inf-absorb1*)

setup $\langle \text{Sign.parent-path} \rangle$

Logical rules ala HOL **lemmas** $\text{SupI} = \text{Sup-upper}$
lemmas $\text{rev-SUPI} = \text{SUP-upper2}$ [of $x A b f$ for $x A b f$]
lemmas $\text{SUPI} = \text{rev-SUPI}$ [rotated]

lemmas $\text{SUPE} = \text{SUP-least}$ [where $u=z$ for z]
lemmas $\text{SupE} = \text{Sup-least}$

lemmas $\text{INFI} = \text{INF-greatest}$
lemmas $\text{InfI} = \text{Inf-greatest}$
lemmas $\text{infI} = \text{semilattice-inf-class.le-infI}$

lemma InfE :
fixes $R :: \text{complete-lattice}$
assumes $P x \leq R$
shows $(\sqcap x. P x) \leq R$
using *assms* **by** (*meson Inf-lower2 rangeI*)

lemma INFE :
fixes $R :: \text{complete-lattice}$
assumes $P x \leq R$
assumes $x \in A$
shows $\sqcap (P \text{ ` } A) \leq R$
using *assms* **by** (*metis INF-lower2*)

lemmas $\text{rev-INFE} = \text{INFE}$ [rotated]

lemma Inf-inf-distrib :
fixes $P :: \text{complete-lattice}$
shows $(\sqcap x. P x \sqcap Q x) = (\sqcap x. P x) \sqcap (\sqcap x. Q x)$
by (*simp add: INF-inf-distrib*)

lemma Sup-sup-distrib :
fixes $P :: \text{complete-lattice}$
shows $(\sqcup x. P x \sqcup Q x) = (\sqcup x. P x) \sqcup (\sqcup x. Q x)$
by (*simp add: SUP-sup-distrib*)

lemma Inf-inf :
fixes $Q :: \text{complete-lattice}$
shows $(\sqcap x. P x \sqcap Q) = (\sqcap x. P x) \sqcap Q$
by (*simp add: INF-inf-const2*)

lemma inf-Inf :
fixes $P :: \text{complete-lattice}$
shows $(\sqcap x. P \sqcap Q x) = P \sqcap (\sqcap x. Q x)$
by (*simp add: INF-inf-const1*)

lemma SUP-sup :
fixes $Q :: \text{complete-lattice}$
assumes $X \neq \{\}$
shows $(\sqcup x \in X. P x \sqcup Q) = (\sqcup x \in X. P x) \sqcup Q$ (*is ?lhs = ?rhs*)

proof(*rule antisym*)

show $?lhs \leq ?rhs$ **by** (*simp add: SUP-le-iff SupI le-supI1*)

from *assms* **show** $?rhs \leq ?lhs$ **by** (*auto simp add: SUP-le-iff intro: SUPI le-supI1*)
qed

lemma *sup-SUP*:

fixes $P :: \text{::complete-lattice}$

assumes $X \neq \{\}$

shows $(\bigsqcup_{x \in X}. P \sqcup Q x) = P \sqcup (\bigsqcup_{x \in X}. Q x)$

using *SUP-sup[OF assms, where P=Q and Q=P]* **by** (*simp add: ac-simps*)

4.1 Boolean lattices and implication

lemma

shows *minus-Not[simp]*: $- \text{Not} = \text{id}$

and *minus-id[simp]*: $- \text{id} = \text{Not}$

by *fastforce+*

definition *boolean-implication* :: $'a::\text{boolean-algebra} \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\langle \longrightarrow_B \rangle$ 60) **where**

$x \longrightarrow_B y = -x \sqcup y$

definition *boolean-eq* :: $'a::\text{boolean-algebra} \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\langle \longleftrightarrow_B \rangle$ 60) **where**

$x \longleftrightarrow_B y = x \longrightarrow_B y \sqcap y \longrightarrow_B x$

setup $\langle \text{Sign.mandatory-path boolean-implication} \rangle$

lemma *bool-alt-def[simp]*:

shows $P \longrightarrow_B Q = (P \longrightarrow Q)$

by (*auto simp: boolean-implication-def*)

lemma *pred--alt-def[simp]*:

shows $(P \longrightarrow_B Q) x = (P x \longrightarrow_B Q x)$

by (*auto simp: boolean-implication-def*)

lemma *set-alt-def*:

shows $P \longrightarrow_B Q = \{x. x \in P \longrightarrow x \in Q\}$

by (*auto simp: boolean-implication-def*)

lemma *member*:

shows $x \in P \longrightarrow_B Q \longleftrightarrow x \in P \longrightarrow x \in Q$

by (*simp add: boolean-implication.set-alt-def*)

lemmas *setI = iffD2[OF boolean-implication.member, rule-format]*

lemma *simps[simp]*:

shows

$\top \longrightarrow_B P = P$

$\perp \longrightarrow_B P = \top$

$P \longrightarrow_B \top = \top$

$P \longrightarrow_B P = \top$

$P \longrightarrow_B \perp = -P$

$P \longrightarrow_B -P = -P$

by (*simp-all add: boolean-implication-def shunt1*)

lemma *Inf-simps[simp]*: — Miniscoping: pushing in universal quantifiers.

shows

$\bigwedge P (Q::\text{::complete-boolean-algebra}). \quad (\bigcap x. P x \longrightarrow_B Q) = ((\bigsqcup x. P x) \longrightarrow_B Q)$

$\bigwedge P (Q::\text{::complete-boolean-algebra}). \quad (\bigcap x \in X. P x \longrightarrow_B Q) = ((\bigsqcup x \in X. P x) \longrightarrow_B Q)$

$\wedge P (Q::\Rightarrow::\text{complete-boolean-algebra}). (\prod x. P \longrightarrow_B Q x) = (P \longrightarrow_B (\prod x. Q x))$
 $\wedge P (Q::\Rightarrow::\text{complete-boolean-algebra}). (\prod x \in X. P \longrightarrow_B Q x) = (P \longrightarrow_B (\prod x \in X. Q x))$
by (*simp-all add: boolean-implication-def INF-sup sup-INF uminus-SUP*)

lemma mono:

assumes $x' \leq x$

assumes $y \leq y'$

shows $x \longrightarrow_B y \leq x' \longrightarrow_B y'$

using *assms* **by** (*simp add: boolean-implication-def sup.coboundedI1 sup.coboundedI2*)

lemma strengthen[*strg*]:

assumes *st-ord* $(\neg F) X X'$

assumes *st-ord* $F Y Y'$

shows *st-ord* $F (X \longrightarrow_B Y) (X' \longrightarrow_B Y')$

using *assms* **by** (*cases F*) (*use boolean-implication.mono in auto*)

lemma eq-conv:

shows $(P = Q) \longleftrightarrow (P \longrightarrow_B Q) \sqcap (Q \longrightarrow_B P) = \top$

unfolding *boolean-implication-def order.eq-iff* **by** (*simp add: sup-shunt top.extremum-unique*)

lemma uminus-imp[*simp*]:

shows $\neg(P \longrightarrow_B Q) = P \sqcap \neg Q$

by (*simp add: boolean-implication-def*)

lemma cases-simp[*simp*]:

shows $(P \longrightarrow_B Q) \sqcap (\neg P \longrightarrow_B Q) = Q$

by (*simp add: boolean-implication-def order.eq-iff boolean-algebra.disj-conj-distrib2 shunt1*)

lemma conv-sup:

shows $(P \longrightarrow_B Q) = \neg P \sqcup Q$

by (*rule boolean-implication-def*)

lemma infL:

shows $P \sqcap Q \longrightarrow_B R = P \longrightarrow_B Q \longrightarrow_B R$

by (*simp add: boolean-implication-def sup-assoc*)

lemmas uncurry = *boolean-implication.infL[symmetric]*

lemma shunt1:

shows $x \sqcap y \leq z \longleftrightarrow x \leq y \longrightarrow_B z$

by (*simp add: boolean-implication-def shunt1*)

lemma shunt2:

shows $x \sqcap y \leq z \longleftrightarrow y \leq x \longrightarrow_B z$

by (*subst inf commute*) (*rule boolean-implication.shunt1*)

lemma mp:

assumes $x \sqcap y \leq z$

shows $x \leq y \longrightarrow_B z$

using *assms* **by** (*simp add: boolean-implication.shunt1*)

lemma imp-trivialI:

assumes $P \sqcap \neg R \leq \neg Q$

shows $P \leq Q \longrightarrow_B R$

using *assms* **by** (*simp add: boolean-implication-def shunt2 sup commute*)

lemma shunt-top:

shows $P \longrightarrow_B Q = \top \longleftrightarrow P \leq Q$

by (simp add: boolean-implication-def sup-shunt)

lemma detachment:

shows $x \sqcap (x \longrightarrow_B y) = x \sqcap y$ (is ?thesis1)

and $(x \longrightarrow_B y) \sqcap x = x \sqcap y$ (is ?thesis2)

proof –

show ?thesis1 by (simp add: boolean-algebra.conj-disj-distrib boolean-implication-def)

then show ?thesis2 by (simp add: ac-simps)

qed

lemma discharge:

assumes $x' \leq x$

shows $x' \sqcap (x \longrightarrow_B y) = x' \sqcap y$ (is ?thesis1)

and $(x \longrightarrow_B y) \sqcap x' = y \sqcap x'$ (is ?thesis2)

proof –

from assms show ?thesis1

by (simp add: boolean-implication-def inf-sup-distrib sup.absorb2 le-supI1 flip: sup-neg-inf)

then show ?thesis2

by (simp add: ac-simps)

qed

lemma trans:

shows $(x \longrightarrow_B y) \sqcap (y \longrightarrow_B z) \leq (x \longrightarrow_B z)$

by (simp add: boolean-implication-def inf-sup-distrib le-supI1 le-supI2)

setup ‹Sign.parent-path›

4.2 Compactness and algebraicity

Fundamental lattice concepts drawn from Davey and Priestley (2002).

context complete-lattice

begin

definition compact-points :: 'a set where — Davey and Priestley (2002, Definition 7.15(ii))

compact-points = $\{x. \forall S. x \leq \bigsqcup S \longrightarrow (\exists T \subseteq S. \text{finite } T \wedge x \leq \bigsqcup T)\}$

lemmas compact-pointsI = subsetD[OF equalityD2[OF compact-points-def], simplified, rule-format]

lemmas compact-pointsD = subsetD[OF equalityD1[OF compact-points-def], simplified, rule-format]

lemma compact-point-bot:

shows $\perp \in \text{compact-points}$

by (rule compact-pointsI) auto

lemma compact-points-sup: — Davey and Priestley (2002, Lemma 7.16)

assumes $x \in \text{compact-points}$

assumes $y \in \text{compact-points}$

shows $x \sqcup y \in \text{compact-points}$

proof(rule compact-pointsI)

fix S assume $x \sqcup y \leq \bigsqcup S$

with compact-pointsD[OF assms(1), of S] compact-pointsD[OF assms(2), of S]

obtain X Y

where $X \subseteq S \wedge \text{finite } X \wedge x \leq \bigsqcup X$

and $Y \subseteq S \wedge \text{finite } Y \wedge y \leq \bigsqcup Y$

by auto

then show $\exists T \subseteq S. \text{finite } T \wedge x \sqcup y \leq \bigsqcup T$

by (simp add: exI[where x=X ∪ Y] Sup-union-distrib le-supI1 le-supI2)

qed

lemma *compact-points-Sup*: — Davey and Priestley (2002, Lemma 7.16)

assumes $X \subseteq \text{compact-points}$

assumes *finite X*

shows $\bigsqcup X \in \text{compact-points}$

using *assms(2,1)* **by** *induct (simp-all add: compact-point-bot compact-points-sup)*

lemma *compact-points-are-ccpo-compact*: — converse should hold

assumes $x \in \text{compact-points}$

shows $\text{ccpo.compact Sup } (\leq) x$

proof(*rule ccpo.compactI[OF complete-lattice-ccpo], rule ccpo.admissibleI, rule notI*)

fix X

assume $\text{Complete-Partial-Order.chain } (\leq) X$ **and** $x \leq \bigsqcup X$ **and** $*$: $X \neq \{\} \forall y \in X. \neg x \leq y$

from $\text{compact-pointsD}[OF \text{ assms } \langle x \leq \bigsqcup X \rangle]$

obtain T **where** $T \subseteq X$ **and** *finite T* **and** $x \leq \bigsqcup T$ **by** *blast*

with $*$ $\text{Complete-Partial-Order.chain-subset}[OF \langle \text{Complete-Partial-Order.chain } (\leq) X \rangle \langle T \subseteq X \rangle]$

show *False*

by (*auto simp: sup.absorb1 sup.absorb2 bot-unique dest: chainD dest!: finite-Sup-in*)

qed

definition *directed* :: ' a set \Rightarrow bool' **where** — Davey and Priestley (2002, Definition 7.7)

directed X $\longleftrightarrow X \neq \{\} \wedge (\forall x \in X. \forall y \in X. \exists z \in X. x \leq z \wedge y \leq z)$

lemmas *directedI = iffD2[OF directed-def, simplified conj-explode, rule-format]*

lemmas *directedD = iffD1[OF directed-def]*

lemma *directed-empty*:

assumes *directed X*

shows $X \neq \{\}$

using *assms* **by** (*simp add: directed-def*)

lemma *chain-directed*:

assumes $\text{Complete-Partial-Order.chain } (\leq) Y$

assumes $Y \neq \{\}$

shows *directed Y*

using *assms* **by** (*metis chainD directedI*)

lemma *directed-alt-def*:

shows $\text{directed } X \longleftrightarrow (\forall Y \subseteq X. \text{finite } Y \longrightarrow (\exists x \in X. \forall y \in Y. y \leq x))$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

have $\exists x \in X. \forall y \in Y. y \leq x$ **if** *finite Y* **and** *directed X* **and** $Y \subseteq X$ **and** $Y \neq \{\}$ **for** Y

using *that* **by** *induct (force dest: directedD)+*

then show *?lhs* \implies *?rhs* **by** (*auto simp: directed-def*)

next

assume *?rhs* **show** *?lhs*

proof(*rule directedI*)

from $\langle ?rhs \rangle$ **show** $X \neq \{\}$ **by** *blast*

fix $x y$ **assume** $x \in X$ **and** $y \in X$ **with** $\langle ?rhs \rangle$ **show** $\exists z \in X. x \leq z \wedge y \leq z$

by (*clarsimp dest!: spec[where x={x, y}]*)

qed

qed

lemma *compact-points-alt-def*: — Davey and Priestley (2002, Definition 7.15(i) (finite points))

shows $\text{compact-points} = \{x :: 'a. \forall D. \text{directed } D \wedge x \leq \bigsqcup D \longrightarrow (\exists d \in D. x \leq d)\}$ (**is** *?lhs* = *?rhs*)

proof(*rule antisym*)

show *?lhs* \subseteq *?rhs*

by (*clarsimp simp: compact-points-def directed-alt-def*) (*metis Sup-least order.trans*)

next

have $*$: $\bigsqcup S = \bigsqcup \{\bigsqcup T \mid T. T \neq \{\} \wedge T \subseteq S \wedge \text{finite } T\}$ **for** $S :: 'a$ set — Davey and Priestley (2002, Exercise 7.5)

```

  by (fastforce intro: order.antisym Sup-subset-mono Sup-least exI[where x={x} for x])
have **: directed {⊔ T | T. T ≠ {} ∧ T ⊆ S ∧ finite T} (is directed ?D) if S ≠ {} for S :: 'a set
proof(rule directedI)
  from ⟨S ≠ {}⟩ show ?D ≠ {} by blast
  fix x y assume x ∈ ?D y ∈ ?D
  then obtain X Y
    where x = ⊔ X ∧ X ≠ {} ∧ X ⊆ S ∧ finite X
      and y = ⊔ Y ∧ Y ≠ {} ∧ Y ⊆ S ∧ finite Y
    by blast
  then show ∃ z ∈ ?D. x ≤ z ∧ y ≤ z
    by - (rule beXI[where x=⊔(X ∪ Y)]; auto simp: Sup-subset-mono)
qed
have ∃ T ⊆ X. finite T ∧ x ≤ ⊔ T if ∀ D. directed D ∧ x ≤ ⊔ D → (∃ d ∈ D. x ≤ d) and x ≤ ⊔ X for x X
  using that *[of X] **[of X] by force
then show ?rhs ⊆ ?lhs
  by (fastforce intro: compact-pointsI)
qed

lemmas compact-points-directedD
  = subsetD[OF equalityDI[OF compact-points-alt-def], simplified, rule-format, simplified conj-explode, rotated
-1]

end

class algebraic-lattice = complete-lattice + — Davey and Priestley (2002, Definition 7.18)
  assumes algebraic: (x :: 'a) = ⊔ ({Y. Y ≤ x} ∩ compact-points)
begin

lemma le-compact:
  shows x ≤ y ↔ (∀ z ∈ compact-points. z ≤ x → z ≤ y)
by (subst algebraic) (auto simp: Sup-le-iff)

end

lemma (in ccpo) compact-alt-def:
  shows ccpo.compact Sup (≤) x ↔ (∀ Y. Y ≠ {} ∧ Complete-Partial-Order.chain (≤) Y ∧ x ≤ Sup Y →
(∃ y ∈ Y. x ≤ y))
by (auto elim!: ccpo.compact.cases dest: ccpo.admissibleD intro!: compactI ccpo.admissibleI)

lemma compact-points-eq-finite-sets: — Davey and Priestley (2002, Examples 7.17)
  shows compact-points = Collect finite (is ?lhs = ?rhs)
proof(rule antisym)
  have *: X ⊆ ⋃ {{x} | x. x ∈ X} for X :: 'a set by blast
  show ?lhs ⊆ ?rhs by (force dest: compact-pointsD[OF - *] elim: finite-subset)
next
  show ?rhs ⊆ ?lhs by (metis CollectD compact-pointsI finite-subset-Union subsetI)
qed

instance set :: (type) algebraic-lattice
by standard (fastforce simp: compact-points-eq-finite-sets)

context semilattice-sup
begin

definition sup-irreducible-on :: 'a set ⇒ 'a ⇒ bool where — Davey and Priestley (2002, Definition 2.42)
  sup-irreducible-on A x ↔ (∀ y ∈ A. ∀ z ∈ A. x = y ⊔ z → x = y ∨ x = z)

abbreviation sup-irreducible :: 'a ⇒ bool where

```

sup-irreducible \equiv *sup-irreducible-on UNIV*

lemma *sup-irreducible-onI*:

assumes $\bigwedge y z. \llbracket y \in A; z \in A; x = y \sqcup z \rrbracket \implies x = y \vee x = z$

shows *sup-irreducible-on A x*

using *assms* **by** (*simp add: sup-irreducible-on-def*)

lemma *sup-irreducible-onD*:

assumes *sup-irreducible-on A x*

assumes $x = y \sqcup z$

assumes $y \in A$

assumes $z \in A$

shows $x = y \vee x = z$

using *assms* **by** (*auto simp: sup-irreducible-on-def*)

lemma *sup-irreducible-on-less*: — [Davey and Priestley \(2002, Definition 2.42 \(alt\)\)](#)

shows *sup-irreducible-on A x* $\longleftrightarrow (\forall y \in A. \forall z \in A. y < x \wedge z < x \longrightarrow y \sqcup z < x)$

by (*simp add: sup-irreducible-on-def ac-simps sup.strict-order-iff*)

(*metis local.sup commute local.sup.left-idem*)

end

lemma *sup-irreducible-bot*:

assumes $\perp \in A$

shows *sup-irreducible-on A* ($\perp :: \text{bounded-semilattice-sup-bot}$)

using *assms* **by** (*auto intro: sup-irreducible-onI*)

lemma *sup-irreducible-le-conv*:

fixes $x :: \text{distrib-lattice}$

assumes *sup-irreducible x*

shows $x \leq y \sqcup z \longleftrightarrow x \leq y \vee x \leq z$

by (*auto simp: inf.absorb-iff2 inf-sup-distrib1 ac-simps*

dest: sup-irreducible-onD[OF assms sym, simplified])

lemma *set-sup-irreducible*:

shows *sup-irreducible X* $\longleftrightarrow (X = \{\} \vee (\exists y. X = \{y\}))$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

by (*auto simp: sup-irreducible-on-def*) (*metis insert-is-Un mk-disjoint-insert*)

show *?rhs* \implies *?lhs* **by** (*auto intro: sup-irreducible-onI*)

qed

definition *Sup-irreducible-on* :: $'a :: \text{complete-lattice set} \Rightarrow 'a \Rightarrow \text{bool}$ **where** — [Davey and Priestley \(2002, Definition 10.26\)](#)

Sup-irreducible-on A x $\longleftrightarrow (\forall S \subseteq A. x = \bigsqcup S \longrightarrow x \in S)$

abbreviation *Sup-irreducible* :: $'a :: \text{complete-lattice} \Rightarrow \text{bool}$ **where**

Sup-irreducible \equiv *Sup-irreducible-on UNIV*

definition *Sup-prime-on* :: $'a :: \text{complete-lattice set} \Rightarrow 'a \Rightarrow \text{bool}$ **where** — [Davey and Priestley \(2002, Definition 10.26\)](#)

Sup-prime-on A x $\longleftrightarrow (\forall S \subseteq A. x \leq \bigsqcup S \longrightarrow (\exists s \in S. x \leq s))$

abbreviation *Sup-prime* :: $'a :: \text{complete-lattice} \Rightarrow \text{bool}$ **where**

Sup-prime \equiv *Sup-prime-on UNIV*

lemma *Sup-irreducible-onI*:

assumes $\bigwedge S. \llbracket S \subseteq A; x = \bigsqcup S \rrbracket \implies x \in S$

shows *Sup-irreducible-on A x*
using *assms* **by** (*simp add: Sup-irreducible-on-def*)

lemma *Sup-irreducible-onD*:
assumes $x = \bigsqcup S$
assumes $S \subseteq A$
assumes *Sup-irreducible-on A x*
shows $x \in S$
using *assms* **by** (*simp add: Sup-irreducible-on-def*)

lemma *Sup-prime-onI*:
assumes $\bigwedge S. \llbracket S \subseteq A; x \leq \bigsqcup S \rrbracket \implies \exists s \in S. x \leq s$
shows *Sup-prime-on A x*
using *assms* **by** (*simp add: Sup-prime-on-def*)

lemma *Sup-prime-onE*:
assumes *Sup-prime-on A x*
assumes $x \leq \bigsqcup S$
assumes $S \subseteq A$
obtains s **where** $s \in S$ **and** $x \leq s$
using *assms* *Sup-prime-on-def* **by** *blast*

lemma *Sup-prime-on-conv*:
assumes *Sup-prime-on A x*
assumes $S \subseteq A$
shows $x \leq \bigsqcup S \longleftrightarrow (\exists s \in S. x \leq s)$
using *assms* **by** (*auto simp: Sup-prime-on-def intro: Sup-upper2*)

lemma *Sup-prime-not-bot*:
assumes *Sup-prime-on A x*
shows $x \neq \perp$
using *assms* **by** (*force simp: Sup-prime-on-def*)

lemma *Sup-prime-on-imp-Sup-irreducible-on*: — the converse holds in Heyting algebras
assumes *Sup-prime-on A x*
shows *Sup-irreducible-on A x*
using *Sup-upper*
by (*fastforce intro!: Sup-irreducible-onI intro: antisym elim!: Sup-prime-onE[OF assms, rotated]*)

lemma *Sup-irreducible-on-imp-sup-irreducible-on*:
assumes *Sup-irreducible-on A x*
assumes $x \in A$
shows *sup-irreducible-on A x*
using *assms* **by** (*fastforce simp: Sup-irreducible-on-def sup-irreducible-on-def*
dest: spec[where $x = \{x, y\}$ for $x y$])

lemma *Sup-prime-is-compact*:
assumes *Sup-prime x*
shows $x \in \text{compact-points}$
using *assms* **by** (*simp add: compact-points-alt-def Sup-prime-on-def*)

5 Closure operators

Our semantic spaces are modelled as lattices arising from the fixed points of various closure operators. We attempt to reduce our proof obligations by defining a locale for Kuratowski's closure axioms, where we do not require strictness (i.e., it need not be the case that the closure maps \perp to \perp). Davey and Priestley (2002, §2.33) term these *topped intersection structures*; see also Pfaltz and Šlapal (2013) for additional useful results.

locale *closure* =
ordering (\leq) ($<$) — We use a partial order as a preorder does not ensure that the closure is idempotent
for *less-eq* :: 'a \Rightarrow 'a \Rightarrow bool (**infix** $\langle \leq \rangle$ 50)
and *less* :: 'a \Rightarrow 'a \Rightarrow bool (**infix** $\langle < \rangle$ 50)
+ **fixes** *cl* :: 'a \Rightarrow 'a
assumes *cl*: $x \leq cl\ y \longleftrightarrow cl\ x \leq cl\ y$ — All-in-one non-strict Kuratowski axiom
begin

definition *closed* :: 'a set **where** — These pre fixed points form a complete lattice ala Tarski/Knaster
closed = $\{x. cl\ x \leq x\}$

lemma *closed-clI*:
assumes $cl\ x \leq x$
shows $x \in closed$
unfolding *closed-def* **using** *assms* **by** *simp*

lemma *expansive*:
shows $x \leq cl\ x$
by (*simp add: cl refl*)

lemma *idempotent*[*simp*]:
shows $cl\ (cl\ x) = cl\ x$
and $cl \circ cl = cl$
using *cl antisym* **by** (*auto iff: expansive*)

lemma *monotone-cl*:
shows *monotone* (\leq) (\leq) *cl*
by (*rule monotoneI*) (*meson cl expansive trans*)

lemmas *strengthen-cl*[*strg*] = *st-monotone*[*OF monotone-cl*]
lemmas *mono-cl*[*trans*] = *monotoneD*[*OF monotone-cl*]

lemma *least*:
assumes $x \leq y$
assumes $y \in closed$
shows $cl\ x \leq y$
using *assms cl closed-def trans* **by** (*blast intro: expansive*)

lemma *least-conv*:
assumes $y \in closed$
shows $cl\ x \leq y \longleftrightarrow x \leq y$
using *assms expansive least trans* **by** *blast*

lemma *closed*[*iff*]:
shows $cl\ x \in closed$
unfolding *closed-def* **by** (*simp add: refl*)

lemma *le-closedE*:
assumes $x \leq cl\ y$
assumes $y \in closed$
shows $x \leq y$
using *assms closed-def trans* **by** *blast*

lemma *closed-conv*: — Typically used to manifest the closure using *subst*
assumes $X \in closed$
shows $X = cl\ X$
using *assms unfolding closed-def* **by** (*blast intro: antisym expansive*)

end

lemma (in *ordering*) *closure-axioms-alt-def*: — Equivalence with the Kuratowski closure axioms
 shows *closure-axioms* $(\leq) cl \longleftrightarrow (\forall x. x \leq cl x) \wedge \text{monotone } (\leq) (\leq) cl \wedge (\forall x. cl (cl x) = cl x)$
unfolding *closure-axioms-def monotone-def* **by** (*metis antisym trans refl*)

lemma (in *ordering*) *closureI*:
 assumes $\bigwedge x. x \leq cl x$
 assumes *monotone* $(\leq) (\leq) cl$
 assumes $\bigwedge x. cl (cl x) = cl x$
 shows *closure* $(\leq) (<) cl$
by (*blast intro: assms closure.intro[OF ordering-axioms, unfolded closure-axioms-alt-def]*)

lemma *closure-inf-closure*:
 fixes $cl_1 :: 'a::\text{semilattice-inf} \Rightarrow 'a$
 assumes *closure-axioms* $(\leq) cl_1$
 assumes *closure-axioms* $(\leq) cl_2$
 shows *closure-axioms* $(\leq) (\lambda X. cl_1 X \sqcap cl_2 X)$
using *assms unfolding closure-axioms-def* **by** (*meson order.trans inf-mono le-inf-iff order-refl*)

5.1 Complete lattices and algebraic closures

locale *closure-complete-lattice* =
 complete-lattice $\sqcap \sqcup (\sqcap) (\leq) (<) (\sqcup) \perp \top$
+ *closure* $(\leq) (<) cl$
 for *less-ega* $:: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\langle \leq \rangle$ 50)
 and *lessa* (**infix** $\langle < \rangle$ 50)
 and *infa* (**infixl** $\langle \sqcap \rangle$ 70)
 and *supa* (**infixl** $\langle \sqcup \rangle$ 65)
 and *bota* $\langle \perp \rangle$
 and *topa* $\langle \top \rangle$
 and *Inf* $\langle \sqcap \rangle$
 and *Sup* $\langle \sqcup \rangle$
 and $cl :: 'a \Rightarrow 'a$

begin

lemma *cl-bot-least*:
 shows $cl \perp \leq cl X$
using *cl* **by** *auto*

lemma *cl-Inf-closed*:
 shows $cl x = \sqcap \{y \in \text{closed}. x \leq y\}$
by (*blast intro: sym[OF Inf-eqI] least expansive*)

lemma *cl-top*:
 shows $cl \top = \top$
by (*simp add: top-le expansive*)

lemma *closed-top[iff]*:
 shows $\top \in \text{closed}$
unfolding *closed-def* **by** *simp*

lemma *Sup-cl-le*:
 shows $\sqcup (cl ' X) \leq cl (\sqcup X)$
by (*meson cl expansive SUP-least Sup-le-iff*)

lemma *sup-cl-le*:
 shows $cl x \sqcup cl y \leq cl (x \sqcup y)$

using *Sup-cl-le*[**where** $X=\{x, y\}$] **by** *simp*

lemma *cl-Inf-le*:

shows $cl (\bigcap X) \leq \bigcap (cl \text{ ` } X)$

by (*meson cl expansive INF-greatest Inf-lower2*)

lemma *cl-inf-le*:

shows $cl (x \sqcap y) \leq cl x \sqcap cl y$

using *cl-Inf-le*[**where** $X=\{x, y\}$] **by** *simp*

lemma *closed-Inf*:

assumes $X \subseteq closed$

shows $\bigcap X \in closed$

unfolding *closed-def* **using** *assms* **by** (*simp add: least Inf-greatest Inf-lower subset-eq*)

lemmas *closed-Inf'*[*intro*] = *closed-Inf*[*OF subsetI*]

lemma *closed-inf*[*intro*]:

assumes $P \in closed$

assumes $Q \in closed$

shows $P \sqcap Q \in closed$

using *assms closed-Inf*[**where** $X=\{P, Q\}$] **by** *simp*

lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF monotone-cl, simplified*]

definition *dense* :: 'a set **where**

$dense = \{x. cl x = \top\}$

lemma *dense-top*:

shows $\top \in dense$

by (*simp add: dense-def cl-top*)

lemma *dense-Sup*:

assumes $X \subseteq dense$

assumes $X \neq \{\}$

shows $\bigsqcup X \in dense$

using *assms* **by** (*fastforce simp: dense-def order.eq-iff intro: order.trans*[*OF - Sup-cl-le*] *elim: SUP-upper2*)

lemma *dense-sup*:

assumes $P \in dense$

assumes $Q \in dense$

shows $P \sqcup Q \in dense$

using *assms dense-def top-le sup-cl-le* **by** *auto*

lemma *dense-le*:

assumes $P \in dense$

assumes $P \leq Q$

shows $Q \in dense$

using *assms dense-def top-le mono-cl* **by** *auto*

lemma *dense-inf-closed*:

shows $dense \cap closed = \{\top\}$

using *dense-def closed-conv closed-top* **by** *fastforce*

end

locale *closure-complete-lattice-class* =

closure-complete-lattice (\leq) ($<$) (\sqcap) (\sqcup) \perp :: - :: *complete-lattice* \top *Inf* *Sup*

Traditionally closures for logical purposes are taken to be “algebraic”, aka “consequence operators” (Davey and Priestley 2002, Definition 7.12), where *compactness* does the work of the finite/singleton sets.

locale *closure-complete-lattice-algebraic* = — Davey and Priestley (2002, Definition 7.12)
closure-complete-lattice

+ **assumes** *algebraic-le*: $cl\ x \leq \bigsqcup (cl\ ' (\{y. y \leq x\} \cap compact\ points))$ — The converse is given by monotonicity
begin

lemma *algebraic*:

shows $cl\ x = \bigsqcup (cl\ ' (\{y. y \leq x\} \cap compact\ points))$

by (*clarsimp simp: order.eq-iff Sup-le-iff algebraic-le expansive simp flip: cl elim!: order.trans*)

lemma *cont-cl*: — Equivalent to *algebraic-le* Davey and Priestley (2002, Theorem 7.14)

shows $cont\ \sqcup (\leq) \sqcup (\leq) cl$

proof(*rule contI*)

fix $X :: 'a\ set$

assume $X: Complete\ Partial\ Order.chain (\leq) X X \neq \{\}$

show $cl (\sqcup X) = \bigsqcup (cl\ ' X)$ (**is** *?lhs = ?rhs*)

proof(*rule order.antisym[OF - Sup-cl-le]*)

have *?lhs* = $\bigsqcup (cl\ ' (\{y. y \leq \sqcup X\} \cap compact\ points))$ **by** (*subst algebraic simp*)

also from X **have** $\dots \leq \bigsqcup (cl\ ' (\{Y \mid Yx. Y \leq x \wedge x \in X\} \cap compact\ points))$

by (*auto dest: chain-directed compact-points-directedD intro: SUP-upper simp: Sup-le-iff*)

also have $\dots \leq ?rhs$

by (*clarsimp simp: Sup-le-iff SUP-upper2 monotoneD[OF monotone-cl]*)

finally show *?lhs* \leq *?rhs* .

qed

qed

lemma *mcont-cl*:

shows $mcont\ \sqcup (\leq) \sqcup (\leq) cl$

by (*simp add: mcontI[OF - cont-cl]*)

lemma *mcont2mcont-cl[cont-intro]*:

assumes $mcont\ luba\ orda\ \sqcup (\leq) P$

shows $mcont\ luba\ orda\ \sqcup (\leq) (\lambda x. cl (P\ x))$

using *assms ccpo.mcont2mcont[OF complete-lattice-ccpo] mcont-cl* **by** *blast*

end

locale *closure-complete-lattice-algebraic-class* =

closure-complete-lattice-algebraic $(\leq) (<) (\cap) (\sqcup) \perp :: - :: complete\ lattice \top Inf Sup$

Our closures often satisfy the stronger condition of *distributivity* (see Scott (1980, §2)).

locale *closure-complete-lattice-distributive* =

closure-complete-lattice

+ **assumes** *cl-Sup-le*: $cl (\sqcup X) \leq \bigsqcup (cl\ ' X) \sqcup cl \perp$

begin

lemma *cl-Sup*:

shows $cl (\sqcup X) = \bigsqcup (cl\ ' X) \sqcup cl \perp$

by (*simp add: order.eq-iff Sup-cl-le cl-Sup-le cl-bot-least*)

lemma *cl-Sup-not-empty*:

assumes $X \neq \{\}$

shows $cl (\sqcup X) = \bigsqcup (cl\ ' X)$

by (*metis assms cl-Sup cl-bot-least SUP-eq-const SUP-upper2 sup.orderE*)

lemma *cl-sup*:

shows $cl (X \sqcup Y) = cl\ X \sqcup cl\ Y$

using *cl-Sup*[**where** $X=\{X, Y\}$] **by** (*simp add: sup-absorb1 cl-bot-least le-supI2*)

lemma *closed-sup*[*intro*]:

assumes $P \in \text{closed}$

assumes $Q \in \text{closed}$

shows $P \sqcup Q \in \text{closed}$

by (*metis assms cl-sup closed-conv closed*)

lemma *closed-Sup*: — Alexandrov: https://en.wikipedia.org/wiki/Alexandrov_topology

assumes $X \subseteq \text{closed}$

shows $\sqcup X \sqcup \text{cl } \perp \in \text{closed}$

using *assms* **by** (*fastforce simp: cl-Sup cl-sup Sup-le-iff simp flip: closed-conv intro: closed-clI Sup-upper le-supI1*)

lemmas *closed-Sup'*[*intro*] = *closed-Sup*[*OF subsetI*]

lemma *cont-cl*:

shows $\text{cont } \sqcup (\leq) \sqcup (\leq) \text{ cl}$

by (*simp add: cl-Sup-not-empty contI*)

lemma *mcont-cl*:

shows $\text{mcont } \sqcup (\leq) \sqcup (\leq) \text{ cl}$

by (*simp add: mcontI[OF - cont-cl]*)

lemma *mcont2mcont-cl*[*cont-intro*]:

assumes $\text{mcont } \text{luba } \text{orda } \sqcup (\leq) F$

shows $\text{mcont } \text{luba } \text{orda } \sqcup (\leq) (\lambda x. \text{cl } (F x))$

using *assms ccpo.mcont2mcont[OF complete-lattice-ccpo]* *mcont-cl* **by** *blast*

lemma *closure-sup-irreducible-on*: — converse requires the closure to be T0

assumes *sup-irreducible-on closed* ($\text{cl } x$)

shows *sup-irreducible-on closed* x

using *assms sup-irreducible-on-def closed-conv closed-sup* **by** *auto*

end

locale *closure-complete-lattice-distributive-class* =

closure-complete-lattice-distributive (\leq) ($<$) (\sqcap) (\sqcup) \perp :: - :: *complete-lattice* \top *Inf Sup*

locale *closure-complete-distrib-lattice-distributive-class* =

closure-complete-lattice-distributive (\leq) ($<$) (\sqcap) (\sqcup) \perp :: - :: *complete-distrib-lattice* \top *Inf Sup*

begin

The lattice arising from the closed elements for a distributive closure is completely distributive, i.e., *Inf* and *Sup* distribute. See Davey and Priestley (2002, Section 10.23).

lemma *closed-complete-distrib-lattice-axiomI'*:

assumes $\forall A \in A. \forall x \in A. x \in \text{closed}$

shows $(\sqcap X \in A. \sqcup X \sqcup \text{cl } \perp)$

$\leq \sqcup (\text{Inf } \{f' \text{ ` } A \mid f. (\forall X \subseteq \text{closed}. f X \in \text{closed}) \wedge (\forall Y \in A. f Y \in Y)\}) \sqcup \text{cl } \perp$

proof –

from *assms*

have $\exists f'. f' \text{ ` } A = f' \text{ ` } A \wedge (\forall X \subseteq \text{closed}. f' X \in \text{closed}) \wedge (\forall Y \in A. f' Y \in Y)$

if $\forall Y \in A. f Y \in Y$ **for** f

using *that* **by** (*fastforce intro!: exI[where x= $\lambda x. \text{if } f x \in \text{closed} \text{ then } f x \text{ else } \text{cl } \perp$]*)

then show *?thesis*

by (*clarsimp simp: Inf-Sup Sup-le-iff simp flip: INF-sup intro!: le-supI1 Sup-upper*)

qed

lemma *closed-complete-distrib-lattice-axiomI*[*intro*]:

assumes $\forall A \in A. \forall x \in A. x \in \text{closed}$
shows $(\prod X \in A. \sqcup X \sqcup \text{cl } \perp)$
 $\leq \sqcup (\text{Inf } \{ B. (\exists f. (\forall x. (\forall x \in x. x \in \text{closed}) \longrightarrow f x \in \text{closed}))$
 $\wedge B = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y) \wedge (\forall x \in B. x \in \text{closed}) \}$
 $\sqcup \text{cl } \perp$
by (*rule order.trans[OF closed-complete-distrib-lattice-axiomI[OF assms]]*)
(use assms in <clarsimp simp: Sup-le-iff ac-simps>; fast intro!: exI imageI Sup-upper le-supI2)

lemma *closed-strict-complete-distrib-lattice-axiomI[intro]:*

assumes $\text{cl } \perp = \perp$
assumes $\forall A \in A. \forall x \in A. x \in \text{closed}$
shows $(\prod X \in A. \sqcup X)$
 $\leq \sqcup (\text{Inf } \{ x. (\exists f. (\forall x. (\forall x \in x. x \in \text{closed}) \longrightarrow f x \in \text{closed}))$
 $\wedge x = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y) \wedge (\forall x \in x. x \in \text{closed}) \}$
using *closed-complete-distrib-lattice-axiomI[simplified assms(1), simplified, OF assms(2)]* .

end

5.2 Closures over powersets

locale *closure-powerset* =

closure-complete-lattice-class cl **for** $\text{cl} :: 'a \text{ set} \Rightarrow 'a \text{ set}$
begin

lemmas *expansive'* = *subsetD[OF expansive]*

lemma *closedI[intro]:*

assumes $\bigwedge x. x \in \text{cl } X \Longrightarrow x \in X$
shows $X \in \text{closed}$
unfolding *closed-def* **using** *assms* **by** *blast*

lemma *closedE:*

assumes $x \in \text{cl } Y$
assumes $Y \in \text{closed}$
shows $x \in Y$
using *assms* *closed-def* **by** *auto*

lemma *cl-mono:*

assumes $x \in \text{cl } X$
assumes $X \subseteq Y$
shows $x \in \text{cl } Y$
using *assms* **by** (*rule subsetD[OF monotoneD[OF monotone-cl], rotated]*)

lemma *cl-bind-le:*

shows $X \gg= \text{cl} \circ f \leq \text{cl } (X \gg= f)$
by (*metis bind-UNION bind-image Sup-cl-le*)

lemma *pointwise-distributive-iff:*

shows $(\forall X. \text{cl } (\bigcup X) = \bigcup (\text{cl } \text{' } X) \cup \text{cl } \{\}) \longleftrightarrow (\forall X. \text{cl } X = (\bigcup_{x \in X} \text{cl } \{x\}) \cup \text{cl } \{\})$ (**is** *?lhs* \longleftrightarrow *?rhs*)
proof(*rule iffI*)

show *?lhs* \Longrightarrow *?rhs* **by** (*metis UN-singleton image-image*)

next

assume *?rhs*
note *distributive* = $\langle ?rhs \rangle$ [*rule-format*]
have *cl-union*: $\text{cl } (X \cup Y) = \text{cl } X \cup \text{cl } Y$ (**is** *?lhs1* = *?rhs1*) **for** $X Y$
proof(*rule antisym[OF - sup-cl-le]*)
from *cl expansive'* **show** $?lhs1 \subseteq ?rhs1$ **by** (*subst distributive*) *auto*
qed

have *cl-insert*: $cl (insert\ x\ X) = cl\ \{x\} \cup cl\ X$ **for** $x\ X$
by (*metis cl-union insert-is-Un*)
show ?*lhs*
proof(*intro allI antisym*)
show $cl (\bigcup X) \subseteq \bigcup (cl\ ' X) \cup cl\ \{\}$ **for** X
by (*subst distributive*) (*clarsimp; metis UnCI cl-insert insert-Diff*)
qed (*simp add: cl-bot-least Sup-cl-le*)
qed

lemma *Sup-prime-on-singleton*:
shows *Sup-prime-on closed* ($cl\ \{x\}$)
unfolding *Sup-prime-on-def* **by** (*meson UnionE expansive' insert-subsetI least bot-least singletonI subsetD*)

end

locale *closure-powerset-algebraic* =
closure-powerset
+ *closure-complete-lattice-algebraic-class*

locale *closure-powerset-distributive* =
closure-powerset
+ *closure-complete-distrib-lattice-distributive-class*

begin

lemmas *distributive = pointwise-distributive-iff*[*THEN iffD1, rule-format, OF cl-Sup*]

lemma *algebraic-axiom*: — Davey and Priestley (2002, Theorem 7.14)
shows $cl\ x \subseteq \bigcup (cl\ ' (\{y.\ y \subseteq x\} \cap local.compact-points))$
unfolding *compact-points-def complete-lattice-class.compact-points-def*[*symmetric*]
by (*metis Int-iff algebraic cl-Sup-not-empty complete-lattice-class.compact-pointsI emptyE*
finite.intros(1) mem-Collect-eq order-bot-class.bot-least order.refl)

lemma *cl-insert*:
shows $cl (insert\ x\ X) = cl\ \{x\} \cup cl\ X$
by (*metis cl-sup insert-is-Un*)

lemma *cl-UNION*:
shows $cl (\bigcup_{i \in I} f\ i) = (\bigcup_{i \in I} cl (f\ i)) \cup cl\ \{\}$
by (*auto simp add: cl-Sup SUP-upper*)

lemma *closed-UNION*:
assumes $\bigwedge i. i \in I \implies f\ i \in closed$
shows $(\bigcup_{i \in I} f\ i) \cup cl\ \{\} \in closed$
using *assms closed-def* **by** (*auto simp: cl-Sup cl-sup*)

lemma *sort-of-inverse*: — Pfaltz and Šlapal (2013, Proposition 2.5)
assumes $y \in cl\ X - cl\ \{\}$
shows $\exists x \in X. y \in cl\ \{x\}$
using *assms* **by** (*subst (asm) distributive*) *blast*

lemma *cl-diff-le*:
shows $cl\ x - cl\ y \subseteq cl (x - y)$
by (*metis Diff-subset-conv Un-Diff-cancel Un-upper2 cl-sup*)

lemma *cl-bind*:
shows $cl (X \gg= f) = (X \gg= cl \circ f) \cup cl\ \{\}$
by (*simp add: bind-UNION cl-Sup*)

lemma *sup-irreducible-on-singleton*:

shows *sup-irreducible-on closed* ($cl \{a\}$)

by (*rule sup-irreducible-onI*)

(*metis Un-iff sup-bot.right-neutral expansive insert-subset least antisym local.sup commute sup-ge2*)

end

5.3 Matroids and antimatroids

The *exchange* axiom characterises *matroids* (see, for instance, §6.1), while the *anti-exchange* axiom characterises *antimatroids* (see e.g. §7.1).

References:

- Pfaltz and Šlapal (2013) provide an overview of these concepts
- <https://en.wikipedia.org/wiki/Antimatroid>

definition *anti-exchange* :: ($'a \text{ set} \Rightarrow 'a \text{ set}$) \Rightarrow *bool* **where**

anti-exchange cl $\longleftrightarrow (\forall X x y. x \neq y \wedge y \in cl (insert x X) - cl X \longrightarrow x \notin cl (insert y X) - cl X)$

definition *exchange* :: ($'a \text{ set} \Rightarrow 'a \text{ set}$) \Rightarrow *bool* **where**

exchange cl $\longleftrightarrow (\forall X x y. y \in cl (insert x X) - cl X \longrightarrow x \in cl (insert y X) - cl X)$

lemmas *anti-exchangeI* = *iffD2[OF anti-exchange-def, rule-format]*

lemmas *exchangeI* = *iffD2[OF exchange-def, rule-format]*

lemma *anti-exchangeD*:

assumes $y \in cl (insert x X) - cl X$

assumes $x \neq y$

assumes *anti-exchange cl*

shows $x \notin cl (insert y X) - cl X$

using *assms unfolding anti-exchange-def by blast*

lemma *exchange-Image*: — Some matroids arise from equivalence relations. Note $sym r \wedge trans r \longrightarrow Refl r$

shows *exchange (Image r)* $\longleftrightarrow sym r \wedge trans r$

by (*auto 6 0 simp: exchange-def sym-def intro!: transI elim: transE*)

locale *closure-powerset-distributive-exchange* =

closure-powerset-distributive

+ **assumes** *exchange: exchange cl*

begin

lemma *exchange-exchange*:

assumes $x \in cl \{y\}$

assumes $x \notin cl \{x\}$

shows $y \in cl \{x\}$

using *assms exchange unfolding exchange-def by blast*

lemma *exchange-closed-inter*:

assumes $Q \in closed$

shows $cl P \cap Q = cl (P \cap Q)$ (**is** *?lhs = ?rhs*)

and $Q \cap cl P = cl (P \cap Q)$ (**is** *?thesis1*)

proof —

show *?lhs = ?rhs*

proof(*rule antisym*)

have $((\bigcup x \in P. cl \{x\}) \cup cl \{x\}) \cap Q \subseteq ((\bigcup x \in P \cap cl Q. cl \{x\}) \cup cl \{x\})$

by *clarsimp (metis IntI UnCI cl-insert exchange-exchange mk-disjoint-insert)*

then show $?lhs \subseteq ?rhs$
by (*simp flip: distributive closed-conv[OF assms]*)
from *assms* **show** $?rhs \subseteq ?lhs$
using *cl-inf-le closed-conv* **by** *blast*
qed
then show *?thesis1* **by** *blast*
qed

lemma *exchange-both-closed-inter*:
assumes $P \in \text{closed}$
assumes $Q \in \text{closed}$
shows $cl (P \cap Q) = P \cap Q$
using *assms closed-conv closed-inf* **by** *force*

end

lemma *anti-exchange-Image*: — when r is asymmetric on distinct points
shows *anti-exchange* (*Image* r) $\longleftrightarrow (\forall x y. x \neq y \wedge (x, y) \in r \longrightarrow (y, x) \notin r)$
by (*auto simp: anti-exchange-def*)

locale *closure-powerset-distributive-anti-exchange* =
closure-powerset-distributive
+ **assumes** *anti-exchange: anti-exchange cl*

5.4 Composition

Conditions under which composing two closures yields a closure. See also Pfaltz and Šlapal (2013).

lemma *closure-comp*:
assumes *closure lesseqa lessa cl₁*
assumes *closure lesseqa lessa cl₂*
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure lesseqa lessa* ($\lambda X. cl_1 (cl_2 X)$)
using *assms* **by** (*clarsimp simp: closure-def closure-axioms-def*) *metis*

lemma *closure-complete-lattice-comp*:
assumes *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa cl₁*
assumes *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa cl₂*
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* ($\lambda X. cl_1 (cl_2 X)$)
using *assms(1)[unfolded closure-complete-lattice-def]*
closure-comp[OF closure-complete-lattice.axioms(2)[OF assms(1)]
closure-complete-lattice.axioms(2)[OF assms(2)]
assms(3)
by (*blast intro: closure-complete-lattice.intro*)

lemma *closure-powerset-comp*:
assumes *closure-powerset cl₁*
assumes *closure-powerset cl₂*
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure-powerset* ($\lambda X. cl_1 (cl_2 X)$)
using *assms* **by** (*simp add: closure-complete-lattice-class-def closure-complete-lattice-comp closure-powerset-def*)

lemma *closure-powerset-distributive-comp*:
assumes *closure-powerset-distributive cl₁*
assumes *closure-powerset-distributive cl₂*
assumes $\bigwedge X. cl_1 (cl_2 X) = cl_2 (cl_1 X)$
shows *closure-powerset-distributive* ($\lambda X. cl_1 (cl_2 X)$)
proof –

```

have cl1 (cl2 (⋃ X)) ⊆ (⋃ X∈X. cl1 (cl2 X)) ∪ cl1 (cl2 {}) for X
  apply (subst (1 2 3) closure-powerset-distributive.distributive[OF assms(1)])
  apply (subst (1 2 3) closure-powerset-distributive.distributive[OF assms(2)])
  apply fast
  done
moreover
from assms have closure-axioms (⊆) (λX. cl1 (cl2 X))
  by (metis closure-powerset-distributive-def closure-complete-lattice-class-def closure-def
    closure-powerset.axioms closure-powerset-comp closure-complete-lattice.axioms(2))
ultimately show ?thesis
  by (intro-locales; blast intro: closure-complete-lattice-distributive-axioms.intro)
qed

```

5.5 Path independence

Pfaltz and Šlapal (2013, Prop 1.1): “an expansive operator is a closure operator iff it is path independent.”

References:

- \$AFP/Stable_Matching/Choice_Functions.thy

context *semilattice-sup*

begin

definition *path-independent* :: ('a ⇒ 'a) ⇒ bool **where**
path-independent f ⟷ (∀ x y. f (x ⊔ y) = f (f x ⊔ f y))

lemma *cl-path-independent*:

shows closure (≤) (<) cl ⟷ *path-independent* cl ∧ (∀ x. x ≤ cl x) (**is** ?lhs ⟷ ?rhs)

proof(*rule iffI*)

show ?lhs ⟹ ?rhs

by (auto 5 0 *simp: closure-def closure-axioms-def path-independent-def order.eq-iff*
elim: le-supE)

show ?rhs ⟹ ?lhs

by *unfold-locales (metis path-independent-def le-sup-iff sup.absorb-iff2 sup.idem)*

qed

end

5.6 Some closures

interpretation *id-cl: closure-powerset-distributive id*

by *standard auto*

5.6.1 Reflexive, symmetric and transitive closures

The reflexive closure *reflcl* is very well behaved. Note the new bottom is *Id*. The reflexive transitive closure *rtrancl* and transitive closure *trancl* are clearly not distributive.

rtrancl is neither matroidal nor antimatroidal.

interpretation *reflcl-cl: closure-powerset-distributive-exchange reflcl*

by *standard (auto simp: exchange-def)*

interpretation *symcl-cl: closure-powerset-distributive-exchange λX. X ∪ X⁻¹*

by *standard (auto simp: exchange-def)*

interpretation *trancl-cl: closure-powerset trancl*

by *standard (metis r-into-trancl' subsetI trancl-id trancl-mono trans-trancl)*

interpretation *rtrancl-cl: closure-powerset rtrancl*

by *standard* (use *rtrancl-subset-rtrancl* in *blast*)

lemma *rtrancl-closed-Id*:

shows $Id \in \text{rtrancl-cl.closed}$

using *rtrancl-cl.idempotent rtrancl-empty* by *fastforce*

lemma *rtrancl-closed-reflcl-closed*:

shows $\text{rtrancl-cl.closed} \subseteq \text{reflcl-cl.closed}$

using *rtrancl-cl.closed-conv* by *fast*

5.6.2 Relation image

lemma *idempotent-Image*:

assumes *refl-on* Y r

assumes *trans* r

assumes $X \subseteq Y$

shows $r \text{ `` } r \text{ `` } X = r \text{ `` } X$

using *assms* by (*auto elim: transE intro: refl-onD refl-onD2*)

lemmas *distributive-Image = Image-eq-UN*

lemma *closure-powerset-distributive-ImageI*:

assumes $cl = \text{Image } r$

assumes *refl* r

assumes *trans* r

shows *closure-powerset-distributive* cl

proof –

from *assms* **have** *closure-axioms* (\subseteq) cl

unfolding *order.closure-axioms-alt-def*

by (*force simp: idempotent-Image Image-mono monotoneI dest: refl-onD*)

with $\langle cl = \text{Image } r \rangle$ **show** *?thesis*

by – (*intro-locales; auto simp: closure-complete-lattice-distributive-axioms-def*)

qed

lemma *closure-powerset-distributive-exchange-ImageI*:

assumes $cl = \text{Image } r$

assumes *equiv UNIV* r — symmetric, transitive and universal domain

shows *closure-powerset-distributive-exchange* cl

using *closure-powerset-distributive-ImageI[OF assms(1)] exchange-Image[of r] assms*

unfolding *closure-powerset-distributive-exchange-def closure-powerset-distributive-exchange-axioms-def*

by (*metis equivE*)

interpretation *Image-rtrancl: closure-powerset-distributive Image* (r^*)

by (*rule closure-powerset-distributive-ImageI*) *auto*

5.6.3 Kleene closure

We define Kleene closure in the traditional way with respect to some axioms that our various lattices satisfy. As trace models are not going to validate $x \cdot \perp = \perp$ (Kozen 1994, Axiom 13), we cannot reuse existing developments of Kleene Algebra (and Concurrent Kleene Algebra (Hoare, Möller, Struth, and Wehrman 2011)). In general it is not distributive.

locale *weak-kleene* =

fixes *unit* :: $'a::\text{complete-lattice}$ ($\langle \varepsilon \rangle$)

fixes *comp* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\langle \cdot \rangle$ 60)

assumes *comp-assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

assumes *weak-comp-unitL*: $\varepsilon \leq x \Longrightarrow \varepsilon \cdot x = x$

assumes *comp-unitR*: $x \cdot \varepsilon = x$

assumes *comp-supL*: $(x \sqcup y) \cdot z = (x \cdot z) \sqcup (y \cdot z)$

```

assumes comp-supR:  $x \cdot (y \sqcup z) = (x \cdot y) \sqcup (x \cdot z)$ 
assumes mcont-compL:  $mcont\ Sup (\leq) Sup (\leq) (\lambda x. x \cdot y)$ 
assumes mcont-compR:  $mcont\ Sup (\leq) Sup (\leq) (\lambda y. x \cdot y)$ 
assumes comp-botL:  $\perp \cdot x = \perp$ 
begin

lemma mcont2mcont-comp:
  assumes mcont Supa orda Sup ( $\leq$ ) f
  assumes mcont Supa orda Sup ( $\leq$ ) g
  shows mcont Supa orda Sup ( $\leq$ )  $(\lambda x. f x \cdot g x)$ 
by (simp add: ccpo.mcont2mcont'[OF complete-lattice-ccpo mcont-compL - assms(1)]
      ccpo.mcont2mcont'[OF complete-lattice-ccpo mcont-compR - assms(2)])

lemma mono2mono-comp:
  assumes monotone orda ( $\leq$ ) f
  assumes monotone orda ( $\leq$ ) g
  shows monotone orda ( $\leq$ )  $(\lambda x. f x \cdot g x)$ 
using mcont-mono[OF mcont-compL] mcont-mono[OF mcont-compR] assms
by (clarsimp simp: monotone-def) (meson order.trans)

context
  notes mcont2mcont-comp[cont-intro]
  notes mono2mono-comp[cont-intro, partial-function-mono]
  notes st-monotone[OF mcont-mono[OF mcont-compL], strg]
  notes st-monotone[OF mcont-mono[OF mcont-compR], strg]
begin

context
  notes [[function-internals]] — Exposes the induction rules we need
begin

partial-function (lfp) star :: 'a  $\Rightarrow$  'a where
  star x =  $(x \cdot star\ x) \sqcup \varepsilon$ 

partial-function (lfp) rev-star :: 'a  $\Rightarrow$  'a where
  rev-star x =  $(rev-star\ x \cdot x) \sqcup \varepsilon$ 

end

lemmas parallel-star-induct-1-1 =
  parallel-fixp-induct-1-1[OF
    complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions
    star.mono star.mono star-def star-def]

lemma star-bot:
  shows star  $\perp = \varepsilon$ 
by (subst star.simps) (simp add: comp-botL)

lemma epsilon-star-le:
  shows  $\varepsilon \leq star\ P$ 
by (subst star.simps) simp

lemma monotone-star:
  shows mono star
proof(rule monotoneI)
  fix x y :: 'a
  assume  $x \leq y$  show star x  $\leq$  star y
proof(induct rule: star.fixp-induct[case-names adm bot step])

```

```

case (step R) show ?case
  apply (strengthen ord-to-strengthen(1)[OF ‹x ≤ y›])
  apply (strengthen ord-to-strengthen(1)[OF step])
  apply (rule order.refl)
  done
qed simp-all
qed

lemma expansive-star:
  shows x ≤ star x
by (subst star.simps, subst star.simps)
  (simp add: comp-supL comp-supR comp-unitR le-supI1 flip: sup.assoc)

lemma star-comp-star:
  shows star x · star x = star x (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
  proof(induct rule: star.fixp-induct[where P=λR. R x · star x ≤ star x, case-names adm bot step])
  case (step R) show ?case
    apply (simp add: comp-supL weak-comp-unitL[OF epsilon-star-le] comp-assoc)
    apply (strengthen ord-to-strengthen(1)[OF step])
    apply (subst (2) star.simps)
    apply simp
    done
  qed (simp-all add: comp-botL)
  show ?rhs ≤ ?lhs
  by (strengthen ord-to-strengthen(2)[OF epsilon-star-le])
  (simp add: weak-comp-unitL[OF epsilon-star-le])
qed

lemma idempotent-star:
  shows star (star x) = star x (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
  proof(induct rule: star.fixp-induct[where P=λR. R (star x) ≤ ?rhs, case-names adm bot step])
  next
  case (step R) then show ?case
    by (metis comp-supR epsilon-star-le le-iff-sup le-sup-iff star-comp-star)
  qed simp-all
  show ?rhs ≤ ?lhs
  by (simp add: expansive-star)
qed

interpretation star: closure-complete-lattice-class star
by standard (metis order.trans expansive-star idempotent-star monotoneD[OF monotone-star])

lemma star-epsilon:
  shows star ε = ε
by (metis idempotent-star star-bot)

lemma epsilon-rev-star-le:
  shows ε ≤ rev-star P
by (subst rev-star.simps) simp

lemma rev-star-comp-rev-star:
  shows rev-star x · rev-star x = rev-star x (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs

```

proof(*induct rule: rev-star.fixp-induct*[**where** $P = \lambda R. \text{rev-star } x \cdot R \ x \leq \text{rev-star } x$, *case-names adm bot step*])
case bot show ?case
by (*subst* (2) *rev-star.simps*) (*simp add: le-supI1 mcont-monoD*[*OF mcont-compR*])
next
case (*step R*) **then show** ?case
by (*simp add: comp-supR epsilon-rev-star-le comp-unitR flip: comp-assoc*)
(*metis comp-supL le-iff-sup le-supI1 rev-star.simps*)
qed simp
show ?rhs \leq ?lhs
by (*metis comp-supR comp-unitR epsilon-rev-star-le le-iff-sup*)
qed

lemma *star-rev-star*:

shows $\text{star} = \text{rev-star}$ (**is** ?lhs = ?rhs)

proof(*intro fun-eqI antisym*)

show ?lhs $P \leq$?rhs P **for** P

proof(*induct rule: star.fixp-induct*[*case-names adm bot step*])

case (*step R*)

have *expansive: x* \leq *rev-star x* **for** x

apply (*subst rev-star.simps*)

apply (*subst rev-star.simps*)

apply (*simp add: comp-supL sup-assoc*)

apply (*metis comp-supR comp-unitR sup-ge2 le-supI2 sup-commute weak-comp-unitL*)

done

show ?case

by (*strengthen ord-to-strengthen(1)*[*OF step*])

(*metis comp-supL epsilon-rev-star-le expansive rev-star-comp-rev-star le-sup-iff sup.absorb-iff2*)

qed simp-all

show ?rhs $P \leq$?lhs P **for** P

proof(*induct rule: rev-star.fixp-induct*[*case-names adm bot step*])

case (*step R*) **show** ?case

by (*strengthen ord-to-strengthen(1)*[*OF step*])

(*metis epsilon-star-le expansive-star mcont-monoD*[*OF mcont-compR*] *le-supI star-comp-star*)

qed simp-all

qed

lemmas *star-fixp-rev-induct* = *rev-star.fixp-induct*[*folded star-rev-star*]

interpretation *rev-star: closure-complete-lattice-class rev-star*

by (*rule star.closure-complete-lattice-class-axioms*[*unfolded star-rev-star*])

lemma *rev-star-bot*:

shows $\text{rev-star } \perp = \varepsilon$

by (*simp add: star-bot flip: star-rev-star*)

lemma *rev-star-epsilon*:

shows $\text{rev-star } \varepsilon = \varepsilon$

by (*simp add: star-epsilon flip: star-rev-star*)

lemmas *star-unfoldL* = *star.simps*

lemma *star-unfoldR*:

shows $\text{star } x = (\text{star } x \cdot x) \sqcup \varepsilon$

by (*subst* (1 2) *star-rev-star*) (*simp flip: rev-star.simps*)

lemmas *rev-star-unfoldR* = *rev-star.simps*

lemma *rev-star-unfoldL*:

shows $rev\text{-}star\ x = (x \cdot rev\text{-}star\ x) \sqcup \varepsilon$
by (*simp flip: star-rev-star star-unfoldL*)

lemma *fold-starL*:
shows $x \cdot star\ x \leq star\ x$
by (*subst (2) star.simps*) *simp*

lemma *fold-starR*:
shows $star\ x \cdot x \leq star\ x$
by (*metis inf-sup-ord(3) star-unfoldR*)

lemma *fold-rev-starL*:
shows $x \cdot rev\text{-}star\ x \leq rev\text{-}star\ x$
by (*simp add: fold-starL flip: star-rev-star*)

lemma *fold-rev-starR*:
shows $rev\text{-}star\ x \cdot x \leq rev\text{-}star\ x$
by (*simp add: fold-starR flip: star-rev-star*)

declare *star.strengthen-cl*[*strg*] *rev-star.strengthen-cl*[*strg*]

end

end

locale *kleene* = *weak-kleene* +
assumes *comp-unitL*: $\varepsilon \cdot x = x$ — satisfied by (*'a*, *'s*, *'v*) *prog* but not (*'a*, *'s*, *'v*) *spec*

6 Galois connections

Here we collect some classical results for Galois connections. These are drawn from Backhouse (2000); Davey and Priestley (2002); Melton, Schmidt, and Strecker (1985); Müller-Olm (1997) amongst others. The canonical reference is likely Gierz, Hofmann, Keimel, Lawson, Mislove, and Scott (2003).

Our focus is on constructing closures (§5) conveniently; we are less interested in the fixed-point story. Many of these results hold for preorders; we simply work with partial orders (via the *ordering* locale). Similarly *conditionally complete lattices* are often sufficient, but for convenience we just assume (unconditional) completeness.

locale *galois* =
orda: *ordering less-eqa lessa*
+ *ordb*: *ordering less-eqb lessb*
for *less-eqa* (**infix** $\langle \leq_a \rangle$ 50)
and *lessa* (**infix** $\langle <_a \rangle$ 50)
and *less-eqb* (**infix** $\langle \leq_b \rangle$ 50)
and *lessb* (**infix** $\langle <_b \rangle$ 50)
+ **fixes** *lower* :: $'a \Rightarrow 'b$
fixes *upper* :: $'b \Rightarrow 'a$
assumes *galois*: $lower\ x \leq_b\ y \iff x \leq_a\ upper\ y$
begin

lemma *monotone-lower*:
shows *monotone* (\leq_a) (\leq_b) *lower*
by (*rule monotoneI*) (*use galois orda.trans ordb.refl in blast*)

lemma *monotone-upper*:
shows *monotone* (\leq_b) (\leq_a) *upper*
by (*rule monotoneI*) (*use galois ordb.trans orda.refl in blast*)

lemmas *strengthen-lower*[*strg*] = *st-monotone*[*OF monotone-lower*]

lemmas *strengthen-upper*[*strg*] = *st-monotone*[*OF monotone-upper*]

lemma *upper-lower-expansive*:

shows $x \leq_a \text{upper } (\text{lower } x)$

using *galois ordb.refl* **by** *blast*

lemma *lower-upper-contractive*:

shows $\text{lower } (\text{upper } x) \leq_b x$

using *galois orda.refl* **by** *blast*

lemma *comp-galois*: — [Backhouse \(2000, Lemma 19\)](#). Observe that the roles of upper and lower have swapped.

fixes *less-egc* :: 'c \Rightarrow 'c \Rightarrow bool (**infix** < \leq_c > 50)

fixes *lessc* :: 'c \Rightarrow 'c \Rightarrow bool (**infix** < \leq_c > 50)

fixes *h* :: 'a \Rightarrow 'c

fixes *k* :: 'b \Rightarrow 'c

assumes *partial-preordering* (\leq_c)

assumes *monotone* (\leq_a) (\leq_c) *h*

assumes *monotone* (\leq_b) (\leq_c) *k*

shows $(\forall x. h (\text{upper } x) \leq_c k x) \iff (\forall x. h x \leq_c k (\text{lower } x))$

using *assms(1) monotoneD[OF assms(2)] monotoneD[OF assms(3)]*

by (*meson lower-upper-contractive partial-preordering.trans upper-lower-expansive*)

lemma *lower-upper-le-iff*: — [Backhouse \(2000, Lemma 23\)](#)

assumes $\forall x y. \text{lower}' x \leq_b y \iff x \leq_a \text{upper}' y$

shows $(\forall x. \text{lower}' x \leq_b \text{lower } x) \iff (\forall y. \text{upper } y \leq_a \text{upper}' y)$

using *assms* **by** (*meson lower-upper-contractive orda.trans ordb.trans upper-lower-expansive*)

lemma *lower-upper-unique*: — [Backhouse \(2000, Lemma 24\)](#)

assumes $\forall x y. \text{lower}' x \leq_b y \iff x \leq_a \text{upper}' y$

shows $\text{lower}' = \text{lower} \iff \text{upper}' = \text{upper}$

using *assms galois lower-upper-contractive orda.eq-iff ordb.eq-iff upper-lower-expansive* **by** *blast*

lemma *upper-lower-idem*:

shows $\text{upper } (\text{lower } (\text{upper } (\text{lower } x))) = \text{upper } (\text{lower } x)$

by (*meson galois lower-upper-contractive orda.antisym ordb.trans upper-lower-expansive*)

lemma *lower-upper-idem*:

shows $\text{lower } (\text{upper } (\text{lower } (\text{upper } x))) = \text{lower } (\text{upper } x)$

by (*metis galois ordb.antisym upper-lower-expansive upper-lower-idem*)

lemma *lower-upper-lower*: — [Melton et al. \(1985, Proposition 1.2\(2\)\)](#)

shows $\text{lower} \circ \text{upper} \circ \text{lower} = \text{lower}$

and $\text{lower } (\text{upper } (\text{lower } x)) = \text{lower } x$

using *galois lower-upper-contractive ordb.antisym upper-lower-expansive upper-lower-idem* **by** *auto*

lemma *upper-lower-upper*: — [Melton et al. \(1985, Proposition 1.2\(2\)\)](#)

shows $\text{upper} \circ \text{lower} \circ \text{upper} = \text{upper}$

and $\text{upper } (\text{lower } (\text{upper } x)) = \text{upper } x$

by (*simp-all add: fun-eq-iff*)

(*metis galois monotone-upper monotoneD orda.antisym orda.refl upper-lower-expansive*)+

definition *cl* :: 'a \Rightarrow 'a **where** — The opposite composition yields a kernel operator

$cl x = \text{upper } (\text{lower } x)$

lemma *cl-axiom*:

shows $(x \leq_a cl y) = (cl x \leq_a cl y)$

by (*metis cl-def galois lower-upper-lower(2)*)

sublocale *closure* (\leq_a) ($<_a$) *cl* — incorporates definitions and lemmas into this namespace
by *standard* (*rule cl-axiom*)

lemma *cl-upper*:

shows $cl (upper P) = upper P$

by (*simp add: cl-def upper-lower-upper*)

lemma *closed-upper*:

shows $upper P \in closed$

by (*simp add: closed-def cl-upper orda.refl*)

lemma *inj-lower-iff-surj-upper*:

shows $inj lower \longleftrightarrow surj upper$

by (*metis inj-def surj-def lower-upper-lower(2) upper-lower-upper(2)*)

lemma *inj-lower-iff-upper-lower-id*:

shows $inj lower \longleftrightarrow upper \circ lower = id$

by (*metis fun.map-comp id-comp inj-iff inj-on-id inj-on-imageI2 lower-upper-lower(1)*)

lemma *upper-inj-iff-surj-lower*:

shows $inj upper \longleftrightarrow surj lower$

by (*metis inj-def surj-def lower-upper-lower(2) upper-lower-upper(2)*)

lemma *inj-upper-iff-lower-upper-id*:

shows $inj upper \longleftrightarrow lower \circ upper = id$

by (*metis fun.map-comp id-comp inj-iff inj-on-id inj-on-imageI2 upper-lower-upper(1)*)

lemma *lower-downset-upper*: — Davey and Priestley (2002, Lemma 7.32): inverse image of lower on a downset is the downset of upper

shows $lower - ' \{a. a \leq_b y\} = \{a. a \leq_a upper y\}$

by (*simp add: galois*)

lemma *lower-downset*: — Davey and Priestley (2002, Lemma 7.32); equivalent to the Galois axiom

shows $\exists!x. lower - ' \{a. a \leq_b y\} = \{a. a \leq_a x\}$

by (*metis lower-downset-upper mem-Collect-eq orda.antisym orda.refl*)

end

setup $\langle Sign.mandatory-path galois \rangle$

lemma *axioms-alt*:

fixes *less-eqa* (**infix** $\langle \leq_a \rangle$ 50)

fixes *less-eqb* (**infix** $\langle \leq_b \rangle$ 50)

fixes *lower* :: $'a \Rightarrow 'b$

fixes *upper* :: $'b \Rightarrow 'a$

assumes *oa*: ordering *less-eqa lessa*

assumes *ob*: ordering *less-eqb lessb*

assumes *ul*: $\forall x. x \leq_a upper (lower x)$

assumes *lu*: $\forall x. lower (upper x) \leq_b x$

assumes *ml*: monotone $(\leq_a) (\leq_b) lower$

assumes *mu*: monotone $(\leq_b) (\leq_a) upper$

shows $lower x \leq_b y \longleftrightarrow x \leq_a upper y$

by (*metis lu ml monotoneD mu oa ob ordering.axioms(1) partial-preordering.trans ul*)

lemma *compose*:

fixes *lower₁* :: $'b \Rightarrow 'c$

fixes *lower₂* :: $'a \Rightarrow 'b$

fixes *less-eqa* :: $'a \Rightarrow 'a \Rightarrow bool$

assumes *galois less-eqb lessb less-eqc lessc lower₁ upper₁*
assumes *galois less-eqa lessa less-eqb lessb lower₂ upper₂*
shows *galois less-eqa lessa less-eqc lessc (lower₁ ◦ lower₂) (upper₂ ◦ upper₁)*
using *assms unfolding galois-def galois-axioms-def* **by** *simp*

locale *complete-lattice* =

cla: *complete-lattice Inf_a Sup_a (⊔_a) (≤_a) (<_a) (⊔_a) ⊥_a ⊤_a*
+ *clb*: *complete-lattice Inf_b Sup_b (⊔_b) (≤_b) (<_b) (⊔_b) ⊥_b ⊤_b*
+ *galois* (≤_a) (<_a) (≤_b) (<_b) *lower upper*
for *less-eqa* :: 'a ⇒ 'a ⇒ *bool* (**infix** <≤_a> 50)
and *lessa* (**infix** <<_a> 50)
and *infa* (**infixl** <⊔_a> 70)
and *supa* (**infixl** <⊔_a> 65)
and *bota* (<⊥_a>)
and *topa* (<⊤_a>)
and *Inf_a Sup_a*
and *less-eqb* :: 'b ⇒ 'b ⇒ *bool* (**infix** <≤_b> 50)
and *lessb* (**infix** <<_b> 50)
and *infb* (**infixl** <⊔_b> 70)
and *supb* (**infixl** <⊔_b> 65)
and *botb* (<⊥_b>)
and *topb* (<⊤_b>)
and *Inf_b Sup_b*
and *lower* :: 'a ⇒ 'b
and *upper* :: 'b ⇒ 'a

begin

lemma *lower-bot*:

shows *lower ⊥_a = ⊥_b*
by (*simp add: clb.le-bot galois*)

lemmas *mono2mono-lower[cont-intro, partial-function-mono] = monotone2monotone[OF monotone-lower, simplified]*

lemma *lower-Sup*: — Melton et al. (1985, Proposition 1.2(6)): *lower* is always a distributive operation

shows *lower (Sup_a X) = Sup_b (lower ' X) (is ?lhs = ?rhs)*
proof(*rule clb.order.antisym*)
show *?lhs ≤_b ?rhs* **by** (*meson cla.Sup-least clb.SUP-upper galois*)
show *?rhs ≤_b ?lhs* **by** (*meson cla.Sup-le-iff clb.SUP-le-iff galois upper-lower-expansive*)

qed

lemma *lower-SUP*:

shows *lower (Sup_a (f ' X)) = Sup_b ((λx. lower (f x)) ' X)*
by (*simp add: lower-Sup image-image*)

lemma *lower-sup*:

shows *lower (X ⊔_a Y) = lower X ⊔_b lower Y*
using *lower-Sup[where X={X, Y}]* **by** *simp*

lemma *lower-Inf-le*:

shows *lower (Inf_a X) ≤_b Inf_b (lower ' X)*
by (*simp add: cla.Inf-lower2 clb.le-INF-iff galois upper-lower-expansive*)

lemma *lower-INF-le*:

shows *lower (Inf_a (f ' X)) ≤_b Inf_b ((λx. lower (f x)) ' X)*
by (*simp add: clb.order.trans[OF lower-Inf-le] image-image*)

lemma *lower-inf-le*:

shows $\text{lower } (x \sqcap_a y) \leq_b \text{lower } x \sqcap_b \text{lower } y$
using *lower-Inf-le*[**where** $X=\{x, y\}$] **by** *simp*

lemma *mcont-lower*: — Backhouse (2000): fixed point theory based on Galois connections is less general than using countable chains

shows $\text{mcont } \text{Sup}_a (\leq_a) \text{Sup}_b (\leq_b) \text{lower}$
by (*meson contI lower-Sup mcontI monotone-lower*)

lemma *mcont2mcont-lower*[*cont-intro*]:

assumes $\text{mcont } \text{luba } \text{orda } \text{Sup}_a (\leq_a) P$
shows $\text{mcont } \text{luba } \text{orda } \text{Sup}_b (\leq_b) (\lambda x. \text{lower } (P x))$
using *assms mcont-lower*
partial-function-definitions.mcont2mcont[*OF clb.complete-lattice-partial-function-definitions*]
by *blast*

lemma *upper-top*:

shows $\text{upper } \top_b = \top_a$
by (*simp add: cla.top-le flip: galois*)

lemma *Sup-upper-le*:

shows $\text{Sup}_a (\text{upper } ' X) \leq_a \text{upper } (\text{Sup}_b X)$
by (*meson cla.SUP-le-iff clb.Sup-upper2 galois lower-upper-contractive*)

lemma *sup-upper-le*:

shows $\text{upper } x \sqcup_a \text{upper } y \leq_a \text{upper } (x \sqcup_b y)$
using *Sup-upper-le*[**where** $X=\{x, y\}$] **by** *simp*

lemma *upper-Inf*: — Melton et al. (1985, Proposition 1.2(6))

shows $\text{upper } (\text{Inf}_b X) = \text{Inf}_a (\text{upper } ' X)$ (**is** *?lhs = ?rhs*)
proof(*rule cla.order.antisym*)
show $?lhs \leq_a ?rhs$ **by** (*meson cla.INF-greatest clb.le-Inf-iff galois lower-upper-contractive*)
show $?rhs \leq_a ?lhs$ **by** (*meson cla.INF-lower clb.le-Inf-iff galois*)
qed

lemma *upper-INF*:

shows $\text{upper } (\text{Inf}_b (f ' X)) = \text{Inf}_a ((\lambda x. \text{upper } (f x)) ' X)$
by (*simp add: image-image upper-Inf*)

lemma *upper-inf*:

shows $\text{upper } (X \sqcap_b Y) = \text{upper } X \sqcap_a \text{upper } Y$
using *upper-Inf*[**where** $X=\{X, Y\}$] **by** *simp*

In a complete lattice *lower* is determined by *upper* and vice-versa.

lemma *lower-Inf-upper*:

shows $\text{lower } X = \text{Inf}_b \{Y. X \leq_a \text{upper } Y\}$
by (*auto simp flip: galois intro: clb.Inf-eqI[symmetric]*)

lemma *upper-Sup-lower*:

shows $\text{upper } X = \text{Sup}_a \{Y. \text{lower } Y \leq_b X\}$
by (*auto simp: galois intro: cla.Sup-eqI[symmetric]*)

lemma *upper-downwards-closure-lower*: — Melton et al. (1985, Lemma 2.1)

shows $\text{upper } x = \text{Sup}_a (\text{lower } - ' \{y. y \leq_b x\})$
by (*simp add: upper-Sup-lower*)

sublocale *closure-complete-lattice* $(\leq_a) (<_a) (\sqcap_a) (\sqcup_a) \perp_a \top_a \text{Inf}_a \text{Sup}_a \text{cl}$

by (*rule closure-complete-lattice.intro*[*OF cla.complete-lattice-axioms closure-axioms*])

end

locale *complete-lattice-distributive* =

galois.complete-lattice

+ **assumes** *upper-Sup-le*: $upper (Sup_b X) \leq_a Sup_a (upper \text{ ' } X)$ — Stronger than Scott continuity, which only asks for this for chain or directed X .

begin

lemma *upper-Sup*:

shows $upper (Sup_b X) = Sup_a (upper \text{ ' } X)$

by (*simp add: Sup-upper-le cla.dual-order.antisym upper-Sup-le*)

lemma *upper-bot*:

shows $upper \perp_b = \perp_a$

using *upper-Sup*[**where** $X=\{\}$] **by** *simp*

lemma *upper-sup*:

shows $upper (x \sqcup_b y) = upper x \sqcup_a upper y$

by (*rule upper-Sup*[**where** $X=\{x, y\}$, *simplified*])

lemmas *mono2mono-upper*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF monotone-upper*, *simplified*]

lemma *mcont-upper*:

shows $mcont Sup_b (\leq_b) Sup_a (\leq_a) upper$

by (*meson contI upper-Sup mcontI monotone-upper*)

lemma *mcont2mcont-upper*[*cont-intro*]:

assumes *mcont luba orda Sup_b* $(\leq_b) P$

shows *mcont luba orda Sup_a* $(\leq_a) (\lambda x. upper (P x))$

by (*simp add: ccpo.mcont2mcont*'[*OF cla.complete-lattice-ccpo mcont-upper - assms*])

sublocale *closure-complete-lattice-distributive* $(\leq_a) (<_a) (\sqcap_a) (\sqcup_a) \perp_a \top_a Inf_a Sup_a cl$

by *standard* (*simp add: cl-def upper-Sup lower-Sup image-image*)

lemma *cl-bot*:

shows $cl \perp_a = \perp_a$

by (*simp add: cl-def lower-bot upper-bot*)

lemma *closed-bot*[*iff*]:

shows $\perp_a \in closed$

by (*simp add: cl-bot closed-clI*)

end

locale *complete-lattice-class* =

galois.complete-lattice

$(\leq) (<) (\sqcap) (\sqcup) \perp :: - :: complete-lattice \top Inf Sup$

$(\leq) (<) (\sqcap) (\sqcup) \perp :: - :: complete-lattice \top Inf Sup$

begin

sublocale *closure-complete-lattice-class* $cl ..$

end

locale *complete-lattice-distributive-class* =

galois.complete-lattice-distributive

$(\leq) (<) (\sqcap) (\sqcup) \perp :: - :: complete-lattice \top Inf Sup$

```

( $\leq$ ) ( $<$ ) ( $\sqcap$ ) ( $\sqcup$ )  $\perp$  :: - :: complete-lattice  $\top$  Inf Sup
begin

sublocale galois.complete-lattice-class ..
sublocale closure-complete-lattice-distributive-class cl ..

end

lemma existence-lower-preserves-Sup: — Hoare and He (1987, p8 of Oxford TR PRG-44) amongst others
  fixes lower :: -::complete-lattice  $\Rightarrow$  -::complete-lattice
  assumes mono lower
  shows ( $\forall x y. lower\ x \leq y \iff x \leq \sqcup\{Y. lower\ Y \leq y\}$ )  $\iff$  ( $\forall X. lower\ (\sqcup X) \leq \sqcup(lower\ 'X)$ ) (is ?lhs
 $\iff$  ?rhs)
proof(rule iffI)
  show ?lhs  $\implies$  ?rhs
  by (metis SUP-upper Sup-least)
  show ?rhs  $\implies$  ?lhs
  by (fastforce intro: Sup-upper SUP-least order.trans elim: order.trans[OF monoD[OF assms]])
qed

lemma lower-preserves-SupI:
  assumes mono lower
  assumes  $\bigwedge X. lower\ (\sqcup X) \leq \sqcup(lower\ 'X)$ 
  assumes  $\bigwedge x. upper\ x = \sqcup\{X. lower\ X \leq x\}$ 
  shows galois.complete-lattice-class lower upper
by standard (metis assms galois.existence-lower-preserves-Sup)

lemma existence-upper-preserves-Inf:
  fixes upper :: -::complete-lattice  $\Rightarrow$  -::complete-lattice
  assumes mono upper
  shows ( $\forall x y. \sqcap\{Y. x \leq upper\ Y\} \leq y \iff x \leq upper\ y$ )  $\iff$  ( $\forall X. \sqcap(upper\ 'X) \leq upper\ (\sqcap X)$ ) (is ?lhs
 $\iff$  ?rhs)
proof(rule iffI)
  assume ?lhs
  interpret gcl: galois.complete-lattice-class  $\lambda x. \sqcap\{Y. x \leq upper\ Y\}$  upper
  by standard (use <?lhs> in blast)
  from gcl.upper-Inf show ?rhs by simp
next
  show ?rhs  $\implies$  ?lhs
  by (auto intro: Inf-lower order.trans[rotated] INF-greatest order.trans[OF - monoD[OF assms], rotated])
qed

lemma upper-preserves-InfI:
  assumes mono upper
  assumes  $\bigwedge X. \sqcap(upper\ 'X) \leq upper\ (\sqcap X)$ 
  assumes  $\bigwedge x. lower\ x = \sqcap\{X. x \leq upper\ X\}$ 
  shows galois.complete-lattice-class lower upper
by standard (metis assms galois.existence-upper-preserves-Inf)

locale powerset =
  galois.complete-lattice-class lower upper
for lower :: 'a set  $\Rightarrow$  'b set
and upper :: 'b set  $\Rightarrow$  'a set
begin

lemma lower-insert:
  shows lower (insert x X) = lower {x}  $\cup$  lower X
by (metis insert-is-Un lower-sup)

```

lemma *lower-distributive*:

shows $lower\ X = (\bigcup_{x \in X}. lower\ \{x\})$
using *lower-Sup*[**where** $X = \{\{x\} \mid x. x \in X\}$] **by** (*auto simp: Union-singleton*)

sublocale *closure-powerset cl ..*

end

locale *powerset-distributive =*

galois.powerset
+ *galois.complete-lattice-distributive-class*
begin

lemma *upper-insert*:

shows $upper\ (insert\ x\ X) = upper\ \{x\} \cup upper\ X$
by (*metis insert-is-Un upper-sup*)

lemma *cl-distributive-axiom*:

shows $cl\ (\bigcup\ X) \subseteq \bigcup\ (cl\ 'X)$
by (*simp add: cl-def lower-Sup upper-Sup*)

sublocale *closure-powerset-distributive cl*

by *standard (simp add: cl-distributive-axiom cla.le-supI1)*

end

Müller-Olm (1997, Theorems 3.3.1, 3.3.2): relation image forms a Galois connection. See also Davey and Priestley (2002, Exercise 7.18).

definition $lower_R :: ('a \times 'b)\ set \Rightarrow 'a\ set \Rightarrow 'b\ set$ **where**
 $lower_R\ R\ A = R\ \text{``}\ A$

definition $upper_R :: ('a \times 'b)\ set \Rightarrow 'b\ set \Rightarrow 'a\ set$ **where**
 $upper_R\ R\ B = \{a. \forall b. (a, b) \in R \longrightarrow b \in B\}$

interpretation *relation: galois.powerset galois.lower_R R galois.upper_R R*

unfolding *galois.lower_R-def galois.upper_R-def* **by** *standard blast*

context *galois.powerset*

begin

lemma *relations-galois*:

defines $R \equiv \{(a, b). b \in lower\ \{a\}\}$
shows $lower = galois.lower_R\ R$
and $upper = galois.upper_R\ R$
proof –
show $lower = galois.lower_R\ R$
proof(*rule HOL.ext*)
fix X
have $lower\ X = (\bigcup_{x \in X}. lower\ \{x\})$ **by** (*rule lower-distributive*)
also have $\dots = (\bigcup_{x \in X}. galois.lower_R\ R\ \{x\})$ **by** (*simp add: galois.lower_R-def R-def*)
also have $\dots = galois.lower_R\ R\ X$ **unfolding** *galois.lower_R-def R-def* **by** *blast*
finally show $lower\ X = galois.lower_R\ R\ X$.

qed

then show $upper = galois.upper_R\ R$

using *galois.galois.relation.lower-upper-unique* **by** *blast*

qed

end

setup $\langle \text{Sign.parent-path} \rangle$

6.1 Some Galois connections

setup $\langle \text{Sign.mandatory-path galois} \rangle$

locale *complete-lattice-class-monomorphic*

= *galois.complete-lattice-class upper lower*

for *upper* :: 'a::complete-lattice \Rightarrow 'a **and** *lower* :: 'a \Rightarrow 'a — Avoid 'a itself parameters

interpretation *conj-imp*: *galois.complete-lattice-class* $(\sqcap) x (\longrightarrow_B) x$ **for** *x* :: -::boolean-algebra — Classic example

by *standard* (*simp add: boolean-implication.conv-sup inf-commute shunt1*)

There are very well-behaved Galois connections arising from the image (and inverse image) of sets under a function; stuttering is one instance (§8.1).

locale *image-vimage* =

fixes *f* :: 'a \Rightarrow 'b

begin

definition *lower* :: 'a set \Rightarrow 'b set **where**

lower X = f ` X

definition *upper* :: 'b set \Rightarrow 'a set **where**

upper X = f -` X

lemma *upper-empty[iff]*:

shows *upper* {} = {}

unfolding *upper-def* **by** *simp*

sublocale *galois.powerset-distributive lower upper*

unfolding *lower-def upper-def* **by** *standard* (*simp-all add: image-subset-iff-subset-vimage vimage-Union*)

abbreviation *equivalent* :: 'a relp **where**

equivalent x y \equiv f x = f y

lemma *equiv*:

shows *Equiv-Relations.equivp equivalent*

by (*simp add: equivpI reflpI symp-def transp-def*)

lemma *equiv-cl-singleton*:

assumes *equivalent* x y

shows cl {x} = cl {y}

using *assms* **by** (*simp add: cl-def galois.image-vimage.lower-def*)

lemma *cl-alt-def*:

shows cl X = {(x, y). *equivalent* x y} “ X

by (*simp add: cl-def lower-def upper-def vimage-image-eq*)

sublocale *closure-powerset-distributive-exchange cl*

by *standard* (*auto simp: cl-alt-def intro: exchangeI*)

lemma *closed-in*:

assumes x \in P

assumes *equivalent* x y

assumes P: P \in closed

shows $y \in P$
using *assms(1-2) closed-conv[OF P] unfolding cl-alt-def by blast*

lemma *clE*:

assumes $x \in cl P$
obtains y **where** *equivalent* $y x$ **and** $y \in P$
using *assms unfolding cl-alt-def by blast*

lemma *clI[intro]*:

assumes $x \in P$
assumes *equivalent* $x y$
shows $y \in cl P$
unfolding *cl-alt-def using assms by blast*

lemma *closed-diff[intro]*:

assumes $X \in closed$
assumes $Y \in closed$
shows $X - Y \in closed$
by (*rule closedI (metis Diff-iff assms clE closed-in)*)

lemma *closed-uminus[intro]*:

assumes $X \in closed$
shows $-X \in closed$
using *closed-diff[where X=UNIV, OF - assms] by fastforce*

end

locale *image-vimage-monomorphic*

= *galois.image-vimage* f
for $f :: 'a \Rightarrow 'a$ — Avoid *'a itself* parameters

locale *image-vimage-idempotent*

= *galois.image-vimage-monomorphic* +
assumes *f-idempotent*: $\bigwedge x. f (f x) = f x$
begin

lemma *f-idempotent-comp*:

shows $f \circ f = f$
by (*simp add: comp-def f-idempotent*)

lemma *idemI*:

assumes $f x \in P$
shows $x \in cl P$
using *assms f-idempotent by (auto simp: cl-alt-def)*

lemma *f-cl*:

shows $f x \in cl P \longleftrightarrow x \in cl P$
by (*simp add: cl-alt-def f-idempotent*)

lemma *f-closed*:

assumes $P \in closed$
shows $f x \in P \longleftrightarrow x \in P$
by (*metis assms closed-conv f-cl*)

lemmas *f-closedI = iffD1[OF f-closed]*

end

setup <Sign.parent-path>

7 Heyting algebras

Our (complete) lattices are Heyting algebras. The following development is oriented towards using the derived Heyting implication in a logical fashion. As there are no standard classes for semi-(complete-)lattices we simply work with complete lattices.

References:

- Esakia, Bezhanishvili, Holliday, and Evseev (2019) – fundamental theory
- van Dalen (2004, Lemma 5.2.1) – some equivalences
- <https://en.wikipedia.org/wiki/Pseudocomplement> – properties

```
class heyting-algebra = complete-lattice +
  assumes inf-Sup-distrib1:  $\bigwedge Y::'a \text{ set. } \bigwedge x::'a. x \sqcap (\bigsqcup Y) = (\bigsqcup y \in Y. x \sqcap y)$ 
begin
```

```
definition heyting :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr <math>\longrightarrow_H\longrightarrow_H y =  $\bigsqcup \{z. x \sqcap z \leq y\}$ 
```

```
lemma heyting: — The Galois property for  $(\sqcap)$  and  $\longrightarrow_H$ 
  shows  $z \leq x \longrightarrow_H y \longleftrightarrow z \sqcap x \leq y$  (is ?lhs  $\longleftrightarrow$  ?rhs)
```

```
proof(rule iffI)
```

```
  from inf-Sup-distrib1 have  $\bigsqcup \{a. x \sqcap a \leq y\} \sqcap x \leq y$  by (simp add: SUP-le-iff inf-commute)
  then show ?lhs  $\Longrightarrow$  ?rhs unfolding heyting-def by (meson inf-mono order.trans order-refl)
  show ?rhs  $\Longrightarrow$  ?lhs by (simp add: heyting-def Sup-upper inf.commute)
```

```
qed
```

```
end
```

```
setup <Sign.mandatory-path heyting>
```

```
context heyting-algebra
begin
```

```
lemma commute:
```

```
  shows  $x \sqcap z \leq y \longleftrightarrow z \leq (x \longrightarrow_H y)$ 
  by (simp add: heyting inf.commute)
```

```
lemmas uncurry = iffD1[OF heyting]
```

```
lemmas curry = iffD2[OF heyting]
```

```
lemma curry-conv:
```

```
  shows  $(x \sqcap y \longrightarrow_H z) = (x \longrightarrow_H y \longrightarrow_H z)$ 
  by (simp add: order-eq-iff) (metis heyting eq-refl inf.assoc)
```

```
lemma swap:
```

```
  shows  $P \longrightarrow_H Q \longrightarrow_H R = Q \longrightarrow_H P \longrightarrow_H R$ 
  by (metis curry-conv inf.commute)
```

```
lemma absorb:
```

```
  shows  $y \sqcap (x \longrightarrow_H y) = y$ 
  and  $(x \longrightarrow_H y) \sqcap y = y$ 
  by (simp-all add: curry inf-absorb1 ac-simps)
```

```
lemma detachment:
```

shows $x \sqcap (x \longrightarrow_H y) = x \sqcap y$ (**is** *?thesis1*)
and $(x \longrightarrow_H y) \sqcap x = x \sqcap y$ (**is** *?thesis2*)
proof –
show *?thesis1* **by** (*metis absorb(1) uncurry inf.assoc inf.commute inf.idem inf-iff-le(2)*)
then show *?thesis2* **by** (*simp add: ac-simps*)
qed

lemma discharge:
assumes $x' \leq x$
shows $x' \sqcap (x \longrightarrow_H y) = x' \sqcap y$ (**is** *?thesis1*)
and $(x \longrightarrow_H y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)

proof –
from *assms* **show** *?thesis1* **by** (*metis curry-conv detachment(2) inf.absorb1*)
then show *?thesis2* **by** (*simp add: ac-simps*)
qed

lemma trans:
shows $(x \longrightarrow_H y) \sqcap (y \longrightarrow_H z) \leq x \longrightarrow_H z$
by (*metis curry detachment(2) swap uncurry inf-le2*)

lemma rev-trans:
shows $(y \longrightarrow_H z) \sqcap (x \longrightarrow_H y) \leq x \longrightarrow_H z$
by (*simp add: inf.commute trans*)

lemma discard:
shows $Q \leq P \longrightarrow_H Q$
by (*simp add: curry*)

lemma infR:
shows $x \longrightarrow_H y \sqcap z = (x \longrightarrow_H y) \sqcap (x \longrightarrow_H z)$
by (*simp add: order-eq-iff curry uncurry detachment le-infI2*)

lemma mono:
assumes $x' \leq x$
assumes $y \leq y'$
shows $x \longrightarrow_H y \leq x' \longrightarrow_H y'$
using *assms* **by** (*metis curry detachment(1) uncurry inf-commute inf-absorb2 le-infI1*)

lemma strengthen[stg]:
assumes *st-ord* $(\neg F) X X'$
assumes *st-ord* $F Y Y'$
shows *st-ord* $F (X \longrightarrow_H Y) (X' \longrightarrow_H Y')$
using *assms* **by** (*cases F; simp add: heyting.mono*)

lemma mono2mono[cont-intro, partial-function-mono]:
assumes *monotone orda* $(\geq) F$
assumes *monotone orda* $(\leq) G$
shows *monotone orda* $(\leq) (\lambda x. F x \longrightarrow_H G x)$
by (*simp add: monotoneI curry discharge le-infI1 monotoneD[OF assms(1)] monotoneD[OF assms(2)]*)

lemma mp:
assumes $x \leq y \longrightarrow_H z$
assumes $x \leq y$
shows $x \leq z$
by (*meson assms uncurry inf-greatest order.refl order-trans*)

lemma botL:
shows $\perp \longrightarrow_H x = \top$

by (*simp add: heyting top-le*)

lemma *top-conv*:

shows $x \longrightarrow_H y = \top \iff x \leq y$

by (*metis curry detachment(2) inf-iff-le(1) inf-top.left-neutral*)

lemma *refl[simp]*:

shows $x \longrightarrow_H x = \top$

by (*simp add: top-conv*)

lemma *topL[simp]*:

shows $\top \longrightarrow_H x = x$

by (*metis detachment(1) inf-top-left*)

lemma *topR[simp]*:

shows $x \longrightarrow_H \top = \top$

by (*simp add: top-conv*)

lemma *K[simp]*:

shows $x \longrightarrow_H (y \longrightarrow_H x) = \top$

by (*simp add: discard top-conv*)

subclass *distrib-lattice*

proof — [Esakia et al. \(2019, Proposition 1.5.3\)](#)

have $x \sqcap (y \sqcup z) \leq x \sqcap y \sqcup x \sqcap z$ **for** $x y z :: 'a$

using *commute* **by** *fastforce*

then have $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ **for** $x y z :: 'a$

by (*simp add: order.eq-iff le-infI2*)

then show $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ **for** $x y z :: 'a$

by (*rule distrib-imp1*)

qed

lemma *supL*:

shows $(x \sqcup y) \longrightarrow_H z = (x \longrightarrow_H z) \sqcap (y \longrightarrow_H z)$

by (*simp add: order-eq-iff mono curry uncurry inf-sup-distrib1*)

subclass (**in** *complete-distrib-lattice*) *heyting-algebra* **by** *standard* (*rule inf-Sup*)

lemma *inf-Sup-distrib*:

shows $x \sqcap \bigsqcup Y = (\bigsqcup y \in Y. x \sqcap y)$

and $\bigsqcup Y \sqcap x = (\bigsqcup y \in Y. x \sqcap y)$

by (*simp-all add: inf-Sup-distrib1 inf-commute*)

lemma *inf-SUP-distrib*:

shows $x \sqcap (\bigsqcup i \in I. Y i) = (\bigsqcup i \in I. x \sqcap Y i)$

and $(\bigsqcup i \in I. Y i) \sqcap x = (\bigsqcup i \in I. Y i \sqcap x)$

by (*simp-all add: inf-Sup-distrib image-image ac-simps*)

end

lemma *eq-boolean-implication*: — the implications coincide in *boolean-algebras*

fixes $x :: \text{boolean-algebra}$

shows $x \longrightarrow_H y = x \longrightarrow_B y$

by (*simp add: order.eq-iff boolean-implication-def heyting.detachement heyting.curry flip: shunt1*)

lemmas *simp-thms* =

heyting.botL

heyting.topL

heyting.topR
 heyting.refl

lemma *Sup-prime-Sup-irreducible-iff*:

fixes $x :: \text{--}: \text{heyting-algebra}$

shows $\text{Sup-prime } x \longleftrightarrow \text{Sup-irreducible } x$

by (*fastforce simp: Sup-prime-on-def Sup-irreducible-on-def inf.order-iff heyting.inf-Sup-distrib*
intro: Sup-prime-on-imp-Sup-irreducible-on)

Logical rules ala HOL lemma *bspec*:

fixes $P :: - \Rightarrow (\text{--}: \text{heyting-algebra})$

shows $x \in X \Longrightarrow (\prod_{x \in X}. P x \longrightarrow_H Q x) \sqcap P x \leq Q x$ (**is** $?X \Longrightarrow ?thesis1$)

and $x \in X \Longrightarrow P x \sqcap (\prod_{x \in X}. P x \longrightarrow_H Q x) \leq Q x$ (**is** $- \Longrightarrow ?thesis2$)

and $(\prod x. P x \longrightarrow_H Q x) \sqcap P x \leq Q x$ (**is** $?thesis3$)

and $P x \sqcap (\prod x. P x \longrightarrow_H Q x) \leq Q x$ (**is** $?thesis4$)

proof –

show $?X \Longrightarrow ?thesis1$ **by** (*meson INF-lower heyting.uncurry*)

then show $?X \Longrightarrow ?thesis2$ **by** (*simp add: inf-commute*)

show $?thesis3$ **by** (*simp add: Inf-lower heyting.commute inf-commute*)

then show $?thesis4$ **by** (*simp add: inf-commute*)

qed

lemma *INFL*:

fixes $Q :: \text{--}: \text{heyting-algebra}$

shows $(\prod_{x \in X}. P x \longrightarrow_H Q) = (\bigsqcup_{x \in X}. P x) \longrightarrow_H Q$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$ **by** (*meson INFE SUPE order.refl heyting.commute heyting.uncurry*)

show $?rhs \leq ?lhs$ **by** (*simp add: INFI SupI heyting.mono*)

qed

lemmas *SUPL* = *heyting.INFL[symmetric]*

lemma *INFR*:

fixes $P :: \text{--}: \text{heyting-algebra}$

shows $(\prod_{x \in X}. P \longrightarrow_H Q x) = (P \longrightarrow_H (\prod_{x \in X}. Q x))$ (**is** $?lhs = ?rhs$)

by (*simp add: order-eq-iff INFI INF-lower heyting.mono*)

(*meson INFI INF-lower heyting.curry heyting.uncurry*)

lemmas *Inf-simps* = — "Miniscoping: pushing in universal quantifiers."

Inf-inf

inf-Inf

INF-inf-const1

INF-inf-const2

heyting.INFL

heyting.INFR

lemma *SUPL-le*:

fixes $Q :: \text{--}: \text{heyting-algebra}$

shows $(\bigsqcup_{x \in X}. P x \longrightarrow_H Q) \leq (\prod_{x \in X}. P x) \longrightarrow_H Q$

by (*simp add: INF-lower SUPE heyting.mono*)

lemma *SUPR-le*:

fixes $P :: \text{--}: \text{heyting-algebra}$

shows $(\prod_{x \in X}. P \longrightarrow_H Q x) \leq P \longrightarrow_H (\bigsqcup_{x \in X}. Q x)$

by (*simp add: SUPE SUP-upper heyting.mono*)

lemma *SUP-inf*:

fixes $Q :: \text{--}: \text{heyting-algebra}$

shows $(\bigsqcup_{x \in X}. P \ x \sqcap Q) = (\bigsqcup_{x \in X}. P \ x) \sqcap Q$
by (*simp add: heyting.inf-SUP-distrib(2)*)

lemma *inf-SUP*:

fixes $P :: \text{'a}::\text{heyting-algebra}$
shows $(\bigsqcup_{x \in X}. P \sqcap Q \ x) = P \sqcap (\bigsqcup_{x \in X}. Q \ x)$
by (*simp add: heyting.inf-SUP-distrib(1)*)

lemmas *Sup-simps* = — "Miniscoping: pushing in universal quantifiers."

sup-SUP
SUP-sup
heyting.inf-SUP
heyting.SUP-inf

lemma *mcont2mcont-inf[cont-intro]*:

fixes $F :: - \Rightarrow \text{'a}::\text{heyting-algebra}$
fixes $G :: - \Rightarrow \text{'a}::\text{heyting-algebra}$
assumes $mcont \ luba \ orda \ Sup \ (\le) \ F$
assumes $mcont \ luba \ orda \ Sup \ (\le) \ G$
shows $mcont \ luba \ orda \ Sup \ (\le) \ (\lambda x. F \ x \sqcap G \ x)$

proof —

have *mcont-inf1*: $mcont \ Sup \ (\le) \ Sup \ (\le) \ (\lambda y. x \sqcap y)$ **for** $x :: \text{'a}::\text{heyting-algebra}$
by (*auto intro!: contI mcontI monotoneI intro: le-infI2 simp flip: heyting.inf-SUP-distrib*)
then have *mcont-inf2*: $mcont \ Sup \ (\le) \ Sup \ (\le) \ (\lambda x. x \sqcap y)$ **for** $y :: \text{'a}::\text{heyting-algebra}$
by (*subst inf commute*) (*rule mcont-inf1*)
from *assms mcont-inf1 mcont-inf2* **show** *?thesis*
by (*best intro: ccpo.mcont2mcont'[OF complete-lattice-ccpo] ccpo.mcont-const[OF complete-lattice-ccpo]*)
qed

lemma *closure-imp-distrib-le*: — [Abadi and Plotkin \(1993, Lemma 3.3\)](#), generalized

fixes $P \ Q :: - :: \text{heyting-algebra}$
assumes $cl: \text{closure-axioms} \ (\le) \ cl$
assumes $cl\text{-inf}: \bigwedge x \ y. cl \ x \sqcap cl \ y \le cl \ (x \sqcap y)$
shows $P \longrightarrow_H Q \le cl \ P \longrightarrow_H cl \ Q$

proof —

from cl **have** $(P \longrightarrow_H Q) \sqcap cl \ P \le cl \ (P \longrightarrow_H Q) \sqcap cl \ P$
by (*metis (mono-tags) closure-axioms-def inf-mono order.refl*)
also have $\dots \le cl \ ((P \longrightarrow_H Q) \sqcap P)$
by (*simp add: cl-inf*)
also from cl **have** $\dots \le cl \ Q$
by (*metis (mono-tags) closure-axioms-def order.refl heyting.mono heyting.uncurry*)
finally show *?thesis*
by (*simp add: heyting*)
qed

setup $\langle \text{Sign.parent-path} \rangle$

Pseudocomplements **definition** *pseudocomplement* :: $\text{'a}::\text{heyting-algebra} \Rightarrow \text{'a} \ (\langle \neg_H \ \rangle \ [75] \ 75)$ **where**
 $\neg_H x = x \longrightarrow_H \perp$

lemma *pseudocomplementI*:

shows $x \le \neg_H y \longleftrightarrow x \sqcap y \le \perp$
by (*simp add: pseudocomplement-def heyting*)

setup $\langle \text{Sign.mandatory-path pseudocomplement} \rangle$

lemma *monotone*:

shows *antimono pseudocomplement*

by (simp add: antimonoI heyting.mono pseudocomplement-def)

lemmas strengthen[strg] = st-monotone[OF pseudocomplement.monotone]

lemmas mono = monotoneD[OF pseudocomplement.monotone]

lemmas mono2mono[cont-intro, partial-function-mono]

= monotone2monotone[OF pseudocomplement.monotone, simplified, of orda P for orda P]

lemma eq-boolean-negation: — the negations coincide in boolean-algebras

fixes $x :: \text{-::}\{boolean\text{-algebra}, heyting\text{-algebra}\}$

shows $\neg_H x = \neg x$

by (simp add: pseudocomplement-def heyting.eq-boolean-implication)

lemma heyting:

shows $x \longrightarrow_H \neg_H x = \neg_H x$

by (simp add: pseudocomplement-def order-eq-iff heyting heyting.detachment)

lemma Inf:

shows $x \sqcap \neg_H x = \perp$

and $\neg_H x \sqcap x = \perp$

by (simp-all add: pseudocomplement-def heyting.detachment)

lemma double-le:

shows $x \leq \neg_H \neg_H x$

by (simp add: pseudocomplement-def heyting.detachment heyting.curry)

interpretation double: closure-complete-lattice-class pseudocomplement \circ pseudocomplement

by standard (simp; meson order.trans pseudocomplement.double-le pseudocomplement.mono)

lemma triple:

shows $\neg_H \neg_H \neg_H x = \neg_H x$

by (simp add: order-eq-iff pseudocomplement.double-le pseudocomplement.mono)

lemma contrapos-le:

shows $x \longrightarrow_H y \leq \neg_H y \longrightarrow_H \neg_H x$

by (simp add: heyting.curry heyting.trans pseudocomplement-def)

lemma sup-inf: — half of de Morgan

shows $\neg_H(x \sqcup y) = \neg_H x \sqcap \neg_H y$

by (simp add: pseudocomplement-def heyting.supL)

lemma inf-sup-weak: — the weakened other half of de Morgan

shows $\neg_H(x \sqcap y) = \neg_H \neg_H(\neg_H x \sqcup \neg_H y)$

by (metis (no-types, opaque-lifting) pseudocomplement-def heyting.curry-conv heyting.supL inf-commute pseudocomplement.triple)

lemma fix-triv:

assumes $x = \neg_H x$

shows $x = y$

using assms by (metis antisym bot.extremum inf.idem inf-le2 pseudocomplementI)

lemma double-top:

shows $\neg_H \neg_H(x \sqcup \neg_H x) = \top$

by (metis pseudocomplement-def heyting.refl pseudocomplement.Inf(1) pseudocomplement.sup-inf)

lemma Inf-inf:

fixes $P :: \text{-} \Rightarrow (\text{-::} heyting\text{-algebra})$

shows $(\bigcap x. P x) \sqcap \neg_H P x = \perp$

by (simp add: pseudocomplement-def Inf-lower heyting.discharge(1))

lemma *SUP-le*: — half of de Morgan

fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$

shows $(\bigsqcup x \in X. P x) \leq \neg_H(\bigsqcap x \in X. \neg_H P x)$

by (*rule SUP-least*) (*meson INF-lower order.trans pseudocomplement.double-le pseudocomplement.mono*)

lemma *SUP-INF-le*:

fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$

shows $(\bigsqcup x \in X. \neg_H P x) \leq \neg_H(\bigsqcap x \in X. P x)$

by (*simp add: INF-lower SUPE pseudocomplement.mono*)

lemma *SUP*:

fixes $P :: - \Rightarrow (-::\text{heyting-algebra})$

shows $\neg_H(\bigsqcup x \in X. P x) = (\bigsqcap x \in X. \neg_H P x)$

by (*simp add: order.eq-iff SUP-upper le-INF-iff pseudocomplement.mono*)

(*metis inf-commute pseudocomplement.SUP-le pseudocomplementI*)

setup $\langle \text{Sign.parent-path} \rangle$

7.1 Downwards closure of preorders (downsets)

A *downset* (also *lower set* and *order ideal*) is a subset of a preorder that is closed under the order relation. (An *ideal* is a downset that is *directed*.) Some results require antisymmetry (a partial order).

References:

- [Vickers \(1989\)](#), early chapters.
- https://en.wikipedia.org/wiki/Alexandrov_topology
- [Abadi and Plotkin \(1991, §3\)](#)

setup $\langle \text{Sign.mandatory-path downwards} \rangle$

definition $cl :: 'a::\text{preorder set} \Rightarrow 'a \text{ set}$ **where**

$cl P = \{x \mid x y. y \in P \wedge x \leq y\}$

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *downwards: closure-powerset-distributive downwards.cl* — On preorders

proof *standard*

show $(P \subseteq \text{downwards.cl } Q) \iff (\text{downwards.cl } P \subseteq \text{downwards.cl } Q)$ **for** $P Q :: 'a \text{ set}$

unfolding *downwards.cl-def* **by** (*auto dest: order-trans*)

show $\text{downwards.cl } (\bigsqcup X) \subseteq \bigcup (\text{downwards.cl } ' X) \cup \text{downwards.cl } \{\}$ **for** $X :: 'a \text{ set set}$

unfolding *downwards.cl-def* **by** *auto*

qed

interpretation *downwards: closure-powerset-distributive-anti-exchange (downwards.cl:::order set \Rightarrow -)*

— On partial orders; see [Pfaltz and Šlapal \(2013\)](#)

by *standard (unfold downwards.cl-def; blast intro: anti-exchangeI antisym)*

setup $\langle \text{Sign.mandatory-path downwards} \rangle$

lemma *cl-empty*:

shows $\text{downwards.cl } \{\} = \{\}$

unfolding *downwards.cl-def* **by** *simp*

lemma *closed-empty[iff]*:

shows $\{\} \in \text{downwards.closed}$

using *downwards.cl-def* by *fastforce*

lemma *clI*[*intro*]:

assumes $y \in P$

assumes $x \leq y$

shows $x \in \text{downwards.cl } P$

unfolding *closure.closed-def* *downwards.cl-def* using *assms* by *blast*

lemma *clE*:

assumes $x \in \text{downwards.cl } P$

obtains y where $y \in P$ and $x \leq y$

using *assms* unfolding *downwards.cl-def* by *fast*

lemma *closed-in*:

assumes $x \in P$

assumes $y \leq x$

assumes $P \in \text{downwards.closed}$

shows $y \in P$

using *assms* unfolding *downwards.cl-def* *downwards.closed-def* by *blast*

lemma *order-embedding*: — On preorders; see Davey and Priestley (2002, §1.35)

fixes $x :: \text{preorder}$

shows $\text{downwards.cl } \{x\} \subseteq \text{downwards.cl } \{y\} \longleftrightarrow x \leq y$

using *downwards.cl* by (*blast elim: downwards.clE*)

The lattice of downsets of a set X is always a *heyting-algebra*.

References:

- Ono (2019, §7.5); uses upsets, points to Stone (1938) as the origin
- Esakia et al. (2019, §2.2)
- https://en.wikipedia.org/wiki/Intuitionistic_logic#Heyting_algebra_semantics

definition *imp* :: ' $a :: \text{preorder set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ where

$\text{imp } P Q = \{\sigma. \forall \sigma' \leq \sigma. \sigma' \in P \longrightarrow \sigma' \in Q\}$

lemma *imp-refl*:

shows $\text{downwards.imp } P P = \text{UNIV}$

by (*simp add: downwards.imp-def*)

lemma *imp-contained*:

assumes $P \subseteq Q$

shows $\text{downwards.imp } P Q = \text{UNIV}$

unfolding *downwards.imp-def* using *assms* by *fast*

lemma *heyting-imp*:

assumes $P \in \text{downwards.closed}$

shows $P \subseteq \text{downwards.imp } Q R \longleftrightarrow P \cap Q \subseteq R$

using *assms* unfolding *downwards.imp-def* *downwards.closed-def* by *blast*

lemma *imp-mp'*:

assumes $\sigma \in \text{downwards.imp } P Q$

assumes $\sigma \in P$

shows $\sigma \in Q$

using *assms* by (*simp add: downwards.imp-def*)

lemma *imp-mp*:

shows $P \cap \text{downwards.imp } P Q \subseteq Q$

and $\text{downwards.imp } P \ Q \cap P \subseteq Q$
by (*meson IntD1 IntD2 downwards.imp-mp' subsetI*)+

lemma *imp-contains*:

assumes $X \subseteq Q$

assumes $X \in \text{downwards.closed}$

shows $X \subseteq \text{downwards.imp } P \ Q$

using *assms* **by** (*auto simp: downwards.imp-def elim: downwards.closed-in*)

lemma *imp-downwards*:

assumes $y \in \text{downwards.imp } P \ Q$

assumes $x \leq y$

shows $x \in \text{downwards.imp } P \ Q$

using *assms order-trans* **by** (*force simp: downwards.imp-def*)

lemma *closed-imp*:

shows $\text{downwards.imp } P \ Q \in \text{downwards.closed}$

by (*meson downwards.clE downwards.closedI downwards.imp-downwards*)

The set $\text{downwards.imp } P \ Q$ is the greatest downset contained in the Boolean implication $P \longrightarrow_B Q$, i.e., downwards.imp is the *kernel* of (\longrightarrow_B) (Zwiers 1989). Note that “kernel” is a choice or interior function.

lemma *imp-boolean-implication-subseteq*:

shows $\text{downwards.imp } P \ Q \subseteq P \longrightarrow_B Q$

unfolding *downwards.imp-def boolean-implication.set-alt-def* **by** *blast*

lemma *downwards-closed-imp-greatest*:

assumes $R \subseteq P \longrightarrow_B Q$

assumes $R \in \text{downwards.closed}$

shows $R \subseteq \text{downwards.imp } P \ Q$

using *assms unfolding boolean-implication.set-alt-def downwards.imp-def downwards.closed-def* **by** *blast*

definition *kernel* :: $'a::\text{order set} \Rightarrow 'a \text{ set}$ **where**

$\text{kernel } X = \bigsqcup \{Q \in \text{downwards.closed}. Q \subseteq X\}$

lemma *kernel-def2*:

shows $\text{downwards.kernel } X = \{\sigma. \forall \sigma' \leq \sigma. \sigma' \in X\}$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

show *?lhs* \subseteq *?rhs*

unfolding *downwards.kernel-def* **using** *downwards.closed-conv* **by** *blast*

next

have $x \in \text{?lhs}$ **if** $x \in \text{?rhs}$ **for** x

unfolding *downwards.kernel-def* **using** *that*

by (*auto elim: downwards.clE intro: exI[where x=downwards.cl {x}]*)

then show *?rhs* \subseteq *?lhs* **by** *blast*

qed

lemma *kernel-contractive*:

shows $\text{downwards.kernel } X \subseteq X$

unfolding *downwards.kernel-def* **by** *blast*

lemma *kernel-idempotent*:

shows $\text{downwards.kernel } (\text{downwards.kernel } X) = \text{downwards.kernel } X$

unfolding *downwards.kernel-def* **by** *blast*

lemma *kernel-monotone*:

shows *mono* *downwards.kernel*

unfolding *downwards.kernel-def* **by** (*rule monotoneI*) *blast*

lemma *closed-kernel-conv*:

shows $X \in \text{downwards.closed} \longleftrightarrow \text{downwards.kernel } X = X$

unfolding *downwards.kernel-def2 downwards.closed-def* **by** (*blast elim: downwards.clE*)

lemma *closed-kernel*:

shows $\text{downwards.kernel } X \in \text{downwards.closed}$

by (*simp add: downwards.closed-kernel-conv downwards.kernel-idempotent*)

lemma *kernel-cl*:

shows $\text{downwards.kernel } (\text{downwards.cl } X) = \text{downwards.cl } X$

using *downwards.closed-kernel-conv* **by** *blast*

lemma *cl-kernel*:

shows $\text{downwards.cl } (\text{downwards.kernel } X) = \text{downwards.kernel } X$

by (*simp flip: downwards.closed-conv add: downwards.closed-kernel*)

lemma *kernel-boolean-implication*:

fixes $P :: \text{::order}$

shows $\text{downwards.kernel } (P \longrightarrow_B Q) = \text{downwards.imp } P Q$

unfolding *downwards.kernel-def2 boolean-implication.set-alt-def downwards.imp-def* **by** *blast*

setup $\langle \text{Sign.parent-path} \rangle$

8 Safety logic

Following Abadi and Lamport (1995); Abadi and Plotkin (1991, 1993) (see also Abadi and Merz (1996, §5.5)), we work in the complete lattice of stuttering-closed safety properties (i.e., stuttering-closed downsets) and use this for logical purposes. We avoid many syntactic issues via a shallow embedding into HOL.

8.1 Stuttering

We define *stuttering equivalence* ala Lamport (1994). This allows any agent to repeat any state at any time. We define a normalisation function (\natural) on (a, s, v) *trace.t* and extract the (matroidal) closure over sets of these from the Galois connection *galois.image-vimage*.

setup $\langle \text{Sign.mandatory-path trace} \rangle$

primrec $\text{natural}' :: 's \Rightarrow ('a \times 's) \text{ list} \Rightarrow ('a \times 's) \text{ list}$ **where**

$\text{natural}' s [] = []$

$|\ \text{natural}' s (x \# xs) = (\text{if } \text{snd } x = s \text{ then } \text{natural}' s xs \text{ else } x \# \text{natural}' (\text{snd } x) xs)$

setup $\langle \text{Sign.mandatory-path final}' \rangle$

lemma $\text{natural}'[\text{simp}]$:

shows $\text{trace.final}' s (\text{trace.natural}' s xs) = \text{trace.final}' s xs$

by (*induct xs arbitrary: s simp-all*)

lemma $\text{natural}'\text{-cong}$:

assumes $s = s'$

assumes $\text{trace.natural}' s xs = \text{trace.natural}' s xs'$

shows $\text{trace.final}' s xs = \text{trace.final}' s' xs'$

by (*metis assms trace.final'.natural'*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path natural}' \rangle$

lemma $\text{natural}'$:

shows $\text{trace.natural}' s (\text{trace.natural}' s xs) = \text{trace.natural}' s xs$
by (*induct xs arbitrary: s*) *simp-all*

lemma *length*:

shows $\text{length} (\text{trace.natural}' s xs) \leq \text{length} xs$
by (*induct xs arbitrary: s*) (*simp-all add: le-SucI*)

lemma *subseq*:

shows *subseq* ($\text{trace.natural}' s xs$) *xs*
by (*induct xs arbitrary: s*) *auto*

lemma *remdups-adj*:

shows $s \# \text{map} \text{snd} (\text{trace.natural}' s xs) = \text{remdups-adj} (s \# \text{map} \text{snd} xs)$
by (*induct xs arbitrary: s*) *simp-all*

lemma *append*:

shows $\text{trace.natural}' s (xs @ ys) = \text{trace.natural}' s xs @ \text{trace.natural}' (\text{trace.final}' s xs) ys$
by (*induct xs arbitrary: s*) *simp-all*

lemma *eq-Nil-conv*:

shows $\text{trace.natural}' s xs = [] \longleftrightarrow \text{snd} \text{ ' set } xs \subseteq \{s\}$
and $[] = \text{trace.natural}' s xs \longleftrightarrow \text{snd} \text{ ' set } xs \subseteq \{s\}$
by (*induct xs arbitrary: s*) *simp-all*

lemma *eq-Cons-conv*:

shows $\text{trace.natural}' s xs = y \# ys$
 $\longleftrightarrow (\exists xs' ys'. xs = xs' @ y \# ys' \wedge \text{snd} \text{ ' set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{trace.natural}' (\text{snd } y) ys' = ys)$ (**is** *?lhs*
 \longleftrightarrow *?rhs*)
and $y \# ys = \text{trace.natural}' s xs$
 $\longleftrightarrow (\exists xs' ys'. xs = xs' @ y \# ys' \wedge \text{snd} \text{ ' set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{trace.natural}' (\text{snd } y) ys' = ys)$ (**is**
?thesis1)

proof –

show *?lhs* \longleftrightarrow *?rhs*

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

proof(*induct xs*)

case (*Cons x xs*) **show** *?case*

proof(*cases s = snd x*)

case *True with Cons*

obtain $xs' ys'$

where $xs = xs' @ y \# ys'$ **and** $\text{snd} \text{ ' set } xs' \subseteq \{s\}$ **and** $\text{snd } y \neq s$

and $\text{trace.natural}' (\text{snd } y) ys' = ys$

by *auto*

with *True* **show** *?thesis* **by** (*simp add: exI[where x=x # xs']*)

qed (*use Cons.prem in force*)

qed *simp*

show *?rhs* \implies *?lhs*

by (*auto simp: trace.natural'.append trace.natural'.eq-Nil-conv*)

qed

then show *?thesis1*

by (*rule eq-commute-conv*)

qed

lemma *eq-append-conv*:

shows $\text{trace.natural}' s xs = ys @ zs$
 $\longleftrightarrow (\exists ys' zs'. xs = ys' @ zs' \wedge \text{trace.natural}' s ys' = ys \wedge \text{trace.natural}' (\text{trace.final}' s ys) zs' = zs)$ (**is** *?lhs*
 $=$ *?rhs*)
and $ys @ zs = \text{trace.natural}' s xs$

$\longleftrightarrow (\exists ys' zs'. xs = ys' @ zs' \wedge \text{trace.natural}' s ys' = ys \wedge \text{trace.natural}' (\text{trace.final}' s ys) zs' = zs)$ (is ?thesis1)

proof –

show ?lhs \longleftrightarrow ?rhs

proof(rule iffI)

show ?lhs \implies ?rhs

proof(induct ys arbitrary: s xs)

case (Cons y ys s xs)

from Cons.prem

obtain ys' zs'

where $xs = ys' @ y \# zs'$ **and** $\text{snd } \text{' set } ys' \subseteq \{s\}$

and $\text{snd } y \neq s$ **and** $\text{trace.natural}' (\text{snd } y) zs' = ys @ zs$

by (clarsimp simp: trace.natural'.eq-Cons-conv)

with Cons.hyps[**where** $s=\text{snd } y$ **and** $xs=zs'$] **show** ?case

by (clarsimp simp: trace.natural'.eq-Cons-conv) (metis append.assoc append-Cons)

qed fastforce

show ?rhs \implies ?lhs

by (auto simp: trace.natural'.append)

qed

then show ?thesis1

by (rule eq-commute-conv)

qed

lemma replicate:

shows $\text{trace.natural}' s (\text{replicate } i \text{ as}) = (\text{if } \text{snd } \text{as} = s \vee i = 0 \text{ then } [] \text{ else } [\text{as}])$

by (auto simp: gr0-conv-Suc trace.natural'.eq-Nil-conv)

lemma map-natural':

shows $\text{trace.natural}' (sf \ s) (\text{map } (\text{map-prod } af \ sf) (\text{trace.natural}' s \ xs))$

$= \text{trace.natural}' (sf \ s) (\text{map } (\text{map-prod } af \ sf) \ xs)$

by (induct xs arbitrary: s; simp; metis)

lemma map-inj-on-sf:

assumes inj-on sf (insert s (snd ' set xs))

shows $\text{trace.natural}' (sf \ s) (\text{map } (\text{map-prod } af \ sf) \ xs) = \text{map } (\text{map-prod } af \ sf) (\text{trace.natural}' s \ xs)$

using assms

proof(induct xs arbitrary: s)

case (Cons x xs s)

from Cons.prem **have** $\text{snd } x \neq sf \ s$ **if** $\text{snd } x \neq s$

by (meson image-eqI inj-onD insert-iff list.set-intros(1) that)

with Cons.prem **show** ?case

by (auto intro: Cons.hyps)

qed simp

lemma amap-noop:

assumes $\text{trace.natural}' s \ xs = \text{map } (\text{map-prod } af \ id) \ zs$

shows $\text{trace.natural}' s \ zs = zs$

using assms **by** (induct xs arbitrary: s zs) (auto split: if-split-asm)

lemma take:

shows $\exists j \leq \text{length } xs. \text{take } i (\text{trace.natural}' s \ xs) = \text{trace.natural}' s (\text{take } j \ xs)$

proof(induct xs arbitrary: s i)

case (Cons x xs s i) **then show** ?case **by** (cases i; fastforce)

qed simp

lemma idle-prefix:

assumes $\text{snd } \text{' set } xs \subseteq \{s\}$

shows $\text{trace.natural}' s (xs @ ys) = \text{trace.natural}' s \ ys$

using *assms* by (*simp* add: *trace.natural'.append trace.natural'.eq-Nil-conv*)

lemma *prefixE*:

assumes *trace.natural' s ys = trace.natural' s (xs @ xsrest)*

obtains *xs' xs'rest* **where** *trace.natural' s xs = trace.natural' s xs'* **and** *ys = xs' @ xs'rest*
by (*metis* *assms trace.natural'.eq-append-conv(2)*)

lemma *aset-conv*:

shows $a \in \text{trace.aset } (\text{trace.T } s \ (\text{trace.natural}' \ s \ xs) \ v)$

$\longleftrightarrow (\exists s' s''. (a, s', s'') \in \text{set } (\text{trace.transitions}' \ s \ xs) \wedge s' \neq s'')$

by (*induct* *xs* *arbitrary: s*) (*auto* *simp: trace.aset.simps*)

setup $\langle \text{Sign.parent-path} \rangle$

definition *natural* :: $(\ 'a, \ 's, \ 'v) \text{ trace.t} \Rightarrow (\ 'a, \ 's, \ 'v) \text{ trace.t} \ (\langle \natural \rangle)$ **where**

$\natural \sigma = \text{trace.T } (\text{trace.init } \sigma) \ (\text{trace.natural}' \ (\text{trace.init } \sigma) \ (\text{trace.rest } \sigma)) \ (\text{trace.term } \sigma)$

setup $\langle \text{Sign.mandatory-path } \text{natural} \rangle$

lemma *sel[simp]*:

shows $\text{trace.init } (\natural \sigma) = \text{trace.init } \sigma$

and $\text{trace.rest } (\natural \sigma) = \text{trace.natural}' \ (\text{trace.init } \sigma) \ (\text{trace.rest } \sigma)$

and $\text{trace.term } (\natural \sigma) = \text{trace.term } \sigma$

by (*simp-all* add: *trace.natural-def*)

lemma *simps*:

shows $\natural(\text{trace.T } s \ [] \ v) = \text{trace.T } s \ [] \ v$

and $\natural(\text{trace.T } s \ ((a, s) \# \ xs) \ v) = \natural(\text{trace.T } s \ xs \ v)$

and $\natural(\text{trace.T } s \ (\text{trace.natural}' \ s \ xs) \ v) = \natural(\text{trace.T } s \ xs \ v)$

by (*simp-all* add: *trace.natural-def trace.natural'.natural'*)

lemma *idempotent[simp]*:

shows $\natural(\natural \sigma) = \natural \sigma$

by (*simp* add: *trace.natural-def trace.natural'.natural'*)

lemma *idle*:

assumes *snd ' set xs* $\subseteq \{s\}$

shows $\natural(\text{trace.T } s \ xs \ v) = \text{trace.T } s \ [] \ v$

by (*simp* add: *assms trace.natural-def trace.natural'.eq-Nil-conv*)

lemma *trace-conv*:

shows $\natural(\text{trace.T } s \ xs \ v) = \natural \sigma \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s \ (\text{trace.rest } \sigma) \wedge \text{trace.term } \sigma = v$

and $\natural \sigma = \natural(\text{trace.T } s \ xs \ v) \longleftrightarrow \text{trace.init } \sigma = s \wedge \text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s \ (\text{trace.rest } \sigma) \wedge \text{trace.term } \sigma = v$

by (*cases* σ ; *fastforce* *simp: trace.natural-def*) $+$

lemma *map-natural*:

shows $\natural(\text{trace.map } af \ sf \ vf \ (\natural \sigma)) = \natural(\text{trace.map } af \ sf \ vf \ \sigma)$

by (*simp* add: *trace.natural-def trace.natural'.map-natural'*)

lemma *continue*:

shows $\natural(\sigma @_{-S} \ xsv) = \natural \sigma @_{-S} (\text{trace.natural}' \ (\text{trace.final } \sigma) \ (\text{fst } \ xsv), \ \text{snd } \ xsv)$

by (*simp* add: *trace.natural-def trace.natural'.append split: option.split*)

lemma *replicate*:

shows $\natural(\text{trace.T } s \ (\text{replicate } i \ as) \ v)$

$= (\text{trace.T } s \ (\text{if } \text{snd } as = s \vee i = 0 \text{ then } [] \ \text{else } [as]) \ v)$

by (*simp add: trace.natural-def trace.natural'.replicate*)

lemma *monotone*:

shows *mono* \Downarrow

using *trace.natural.continue* **by** (*fastforce intro: monoI simp: trace.less-eq-t-def*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF trace.natural.monotone*]

lemmas *mono* = *monotoneD*[*OF trace.natural.monotone*]

lemmas *mono2mono*[*cont-intro, partial-function-mono*]

= *monotone2monotone*[*OF trace.natural.monotone, simplified, of orda P for orda P*]

lemma *less-eqE*:

assumes $t \leq u$

assumes $\Downarrow u' = \Downarrow u$

obtains t' **where** $\Downarrow t = \Downarrow t'$ **and** $t' \leq u'$

using *assms*

by *atomize-elim*

(*fastforce simp: trace.natural-def trace.split-Ex trace.less-eq-None(2)[unfolded prefix-def]*)

elim!:: trace.less-eqE prefixE trace.natural'.prefixE)

lemma *less-eq-natural*:

assumes $\sigma_1 \leq \Downarrow \sigma_2$

shows $\Downarrow \sigma_1 = \sigma_1$

using *assms*

by (*cases* σ_1)

(*auto simp: trace.natural-def prefix-def trace.natural'.eq-append-conv trace.natural'.natural'*)

elim!:: trace.less-eqE)

lemma *map-le*:

assumes $\Downarrow \sigma_1 \leq \Downarrow \sigma_2$

shows $\Downarrow(\text{trace.map af sf vf } \sigma_1) \leq \Downarrow(\text{trace.map af sf vf } \sigma_2)$

using *trace.natural.mono[OF trace.map.mono[OF assms], simplified trace.natural.map-natural]* .

lemma *map-inj-on-sf*:

assumes *inj-on sf* (*trace.sset* σ)

shows $\Downarrow(\text{trace.map af sf vf } \sigma) = \text{trace.map af sf vf } (\Downarrow \sigma)$

using *assms* **by** (*cases* σ) (*simp add: trace.natural-def trace.natural'.map-inj-on-sf trace.sset.simps*)

lemma *take*:

shows $\exists j. \Downarrow(\text{trace.take } i \sigma) = \text{trace.take } j (\Downarrow \sigma)$

by (*meson trace.natural.mono trace.less-eq-take-def*)

lemma *take-natural*:

shows $\Downarrow(\text{trace.take } i (\Downarrow \sigma)) = \text{trace.take } i (\Downarrow \sigma)$

using *trace.natural.less-eq-natural* **by** *blast*

lemma *takeE*:

shows $\llbracket \sigma_1 = \Downarrow(\text{trace.take } i \sigma_2); \bigwedge j. \llbracket \sigma_1 = \text{trace.take } j (\Downarrow \sigma_2) \rrbracket \implies \text{thesis} \rrbracket \implies \text{thesis}$

and $\llbracket \Downarrow(\text{trace.take } i \sigma_2) = \sigma_1; \bigwedge j. \llbracket \sigma_1 = \text{trace.take } j (\Downarrow \sigma_2) \rrbracket \implies \text{thesis} \rrbracket \implies \text{thesis}$

using *trace.natural.take* **by** *blast+*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path aset} \rangle$

lemma *natural-conv*:

shows $a \in \text{trace.aset } (\Downarrow \sigma) \iff (\exists s s'. (a, s, s') \in \text{trace.steps } \sigma)$

by (*simp add: trace.natural-def trace.steps'-def trace.natural'.aset-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path sset} \rangle$

lemma *natural'[simp]*:

shows $\text{trace.sset} (\text{trace.T } s_0 (\text{trace.natural}' s_0 xs) v) = \text{trace.sset} (\text{trace.T } s_0 xs v)$
by (*induct xs arbitrary: s₀*) (*simp-all add: trace.sset.simps*)

lemma *natural[simp]*:

shows $\text{trace.sset} (\natural \sigma) = \text{trace.sset } \sigma$
by (*simp add: trace.natural-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path vset} \rangle$

lemma *natural[simp]*:

shows $\text{trace.vset} (\natural \sigma) = \text{trace.vset } \sigma$
by (*cases* σ) (*simp add: trace.natural-def trace.t.simps(8)*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path take} \rangle$

lemma *natural*:

shows $\exists j \leq \text{Suc} (\text{length} (\text{trace.rest } \sigma)). \text{trace.take } i (\natural \sigma) = \natural (\text{trace.take } j \sigma)$
using *trace.natural'.take* [**where** $i=i$ **and** $s = \text{trace.init } \sigma$ **and** $xs = \text{trace.rest } \sigma$]
by (*auto simp: trace.natural-def trace.take-def not-le*) (*use Suc-n-not-le-n in blast*)

lemma *naturalE*:

shows $\llbracket \sigma_1 = \text{trace.take } i (\natural \sigma_2); \bigwedge j. \llbracket j \leq \text{Suc} (\text{length} (\text{trace.rest } \sigma_2)); \sigma_1 = \natural (\text{trace.take } j \sigma_2) \rrbracket \implies \text{thesis} \rrbracket \implies$
thesis
and $\llbracket \text{trace.take } i (\natural \sigma_2) = \sigma_1; \bigwedge j. \llbracket j \leq \text{Suc} (\text{length} (\text{trace.rest } \sigma_2)); \natural (\text{trace.take } j \sigma_2) = \sigma_1 \rrbracket \implies \text{thesis} \rrbracket \implies$
thesis
using *trace.take.natural[of σ_2 i]* **by** *force+*

setup $\langle \text{Sign.parent-path} \rangle$

lemma *steps'-alt-def*:

shows $\text{trace.steps}' s xs = \text{set} (\text{trace.transitions}' s (\text{trace.natural}' s xs))$
by (*induct xs arbitrary: s*) *auto*

setup $\langle \text{Sign.mandatory-path steps}' \rangle$

lemma *natural'*:

shows $\text{trace.steps}' s (\text{trace.natural}' s xs) = \text{trace.steps}' s xs$
unfolding *trace.steps'-def* **by** (*induct xs arbitrary: s*) *auto*

lemma *asetD*:

assumes $\text{trace.steps } \sigma \subseteq r$
shows $\forall a. a \in \text{trace.aset} (\natural \sigma) \longrightarrow a \in \text{fst } ' r$
using *assms* **by** (*force simp: trace.aset.natural-conv*)

lemma *range-initE*:

assumes $\text{trace.steps}' s_0 xs \subseteq \text{range } af \times \text{range } sf \times \text{range } sf$
assumes $(a, s, s') \in \text{trace.steps}' s_0 xs$
obtains s_0' **where** $s_0 = sf s_0'$

using *assms* by (induct *xs* arbitrary: *s* *s*₀) (auto simp: trace.steps'-alt-def split: if-split-asm)

lemma *map-range-conv*:

shows trace.steps' (sf *s*) *xs* ⊆ range *af* × range *sf* × range *sf*
←→ (∃ *xs'*. trace.natural' (sf *s*) *xs* = map (map-prod *af* *sf*) *xs'*) (is ?lhs ←→ ?rhs)

proof(rule iffI)

show ?lhs ⇒ ?rhs

by (induct *xs* arbitrary: *s*) (auto 0 3 simp: Cons-eq-map-conv)

show ?rhs ⇒ ?lhs

by (force simp: trace.steps'-alt-def trace.transitions'.map map-prod-conv)

qed

lemma *step-conv*:

shows trace.steps' *s* *xs* = {*x*}
←→ fst (snd *x*) = *s* ∧ fst (snd *x*) ≠ snd (snd *x*)
∧ (∃ *ys zs*. snd ' set *ys* ⊆ {*s*} ∧ snd ' set *zs* ⊆ {snd (snd *x*)}
∧ *xs* = *ys* @ [(fst *x*, snd (snd *x*))] @ *zs*) (is ?lhs ←→ ?rhs)

proof(rule iffI)

show ?lhs ⇒ ?rhs

by (fastforce dest!: arg-cong[where *f*=set]

simp: trace.steps'-alt-def set-singleton-conv set-replicate-conv-if
trace.transitions'.eq-Nil-conv trace.transitions'.eq-Cons-conv
trace.natural'.eq-Nil-conv trace.natural'.eq-Cons-conv
split: if-split-asm)

show ?rhs ⇒ ?lhs

by (clarsimp simp: trace.steps'.append)

qed

setup ⟨Sign.parent-path⟩

interpretation *stuttering*: galois.image-vimage-idempotent †

by (simp add: galois.image-vimage-idempotent.intro)

abbreviation *stuttering-equiv-syn* :: ('*a*, '*s*, '*v*) trace.*t* ⇒ ('*a*, '*s*, '*v*) trace.*t* ⇒ bool (infix <≈_{*S*}> 50) where

$\sigma_1 \simeq_S \sigma_2 \equiv \text{trace.stuttering.equivalent } \sigma_1 \sigma_2$

setup ⟨Sign.mandatory-path *stuttering*⟩

setup ⟨Sign.mandatory-path *cl*⟩

setup ⟨Sign.mandatory-path *downwards*⟩

lemma *cl*:

shows trace.stuttering.cl (downwards.cl *P*) = downwards.cl (trace.stuttering.cl *P*) (is ?lhs = ?rhs)

proof(rule antisym)

show ?lhs ⊆ ?rhs

by (clarsimp simp: trace.stuttering.cl-alt-def downwards.cl-def trace.less-eq-t-def)
(metis trace.final'.natural' trace.natural.continue trace.natural.sel(1,2))

next

show ?rhs ⊆ ?lhs

by (clarsimp elim!: downwards.clE trace.stuttering.clE)
(erule (1) trace.natural.less-eqE; fastforce)

qed

lemma *closed*:

assumes *P* ∈ downwards.closed

shows trace.stuttering.cl *P* ∈ downwards.closed

by (metis assms downwards.closedI downwards.closed-conv trace.stuttering.cl.downwards.cl)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *downwards-imp*: — Abadi and Plotkin (1993, p13)

assumes $P \in \text{trace.stuttering.closed}$

assumes $Q \in \text{trace.stuttering.closed}$

shows $\text{downwards.imp } P \ Q \in \text{trace.stuttering.closed}$

by (*simp add*: *assms downwards.closed-imp downwards.heyting-imp downwards.imp-mp*
trace.stuttering.cl.downwards.closed trace.stuttering.closed-clI
trace.stuttering.exchange-closed-inter trace.stuttering.least)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path equiv} \rangle$

lemma *simps*:

shows $\text{snd } ' \text{ set } xs \subseteq \{s\} \implies \text{trace.T } s \ (xs \ @ \ ys) \ v \simeq_S \text{trace.T } s \ ys \ v$

and $\text{snd } ' \text{ set } ys \subseteq \{\text{trace.final}' \ s \ xs\} \implies \text{trace.T } s \ (xs \ @ \ ys) \ v \simeq_S \text{trace.T } s \ xs \ v$

and $\text{snd } ' \text{ set } xs \subseteq \{\text{snd } x\} \implies \text{trace.T } s \ (x \ # \ xs \ @ \ ys) \ v \simeq_S \text{trace.T } s \ (x \ # \ ys) \ v$

by (*fastforce simp*: *trace.natural-def trace.natural'.append trace.natural'.eq-Nil-conv*)+

lemma *append-cong*:

assumes $s = s'$

assumes $\text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s \ xs'$

assumes $\text{trace.natural}' \ (\text{trace.final}' \ s \ xs) \ ys = \text{trace.natural}' \ (\text{trace.final}' \ s \ xs) \ ys'$

assumes $v = v'$

shows $\text{trace.T } s \ (xs \ @ \ ys) \ v \simeq_S \text{trace.T } s' \ (xs' \ @ \ ys') \ v'$

using *assms* by (*simp add*: *trace.natural-def trace.natural'.append cong: trace.final'.natural'-cong*)

lemma *E*:

assumes $\text{trace.T } s \ xs \ v \simeq_S \text{trace.T } s' \ xs' \ v'$

obtains $\text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s' \ xs'$ and $s = s'$ and $v = v'$

using *assms* by (*fastforce simp*: *trace.natural-def*)

lemma *append-conv*:

shows $\text{trace.T } s \ (xs \ @ \ ys) \ v \simeq_S \sigma$

$\longleftrightarrow (\exists xs' \ ys'. \sigma = \text{trace.T } s \ (xs' \ @ \ ys') \ v \wedge \text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s \ xs'$
 $\wedge \text{trace.natural}' \ (\text{trace.final}' \ s \ xs) \ ys = \text{trace.natural}' \ (\text{trace.final}' \ s \ xs) \ ys')$ (**is** *?thesis1*)

and $\sigma \simeq_S \text{trace.T } s \ (xs \ @ \ ys) \ v$

$\longleftrightarrow (\exists xs' \ ys'. \sigma = \text{trace.T } s \ (xs' \ @ \ ys') \ v \wedge \text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s \ xs'$
 $\wedge \text{trace.natural}' \ (\text{trace.final}' \ s \ xs) \ ys = \text{trace.natural}' \ (\text{trace.final}' \ s \ xs) \ ys')$ (**is** *?thesis2*)

proof –

show *?thesis1*

by (*cases* σ)

(*fastforce simp*: *trace.natural'.append trace.natural'.eq-append-conv*

elim: trace.stuttering.equiv.E

intro: trace.stuttering.equiv.append-cong)

then show *?thesis2*

by (*rule eq-commute-conv*)

qed

lemma *map*:

assumes $\sigma_1 \simeq_S \sigma_2$

shows $\text{trace.map } af \ sf \ vf \ \sigma_1 \simeq_S \text{trace.map } af \ sf \ vf \ \sigma_2$

by (*metis* *assms* *trace.natural.map-natural*)

lemma *steps*:

assumes $\sigma_1 \simeq_S \sigma_2$

shows *trace.steps* $\sigma_1 = \text{trace.steps } \sigma_2$

using *assms* **by** (*force simp: trace.steps'-alt-def trace.natural-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.2 The ('a, 's, 'v) spec lattice

Our workhorse lattice consists of all sets of traces that are downwards and stuttering closed. This combined closure is neither matroidal nor antimatroidal (§5.3).

We define the lattice as a type and instantiate the relevant type classes. In the following read $P \leq Q$ ($P \subseteq Q$ in the powerset model) as “Q follows from P” or “P entails Q”.

setup $\langle \text{Sign.mandatory-path raw} \rangle$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition *cl* :: ('a, 's, 'v) *trace.t set* \Rightarrow ('a, 's, 'v) *trace.t set* **where**

cl *P* = *downwards.cl* (*trace.stuttering.cl* *P*)

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *spec: closure-powerset-distributive raw.spec.cl*

unfolding *raw.spec.cl-def*

by (*simp add: closure-powerset-distributive-comp downwards.closure-powerset-distributive-axioms trace.stuttering.closure-powerset-distributive-axioms trace.stuttering.cl.downwards.cl*)

setup $\langle \text{Sign.mandatory-path spec} \rangle$

setup $\langle \text{Sign.mandatory-path cl} \rangle$

lemma *empty[simp]*:

shows *raw.spec.cl* $\{\} = \{\}$

by (*simp add: raw.spec.cl-def downwards.cl-empty trace.stuttering.cl-bot*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *I*:

assumes $P \in \text{downwards.closed}$

assumes $P \in \text{trace.stuttering.closed}$

shows $P \in \text{raw.spec.closed}$

by (*metis* *assms* *raw.spec.cl-def downwards.closed-conv raw.spec.closed trace.stuttering.closed-conv*)

lemma *empty[intro]*:

shows $\{\} \in \text{raw.spec.closed}$

using *raw.spec.cl.empty* **by** *blast*

lemma *downwards-closed*:

assumes $P \in \text{raw.spec.closed}$

```

  shows  $P \in \text{downwards.closed}$ 
by (metis assms downwards.closed raw.spec.cl-def raw.spec.closed-conv)

lemma stuttering-closed:
  assumes  $P \in \text{raw.spec.closed}$ 
  shows  $P \in \text{trace.stuttering.closed}$ 
using assms raw.spec.cl-def raw.spec.closed-conv by fast

lemma downwards-imp:
  assumes  $P \in \text{raw.spec.closed}$ 
  assumes  $Q \in \text{raw.spec.closed}$ 
  shows  $\text{downwards.imp } P \ Q \in \text{raw.spec.closed}$ 
by (meson assms downwards.closed-imp raw.spec.closed.I raw.spec.closed.stuttering-closed
    trace.stuttering.cl.closed.downwards-imp)

lemma heyting-downwards-imp:
  assumes  $P \in \text{raw.spec.closed}$ 
  shows  $P \subseteq \text{downwards.imp } Q \ R \longleftrightarrow P \cap Q \subseteq R$ 
by (simp add: assms downwards.heyting-imp raw.spec.closed.downwards-closed)

lemma takeE:
  assumes  $\sigma \in P$ 
  assumes  $P \in \text{raw.spec.closed}$ 
  shows  $\text{trace.take } i \ \sigma \in P$ 
by (meson assms downwards.closed-in raw.spec.closed.downwards-closed trace.less-eq-take)

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.parent-path>

typedef ('a, 's, 'v) spec = raw.spec.closed :: ('a, 's, 'v) trace.t set set
morphisms unMkS MkS
by blast

setup-lifting type-definition-spec

instantiation spec :: (type, type, type) complete-distrib-lattice
begin

lift-definition bot-spec :: ('a, 's, 'v) spec is empty ..
lift-definition top-spec :: ('a, 's, 'v) spec is UNIV ..
lift-definition sup-spec :: ('a, 's, 'v) spec  $\Rightarrow$  ('a, 's, 'v) spec  $\Rightarrow$  ('a, 's, 'v) spec is sup ..
lift-definition inf-spec :: ('a, 's, 'v) spec  $\Rightarrow$  ('a, 's, 'v) spec  $\Rightarrow$  ('a, 's, 'v) spec is inf ..
lift-definition less-eq-spec :: ('a, 's, 'v) spec  $\Rightarrow$  ('a, 's, 'v) spec  $\Rightarrow$  bool is less-eq .
lift-definition less-spec :: ('a, 's, 'v) spec  $\Rightarrow$  ('a, 's, 'v) spec  $\Rightarrow$  bool is less .
lift-definition Inf-spec :: ('a, 's, 'v) spec set  $\Rightarrow$  ('a, 's, 'v) spec is Inf ..
lift-definition Sup-spec :: ('a, 's, 'v) spec set  $\Rightarrow$  ('a, 's, 'v) spec is  $\lambda X. \text{Sup } X \sqcup \text{raw.spec.cl } \{ \}$  ..

instance
by (standard; transfer; auto simp: raw.spec.closed-strict-complete-distrib-lattice-axiomI[OF raw.spec.cl.empty])

end

declare
  SUPE[where 'a=('a, 's, 'v) spec, intro!]
  SupE[where 'a=('a, 's, 'v) spec, intro!]

```

Sup-le-iff[**where** $'a=('a, 's, 'v)$ *spec, simp*]
SupI[**where** $'a=('a, 's, 'v)$ *spec, intro*]
SUPI[**where** $'a=('a, 's, 'v)$ *spec, intro*]
rev-SUPI[**where** $'a=('a, 's, 'v)$ *spec, intro?*]
INFE[**where** $'a=('a, 's, 'v)$ *spec, intro*]

Observations about this type:

- it is not a BNF (datatype) as it uses the powerset
- it fails to be T0 or sober due to the lack of limit points (completeness) in $('a, 's, 'v)$ *trace.t*
 - also stuttering closure precludes T0
- the *complete-distrib-lattice* instance shows that arbitrary/infinitary *Sups* and *Infs* distribute
 - in other words: safety properties are closed under arbitrary intersections and unions
 - in other words: Alexandrov
- conclude: the lack of limit points makes this model easier to work in and adds expressivity
 - see §24 for further discussion

setup \langle *Sign.mandatory-path spec* \rangle

lemmas *antisym* = *antisym*[**where** $'a=('a, 's, 'v)$ *spec*]

lemmas *eq-iff* = *order.eq-iff*[**where** $'a=('a, 's, 'v)$ *spec*]

setup \langle *Sign.parent-path* \rangle

8.3 Irreducible elements

The irreducible elements of $('a, 's, 'v)$ *trace.t* are the closures of singletons.

setup \langle *Sign.mandatory-path raw* \rangle

definition *singleton* :: $('a, 's, 'v)$ *trace.t* \Rightarrow $('a, 's, 'v)$ *trace.t* *set* **where**
singleton σ = *raw.spec.cl* $\{\sigma\}$

lemma *singleton-le-conv*:

shows *raw.singleton* $\sigma_1 \leq$ *raw.singleton* $\sigma_2 \iff \Downarrow \sigma_1 \leq \Downarrow \sigma_2$ (**is** *?lhs* \iff *?rhs*)

proof(*rule iffI*)

assume *?lhs*

then have $\sigma \in$ *downwards.cl* $\{\Downarrow \sigma_2\}$ **if** $\sigma \leq \Downarrow \sigma_1$ **for** σ

using *that trace.natural.mono*

by (*force simp: raw.singleton-def raw.spec.cl-def*

intro: downwards.clI[**where** $y=\Downarrow \sigma_1$]

elim!: *downwards.clE trace.stuttering.clE*

dest!: *subsetD*[**where** $c=\sigma$]

dest: trace.natural.less-eq-natural)

then show *?rhs*

by (*fastforce simp flip: downwards.order-embedding*[**where** $x=\Downarrow \sigma_1$]

elim: downwards.clE trace.stuttering.clE)

next

show *?rhs* \implies *?lhs*

by (*clarsimp simp: raw.singleton-def raw.spec.cl-def*)

(*metis downwards.order-embedding trace.natural.idempotent trace.stuttering.cl.downwards.cl*
trace.stuttering.cl-mono trace.stuttering-equiv-cl-singleton)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

lift-definition $\text{singleton} :: ('a, 's, 'v) \text{trace.t} \Rightarrow ('a, 's, 'v) \text{spec} (\langle \! \langle - \! \rangle \! \rangle)$ **is** raw.singleton
by $(\text{simp add: raw.singleton-def})$

abbreviation $\text{singleton-trace-syn} :: 's \Rightarrow ('a \times 's) \text{list} \Rightarrow 'v \text{option} \Rightarrow ('a, 's, 'v) \text{spec} (\langle \! \langle -, -, - \! \rangle \! \rangle)$ **where**
 $\langle \! \langle s, xs, v \! \rangle \! \rangle \equiv \langle \! \langle \text{trace.T } s \text{ } xs \text{ } v \! \rangle \! \rangle$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma Sup-prime :

shows $\text{Sup-prime} \langle \! \langle \sigma \! \rangle \! \rangle$

by $(\text{clarsimp simp: Sup-prime-on-def})$

$(\text{transfer; auto simp: raw.singleton-def elim!: Sup-prime-onE[OF raw.spec.Sup-prime-on-singleton]})$

lemma nchotomy :

shows $\exists X \in \text{raw.spec.closed. } x = \bigsqcup (\text{spec.singleton } ' X)$

by transfer

$(\text{use raw.spec.closed-conv in } \langle \text{auto simp: raw.singleton-def} \\ \text{simp flip: raw.spec.distributive[simplified]} \rangle)$

lemmas $\text{exhaust} = \text{bexE[OF spec.singleton.nchotomy]}$

lemma collapse[simp] :

shows $\bigsqcup (\text{spec.singleton } ' \{\sigma. \langle \! \langle \sigma \! \rangle \! \rangle \leq P\}) = P$

by $(\text{rule spec.singleton.exhaust[of P]; blast intro: antisym})$

lemmas $\text{not-bot} = \text{Sup-prime-not-bot[OF spec.singleton.Sup-prime]}$ — Non-triviality

setup $\langle \text{Sign.parent-path} \rangle$

lemma $\text{singleton-le-ext-conv}$:

shows $P \leq Q \iff (\forall \sigma. \langle \! \langle \sigma \! \rangle \! \rangle \leq P \longrightarrow \langle \! \langle \sigma \! \rangle \! \rangle \leq Q)$ **(is** $?lhs \iff ?rhs)$

proof (rule iffI)

show $?rhs \implies ?lhs$

by $(\text{rule spec.singleton.exhaust[where } x=P]; \text{rule spec.singleton.exhaust[where } x=Q]; \text{blast})$

qed fastforce

lemmas $\text{singleton-le-conv} = \text{raw.singleton-le-conv[transferred]}$

lemmas $\text{singleton-le-extI} = \text{iffD2[OF spec.singleton-le-ext-conv, rule-format]}$

lemma $\text{singleton-eq-conv[simp]}$:

shows $\langle \! \langle \sigma \! \rangle \! \rangle = \langle \! \langle \sigma' \! \rangle \! \rangle \iff \sigma \simeq_S \sigma'$

by $(\text{auto simp: spec.eq-iff spec.singleton-le-conv})$

lemma singleton-cong :

assumes $\sigma \simeq_S \sigma'$

shows $\langle \! \langle \sigma \! \rangle \! \rangle = \langle \! \langle \sigma' \! \rangle \! \rangle$

using assms **by** simp

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

named-theorems $\text{le-conv} \langle \text{simplification rules for } \langle \! \langle \sigma \! \rangle \! \rangle \leq \text{const } \dots \rangle$

lemmas $\text{antisym} = \text{antisym[OF spec.singleton-le-extI spec.singleton-le-extI]}$

lemmas *top* = *spec.singleton.collapse*[*of* \top , *simplified*, *symmetric*]

lemma *monotone*:

shows *mono spec.singleton*

by (*simp add: monoI trace.natural.mono spec.singleton-le-conv*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.singleton.monotone*]

lemmas *mono* = *monoD*[*OF spec.singleton.monotone*]

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]

= *monotone2monotone*[*OF spec.singleton.monotone*, *simplified*]

lemma *simps*[*simp*]:

shows $\langle \natural\sigma \rangle = \langle \sigma \rangle$

and $\langle s, xs, v \rangle \leq \langle s, \text{trace.natural}' s xs, v \rangle$

and *snd* ' *set* $xs \subseteq \{s\} \implies \langle s, xs @ ys, v \rangle = \langle s, ys, v \rangle$

and *snd* ' *set* $ys \subseteq \{\text{trace.final}' s xs\} \implies \langle s, xs @ ys, v \rangle = \langle s, xs, v \rangle$

and *snd* ' *set* $xs \subseteq \{\text{snd } x\} \implies \langle s, x \# xs @ ys, v \rangle = \langle s, x \# ys, v \rangle$

and $\langle s, (a, s) \# xs, v \rangle = \langle s, xs, v \rangle$

by (*simp-all add: antisym spec.singleton-le-conv trace.stuttering.equiv.simps trace.natural.simps*)

lemma *Cons*: — self-applies, not usable by *simp*

assumes *snd* ' *set* $as \subseteq \{s'\}$

shows $\langle s, (a, s') \# as, v \rangle = \langle s, [(a, s')], v \rangle$

by (*simp add: asms spec.singleton.simps(4)[where $xs=[(a, s')]$ and $ys=as$, simplified]*)

lemmas *Sup-irreducible* = *iffD1*[*OF heyting.Sup-prime-Sup-irreducible-iff spec.singleton.Sup-prime*]

lemmas *sup-irreducible* = *Sup-irreducible-on-imp-sup-irreducible-on*[*OF spec.singleton.Sup-irreducible*, *simplified*]

lemmas *Sup-leE*[*elim*] = *Sup-prime-onE*[*OF spec.singleton.Sup-prime*, *simplified*]

lemmas *sup-le-conv*[*simp*] = *sup-irreducible-le-conv*[*OF spec.singleton.sup-irreducible*]

lemmas *Sup-le-conv*[*simp*] = *Sup-prime-on-conv*[*OF spec.singleton.Sup-prime*, *simplified*]

lemmas *compact-point* = *Sup-prime-is-compact*[*OF spec.singleton.Sup-prime*]

lemmas *compact*[*cont-intro*] = *compact-points-are-ccpo-compact*[*OF spec.singleton.compact-point*]

lemma *Inf*:

shows $\bigcap (\text{spec.singleton} \text{ ' } X) = \bigsqcup (\text{spec.singleton} \text{ ' } \{\sigma. \forall \sigma_1 \in X. \sigma \leq \natural\sigma_1\})$

by (*fastforce simp: le-INF-iff spec.singleton-le-conv*

dest: spec.singleton.mono

intro: spec.singleton.antisym)

lemmas *inf* = *spec.singleton.Inf*[**where** $X=\{\sigma_1, \sigma_2\}$, *simplified*] **for** $\sigma_1 \sigma_2$

lemma *less-eq-Some*[*simp*]:

shows $\langle s, xs, \text{Some } v \rangle \leq \langle \sigma \rangle$

$\longleftrightarrow \text{trace.term } \sigma = \text{Some } v \wedge \text{trace.init } \sigma = s \wedge \text{trace.natural}' s (\text{trace.rest } \sigma) = \text{trace.natural}' s xs$

by (*auto simp: spec.singleton-le-conv trace.natural-def*)

lemma *less-eq-None*:

shows [*iff*]: $\langle s, xs, \text{None} \rangle \leq \langle s, xs, v' \rangle$

by (*auto simp: spec.singleton-le-conv trace.natural-def trace.less-eq-None*)

lemma *map-cong*:

assumes $\bigwedge a. a \in \text{trace.aset } (\natural\sigma) \implies af a = af' a$

assumes $\bigwedge x. x \in \text{trace.sset } (\natural\sigma) \implies sf x = sf' x$

assumes $\bigwedge v. v \in \text{trace.vset } (\natural\sigma) \implies vf v = vf' v$

assumes $\natural\sigma = \natural\sigma'$

shows $\langle \text{trace.map } af sf vf \sigma \rangle = \langle \text{trace.map } af' sf' vf' \sigma' \rangle$

proof —

from *assms* **have** $\text{trace.map af sf vf } (\natural \sigma) \simeq_S \text{trace.map af' sf' vf' } (\natural \sigma')$
by (*simp del: trace.sset.natural trace.vset.natural cong: trace.t.map-cong*)
then show *?thesis*
by (*simp add: trace.natural.map-natural*)
qed

lemma *map-le*:

assumes $\langle \sigma \rangle \leq \langle \sigma' \rangle$
shows $\langle \text{trace.map af sf vf } \sigma \rangle \leq \langle \text{trace.map af sf vf } \sigma' \rangle$
using *assms* **by** (*simp add: spec.singleton-le-conv trace.natural.map-le*)

lemma *takeI*:

assumes $\langle \sigma \rangle \leq P$
shows $\langle \text{trace.take } i \sigma \rangle \leq P$
by (*meson assms order.trans spec.singleton.mono trace.less-eq-take*)

setup $\langle \text{Sign.parent-path} \rangle$

lemmas *assms-cong* = *order.assms-cong*[**where** $'a=('a, 's, 'v)$ *spec*]

lemmas *concl-cong* = *order.concl-cong*[**where** $'a=('a, 's, 'v)$ *spec*]

declare *spec.singleton.transfer*[*transfer-rule del*]

setup $\langle \text{Sign.parent-path} \rangle$

8.4 Maps

Lift *trace.map* to the $('a, 's, 'v)$ *spec* lattice via image and inverse image.

Note that the image may yield a set that is not stuttering closed (i.e., we need to close the obvious model-level definition of *spec.map* under stuttering) as arbitrary *sf* may introduce stuttering not present in *P*. In contrast the inverse image preserves stuttering. These issues are elided here through the use of *spec.singleton*.

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('b, 't, 'w) \text{ spec}$ **where**
 $\text{map af sf vf } P = \bigsqcup (\text{spec.singleton } \langle \text{trace.map af sf vf } \langle \sigma. \langle \sigma \rangle \leq P \rangle)$

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**
 $\text{invmap af sf vf } P = \bigsqcup (\text{spec.singleton } \langle \text{trace.map af sf vf } - \langle \sigma. \langle \sigma \rangle \leq P \rangle)$

abbreviation *amap* :: $('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('b, 's, 'v) \text{ spec}$ **where**

$\text{amap af} \equiv \text{spec.map af id id}$

abbreviation *ainvmap* :: $('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{ainvmap af} \equiv \text{spec.invmap af id id}$

abbreviation *smap* :: $('s \Rightarrow 't) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 't, 'v) \text{ spec}$ **where**

$\text{smap sf} \equiv \text{spec.map id sf id}$

abbreviation *sinvmap* :: $('s \Rightarrow 't) \Rightarrow ('a, 't, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{sinvmap sf} \equiv \text{spec.invmap id sf id}$

abbreviation *vmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'w) \text{ spec}$ **where** — aka *liftM*

$\text{vmap vf} \equiv \text{spec.map id id vf}$

abbreviation *vinvmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'w) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{vinvmap vf} \equiv \text{spec.invmap id id vf}$

interpretation *map-invmap*: *galois.complete-lattice-distributive-class*

spec.map af sf vf

$\text{spec.invmap af sf vf}$ **for** af sf vf

proof *standard*

show $\text{spec.map af sf vf } P \leq Q \iff P \leq \text{spec.invmap af sf vf } Q$ (**is** *?lhs* \iff *?rhs*) **for** $P Q$

proof(*rule iffI*)

show $?lhs \implies ?rhs$

by (*fastforce simp: spec.map-def spec.invmap-def intro: spec.singleton-le-extI*)

show $?rhs \implies ?lhs$

by (*fastforce simp: spec.map-def spec.invmap-def*

dest: order.trans[of - P] spec.singleton.map-le[where af=af and sf=sf and vf=vf])

qed

show $spec.invmap\ af\ sf\ vf\ (\bigsqcup X) \leq \bigsqcup (spec.invmap\ af\ sf\ vf\ 'X)$ **for** X

by (*fastforce simp: spec.invmap-def*)

qed

setup $\langle Sign.mandatory-path\ singleton \rangle$

lemma *map-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq spec.map\ af\ sf\ vf\ P \iff (\exists \sigma'. \langle \sigma' \rangle \leq P \wedge \langle \sigma \rangle \leq \langle trace.map\ af\ sf\ vf\ \sigma' \rangle)$

by (*simp add: spec.map-def*)

lemma *invmap-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq spec.invmap\ af\ sf\ vf\ P \iff \langle trace.map\ af\ sf\ vf\ \sigma \rangle \leq P$

by (*simp add: spec.invmap-def*) (*meson order.refl order.trans spec.singleton.map-le*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ map \rangle$

lemmas *bot = spec.map-invmap.lower-bot*

lemmas *monotone = spec.map-invmap.monotone-lower*

lemmas *mono = monotoneD[OF spec.map.monotone]*

lemmas *Sup = spec.map-invmap.lower-Sup*

lemmas *sup = spec.map-invmap.lower-sup*

lemmas *Inf-le = spec.map-invmap.lower-Inf-le* — Converse does not hold

lemmas *inf-le = spec.map-invmap.lower-inf-le* — Converse does not hold

lemmas *invmap-le = spec.map-invmap.lower-upper-contractive*

lemma *singleton*:

shows $spec.map\ af\ sf\ vf\ \langle \sigma \rangle = \langle trace.map\ af\ sf\ vf\ \sigma \rangle$

by (*auto simp: spec.map-def spec.eq-iff spec.singleton-le-conv intro: trace.natural.map-le*)

lemma *top*:

assumes *surj af*

assumes *surj sf*

assumes *surj vf*

shows $spec.map\ af\ sf\ vf\ \top = \top$

by (*rule antisym*)

(*auto simp: assms spec.singleton.top spec.map.Sup spec.map.singleton surj-f-inv-f*

intro: exI[where x=trace.map (inv af) (inv sf) (inv vf) σ for σ])

lemma *id*:

shows $spec.map\ id\ id\ id\ P = P$

and $spec.map\ (\lambda x. x)\ (\lambda x. x)\ (\lambda x. x)\ P = P$

by (*simp-all add: spec.map-def flip: id-def*)

lemma *comp*:

shows $spec.map\ af\ sf\ vf \circ spec.map\ ag\ sg\ vg = spec.map\ (af \circ ag)\ (sf \circ sg)\ (vf \circ vg)$ (**is** $?lhs = ?rhs$)

and $\text{spec.map } af \text{ } sf \text{ } vf \text{ } (\text{spec.map } ag \text{ } sg \text{ } vg \text{ } P) = \text{spec.map } (\lambda a. af \text{ } (ag \text{ } a)) (\lambda s. sf \text{ } (sg \text{ } s)) (\lambda v. vf \text{ } (vg \text{ } v)) P$ (**is** *?thesis1*)

proof –

have $?lhs \text{ } P = ?rhs \text{ } P$ **for** P

by (*rule spec.singleton.exhaust[where x=P]*)

(*simp add: spec.map.Sup spec.map.singleton image-image comp-def*)

then show $?lhs = ?rhs$ **and** *?thesis1* **by** (*simp-all add: comp-def*)

qed

lemmas $\text{map} = \text{spec.map.comp}$

lemma *inf-distr*:

shows $\text{spec.map } af \text{ } sf \text{ } vf \text{ } P \sqcap Q = \text{spec.map } af \text{ } sf \text{ } vf \text{ } (P \sqcap \text{spec.invmap } af \text{ } sf \text{ } vf \text{ } Q)$ (**is** $?lhs = ?rhs$)

and $Q \sqcap \text{spec.map } af \text{ } sf \text{ } vf \text{ } P = \text{spec.map } af \text{ } sf \text{ } vf \text{ } (\text{spec.invmap } af \text{ } sf \text{ } vf \text{ } Q \sqcap P)$ (**is** *?thesis1*)

proof –

show $?lhs = ?rhs$

proof(*rule antisym*)

obtain X **where** $Q: Q = \bigsqcup (\text{spec.singleton } ' X)$ **using** *spec.singleton.nchotomy[of Q]* **by** *blast*

then have $*$: $\langle \text{trace.take } j \text{ } (\natural \sigma_Q) \rangle \leq ?rhs$

if $\langle \sigma_P \rangle \leq P$

and $\sigma_Q \in X$

and $\natural (\text{trace.take } i \text{ } (\text{trace.map } af \text{ } sf \text{ } vf \text{ } \sigma_P)) = \text{trace.take } j \text{ } (\natural \sigma_Q)$

for $\sigma_P \sigma_Q i j$

using *that*

by (*auto simp: spec.singleton.le-conv spec.singleton-le-conv heyting.inf-SUP-distrib*

spec.map-def spec.singleton.inf trace.less-eq-take-def

trace.take.map spec.singleton.takeI trace.take.take trace.natural.take-natural

intro!: exI[where x=trace.take i σ_P] exI[where x=j])

with Q **show** $?lhs \leq ?rhs$

by (*subst spec.map-def*)

(*fastforce simp: heyting.inf-SUP-distrib spec.singleton.inf trace.less-eq-take-def*

elim: trace.take.naturalE(2))

show $?rhs \leq ?lhs$

by (*simp add: le-infI1 spec.map-invmap.galois spec.map-invmap.upper-lower-expansive*)

qed

then show *?thesis1*

by (*simp add: inf commute*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path smap} \rangle$

lemma *comp*:

shows $\text{spec.smap } sf \circ \text{spec.smap } sg = \text{spec.smap } (sf \circ sg)$

and $\text{spec.smap } sf \text{ } (\text{spec.smap } sg \text{ } P) = \text{spec.smap } (\lambda s. sf \text{ } (sg \text{ } s)) P$

by (*simp-all add: comp-def spec.map.comp id-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemmas $\text{bot} = \text{spec.map-invmap.upper-bot}$

lemmas $\text{top} = \text{spec.map-invmap.upper-top}$

lemmas $\text{monotone} = \text{spec.map-invmap.monotone-upper}$

lemmas $\text{mono} = \text{monotoneD}[OF \text{spec.invmap.monotone}]$

lemmas *Sup* = *spec.map-invmap.upper-Sup*

lemmas *sup* = *spec.map-invmap.upper-sup*

lemmas *Inf* = *spec.map-invmap.upper-Inf*

lemmas *inf* = *spec.map-invmap.upper-inf*

lemma *singleton*:

shows *spec.invmap af sf vf* $\langle \sigma \rangle = \bigsqcup \{ \text{spec.singleton } \sigma'. \langle \text{trace.map af sf vf } \sigma' \rangle \leq \langle \sigma \rangle \}$
by (*simp add: spec.invmap-def*)

lemma *id*:

shows *spec.invmap id id id* $P = P$
and *spec.invmap* $(\lambda x. x) (\lambda x. x) (\lambda x. x) P = P$
unfolding *id-def[symmetric]* **by** (*metis spec.map.id(1) spec.map-invmap.lower-upper-lower(2)*)⁺

lemma *comp*:

shows *spec.invmap af sf vf* (*spec.invmap ag sg vg* P) = *spec.invmap* $(\lambda x. ag (af x)) (\lambda s. sg (sf s)) (\lambda v. vg (vf v)) P$ (**is** *?lhs P = ?rhs P*)

and *spec.invmap af sf vf* \circ *spec.invmap ag sg vg* = *spec.invmap* $(ag \circ af) (sg \circ sf) (vg \circ vf)$ (**is** *?thesis1*)

proof –

show *?lhs P = ?rhs P* **for** P

by (*auto intro: spec.singleton.antisym spec.singleton-le-extI simp: spec.singleton.le-conv*)

then show *?thesis1*

by (*simp add: fun-eq-iff comp-def*)

qed

lemmas *invmap* = *spec.invmap.comp*

lemma *invmap-inf-distr-le*:

fixes *af* :: $'a \Rightarrow 'b$

fixes *sf* :: $'s \Rightarrow 't$

fixes *vf* :: $'v \Rightarrow 'w$

shows *spec.invmap af sf vf* $P \sqcap Q \leq \text{spec.invmap af sf vf } (P \sqcap \text{spec.map af sf vf } Q)$

and $Q \sqcap \text{spec.invmap af sf vf } P \leq \text{spec.invmap af sf vf } (\text{spec.map af sf vf } Q \sqcap P)$

by (*metis order.refl inf-mono spec.map-invmap.upper-inf spec.map-invmap.upper-lower-expansive*)⁺

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path amap} \rangle$

lemma *invmap-le*: — *af = id* in *spec.invmap*

shows *spec.amap af* (*spec.invmap id sf vf* P) $\leq \text{spec.invmap id sf vf } (\text{spec.amap af } P)$

proof –

have *spec.invmap id sf vf* $P \leq \text{spec.invmap af sf vf } (\text{spec.amap af } P)$ (**is** *?lhs \leq ?rhs*)

proof(*rule spec.singleton-le-extI*)

show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ

using that **by** (*simp add: spec.singleton.le-conv extI[where x=trace.map id sf vf σ] flip: id-def*)

qed

then show *?thesis*

by (*simp add: spec.map-invmap.galois spec.invmap.comp flip: id-def*)

qed

lemma *surj-invmap*: — *af = id* in *spec.invmap*

fixes $P :: ('a, 't, 'w) \text{spec}$

fixes *af* :: $'a \Rightarrow 'b$

fixes *sf* :: $'s \Rightarrow 't$

fixes *vf* :: $'v \Rightarrow 'w$

assumes *surj af*
shows $\text{spec.amap } af \ (\text{spec.invmap } id \ sf \ vf \ P) = \text{spec.invmap } id \ sf \ vf \ (\text{spec.amap } af \ P) \ (\text{is } ?lhs = ?rhs)$
proof(*rule antisym[OF spec.amap.invmap-le spec.singleton-le-extI]*)
have $1: \exists \sigma_3. \sigma_2 = \text{trace.map } af \ id \ id \ \sigma_3 \wedge \sigma_1 \simeq_S \sigma_3$
if $\text{trace.map } af \ id \ id \ \sigma_1 \simeq_S \sigma_2$
for $\sigma_1 :: ('a, 't, 'w) \text{ trace.t}$ **and** σ_2
proof –
have $** : \exists ys'. ys = \text{map } (\text{map-prod } af \ id) \ ys' \wedge \text{trace.natural}' \ s \ xs = \text{trace.natural}' \ s \ ys'$
if $\text{trace.natural}' \ s \ (\text{map } (\text{map-prod } af \ id) \ xs) = \text{trace.natural}' \ s \ ys$
for $s :: 't$ **and** $xs \ ys$
using *that*
proof(*induct ys arbitrary: s xs*)
case *Nil* **then show** *?case*
by (*fastforce simp: trace.natural'.eq-Nil-conv*)
next
case (*Cons y ys s xs*) **show** *?case*
proof(*cases snd y = s*)
case *True* **with** *Cons.prem*s **show** *?thesis*
by (*fastforce dest: Cons.hyps*
simp: iffD1[OF surj-iff ‹surj af›]
simp flip: id-def
intro: exI[where x=map-prod (inv af) id y # ys' for ys'])
next
case *False* **with** *Cons.prem*s **show** *?thesis*
by (*force dest!: Cons.hyps*
simp: trace.natural'.eq-Cons-conv trace.natural'.idle-prefix
map-eq-append-conv snd-image-map-prod
simp flip: id-def
intro: exI[where x=(a, s) # xs for a s xs])
qed
qed
from *that* **show** *?thesis*
by (*cases* σ_2) (*clarsimp simp: ** trace.natural-def trace.split-Ex*)
qed
have $2: \exists zs. xs = \text{map } (\text{map-prod } af \ id) \ zs \wedge \text{map } (\text{map-prod } id \ sf) \ zs = ys$
if $xs\text{-ys}: \text{map } (\text{map-prod } id \ sf) \ xs = \text{map } (\text{map-prod } af \ id) \ ys$
for $xs \ ys$
proof –
have $\exists zs. xs' = \text{map } (\text{map-prod } af \ id) \ zs \wedge \text{map } (\text{map-prod } id \ sf) \ zs = ys'$
if $\text{length } xs' = \text{length } ys'$
and *prefix* $xs' \ xs$
and *prefix* $ys' \ ys$
and $\text{map } (\text{map-prod } id \ sf) \ xs = \text{map } (\text{map-prod } af \ id) \ ys$ **for** $xs' \ ys'$
using *that*
proof(*induct xs' ys' rule: rev-induct2*)
case (*snoc x xs y ys*) **then show** *?case*
by (*cases* $x; \text{cases } y$)
(force simp: prefix-def simp flip: id-def intro: exI[where x=zs @ [(fst y, snd x)] for zs])
qed *simp*
from *this*[*OF map-eq-imp-length-eq[OF xs-ys] prefix-order.refl prefix-order.refl xs-ys*]
show *?thesis* .
qed
fix σ
assume $\langle \sigma \rangle \leq ?rhs$
then obtain σ_P **where** $\sigma_P: \langle \sigma_P \rangle \leq P \ \langle \text{trace.map } id \ sf \ vf \ \sigma \rangle \leq \langle \text{trace.map } af \ id \ id \ \sigma_P \rangle$
by (*clarsimp simp: spec.singleton.le-conv*)
then obtain $i \ \sigma_P'$ **where** $\sigma_P': \text{trace.map } id \ sf \ vf \ \sigma = \text{trace.map } af \ id \ id \ \sigma_P' \ \text{trace.take } i \ \sigma_P \simeq_S \sigma_P'$
by (*fastforce simp: spec.singleton-le-conv trace.less-eq-take-def trace.take.map*)

$dest: 1[OF\ sym]$
 $elim!: trace.take.naturalE)$
then obtain zs **where** $zs: trace.rest\ \sigma = map\ (map\text{-}prod\ af\ id)\ zs\ map\ (map\text{-}prod\ id\ sf)\ zs = trace.rest\ \sigma_{P'}$
by $(cases\ \sigma, cases\ \sigma_{P'})\ (clarsimp\ dest!: 2)$
from $\langle \langle \sigma_P \rangle \leq P \rangle\ \sigma_{P'}(1)\ \sigma_{P'}(2)[symmetric]\ zs$ **show** $\langle \sigma \rangle \leq ?lhs$
by $(cases\ \sigma, cases\ \sigma_{P'})$
 $(clarsimp\ intro!: exI[where\ x=trace.T\ (trace.init\ \sigma)\ zs\ (trace.term\ \sigma)])$
 $elim!: order.trans[rotated]$
 $simp: spec.singleton.le-conv\ spec.singleton-le-conv\ trace.natural.mono)$

qed

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle Sign.parent\text{-}path \rangle$

8.5 The idle process

As observed by Abadi and Plotkin (1991), many laws require the processes involved to accept all initial states (see, for instance, §8.8). We call the minimal such process *spec.idle*. It is also the lower bound on specification by transition relation (§8.10).

setup $\langle Sign.mandatory\text{-}path\ spec \rangle$

definition $idle :: ('a, 's, 'v)\ spec$ **where**
 $idle = (\sqcup s. \langle s, [], None \rangle)$

named-theorems $idle\text{-}le$ $\langle rules\ for\ \langle spec.idle \leq const\ \dots \rangle \rangle$

setup $\langle Sign.mandatory\text{-}path\ singleton \rangle$

lemma $idle\text{-}le\text{-}conv[spec.singleton.le-conv]:$

shows $\langle \sigma \rangle \leq spec.idle \iff trace.steps\ \sigma = \{\} \wedge trace.term\ \sigma = None$
by $(auto\ simp: spec.idle\text{-}def\ spec.singleton-le-conv\ trace.natural.simps\ trace.natural'.eq\text{-}Nil\text{-}conv\ trace.less\text{-}eq\text{-}None)$

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle Sign.mandatory\text{-}path\ idle \rangle$

lemma $minimal\text{-}le:$

shows $\langle s, [], None \rangle \leq spec.idle$
by $(simp\ add: spec.singleton.idle\text{-}le\text{-}conv)$

lemma $map\text{-}le[spec.idle\text{-}le]:$

assumes $spec.idle \leq P$
assumes $surj\ sf$
shows $spec.idle \leq spec.map\ af\ sf\ vf\ P$
by $(strengthen\ ord\text{-}to\text{-}strengthen(2)[OF\ assms(1)])$
 $(use\ \langle surj\ sf \rangle\ in\ \langle auto\ simp: spec.idle\text{-}def\ spec.map.Sup\ spec.map.singleton\ image\text{-}image\ spec.singleton-le-conv\ trace.natural.simps\ trace.less\text{-}eq\text{-}None \rangle)$

lemma $invmap\text{-}le:$

assumes $spec.idle \leq P$
shows $spec.idle \leq spec.invmap\ af\ sf\ vf\ P$
by $(strengthen\ ord\text{-}to\text{-}strengthen(2)[OF\ assms])$
 $(auto\ simp: spec.idle\text{-}def\ spec.map.Sup\ spec.map.singleton\ simp\ flip: spec.map\text{-}invmap.galois)$

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle \text{Sign.mandatory-path map-invmap} \rangle$

lemma *cl-alt-def*:

shows $\text{spec.map-invmap.cl} \text{ - - - af sf vf } P$
 $= \sqcup \{ \langle \sigma \rangle \mid \sigma \sigma'. \langle \sigma' \rangle \leq P \wedge \langle \text{trace.map af sf vf } \sigma \rangle \leq \langle \text{trace.map af sf vf } \sigma' \rangle \}$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

by (*rule spec.singleton.exhaust[of P]*)

(*fastforce simp: spec.map-invmap.cl-def*

spec.map.Sup spec.map.singleton spec.invmap.Sup spec.invmap.singleton

intro: spec.singleton.mono)

show $?rhs \leq ?lhs$

by (*clarsimp simp: spec.map-invmap.cl-def simp flip: spec.map.singleton*)

(*simp add: order.trans[OF - spec.map.mono] flip: spec.map-invmap.galois*)

qed

lemma *cl-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.map-invmap.cl} \text{ - - - af sf vf } P \iff \langle \text{trace.map af sf vf } \sigma \rangle \leq \text{spec.map af sf vf } P$

by (*simp add: spec.map-invmap.cl-def spec.singleton.invmap-le-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.6 Actions

Our primitive actions are arbitrary relations on the state, labelled by the agent performing the state transition and a value to return.

For refinement purposes we need $\text{idle} \leq \text{action } a \ F$; see §12.1.1.

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition *action* $:: ('v \times 'a \times 's \times 's) \text{ set} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{action } F = (\sqcup (v, a, s, s') \in F. \langle s, [(a, s')], \text{Some } v \rangle) \sqcup \text{spec.idle}$

definition *guard* $:: ('s \Rightarrow \text{bool}) \Rightarrow ('a, 's, \text{unit}) \text{ spec}$ **where**

$\text{guard } g = \text{spec.action } (\{()\} \times \text{UNIV} \times \text{Diag } g)$

definition *return* $:: 'v \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{return } v = \text{spec.action } (\{v\} \times \text{UNIV} \times \text{Id})$

abbreviation (*input*) *read* $:: ('s \Rightarrow 'v \text{ option}) \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$\text{read } f \equiv \text{spec.action } \{(v, a, s, s) \mid a \ s \ v. \ f \ s = \text{Some } v\}$

abbreviation (*input*) *write* $:: 'a \Rightarrow ('s \Rightarrow 's) \Rightarrow ('a, 's, \text{unit}) \text{ spec}$ **where**

$\text{write } a \ f \equiv \text{spec.action } \{((), a, s, f \ s) \mid s. \ \text{True}\}$

lemma *action-le[case-names idle step]*:

assumes $\text{spec.idle} \leq P$

assumes $\bigwedge v \ a \ s \ s'. (v, a, s, s') \in F \implies \langle s, [(a, s')], \text{Some } v \rangle \leq P$

shows $\text{spec.action } F \leq P$

by (*simp add: assms spec.action-def split-def*)

setup $\langle \text{Sign.mandatory-path idle} \rangle$

lemma *action-le[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.action } F$

by (*simp add: spec.action-def*)

lemma *guard-le[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.guard } g$

by (*simp add: spec.guard-def spec.idle-le*)

lemma *return-le[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.return } v$

by (*simp add: spec.return-def spec.idle-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path map} \rangle$

lemma *action-le*:

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$

shows $\text{spec.map } af \text{ } sf \text{ } vf \text{ } (\text{spec.action } F) \leq \text{spec.action } (\text{map-prod } vf \text{ } (\text{map-prod } af \text{ } (\text{map-prod } sf \text{ } sf))) \text{ } 'F$

by (*force simp: spec.action-def spec.idle-def spec.map.Sup spec.map.sup spec.map.singleton SUP-le-iff*)

lemma *action*:

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$

shows $\text{spec.map } af \text{ } sf \text{ } vf \text{ } (\text{spec.action } F) \sqcup \text{spec.idle}$

$= \text{spec.action } (\text{map-prod } vf \text{ } (\text{map-prod } af \text{ } (\text{map-prod } sf \text{ } sf))) \text{ } 'F \text{ } (\text{is } ?lhs = ?rhs)$

proof(*rule antisym*)

show $?lhs \leq ?rhs$

by (*simp add: spec.idle-le spec.map.action-le*)

show $?rhs \leq ?lhs$

by (*force simp: spec.action-def spec.idle-def spec.map.sup spec.map.singleton spec.map.Sup*)

qed

lemma *surj-sf-action*:

assumes *surj sf*

shows $\text{spec.map } af \text{ } sf \text{ } vf \text{ } (\text{spec.action } F) = \text{spec.action } (\text{map-prod } vf \text{ } (\text{map-prod } af \text{ } (\text{map-prod } sf \text{ } sf))) \text{ } 'F$

by (*simp add: assms sup.absorb1 spec.idle-le flip: spec.map.action*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma *empty*:

shows $\text{spec.action } \{\} = \text{spec.idle}$

by (*simp add: spec.action-def*)

lemma *idleI*:

assumes $\text{snd } ' \text{ set } xs \subseteq \{s\}$

shows $\langle s, xs, \text{None} \rangle \leq \text{spec.action } F$

using *assms* **by** (*simp add: spec.action-def spec.singleton.le-conv*)

lemma *stepI*:

assumes $(v, a, s, s') \in F$

assumes $\forall v''. w = \text{Some } v'' \longrightarrow v'' = v$

shows $\langle s, [(a, s')], w \rangle \leq \text{spec.action } F$

using *assms* **by** (*cases w; force simp: spec.action-def spec.singleton.mono trace.less-eq-None*)

lemma *stutterI*:

assumes $(v, a, s, s) \in F$

shows $\langle s, [], \text{Some } v \rangle \leq \text{spec.action } F$

by (*fastforce simp: spec.singleton-le-conv trace.natural.simps*
intro: order.trans[OF - spec.action.stepI[OF assms]])

lemma *stutter-stepI*:

assumes $(v, a, s, s) \in F$

shows $\langle s, [(b, s)], \text{Some } v \rangle \leq \text{spec.action } F$

by (*fastforce simp: spec.singleton-le-conv trace.natural.simps*
intro: order.trans[OF - spec.action.stepI[OF assms]])

lemma *stutter-stepsI*:

assumes $(v, a, s, s) \in F$

assumes *snd* ' set $xs \subseteq \{s\}$

shows $\langle s, xs, \text{Some } v \rangle \leq \text{spec.action } F$

by (*simp add: assms trace.natural'.eq-Nil-conv order.trans[OF - spec.action.stutterI[OF assms(1)]]*)

lemma *monotone*:

shows *mono spec.action*

by (*force simp: spec.action-def intro: monoI*)

lemmas *strengthen[strg] = st-monotone[OF spec.action.monotone]*

lemmas *mono = monotoneD[OF spec.action.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*
= monotone2monotone[OF spec.action.monotone, simplified]

lemma *Sup*:

shows *spec.action* $(\bigcup X) = (\bigsqcup_{F \in X} \text{spec.action } F) \sqcup \text{spec.idle}$

by (*force simp: spec.eq-iff spec.action-def*)

lemma

shows *SUP: spec.action* $(\bigcup_{x \in X} F x) = (\bigsqcup_{x \in X} \text{spec.action } (F x)) \sqcup \text{spec.idle}$

and *SUP-not-empty: $X \neq \{\}$ $\implies \text{spec.action } (\bigcup_{x \in X} F x) = (\bigsqcup_{x \in X} \text{spec.action } (F x))$*

by (*auto simp: spec.action.Sup image-image sup.absorb1 SUPI spec.idle-le simp flip: ex-in-conv*)

lemma *sup*:

shows *spec.action* $(F \cup G) = \text{spec.action } F \sqcup \text{spec.action } G$

using *spec.action.Sup[where $X = \{F, G\}$]* **by** (*simp add: sup-absorb1 le-supI1 spec.idle-le*)

lemma *Inf-le*:

shows *spec.action* $(\bigcap Fs) \leq \prod (\text{spec.action } ' Fs)$

by (*simp add: spec.action-def ac-simps SUP-le-iff SUP-upper le-INF-iff le-supI2*)

lemma *inf-le*:

shows *spec.action* $(F \cap G) \leq \text{spec.action } F \sqcap \text{spec.action } G$

using *spec.action.Inf-le[where $Fs = \{F, G\}$]* **by** *simp*

lemma *stutter-agents-le*:

assumes $\llbracket A \neq \{\}; r \neq \{\} \rrbracket \implies B \neq \{\}$

assumes $r \subseteq \text{Id}$

shows *spec.action* $(\{v\} \times A \times r) \leq \text{spec.action } (\{v\} \times B \times r)$

using *assms*

by (*subst spec.action-def*) (*fastforce simp: spec.idle-le intro!: spec.action.stutter-stepI*)

lemma *read-agents*:

assumes $A \neq \{\}$

assumes $B \neq \{\}$

assumes $r \subseteq \text{Id}$

shows *spec.action* $(\{v\} \times A \times r) = \text{spec.action } (\{v\} \times B \times r)$

by (*rule antisym[OF spec.action.stutter-agents-le spec.action.stutter-agents-le]; rule assms*)

lemma *invmap-le*: — A typical refinement

fixes *af* :: 'a ⇒ 'b

fixes *sf* :: 's ⇒ 't

fixes *vf* :: 'v ⇒ 'w

shows *spec.action* (map-prod *vf* (map-prod *af* (map-prod *sf* *sf*)) - ' *F*) ≤ *spec.invmap* *af* *sf* *vf* (*spec.action* *F*)

by (*meson order.trans image-vimage-subset spec.action.mono spec.map.action-le spec.map-invmap.galois*)

setup ⟨*Sign.parent-path*⟩

setup ⟨*Sign.mandatory-path singleton*⟩

lemma *action-le-conv*:

shows ⟨ σ ⟩ ≤ *spec.action* *F*

⟷ (trace.steps σ = {} ∧ case-option True (λ*v*. ∃ *a*. (*v*, *a*, trace.init σ , trace.init σ) ∈ *F*) (trace.term σ))

∨ (∃ *x* ∈ *F*. trace.steps σ = {snd *x*} ∧ case-option True ((=) (fst *x*)) (trace.term σ)) (is ?lhs ⟷ ?rhs)

proof(rule *iffI*)

show ?lhs ⇒ ?rhs

unfolding *spec.action-def spec.singleton.sup-le-conv*

proof(induct rule: *disjE*[consumes 1, case-names step idle])

case *step*

then obtain *v a s s'* **where** *: $\sigma \leq \natural(\text{trace.T } s \ [(a, s') \ (\text{Some } v)])$ **and** *F*: (*v*, *a*, *s*, *s'*) ∈ *F*

by (*clarsimp simp: spec.singleton-le-conv*)

from * **show** ?case

proof(induct rule: *trace.less-eqE*)

case *prefix* **with** *F* **show** ?case

by (*clarsimp simp add: trace.natural'.eq-Nil-conv prefix-Cons split: if-splits*)

(*force simp: trace.steps'-alt-def*)

next

case (*maximal v*) **with** *F* **show** ?case

by (*clarsimp simp: trace.natural.trace-conv trace.natural'.eq-Nil-conv*

trace.natural'.eq-Cons-conv trace.steps'.append split: if-splits;

force)

qed

qed (*simp add: spec.singleton.le-conv*)

show ?rhs ⇒ ?lhs

by (*cases* σ)

(*auto simp: trace.steps'.step-conv*

intro: spec.action.idleI spec.action.stutter-stepsI spec.action.stepI

elim!: order.trans[OF eq-refl[OF spec.singleton.Cons]]

split: option.split-asm)

qed

lemma *action-Some-leE*:

assumes ⟨ σ ⟩ ≤ *spec.action* *F*

assumes trace.term σ = Some *v*

obtains *x*

where *x* ∈ *F*

and trace.init σ = fst (snd (snd *x*))

and trace.final σ = snd (snd (snd *x*))

and trace.steps $\sigma \subseteq \{\text{snd } x\}$

and *v* = fst *x*

using *assms* **by** (*auto simp: spec.singleton.action-le-conv trace.steps'.step-conv trace.steps'.append*)

lemma *action-not-idle-leE*:

assumes ⟨ σ ⟩ ≤ *spec.action* *F*

assumes $\sigma \neq \text{trace.T } (\text{trace.init } \sigma) \ [] \text{ None}$

obtains *x*

where *x* ∈ *F*

and $trace.init\ \sigma = fst\ (snd\ (snd\ x))$
and $trace.final\ \sigma = snd\ (snd\ (snd\ x))$
and $trace.steps\ \sigma \subseteq \{snd\ x\}$
and $case-option\ True\ ((=)\ (fst\ x))\ (trace.term\ \sigma)$
using $assms$
by $(cases\ \sigma)$
 $(auto\ 0\ 0\ simp: spec.singleton.action-le-conv\ trace.natural.idle\ option.case-eq-if$
 $trace.steps'.step-conv\ trace.steps'.append)$

lemma $action-not-idle-le-splitE$:

assumes $\langle\sigma\rangle \leq spec.action\ F$
assumes $\not\vdash\sigma \neq trace.T\ (trace.init\ \sigma)\ []\ None$
obtains $(return)\ v\ a$
where $(v, a, trace.init\ \sigma, trace.init\ \sigma) \in F$
and $trace.steps\ \sigma = \{\}$
and $trace.term\ \sigma = Some\ v$
 $| (step)\ v\ a\ ys\ zs$
where $(v, a, trace.init\ \sigma, trace.final\ \sigma) \in F$
and $trace.init\ \sigma \neq trace.final\ \sigma$
and $snd\ 'set\ ys \subseteq \{trace.init\ \sigma\}$
and $snd\ 'set\ zs \subseteq \{trace.final\ \sigma\}$
and $trace.rest\ \sigma = ys\ @\ [(a, trace.final\ \sigma)]\ @\ zs$
and $case-option\ True\ ((=)\ v)\ (trace.term\ \sigma)$

using $assms$
by $(cases\ \sigma)$
 $(auto\ 0\ 0\ simp: spec.singleton.action-le-conv\ trace.natural.idle\ option.case-eq-if$
 $trace.steps'.step-conv\ trace.steps'.append$
 $cong: if-cong)$

lemma $guard-le-conv[spec.singleton.le-conv]$:

shows $\langle\sigma\rangle \leq spec.guard\ g \longleftrightarrow trace.steps\ \sigma = \{\} \wedge (case-option\ True\ \langle g\ (trace.init\ \sigma)\ \rangle\ (trace.term\ \sigma))$
by $(fastforce\ simp: spec.guard-def\ spec.singleton.action-le-conv\ trace.steps'.step-conv$
 $split: option.split)$

lemma $return-le-conv[spec.singleton.le-conv]$:

shows $\langle\sigma\rangle \leq spec.return\ v$
 $\longleftrightarrow trace.steps\ \sigma = \{\} \wedge (case-option\ True\ ((=)\ v)\ (trace.term\ \sigma))$
by $(fastforce\ simp: spec.return-def\ spec.singleton.action-le-conv\ trace.steps'.step-conv$
 $split: option.split)$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ action \rangle$

lemma $mono-stronger$:

assumes $\bigwedge v\ a\ s\ s'. [(v, a, s, s') \in F; s \neq s'] \implies (v, a, s, s') \in F'$
assumes $\bigwedge v\ a\ s. (v, a, s, s) \in F \implies \exists a'. (v, a', s, s) \in F'$
shows $spec.action\ F \leq spec.action\ F'$
proof $(induct\ rule: spec.action-le[OF\ spec.idle.action-le,\ case-names\ step])$
case $(step\ v\ a\ s\ s')$ **then show** $?case$
by $(cases\ s = s')$
 $(auto\ dest: assms\ intro: spec.action.stutterI\ spec.action.stepI)$

qed

lemma $cong$:

assumes $\bigwedge v\ a\ s\ s'. s \neq s' \implies (v, a, s, s') \in F \longleftrightarrow (v, a, s, s') \in F'$
assumes $\bigwedge v\ a\ s. (v, a, s, s) \in F \implies \exists a'. (v, a', s, s) \in F'$
assumes $\bigwedge v\ a\ s. (v, a, s, s) \in F' \implies \exists a'. (v, a', s, s) \in F$

shows $\text{spec.action } F = \text{spec.action } F'$
using *assms* **by** (*blast intro!*: *spec.antisym spec.action.mono-stronger*)

lemma *le-actionD*:

assumes $\text{spec.action } F \leq \text{spec.action } F'$
shows $\llbracket (v, a, s, s') \in F; s \neq s' \rrbracket \implies (v, a, s, s') \in F'$
and $(v, a, s, s) \in F \implies \exists a'. (v, a', s, s) \in F'$

proof –

fix $v a s s'$
show $(v, a, s, s') \in F'$ **if** $(v, a, s, s') \in F$ **and** $s \neq s'$
using *iffD1[OF spec.singleton-le-ext-conv assms]* **that**
by (*fastforce simp: spec.singleton.action-le-conv*
dest: spec[where x=trace.T s [(a, s')] (Some v)])
show $\exists a'. (v, a', s, s) \in F'$ **if** $(v, a, s, s) \in F$
using *iffD1[OF spec.singleton-le-ext-conv assms]* **that**
by (*fastforce simp: spec.singleton.action-le-conv*
dest: spec[where x=trace.T s [] (Some v)])

qed

lemma *eq-action-conv*:

shows $\text{spec.action } F = \text{spec.action } F'$
 $\longleftrightarrow (\forall v a s s'. s \neq s' \longrightarrow (v, a, s, s') \in F \longleftrightarrow (v, a, s, s') \in F')$
 $\wedge (\forall v a s. (v, a, s, s) \in F \longrightarrow (\exists a'. (v, a', s, s) \in F'))$
 $\wedge (\forall v a s. (v, a, s, s) \in F' \longrightarrow (\exists a'. (v, a', s, s) \in F))$

by (*rule iffI,metis order.refl spec.action.le-actionD,blast intro!: spec.action.cong*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *return-alt-def*:

assumes $A \neq \{\}$
shows $\text{spec.return } v = \text{spec.action } (\{v\} \times A \times \text{Id})$

unfolding *spec.return-def* **using** *assms* **by** (*blast intro: spec.action.cong*)

setup $\langle \text{Sign.mandatory-path return} \rangle$

lemma *cong*:

assumes $\bigwedge v a s s'. (v, a, s, s') \in F \implies s' = s$
assumes $\bigwedge v s. v \in \text{fst } ' F \implies \exists a. (v, a, s, s) \in F$
shows $\text{spec.action } F = \bigsqcup (\text{spec.return } ' \text{fst } ' F) \sqcup \text{spec.idle}$

by (*simp add: spec.return-def image-image flip: spec.action.SUP*)
(rule spec.action.cong; auto intro: rev-bexI dest: assms(1) intro: assms(2))

lemma *action-le*:

assumes $\text{Id} \subseteq \text{snd } ' \text{snd } ' F$
shows $\text{spec.return } () \leq \text{spec.action } F$

unfolding *spec.return-def*

proof(*induct rule: spec.action-le*)

case (*step v a s s'*) **with** *subsetD[OF assms, where c=(s, s)]* **show** *?case*
by (*force intro: spec.action.stutterI*)

qed (*simp add: spec.idle-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path guard} \rangle$

lemma *alt-def*:

assumes $A \neq \{\}$
shows $\text{spec.guard } g = \text{spec.action } (\{()\} \times A \times \text{Diag } g)$

unfolding *spec.guard-def using assms by (fastforce simp: intro: spec.action.cong)*

lemma *bot:*

shows *spec.guard* $\perp = \text{spec.idle}$
and *spec.guard* $\langle \text{False} \rangle = \text{spec.idle}$

by (*simp-all add: spec.guard-def spec.action.empty*)

lemma *top:*

shows *spec.guard* $\top = \text{spec.return } ()$
and *spec.guard* $\langle \text{True} \rangle = \text{spec.return } ()$

by (*simp-all add: spec.guard-def spec.return-def flip: Id-def*)

lemma *monotone:*

shows *mono spec.guard*

proof(*rule monotoneI*)

show *spec.guard* $g \leq \text{spec.guard } g'$ **if** $g \leq g'$ **for** $g g' :: 's \text{ pred}$

unfolding *spec.guard-def by (strengthen ord-to-strengthen(1)[OF $\langle g \leq g' \rangle$]) simp*
qed

lemmas *strengthen[strg] = st-monotone[OF spec.guard.monotone]*

lemmas *mono = monotoneD[OF spec.guard.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF spec.guard.monotone, simplified]*

lemma *Sup:*

shows *spec.guard* $(\sqcup X) = \sqcup(\text{spec.guard } \langle X \rangle \sqcup \text{spec.idle}$

by (*auto simp: spec.guard-def Diag-Sup*

simp flip: spec.action.Sup[where $X = (\lambda x. \{()\} \times \text{UNIV} \times \text{Diag } x) \langle X \rangle$, simplified image-image]

intro: arg-cong[where $f = \text{spec.action}$])

lemma *sup:*

shows *spec.guard* $(g \sqcup h) = \text{spec.guard } g \sqcup \text{spec.guard } h$

by (*simp add: spec.guard.Sup[where $X = \{g, h\}$ for g, h , simplified]*

ac-simps sup-absorb2 le-supI2 spec.idle-le)

lemma *return-le:*

shows *spec.guard* $g \leq \text{spec.return } ()$

by (*simp add: spec.guard-def spec.return-def Sigma-mono spec.action.mono*)

lemma *guard-less: — Non-triviality*

assumes $g < g'$

shows *spec.guard* $g < \text{spec.guard } g'$

proof(*rule le-neq-trans*)

show *spec.guard* $g \leq \text{spec.guard } g'$

by (*strengthen ord-to-strengthen(1)[OF order-less-imp-le[OF assms]]*) *simp*

from *assms obtain* s **where** $g' s \neg g s$ **by** (*metis leD predicate1I*)

from $\langle \neg g s \rangle$ **have** $\neg \langle s, [], \text{Some } () \rangle \leq \text{spec.guard } g$

by (*clarsimp simp: spec.guard-def spec.action-def*

spec.singleton-le-conv spec.singleton.le-conv trace.natural.simps)

moreover

from $\langle g' s \rangle$ **have** $\langle s, [], \text{Some } () \rangle \leq \text{spec.guard } g'$

by (*simp add: spec.guard-def spec.action.stutterI*)

ultimately show *spec.guard* $g \neq \text{spec.guard } g'$ **by** *metis*

qed

lemma *cong:*

assumes $\bigwedge v a s s'. (v, a, s, s') \in F \implies s' = s$

shows *spec.action* $F = \text{spec.guard } (\lambda s. s \in \text{fst } \langle \text{snd } \langle \text{snd } \langle F \rangle \rangle \rangle)$ (**is** *?lhs = ?rhs*)

```

proof(rule antisym)
  show ?lhs ≤ ?rhs
  by (force simp: spec.guard-def intro: spec.action.mono dest: assms)
  show ?rhs ≤ ?lhs
  unfolding spec.guard-def
  by (rule spec.action-le;
    clarsimp simp: spec.idle-le; blast intro: spec.action.stutterI dest: assms)
qed

```

```

lemma action-le:
  assumes  $Diag\ g \subseteq snd\ 's\nd\ 'F$ 
  shows  $spec.guard\ g \leq spec.action\ F$ 
unfolding spec.guard-def
proof(induct rule: spec.action-le)
  case (step v a s s') with subsetD[OF assms, where c=(s, s)] show ?case
  by (force intro: spec.action.stutterI)
qed (simp add: spec.idle-le)

```

```

setup ‹Sign.parent-path›

```

```

setup ‹Sign.parent-path›

```

8.7 Operations on return values

For various purposes, including defining a history-respecting sequential composition (bind, see §8.8), we use a Galois pair of operations that saturate or eradicate return values.

```

setup ‹Sign.mandatory-path spec›

```

```

setup ‹Sign.mandatory-path term›

```

```

definition none :: ('a, 's, 'v) spec ⇒ ('a, 's, 'w) spec where
  none P =  $\bigsqcup \{ \langle s, xs, None \rangle \mid s\ xs\ v.\ \langle s, xs, v \rangle \leq P \}$ 

```

```

definition all :: ('a, 's, 'v) spec ⇒ ('a, 's, 'w) spec where
  all P =  $\bigsqcup \{ \langle s, xs, v \rangle \mid s\ xs\ v.\ \langle s, xs, None \rangle \leq P \}$ 

```

```

setup ‹Sign.parent-path›

```

```

interpretation term: galois.complete-lattice-distributive-class spec.term.none spec.term.all

```

```

proof standard

```

```

  show  $spec.term.none\ P \leq Q \iff P \leq spec.term.all\ Q$  (is ?lhs  $\iff$  ?rhs)

```

```

  for P :: ('a, 'b, 'c) spec

```

```

  and Q :: ('a, 'b, 'f) spec

```

```

proof(rule iffI)

```

```

  show ?lhs  $\implies$  ?rhs

```

```

  by (fastforce simp: spec.term.none-def spec.term.all-def trace.split-all
    intro: spec.singleton-le-extI)

```

```

  show ?rhs  $\implies$  ?lhs

```

```

  by (fastforce simp: spec.term.none-def spec.term.all-def
    spec.singleton-le-conv trace.natural-def trace.less-eq-None
    elim: trace.less-eqE order.trans[rotated]
    dest: order.trans[of - P])

```

```

qed

```

```

  show  $spec.term.all\ (\bigsqcup\ X) \leq \bigsqcup\ (spec.term.all\ 'X)$  for X :: ('a, 'b, 'f) spec set

```

```

  by (auto 0 5 simp: spec.term.all-def)

```

```

qed

```

```

setup ‹Sign.mandatory-path singleton.term›

```

lemma *none-le-conv*[*spec.singleton.le-conv*]:

shows $\langle \sigma \rangle \leq \text{spec.term.none } P \longleftrightarrow \text{trace.term } \sigma = \text{None} \wedge \langle \text{trace.init } \sigma, \text{trace.rest } \sigma, \text{None} \rangle \leq P$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof(*rule iffI*)

show $?lhs \implies ?rhs$

by (*fastforce simp: spec.term.none-def trace.natural-def spec.singleton-le-conv trace.less-eq-None*
intro: order.trans[rotated])

show $?rhs \implies ?lhs$

by (*cases* σ) (*fastforce simp: spec.term.none-def*)

qed

lemma *all-le-conv*[*spec.singleton.le-conv*]:

shows $\langle \sigma \rangle \leq \text{spec.term.all } P \longleftrightarrow (\exists w. \langle \text{trace.init } \sigma, \text{trace.rest } \sigma, w \rangle \leq P)$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof(*rule iffI*)

show $?lhs \implies ?rhs$

by (*cases* σ) (*fastforce simp: spec.term.none-def simp flip: spec.term.galois*)

show $?rhs \implies ?lhs$

by (*cases* σ) (*fastforce simp: spec.term.all-def intro: order.trans[rotated]*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

lemma *singleton*:

shows $\text{spec.term.none } \langle \sigma \rangle = \langle \text{trace.init } \sigma, \text{trace.rest } \sigma, \text{None} \rangle$

by (*force simp: spec.eq-iff spec.term.galois spec.singleton.le-conv spec.singleton.mono trace.less-eq-None*)

lemmas $\text{bot}[simp] = \text{spec.term.lower-bot}$

lemmas $\text{monotone} = \text{spec.term.monotone-lower}$

lemmas $\text{mono} = \text{monotoneD}[OF \text{spec.term.none.monotone}]$

lemmas $\text{Sup} = \text{spec.term.lower-Sup}$

lemmas $\text{sup} = \text{spec.term.lower-sup}$

lemmas $\text{Inf-le} = \text{spec.term.lower-Inf-le}$

lemma *Inf-not-empty*:

assumes $X \neq \{\}$

shows $\text{spec.term.none } (\bigsqcap X) = (\bigsqcap_{x \in X. \text{spec.term.none } x)$

by (*rule antisym[OF spec.term.lower-Inf-le]*)

(*use assms in* $\langle \text{auto intro: spec.singleton-le-extI simp: spec.singleton.term.none-le-conv le-Inf-iff} \rangle$)

lemma *inf*:

shows $\text{spec.term.none } (P \sqcap Q) = \text{spec.term.none } P \sqcap \text{spec.term.none } Q$

and $\text{spec.term.none } (Q \sqcap P) = \text{spec.term.none } Q \sqcap \text{spec.term.none } P$

using $\text{spec.term.none.Inf-not-empty}[\text{where } X = \{P, Q\}]$ **by** (*simp-all add: ac-simps*)

lemma *inf-unit*:

fixes $P Q :: (-, -, \text{unit}) \text{spec}$

shows $\text{spec.term.none } (P \sqcap Q) = \text{spec.term.none } P \sqcap Q$ (**is** $?thesis1 P Q$)

and $\text{spec.term.none } (P \sqcap Q) = P \sqcap \text{spec.term.none } Q$ (**is** $?thesis2$)

proof –

show $*$: $?thesis1 P Q$ **for** $P Q$

by (*rule spec.singleton.antisym; metis le-inf-iff spec.singleton.term.none-le-conv trace.t.collapse*)
from $*[\text{where } P=Q \text{ and } Q=P]$ **show** *?thesis2*
by (*simp add: ac-simps*)
qed

lemma *idempotent[simp]*:
shows *spec.term.none (spec.term.none P) = spec.term.none P*
by (*rule spec.singleton.exhaust[of P]*)
(simp add: spec.term.none.Sup spec.term.none.singleton image-image)

lemma *contractive[iff]*:
shows *spec.term.none P ≤ P*
by (*rule spec.singleton-le-extI (simp add: spec.singleton-le-conv trace.split-all)*)

lemma *map-gen*:
fixes *vf :: 'v ⇒ 'w*
fixes *vf' :: 'a ⇒ 'b* — arbitrary type
shows *spec.term.none (spec.map af sf vf P) = spec.map af sf vf' (spec.term.none P) (is ?lhs = ?rhs)*
by (*fastforce simp: spec.map-def spec.eq-iff image-image trace.split-all trace.split-Ex*
spec.term.none.Sup spec.term.none.singleton spec.singleton.term.none-le-conv
elim: order.trans[rotated])

lemmas *map = spec.term.none.map-gen[where vf'=id]* — *simp-friendly*

lemma *invmap-gen*:
fixes *vf :: 'v ⇒ 'w*
fixes *vf' :: 'a ⇒ 'b* — arbitrary type
shows *spec.term.none (spec.invmap af sf vf P) = spec.invmap af sf vf' (spec.term.none P) (is ?lhs = ?rhs)*
proof(*rule antisym*)
show *?lhs ≤ ?rhs*
by (*simp add: spec.map-invmap.lower-upper-contractive spec.term.none.mono*
flip: spec.map-invmap.galois spec.term.none.map-gen[where vf=vf'])
show *?rhs ≤ ?lhs*
by (*rule spec.singleton-le-extI*)
(clarsimp simp: spec.singleton.invmap-le-conv spec.singleton.term.none-le-conv)
qed

lemmas *invmap = spec.term.none.invmap-gen[where vf'=id]* — *simp-friendly*

lemma *idle*:
shows *spec.term.none spec.idle = spec.idle*
by (*simp add: spec.idle-def spec.term.none.Sup spec.term.none.singleton image-image*)

lemma *return*:
shows *spec.term.none (spec.return v) = spec.idle*
by (*auto simp: spec.eq-iff spec.return-def spec.action-def spec.term.none.idle spec.singleton.idle-le-conv*
spec.term.none.sup spec.term.none.Sup spec.term.none.singleton)

lemma *guard*:
shows *spec.term.none (spec.guard g) = spec.idle*
by (*rule antisym[OF spec.term.none.mono[OF spec.guard.return-le, simplified spec.term.none.return]*
spec.term.none.mono[OF spec.idle.guard-le, simplified spec.term.none.idle]])

setup $\langle \text{Sign.parent-path} \rangle$

lemma *none-all-le*:
shows *spec.term.none P ≤ spec.term.all P*
using *spec.term.galois* **by** *fastforce*

lemma *none-all*[simp]:

shows *spec.term.none* (*spec.term.all* P) = *spec.term.none* P

by (*metis spec.eq-iff spec.term.lower-upper-contractive*
spec.term.none.idempotent spec.term.none.mono spec.term.none-all-le)

lemma *all-none*[simp]:

shows *spec.term.all* (*spec.term.none* P) = *spec.term.all* P

by (*metis spec.eq-iff spec.term.galois spec.term.none-all*)

setup ‹*Sign.mandatory-path all*›

lemmas *bot*[simp] = *spec.term.upper-bot*

lemmas *top* = *spec.term.upper-top*

lemmas *monotone* = *spec.term.monotone-upper*

lemmas *mono* = *monotoneD*[OF *spec.term.all.monotone*]

lemma *expansive*:

shows $P \leq \text{spec.term.all } P$

using *spec.term.galois* **by** *blast*

lemmas *Sup* = *spec.term.upper-Sup*

lemmas *sup* = *spec.term.upper-sup*

lemmas *Inf* = *spec.term.upper-Inf*

lemmas *inf* = *spec.term.upper-inf*

lemmas *singleton* = *spec.term.all-def*[**where** $P = \langle \sigma \rangle$] **for** σ

lemma *monomorphic*:

shows *spec.term.cl* - = *spec.term.all*

unfolding *spec.term.cl-def* **by** *simp*

lemma *closed-conv*:

assumes $P \in \text{spec.term.closed}$ -

shows $P = \text{spec.term.all } P$

using *assms spec.term.closed-conv* **by** (*auto simp: spec.term.all.monomorphic*)

lemma *closed*[iff]:

shows *spec.term.all* $P \in \text{spec.term.closed}$ -

using *spec.term.closed-upper* **by** (*auto simp: spec.term.all.monomorphic*)

lemma *idempotent*[simp]:

shows *spec.term.all* (*spec.term.all* P) = *spec.term.all* P

by (*metis antisym spec.term.galois spec.term.lower-upper-contractive spec.term.none.idempotent*)

lemma *map*: — $vf = id$ on the RHS

fixes $vf :: 'v \Rightarrow 'w$

shows *spec.term.all* (*spec.map af sf vf* P) = *spec.map af sf id* (*spec.term.all* P) (**is** ?lhs = ?rhs)

proof(*rule antisym*[OF *spec.singleton-le-extI*])

fix σ

assume $\langle \sigma \rangle \leq ?lhs$

then obtain $\sigma' i w$

where $\langle \sigma' \rangle \leq P$

and *trace.T* (*trace.init* σ) (*trace.rest* σ) $w \simeq_S \text{trace.map af sf vf}$ (*trace.take* $i \sigma'$)

using *that* **by** (*fastforce elim!: trace.less-eq-takeE trace.take.naturalE*)

simp: trace.take.map spec.singleton.le-conv spec.singleton-le-conv

then show $\langle \sigma \rangle \leq ?rhs$
by (*simp add: spec.singleton.le-conv spec.singleton-le-conv*)
*(fastforce intro: exI[**where** $x = \text{trace}.T (\text{trace}.init \sigma') (\text{trace}.rest (\text{trace}.take i \sigma')) (\text{trace}.term \sigma)$] exI[**where** $x = \text{None}$] elim: order.trans[rotated] simp: trace.natural-def spec.singleton.mono trace.less-eq-None take-is-prefix)*

next
show $?rhs \leq ?lhs$
by (*simp add: spec.term.none.map flip: spec.term.galois*)
*(simp flip: spec.term.none.map[**where** $vf = vf$])*

qed

lemma *invmap*: — $vf = id$ on the RHS
fixes $vf :: 'v \Rightarrow 'w$
shows *spec.term.all* (*spec.invmap af sf vf P*) = *spec.invmap af sf id (spec.term.all P)* (**is** $?lhs = ?rhs$)

proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*simp add: order.trans[OF spec.term.none.contractive spec.map-invmap.lower-upper-contractive] flip: spec.map-invmap.galois spec.term.galois spec.term.all.map[**where** $vf = vf$]*)

show $?rhs \leq ?lhs$
by (*simp add: spec.term.none.invmap flip: spec.term.galois*)
*(simp flip: spec.term.none.invmap-gen[**where** $vf = vf$])*

qed

lemma *vmap-unit-absorb*:
shows *spec.vmap* $\langle () \rangle$ (*spec.term.all P*) = *spec.term.all P* (**is** $?lhs = ?rhs$)

proof(*rule antisym[OF - spec.singleton-le-extI]*)
show $?lhs \leq ?rhs$
by (*simp add: spec.term.none.map spec.map.id flip: spec.term.galois*)

show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ
using *that*
by (*clarsimp simp: spec.singleton.le-conv*
*intro!: exI[**where** $x = \text{trace}.map id id \langle \text{undefined} \rangle \sigma$] (metis (mono-tags) order.refl fun-unit-id trace.t.map-ident)*)

qed

lemma *vmap-unit*:
shows *spec.vmap* $\langle () \rangle$ (*spec.term.all P*) = *spec.term.all (spec.vmap* $\langle () \rangle$ *P)*

by (*simp add: spec.map.id spec.term.all.map spec.term.all.vmap-unit-absorb*)

lemma *idle*:
shows *spec.term.all spec.idle* = $(\sqcup v. \text{spec}.return v)$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*clarsimp simp: spec.term.all-def spec.singleton.le-conv option.case-eq-if*)

show $?rhs \leq ?lhs$
by (*simp add: spec.term.none.Sup spec.term.none.return flip: spec.term.galois*)

qed

lemma *action*:
fixes $F :: ('v \times 'a \times 's \times 's)$ *set*
shows *spec.term.all (spec.action F)* = *spec.action (UNIV \times snd $' F$)* \sqcup $(\sqcup v. \text{spec}.return v)$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*clarsimp simp: spec.term.all-def spec.singleton.le-conv spec.singleton.action-le-conv*
split: option.split)

meson

show $?rhs \leq ?lhs$
by (*force simp: spec.action-def spec.idle-le spec.term.none.idle spec.term.none.return*
spec.term.none.Sup spec.term.none.sup spec.term.none.singleton
simp flip: spec.term.galois)

qed

lemma *return*:
shows $spec.term.all (spec.return v) = (\bigsqcup v. spec.return v)$
by (*auto simp: spec.return-def spec.term.all.action*
simp flip: spec.action.SUP-not-empty spec.action.sup
intro: arg-cong[where f=spec.action])

lemma *guard*:
shows $spec.term.all (spec.guard g) = (\bigsqcup v. spec.return v)$
by (*simp add: spec.eq-iff spec.idle.guard-le spec.term.none.Sup spec.term.none.return*
spec.term.all.mono[OF spec.guard.return-le, unfolded spec.term.all.return]
flip: spec.term.galois)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path idle.term \rangle$

lemma *none-le-conv[spec.idle-le]*:
shows $spec.idle \leq spec.term.none P \longleftrightarrow spec.idle \leq P$
by (*metis spec.term.all.monomorphic spec.term.cl-def spec.term.galois spec.term.none.idle*)

lemma *all-le-conv[spec.idle-le]*:
shows $spec.idle \leq spec.term.all P \longleftrightarrow spec.idle \leq P$
by (*simp add: spec.term.none.idle flip: spec.term.galois*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path term.closed \rangle$

lemma *return-unit*:
shows $spec.return () \in spec.term.closed -$
by (*rule spec.term.closed-clI*) (*simp add: spec.term.all.return spec.term.all.monomorphic*)

lemma *none-inf*:
fixes $P :: ('a, 's, 'v) spec$
fixes $Q :: ('a, 's, 'w) spec$
assumes $P \in spec.term.closed -$
shows $P \sqcap spec.term.none Q = spec.term.none (spec.term.none P \sqcap Q)$ (**is** $?lhs = ?rhs$)
and $spec.term.none Q \sqcap P = spec.term.none (Q \sqcap spec.term.none P)$ (**is** $?thesis1$)

proof –

show $?lhs = ?rhs$
proof(*rule antisym[OF spec.singleton-le-extI]*)
show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ
using *that* **by** (*cases* σ) (*simp add: spec.singleton.le-conv*)
show $?rhs \leq ?lhs$
by (*auto simp: spec.term.galois intro: le-infI1 le-infI2 spec.term.none-all-le spec.term.all.expansive*)

qed

then show $?thesis1$
by (*simp add: ac-simps*)

qed

lemma *none-inf-monomorphic*:

fixes $P :: ('a, 's, 'v) \text{ spec}$

fixes $Q :: ('a, 's, 'v) \text{ spec}$

assumes $P \in \text{spec.term.closed}$ -

shows $P \sqcap \text{spec.term.none } Q = \text{spec.term.none } (P \sqcap Q)$ (**is** *?thesis1*)

and $\text{spec.term.none } Q \sqcap P = \text{spec.term.none } (Q \sqcap P)$ (**is** *?thesis2*)

by (*simp-all add: spec.term.closed.none-inf[OF assms, simplified] spec.term.none.inf*)

lemma *singleton-le-extI*:

assumes $Q \in \text{spec.term.closed}$ -

assumes $\bigwedge s \text{ xs. } \langle s, \text{xs}, \text{None} \rangle \leq P \implies \langle s, \text{xs}, \text{None} \rangle \leq Q$

shows $P \leq Q$

by (*subst spec.term.closed-conv[OF assms(1)], rule spec.singleton-le-extI*)

(*auto simp: trace.split-all spec.term.none.singleton spec.term.all.monomorphic*)

simp flip: spec.term.galois

intro: assms(2)

elim: order.trans[rotated])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.8 Bind

We define monadic *bind* in terms of bi-strict *continue*. The latter supports left and right residuals (see, amongst many others, Hoare and He (1987); Hoare, He, and Sanders (1987b); Pratt (1990)), whereas *bind* encodes the non-retractability of observable actions, i.e., $\text{spec.term.none } f \leq f \ggg g$, which defeats a general right residual.

It is tempting to write this in a more direct style (using *case-option*) but the set comprehension syntax is not friendly to strengthen/monotonicity facts.

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition $\text{continue} :: ('a, 's, 'v) \text{ spec} \Rightarrow ('v \Rightarrow ('a, 's, 'w) \text{ spec}) \Rightarrow ('a, 's, 'w) \text{ spec}$ **where**

$\text{continue } f g =$

$\bigsqcup \{ \langle \text{trace.init } \sigma_f, \text{trace.rest } \sigma_f @ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle$

$|\sigma_f \sigma_g v. \langle \sigma_f \rangle \leq f \wedge \text{trace.init } \sigma_g = \text{trace.final } \sigma_f \wedge \text{trace.term } \sigma_f = \text{Some } v \wedge \langle \sigma_g \rangle \leq g v \}$

definition $\text{bind} :: ('a, 's, 'v) \text{ spec} \Rightarrow ('v \Rightarrow ('a, 's, 'w) \text{ spec}) \Rightarrow ('a, 's, 'w) \text{ spec}$ **where**

$\text{bind } f g = \text{spec.term.none } f \sqcup \text{spec.continue } f g$

ad hoc-overloading

$\text{Monad-Syntax.bind} \equiv \text{spec.bind}$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma *continue-le-conv*:

shows $\langle \sigma \rangle \leq \text{spec.continue } f g$

$\longleftrightarrow (\exists \text{xs ys } v w. \langle \text{trace.init } \sigma, \text{xs}, \text{Some } v \rangle \leq f$

$\wedge \langle \text{trace.final}' (\text{trace.init } \sigma) \text{xs}, \text{ys}, w \rangle \leq g v$

$\wedge \sigma \leq \text{trace.T } (\text{trace.init } \sigma) (\text{xs} @ \text{ys}) w)$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

assume *?lhs*

then obtain $s \text{ xs } \text{ys } v w$

where $\sigma: \langle \sigma \rangle \leq \langle \text{trace.T } s (\text{xs} @ \text{ys}) w \rangle$

and $f: \langle s, \text{xs}, \text{Some } v \rangle \leq f$

and $g: \langle \text{trace.final}' s \text{xs}, \text{ys}, w \rangle \leq g v$

by (*clarsimp simp: spec.continue-def trace.split-all spec.singleton-le-conv*)

from σ **show** *?rhs*

```

proof(cases rule: trace.less-eqE)
  case prefix
  from prefix(3)[simplified, simplified trace.natural'.append] show ?thesis
  proof(cases rule: prefix-append-not-NilE)
    case incomplete
    then obtain zs where zs: trace.natural' s xs = trace.natural' (trace.init  $\sigma$ ) (trace.rest  $\sigma$ ) @ zs
      by (rule prefixE)
    from f prefix(2) zs
    have  $\langle \text{trace.init } \sigma, \text{trace.rest } \sigma @ zs, \text{Some } v \rangle \leq f$ 
      by (clarsimp elim!: order.trans[rotated])
      (metis trace.natural'.append trace.final'.natural' trace.natural'.natural')
    moreover
    from g prefix(2) zs
    have  $\langle \text{trace.final}' (trace.init \sigma) (trace.rest \sigma @ zs), ys, \text{None} \rangle \leq g v$ 
      by (clarsimp elim!: order.trans[rotated])
      (metis spec.singleton.less-eq-None trace.final'.natural' trace.final'.sims(3))
    moreover note  $\langle \text{trace.term } (\natural\sigma) = \text{None} \rangle$ 
    ultimately show ?thesis
      by (fastforce simp: trace.less-eq-None)
  next
  case (continue us)
  from continue(1)
  obtain ys' zs'
    where trace.rest  $\sigma = ys' @ zs'$ 
      and trace.natural' (trace.init  $\sigma$ ) ys' = trace.natural' s xs
      and trace.natural' (trace.final' (trace.init  $\sigma$ ) (trace.natural' s xs)) zs' = us
    by (clarsimp simp: trace.natural'.eq-append-conv)
  with f g prefix(1,2) continue(2-) show ?thesis
    by - (rule exI[where x=ys'];
      force simp: spec.singleton-le-conv trace.less-eq-None trace.natural-def
        cong: trace.final'.natural'-cong
        elim!: order.trans[rotated]
        intro: exI[where x=None])
  qed
next
case (maximal x) with f g show ?thesis
  by (fastforce simp: trace.stuttering.equiv.append-conv
    cong: trace.final'.natural'-cong
    elim!: order.trans[rotated])
qed
next
show ?rhs  $\implies$  ?lhs
  using spec.singleton.mono by (auto 10 0 simp: spec.continue-def trace.split-Ex)
qed

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.mandatory-path continue} \rangle$ 

lemma mono:
  assumes  $f \leq f'$ 
  assumes  $\bigwedge v. g v \leq g' v$ 
  shows  $\text{spec.continue } f g \leq \text{spec.continue } f' g'$ 
unfolding spec.continue-def
apply (strengthen ord-to-strengthen(1)[OF assms(1)])
apply (strengthen ord-to-strengthen(1)[OF assms(2)])
apply (rule order.refl)
done

```

lemma *strengthen*[*strg*]:
assumes *st-ord* $F f f'$
assumes $\bigwedge x. \textit{st-ord } F (g x) (g' x)$
shows *st-ord* $F (\textit{spec.continue } f g) (\textit{spec.continue } f' g')$
using *assms* **by** (*cases* F ; *simp* *add*: *spec.continue.mono*)

lemma *mono2mono*[*cont-intro, partial-function-mono*]:
assumes *monotone* *orda* $(\leq) f$
assumes $\bigwedge x. \textit{monotone } \textit{orda } (\leq) (\lambda y. g y x)$
shows *monotone* *orda* $(\leq) (\lambda x. \textit{spec.continue } (f x) (g x))$
using *assms* **by** (*simp* *add*: *monotone-def spec.continue.mono*)

definition *resL* :: $(v \Rightarrow (a, s, w) \textit{spec}) \Rightarrow (a, s, w) \textit{spec} \Rightarrow (a, s, v) \textit{spec}$ **where**
resL $g P = \bigsqcup \{f. \textit{spec.continue } f g \leq P\}$

definition *resR* :: $(a, s, v) \textit{spec} \Rightarrow (a, s, w) \textit{spec} \Rightarrow (v \Rightarrow (a, s, w) \textit{spec})$ **where**
resR $f P = \bigsqcup \{g. \textit{spec.continue } f g \leq P\}$

interpretation *L*: *galois.complete-lattice-class* $\lambda f. \textit{spec.continue } f g \textit{spec.continue.resL } g$ **for** g
proof

show $\textit{spec.continue } f g \leq P \iff f \leq \textit{spec.continue.resL } g P$ (**is** *?lhs* \iff *?rhs*) **for** $f P$
proof(*rule iffI*)
assume *?rhs*
then have $\textit{spec.continue } f g \leq \textit{spec.continue } (\textit{spec.continue.resL } g P) g$
by (*simp* *add*: *spec.continue.mono*)
also have $\dots \leq P$
by (*auto simp*: *spec.continue.resL-def spec.continue-def*)
finally show *?lhs* .
qed (*simp* *add*: *spec.continue.resL-def Sup-upper*)
qed

interpretation *R*: *galois.complete-lattice-class* $\lambda g. \textit{spec.continue } f g \textit{spec.continue.resR } f$
for $f :: (a, s, v) \textit{spec}$

proof
show $\textit{spec.continue } f g \leq P \iff g \leq \textit{spec.continue.resR } f P$ (**is** *?lhs* \iff *?rhs*)
for $g :: (a, s, w) \textit{spec}$
and $P :: (a, s, v) \textit{spec}$
proof(*rule iffI*)
assume *?rhs*
then have $\textit{spec.continue } f g \leq \textit{spec.continue } f (\textit{spec.continue.resR } f P)$
by (*simp* *add*: *le-fun-def spec.continue.mono*)
also have $\dots \leq P$
by (*auto simp*: *spec.continue.resR-def spec.continue-def*)
finally show *?lhs* .
qed (*simp* *add*: *spec.continue.resR-def Sup-upper*)
qed

setup $\langle \textit{Sign.parent-path} \rangle$

setup $\langle \textit{Sign.mandatory-path singleton} \rangle$

lemma *bind-le-conv*:

shows $\langle \sigma \rangle \leq \textit{spec.bind } f g \iff \langle \sigma \rangle \leq \textit{spec.term.none } f \vee \langle \sigma \rangle \leq \textit{spec.continue } f g$
by (*simp* *add*: *spec.bind-def*)

lemma *bind-le*[*consumes 1*]:

assumes $\langle \sigma \rangle \leq f \ggg g$

obtains
(incomplete) $\langle \sigma \rangle \leq \text{spec.term.none } f$
| *(continue)* $\sigma_f \sigma_g v_f$
where $\langle \sigma_f \rangle \leq f$ **and** $\text{trace.final } \sigma_f = \text{trace.init } \sigma_g$ **and** $\text{trace.term } \sigma_f = \text{Some } v_f$
and $\langle \sigma_g \rangle \leq g v_f$ **and** $\not\vdash \sigma_g \neq \text{trace.T } (\text{trace.init } \sigma_g) \square \text{None}$
and $\sigma = \text{trace.T } (\text{trace.init } \sigma_f) (\text{trace.rest } \sigma_f @ \text{trace.rest } \sigma_g) (\text{trace.term } \sigma_g)$
using *assms*[*unfolded spec.singleton.bind-le-conv*]
proof(*atomize-elim, induct rule: stronger-disjE*[*consumes 1, case-names incomplete continue*])
case *continue*
from $\langle \sigma \rangle \leq \text{spec.continue } f g$ **obtain** $xs \ ys \ v \ w$
where $f: \langle \text{trace.init } \sigma, xs, \text{Some } v \rangle \leq f$
and $g: \langle \text{trace.final}' (\text{trace.init } \sigma) xs, ys, w \rangle \leq g v$
and $\sigma: \sigma \leq \text{trace.T } (\text{trace.init } \sigma) (xs @ ys) w$
by (*clarsimp simp: spec.singleton.continue-le-conv*)
with $\langle \neg \sigma \rangle \leq \text{spec.term.none } f$ **obtain** ys'
where $\langle \text{trace.final}' (\text{trace.init } \sigma) xs, ys', \text{trace.term } \sigma \rangle \leq g v$
and $\text{trace.rest } \sigma = xs @ ys'$
and $\text{trace.natural}' (\text{trace.final}' (\text{trace.init } \sigma) xs) ys' = \square \longrightarrow (\exists y. \text{trace.term } \sigma = \text{Some } y)$
by (*atomize-elim, cases* σ)
(*auto elim!: trace.less-eqE prefix-append-not-NilE*
elim: order.trans[*OF spec.singleton.mono, rotated*]
dest: spec.singleton.mono[*OF iffD2*][*OF trace.less-eq-None(2)*][**where** $s = \text{trace.init } \sigma$ **and** $\sigma = \text{trace.T}$
(trace.init } \sigma) xs (Some v)], *simplified*]
order.trans[*OF spec.singleton.less-eq-None*]
simp: trace.less-eq-None spec.singleton.le-conv trace.natural'.eq-Nil-conv)+
with f **show** *?case*
by (*cases* σ) (*force simp: trace.natural-def*)
qed *blast*

setup $\langle \text{Sign.parent-path} \rangle$

lemma *bind-le*[*case-names incomplete continue*]:

assumes $\text{spec.term.none } f \leq P$
assumes $\bigwedge \sigma_f \sigma_g v. \llbracket \langle \sigma_f \rangle \leq f; \text{trace.init } \sigma_g = \text{trace.final } \sigma_f; \text{trace.term } \sigma_f = \text{Some } v; \langle \sigma_g \rangle \leq g v;$
 $\not\vdash \sigma_g \neq \text{trace.T } (\text{trace.init } \sigma_g) \square \text{None} \rrbracket$
 $\implies \langle \text{trace.init } \sigma_f, \text{trace.rest } \sigma_f @ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle \leq P$
shows $f \ggg g \leq P$
by (*rule spec.singleton-le-extI*) (*use assms in* $\langle \text{fastforce elim: spec.singleton.bind-le} \rangle$)

setup $\langle \text{Sign.mandatory-path bind} \rangle$

definition *resL* :: $(v \Rightarrow (a, s, w) \text{ spec}) \Rightarrow (a, s, w) \text{ spec} \Rightarrow (a, s, v) \text{ spec}$ **where**
 $\text{resL } g P = \bigsqcup \{f. f \ggg g \leq P\}$

lemma *incompleteI*:

assumes $\langle s, xs, \text{None} \rangle \leq f$
shows $\langle s, xs, \text{None} \rangle \leq f \ggg g$
using *assms* **by** (*auto simp: spec.bind-def spec.singleton.term.none-le-conv*)

lemma *continueI*:

assumes $f: \langle s, xs, \text{Some } v \rangle \leq f$
assumes $g: \langle \text{trace.final}' s xs, ys, w \rangle \leq g v$
shows $\langle s, xs @ ys, w \rangle \leq f \ggg g$
using *assms* **by** (*force simp: spec.bind-def spec.continue-def intro!: disjI2*)

lemma *singletonL*:

shows $\langle \sigma \rangle \ggg g$
 $= \text{spec.term.none } \langle \sigma \rangle$

$\sqcup \sqcup \{ \langle \text{trace.init } \sigma, \text{trace.rest } \sigma @ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle | \sigma_g. \text{trace.final } \sigma = \text{trace.init } \sigma_g \wedge (\exists v. \text{trace.term } \sigma = \text{Some } v \wedge \langle \sigma_g \rangle \leq g v) \}$ (is ?lhs = ?rhs)

proof(rule antisym)

show ?lhs \leq ?rhs

proof(induct rule: spec.bind-le)

case (continue $\sigma_f \sigma_g v$) **then show** ?case

by (cases σ_f ; cases σ_g)

(simp add: trace.split-E x ;

metis order.refl spec.singleton.simps(1) trace.final'.natural' trace.stuttering.equiv.append-cong)

qed force

show ?rhs \leq ?lhs

by (cases σ)

(force simp: spec.term.none.singleton spec.singleton.bind-le-conv spec.singleton.continue-le-conv)

qed

lemma mono:

assumes $f \leq f'$

assumes $\bigwedge v. g v \leq g' v$

shows spec.bind $f g \leq$ spec.bind $f' g'$

unfolding spec.bind-def

apply (strengthen ord-to-strengthen(1)[OF assms(1)])

apply (strengthen ord-to-strengthen(1)[OF assms(2)])

apply (rule order.refl)

done

lemma strengthen[strg]:

assumes st-ord $F f f'$

assumes $\bigwedge x. \text{st-ord } F (g x) (g' x)$

shows st-ord $F (\text{spec.bind } f g) (\text{spec.bind } f' g')$

using assms **by** (cases F ; simp add: spec.bind.mono)

lemma mono2mono[cont-intro, partial-function-mono]:

assumes monotone orda (\leq) f

assumes $\bigwedge x. \text{monotone orda } (\leq) (\lambda y. g y x)$

shows monotone orda (\leq) ($\lambda x. \text{spec.bind } (f x) (g x)$)

using assms **by** (simp add: monotone-def spec.bind.mono)

interpretation L: galois.complete-lattice-class $\lambda f. f \ggg g \text{ spec.bind.resL } g$ **for** g

proof

show $f \ggg g \leq P \longleftrightarrow f \leq \text{spec.bind.resL } g P$ (is ?lhs \longleftrightarrow ?rhs) **for** $f P$

proof(rule iffI)

assume ?rhs

then have $f \ggg g \leq \text{spec.bind.resL } g P \ggg g$

by (simp add: spec.bind.mono)

also have $\dots \leq P$

by (simp add: spec.bind.resL-def spec.bind-def spec.term.none.Sup spec.continue.L.lower-Sup)

finally show ?lhs .

qed (simp add: spec.bind.resL-def Sup-upper)

qed

lemmas SUPL = spec.bind.L.lower-SUP

lemmas SupL = spec.bind.L.lower-Sup

lemmas supL = spec.bind.L.lower-sup[of $f_1 f_2 g$] **for** $f_1 f_2 g$

lemmas INFL-le = spec.bind.L.lower-INF-le

lemmas InfL-le = spec.bind.L.lower-Inf-le

lemmas infL-le = spec.bind.L.lower-inf-le[of $f_1 f_2 g$] **for** $f_1 f_2 g$

lemma SUPR:

shows $\text{spec.bind } f (\lambda v. \bigsqcup_{x \in X}. g \ x \ v) = (\bigsqcup_{x \in X}. f \ggg g \ x) \sqcup (f \ggg \perp)$ (**is** *?thesis1*) — *Sup over ('a, 's, 'v) spec*

and $\text{spec.bind } f (\bigsqcup_{x \in X}. g \ x) = (\bigsqcup_{x \in X}. f \ggg g \ x) \sqcup (f \ggg \perp)$ (**is** *?thesis2*) — *Sup over functions*

proof –

show *?thesis1*

by (*cases* $X = \{\}$)

(*simp-all add: spec.bind-def spec.continue.R.lower-bot sup-SUP ac-simps*

spec.continue.R.lower-SUP[where f=g and X=X, unfolded Sup-fun-def image-image]

flip: bot-fun-def)

then show *?thesis2*

by (*simp add: Sup-fun-def image-image*)

qed

lemma SUPR-not-empty:

assumes $X \neq \{\}$

shows $\text{spec.bind } f (\lambda v. \bigsqcup_{x \in X}. g \ x \ v) = (\bigsqcup_{x \in X}. f \ggg g \ x)$

using *assms* **by** (*clarsimp simp: spec.bind.SUPR spec.bind.mono sup.absorb1 SUPI simp flip: ex-in-conv*)

lemmas $\text{supR} = \text{spec.bind.SUPR-not-empty}[\text{where } g=\text{id} \text{ and } X=\{g_1, g_2\} \text{ for } g_1 \ g_2, \text{ simplified}]$

lemma InfR-le:

shows $\text{spec.bind } f (\lambda v. \prod_{x \in X}. g \ x \ v) \leq (\prod_{x \in X}. f \ggg g \ x)$

by (*meson INF-lower order.refl le-INF-iff spec.bind.mono*)

lemma infR-le:

shows $\text{spec.bind } f (g_1 \sqcap g_2) \leq (f \ggg g_1) \sqcap (f \ggg g_2)$

and $\text{spec.bind } f (\lambda v. g_1 \ v \sqcap g_2 \ v) \leq (f \ggg g_1) \sqcap (f \ggg g_2)$

by (*simp-all add: spec.bind.mono*)

lemma Inf-le:

shows $\text{spec.bind } (\prod_{x \in X}. f \ x) (\lambda v. (\prod_{x \in X}. g \ x \ v)) \leq (\prod_{x \in X}. \text{spec.bind } (f \ x) (g \ x))$

by (*auto simp: le-INF-iff intro: spec.bind.mono*)

lemma inf-le:

shows $\text{spec.bind } (f_1 \sqcap f_2) (\lambda v. g_1 \ v \sqcap g_2 \ v) \leq \text{spec.bind } f_1 \ g_1 \sqcap \text{spec.bind } f_2 \ g_2$

by (*simp add: spec.bind.mono*)

lemma mcont2mcont[cont-intro]:

assumes $\text{mcont luba orda Sup } (\leq) \ f$

assumes $\bigwedge v. \text{mcont luba orda Sup } (\leq) (\lambda x. g \ x \ v)$

shows $\text{mcont luba orda Sup } (\leq) (\lambda x. \text{spec.bind } (f \ x) (g \ x))$

proof(*rule ccpo.mcont2mcont'[OF complete-lattice-ccpo - - assms(1)]*)

show $\text{mcont Sup } (\leq) \text{Sup } (\leq) (\lambda f. \text{bind } f (g \ x))$ **for** x

by (*intro mcontI contI monotoneI*) (*simp-all add: spec.bind.mono flip: spec.bind.SUPL*)

show $\text{mcont luba orda Sup } (\leq) (\lambda x. \text{bind } f (g \ x))$ **for** f

by (*intro mcontI monotoneI contI*)

(*simp-all add: mcont-monoD[OF assms(2)] spec.bind.mono*

flip: spec.bind.SUPR-not-empty contD[OF mcont-cont[OF assms(2)]])

qed

lemmas $\text{botL}[simp] = \text{spec.bind.L.lower-bot}$

lemma botR:

shows $f \ggg \perp = \text{spec.term.none } f$

by (*simp add: spec.bind-def spec.continue.R.lower-bot*)

lemma eq-bot-conv:

shows $\text{spec.bind } f \ g = \perp \iff f = \perp$
by (*fastforce simp: spec.continue.L.lower-bot spec.bind-def spec.term.galois simp flip: bot.extremum-unique*)

lemma *idleL[simp]*:

shows $\text{spec.idle} \ggg g = \text{spec.idle}$
by (*simp add: spec.idle-def spec.bind.SupL image-image spec.bind.singletonL spec.term.none.singleton*)

lemma *idleR*:

shows $f \gg \text{spec.idle} = f \ggg \perp$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*fastforce simp: spec.bind.botR trace.split-all spec.singleton.le-conv*
intro!: spec.bind-le
intro: spec.bind.incompleteI order.trans[rotated])
show $?rhs \leq ?lhs$
by (*simp add: spec.bind.mono*)

qed

lemmas *ifL = if-distrib[where f= $\lambda f.$ spec.bind f g for g]*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path idle} \rangle$

lemma *bind-le-conv[spec.idle-le]*:

shows $\text{spec.idle} \leq f \ggg g \iff \text{spec.idle} \leq f$ (**is** $?lhs \iff ?rhs$)
proof(*rule iffI*)
show $?lhs \implies ?rhs$
by (*fastforce simp: spec.idle-def spec.singleton.mono trace.less-eq-None spec.singleton.bind-le-conv*
spec.singleton.term.none-le-conv spec.singleton.continue-le-conv
elim: order.trans[rotated])
show $?rhs \implies ?lhs$
by (*simp add: spec.bind-def spec.idle.term.none-le-conv le-supI1*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

lemma *bindL-le[iff]*:

shows $\text{spec.term.none } f \leq f \ggg g$
by (*simp add: spec.bind-def*)

lemma *bind*:

shows $\text{spec.term.none } (f \ggg g) = f \ggg (\lambda v. \text{spec.term.none } (g \ v))$
by (*rule spec.singleton.antisym*)
(auto elim: spec.singleton.bind-le
simp: trace.split-all spec.bind.incompleteI spec.bind.continueI spec.singleton.term.none-le-conv)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *bind*:

shows $\text{spec.term.all } (f \ggg g) = \text{spec.term.all } f \sqcup (f \ggg (\lambda v. \text{spec.term.all } (g \ v)))$ (**is** $?lhs = ?rhs$)
proof(*rule antisym[OF spec.singleton-le-extI]*)

```

show  $\langle \sigma \rangle \leq ?rhs$  if  $\langle \sigma \rangle \leq ?lhs$  for  $\sigma$ 
  using that
  by (cases  $\sigma$ )
    (fastforce simp: trace.split-all spec.singleton.le-conv
      elim!: spec.singleton.bind-le
      intro: spec.bind.continueI)
show  $?rhs \leq ?lhs$ 
  by (simp add: spec.term.none.sup spec.term.none.bind spec.bind.mono flip: spec.term.galois)
qed

```

```

setup  $\langle Sign.parent-path \rangle$ 

```

```

setup  $\langle Sign.parent-path \rangle$ 

```

```

The monad laws for  $(\gg=)$ .  setup  $\langle Sign.mandatory-path bind \rangle$ 

```

```

lemma bind:

```

```

  fixes  $f :: (-, -, -)$  spec
  shows  $f \gg= g \gg= h = f \gg= (\lambda v. g v \gg= h)$  (is  $?lhs = ?rhs$ )

```

```

proof(rule antisym)

```

```

  show  $?lhs \leq ?rhs$ 

```

```

  proof(induct rule: spec.bind-le)

```

```

    case incomplete show ?case

```

```

      by (simp add: spec.bind.mono spec.term.none.bind)

```

```

  next

```

```

    case (continue  $\sigma_f g \sigma_h v$ ) then show ?case

```

```

      by (cases  $\sigma_h$ )

```

```

        (fastforce elim: spec.singleton.bind-le spec.bind.continueI
          simp: spec.singleton.le-conv trace.split-all)

```

```

  qed

```

```

  show  $?rhs \leq ?lhs$ 

```

```

  proof(induct rule: spec.bind-le)

```

```

    case incomplete show ?case

```

```

      by (strengthen ord-to-strengthen(2)[OF spec.term.none.bindL-le])

```

```

        (simp add: spec.term.none.bind)

```

```

  next

```

```

    case (continue  $\sigma_f \sigma_{gh} v$ )

```

```

    note  $*$  = continue.hyps(1-3)

```

```

    from  $\langle \sigma_{gh} \rangle \leq g v \gg= h$  show ?case

```

```

    proof(cases rule: spec.singleton.bind-le)

```

```

      case incomplete with * show ?thesis

```

```

        by (cases  $\sigma_f$ )

```

```

          (clarsimp simp: spec.singleton.le-conv spec.bind.incompleteI spec.bind.continueI)

```

```

    next

```

```

      case (continue  $\sigma_g \sigma_h v_g$ ) with * show ?thesis

```

```

        by (cases  $\sigma_f$ ; cases  $\sigma_g$ )

```

```

          (simp flip: append-assoc; fastforce intro!: spec.bind.continueI)

```

```

  qed

```

```

  qed

```

```

qed

```

```

lemmas assoc = spec.bind.bind

```

```

lemma returnL-le:

```

```

  shows  $g v \leq spec.return v \gg= g$  (is  $?lhs \leq ?rhs$ )

```

```

proof(rule spec.singleton-le-extI)

```

```

  show  $\langle \sigma \rangle \leq ?rhs$  if  $\langle \sigma \rangle \leq ?lhs$  for  $\sigma$ 

```

```

  by (rule spec.bind.continueI[where  $xs=[]$  and  $s=trace.init \sigma$  and  $ys=trace.rest \sigma$  and  $w=trace.term \sigma$  and

```

$v=v$, *simplified*)
(simp-all add: spec.return-def spec.action.stutterI that)

qed

lemma *returnL*:

assumes $spec.idle \leq g v$

shows $spec.return v \ggg g = g v$

by (*rule antisym[OF spec.bind-le spec.bind.returnL-le]*)

(simp-all add: assms spec.term.none.return spec.singleton.return-le-conv trace.split-all)

lemma *returnR[simp]*:

shows $f \ggg spec.return = f$ (**is** $?lhs = ?rhs$)

proof(*rule antisym[OF - spec.singleton-le-extI]*)

show $?lhs \leq ?rhs$

by (*auto intro: spec.bind-le*

simp: trace.split-all spec.singleton.return-le-conv order.trans[OF spec.singleton.less-eq-None(1)]

split: option.split-asm)

show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ

using *that*

by (*cases* σ ; *cases trace.term* σ ;

clarsimp simp: spec.bind.incompleteI spec.bind.continueI[where ys=[], simplified] spec.singleton.le-conv)

qed

lemma *return*: — Does not require $spec.idle \leq g v$

fixes $f :: ('a, 's, 'v) spec$

fixes $g :: 'v \Rightarrow 'x \Rightarrow ('a, 's, 'w) spec$

shows $f \ggg (\lambda v. spec.return x \ggg g v) = f \ggg (\lambda v. g v x)$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

proof(*induct rule: spec.bind-le*)

case (*continue* $\sigma_f \sigma_{rg} v$)

from $\langle \sigma_{rg} \rangle \leq spec.return x \ggg g v$ **show** *?case*

proof(*induct rule: spec.singleton.bind-le[case-names incomplete continue2]*)

case *incomplete* **with** $\langle \sigma_f \rangle \leq f$ $\langle trace.init \sigma_{rg} = trace.final \sigma_f \rangle$ **show** *?thesis*

by (*cases* σ_f)

(auto simp: spec.term.none.return spec.singleton.le-conv

intro: spec.bind.incompleteI order.trans[rotated])

next

case (*continue2* $\sigma_r \sigma_g v_r$) **with** *continue* **show** *?case*

by (*cases* σ_f) (*simp add: trace.split-all spec.singleton.le-conv spec.bind.continueI*)

qed

qed *simp*

show $?rhs \leq ?lhs$

by (*simp add: spec.bind.mono spec.bind.returnL-le*)

qed

setup $\langle Sign.mandatory-path term \rangle$

lemma *noneL[simp]*:

shows $spec.term.none f \ggg g = spec.term.none f$

by (*simp add: spec.bind.bind flip: spec.bind.botR bot-fun-def*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path map \rangle$

lemma *bind-le*: — Converse does not hold: it may be that no final states of f satisfy g

fixes $f :: ('a, 's, 'v) \text{ spec}$

fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{ spec}$

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'w \Rightarrow 'x$

shows $\text{spec.map } af \ sf \ vf \ (f \ggg g) \leq \text{spec.map } af \ sf \ id \ f \ggg (\lambda v. \text{spec.map } af \ sf \ vf \ (g \ v))$

by (*subst* (1) *spec.map-def*)

(*force simp: spec.singleton.le-conv trace.split-all trace.final'.map*

intro: spec.bind.incompleteI spec.bind.continueI

elim: spec.singleton.bind-le)

lemma *bind-inj-sf*:

fixes $f :: ('a, 's, 'x) \text{ spec}$

fixes $g :: 'x \Rightarrow ('a, 's, 'v) \text{ spec}$

assumes *inj sf*

shows $\text{spec.map } af \ sf \ vf \ (f \ggg g) = \text{spec.map } af \ sf \ id \ f \ggg (\lambda v. \text{spec.map } af \ sf \ vf \ (g \ v))$ (**is** *?lhs = ?rhs*)

proof(*rule antisym[OF spec.map.bind-le]*)

show *?rhs* \leq *?lhs*

proof(*induct rule: spec.bind-le*)

case *incomplete* **show** *?case*

by (*metis spec.map.mono spec.term.none.bindL-le spec.term.none.map-gen*)

next

case (*continue* $\sigma_f \ \sigma_g \ v$)

from *continue*(1,4) **obtain** $\sigma_f' \ \sigma_g'$

where $*$: $\langle \sigma_f' \rangle \leq f \ \langle \sigma_f \rangle \leq \langle \text{trace.map } af \ sf \ id \ \sigma_f' \rangle$

$\langle \sigma_g' \rangle \leq g \ v \ \langle \sigma_g \rangle \leq \langle \text{trace.map } af \ sf \ vf \ \sigma_g' \rangle$

by (*clarsimp simp: spec.singleton.le-conv*)

with *continue*(2,3)

have $sf \ (\text{trace.init } \sigma_g') = sf \ (\text{trace.final } \sigma_f')$

by (*cases* σ_f ; *cases* σ_g ; *cases* σ_f' ; *cases* σ_g' ; *clarsimp*)

(*clarsimp simp: spec.singleton-le-conv simp flip: trace.final'.map[where af=af and sf=sf];*

erule trace.less-eqE; simp add: trace.natural.trace-conv; metis trace.final'.natural')

with *continue*(2,3) $*$ **show** *?case*

by (*cases* σ_f ; *cases* σ_g ; *cases* σ_f' ; *cases* σ_g')

(*fastforce dest: inj-onD[OF assms, simplified]*

elim: trace.less-eqE spec.bind.continueI

simp: spec.singleton.le-conv trace.final'.map trace.less-eq-None

spec.singleton-le-conv trace.natural-def trace.natural'.append)

qed

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path vmap} \rangle$

lemma *eq-return*: — generalizes *spec.bind.returnR*

shows $\text{spec.vmap } vf \ P = P \ggg \text{spec.return} \circ vf$ (**is** *?thesis1*)

and $\text{spec.vmap } vf \ P = P \ggg (\lambda v. \text{spec.return} \ (vf \ v))$ (**is** *?lhs = ?rhs*) — useful for flip/symmetric

proof —

show *?lhs = ?rhs*

proof(*rule antisym*)

show *?lhs* \leq *?rhs*

by (*rule spec.singleton.exhaust[of P]*)

(*fastforce simp: trace.split-all spec.singleton.le-conv spec.map.Sup spec.map.singleton map-option-case*

intro: spec.bind.incompleteI spec.bind.continueI[where ys=[], simplified]

split: option.split)

next

```

show ?rhs ≤ ?lhs
  by (rule spec.bind-le)
    (force simp: trace.split-all spec.singleton.le-conv trace.less-eq-None trace.natural.mono
      spec.term.galois spec.term.all.expansive spec.term.all.map spec.map.id
      split: option.split-asm)+
qed
then show ?thesis1
  by (simp add: comp-def)
qed

lemma unitL: — monomorphise ignored return values
  shows f ≫ g = spec.vmap ⟨()⟩ f ≫ g
by (simp add: spec.vmap.eq-return comp-def spec.bind.bind spec.bind.return)

setup ⟨Sign.parent-path⟩

setup ⟨Sign.mandatory-path invmap⟩

lemma bind:
  fixes f :: ('b, 't, 'v) spec
  fixes g :: 'v ⇒ ('b, 't, 'x) spec
  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: 'w ⇒ 'x
  shows spec.invmap af sf vf (f ≫ g) = spec.invmap af sf id f ≫ (λv. spec.invmap af sf vf (g v)) (is ?lhs =
    ?rhs)
proof(rule antisym[OF spec.singleton-le-extI])
  fix σ assume ⟨σ⟩ ≤ ?lhs
  then have ⟨trace.map af sf vf σ⟩ ≤ f ≫ g by (simp add: spec.singleton.le-conv)
  then show ⟨σ⟩ ≤ ?rhs
  proof(induct rule: spec.singleton.bind-le)
  case incomplete then show ?case
    by (cases σ) (clarsimp simp: spec.singleton.le-conv spec.bind.incompleteI)
  next
  case (continue σf σg vf) then show ?case
    by (cases σ; cases σf; cases σg)
      (clarsimp simp: spec.bind.continueI map-eq-append-conv spec.singleton.le-conv trace.final'.map)
  qed
next
  show ?rhs ≤ ?lhs
  by (simp add: order.trans[OF spec.map.bind-le] spec.bind.mono spec.map-invmap.lower-upper-contractive
    flip: spec.map-invmap.galois)
qed

lemma split-vinvmap:
  shows spec.invmap af sf vf P = spec.invmap af sf id P ≫ (λv. ⌊ v' ∈ vf - {v} . spec.return v' ) (is ?lhs = ?rhs)
proof(rule antisym[OF spec.singleton-le-extI])
  show ⟨σ⟩ ≤ ?rhs if ⟨σ⟩ ≤ ?lhs for σ
  using that
  by (cases σ; cases trace.term σ)
    (auto simp: spec.singleton.le-conv
      intro: spec.bind.incompleteI spec.bind.continueI[where ys=[], simplified])
  show ?rhs ≤ ?lhs
  proof(induct rule: spec.bind-le)
  case (continue σf σg v) then show ?case
    by (cases σf; cases trace.term σg)
      (auto simp: spec.singleton.le-conv split: option.split-asm elim: order.trans[rotated])
  qed (simp add: spec.term.none.invmap-gen[where vf'=vf] spec.invmap.mono)

```

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma *return-const*:

assumes $V \neq \{\}$

assumes $W \neq \{\}$

shows $\text{spec.action } (V \times F) = \text{spec.action } (W \times F) \gg (\bigsqcup v \in V. \text{spec.return } v)$ (is ?lhs = ?rhs)

proof(rule antisym)

from $\langle W \neq \{\} \rangle$ show ?lhs \leq ?rhs

by – (rule spec.action-le;

fastforce intro: spec.bind.continueI[where xs=[x] and v=SOME w. w \in W for x, simplified]

spec.action.stepI

simp: some-in-eq spec.singleton.le-conv spec.singleton.action-le-conv

spec.idle.action-le spec.idle.bind-le-conv)

from $\langle V \neq \{\} \rangle$ show ?rhs \leq ?lhs

by – (rule spec.bind-le,

fastforce simp: spec.term.galois spec.term.all.action intro: le-supI1 spec.action.mono,

auto 0 3 simp: spec.singleton.le-conv spec.singleton.action-le-conv

simp flip: trace.steps'.empty-conv

simp del: trace.steps'.simps split: option.splits)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term.closed} \rangle$

lemma *bind-all-return*:

assumes $f \in \text{spec.term.closed}$ -

shows $f \gg (\bigsqcup \text{range spec.return}) = \text{spec.term.all } f$ (is ?lhs = ?rhs)

proof(rule antisym[OF - spec.singleton-le-extI])

show ?lhs \leq ?rhs

by (subst (2) spec.term.closed-conv[OF assms])

(simp add: spec.term.none.bind spec.term.none.Sup image-image spec.term.none.return

spec.bind.botR spec.bind.idleR

spec.term.all.monomorphic

flip: spec.term.galois)

next

fix σ

assume $\langle \sigma \rangle \leq ?rhs$

then obtain v where $\langle \text{trace.init } \sigma, \text{trace.rest } \sigma, \text{Some } v \rangle \leq f$

by (subst (asm) spec.term.closed-conv[OF assms])

(force simp: spec.singleton.le-conv spec.term.all.monomorphic)

then show $\langle \sigma \rangle \leq ?lhs$

by (cases σ ; cases trace.term σ)

(auto simp: spec.singleton.le-conv spec.bind.continueI[where ys=[], simplified]

split: option.split)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.9 Kleene star

We instantiate the generic Kleene locale with monomorphic $\text{spec.return } ()$. The polymorphic $(\sqcup v. \text{spec.return } v)$ fails the comp-unitR axiom ($\varepsilon \leq x \implies x \cdot \varepsilon = x$).

setup $\langle \text{Sign.mandatory-path spec} \rangle$

interpretation kleene : $\text{weak-kleene spec.return } () \lambda x y. \text{spec.bind } x \langle y \rangle$
by standard ($\text{simp-all add: spec.bind.bind spec.bind.supL spec.bind.supR}$
 $\text{spec.bind.returnL order.trans[OF spec.idle.return-le]}$)

setup $\langle \text{Sign.mandatory-path idle.kleene} \rangle$

lemmas $\text{star-le[spec.idle-le]} = \text{order.trans[OF spec.idle.return-le spec.kleene.epsilon-star-le]}$

lemmas $\text{rev-star-le[spec.idle-le]} = \text{spec.idle.kleene.star-le[unfolded spec.kleene.star-rev-star]}$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path return.kleene} \rangle$

lemmas $\text{star-le} = \text{spec.kleene.epsilon-star-le}$

lemmas $\text{rev-star-le} = \text{spec.return.kleene.star-le[unfolded spec.kleene.star-rev-star]}$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path kleene} \rangle$

lemma star-idle :

shows $\text{spec.kleene.star spec.idle} = \text{spec.return } ()$

by ($\text{subst spec.kleene.star.simps}$) ($\text{simp add: sup.absorb2 spec.idle.return-le}$)

lemmas $\text{rev-star-idle} = \text{spec.kleene.star-idle[unfolded spec.kleene.star-rev-star]}$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term.all.kleene} \rangle$

lemma star-closed-le :

fixes $P :: (-, -, \text{unit}) \text{ spec}$

assumes $P \in \text{spec.term.closed -}$

shows $\text{spec.term.all (spec.kleene.star } P) \leq \text{spec.kleene.star } P$ (**is** $- \leq ?\text{rhs}$)

proof($\text{induct rule: spec.kleene.star.fixp-induct[where } P=\lambda R. \text{spec.term.all (} R \text{ } P) \leq ?\text{rhs, case-names adm bot step}]$)

case ($\text{step } R$) **show** $?case$

by ($\text{auto simp: spec.term.all.sup spec.term.all.bind spec.kleene.expansive-star spec.term.all.return}$

$\text{simp flip: spec.term.all.closed-conv[OF assms]}$

$\text{intro: spec.kleene.epsilon-star-le}$

$\text{order.trans[OF spec.bind.mono[OF order.refl step] spec.kleene.fold-starL]}$)

qed simp-all

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term.closed.kleene} \rangle$

lemma star :

assumes $P \in \text{spec.term.closed -}$

shows $\text{spec.kleene.star } P \in \text{spec.term.closed -}$

by (rule spec.term.closed-clI)
 (simp add: spec.term.all.kleene.star-closed-le[OF assms] spec.term.all.monomorphic)

setup <Sign.parent-path>

setup <Sign.parent-path>

8.10 Transition relations

Using *spec.kleene.star* we can specify the transitions each agent is allowed to perform. These constraints ((\sqcap) *spec.rel r*) distribute through all program constructs (for suitable *r*).

Observations:

- the Galois connection between *spec.rel* and *spec.steps* is much easier to show in the powerset model
 - see [van Staden \(2015, Footnote 2\)](#)
- most useful facts about *spec.steps* depend on the model

setup <Sign.mandatory-path spec>

setup <Sign.mandatory-path rel>

definition *act* :: ('a, 's) steps \Rightarrow ('a, 's, unit) spec **where** — lift above *spec.return* to ease some proofs
act r = *spec.action* ({()} \times (r \cup UNIV \times Id))

abbreviation *monomorphic* :: ('a, 's) steps \Rightarrow ('a, 's, unit) spec **where**
monomorphic r \equiv *spec.kleene.star* (*spec.rel.act r*)

lemma *act-alt-def*:

shows *spec.rel.act r* = *spec.action* ({()} \times r) \sqcup *spec.return* ()

by (simp add: spec.rel.act-def spec.return-def Sigma-Un-distrib2 flip: spec.action.sup)

setup <Sign.parent-path>

definition *rel* :: ('a, 's) steps \Rightarrow ('a, 's, 'v) spec **where**
rel r = *spec.term.all* (*spec.rel.monomorphic r*)

definition *steps* :: ('a, 's, 'v) spec \Rightarrow ('a, 's) steps **where**
steps P = $\bigcap \{r. P \leq \text{spec.rel } r\}$

setup <Sign.mandatory-path rel.act>

lemma *monotone*:

shows *mono spec.rel.act*

proof(rule *monotoneI*)

show *spec.rel.act r* \leq *spec.rel.act r'* **if** $r \subseteq r'$ **for** $r, r' :: ('a, 's) \text{ steps}$

using *that unfolding spec.rel.act-def by (strengthen ord-to-strengthen(1)[OF <r \leq r'>]) simp*
qed

lemmas *strengthen[strg]* = *st-monotone[OF spec.rel.act.monotone]*

lemmas *mono* = *monotoneD[OF spec.rel.act.monotone]*

lemma *empty*:

shows *spec.rel.act* {} = *spec.return* ()

by (simp add: spec.rel.act-def spec.return-def spec.action.empty)

lemma *UNIV*:

shows $\text{spec.rel.act UNIV} = \text{spec.action} (\{\()\} \times \text{UNIV})$

by (*simp add: spec.rel.act-def*)

lemma *sup*:

shows $\text{spec.rel.act} (r \cup s) = \text{spec.rel.act} r \sqcup \text{spec.rel.act} s$

by (*fastforce simp: spec.rel.act-def simp flip: spec.action.sup intro: arg-cong[where f=spec.action]*)

lemma *stutter*:

shows $\text{spec.rel.act} (\text{UNIV} \times \text{Id}) = \text{spec.return} ()$

by (*simp add: spec.rel.act-def spec.return-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

lemma *act-mono*:

shows $\text{spec.term.all} (\text{spec.rel.act} r) = \text{spec.rel.act} r$

by (*simp add: spec.rel.act-alt-def spec.term.all.sup spec.term.all.action spec.term.all.return UNIV-unit*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *rel*:

shows $\text{spec.term.all} (\text{spec.rel} r) = \text{spec.rel} r$

by (*simp add: spec.rel-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

lemma *act*:

shows $\text{spec.rel.act} r \in \text{spec.term.closed}$ -

by (*metis spec.term.all.rel.act-mono spec.term.all.closed*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *rel*:

shows $\text{spec.rel} r \in \text{spec.term.closed}$ -

by (*metis spec.term.all.closed spec.term.all.rel*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

lemma *inf-none-rel*: — polymorphic constants

shows $\text{spec.term.none} (\text{spec.rel} r :: ('a, 's, 'v) \text{spec}) \sqcap \text{spec.term.none} P$

$= \text{spec.rel} r \sqcap (\text{spec.term.none} P :: ('a, 's, 'v) \text{spec})$ (**is** *?thesis1*)

and $\text{spec.term.none} P \sqcap \text{spec.term.none} (\text{spec.rel} r :: ('a, 's, 'v) \text{spec})$

$= \text{spec.term.none} P \sqcap (\text{spec.rel} r :: ('a, 's, 'v) \text{spec})$ (**is** *?thesis2*)

proof -

show *?thesis1*

by (*metis spec.term.closed.rel spec.term.closed.none-inf(1)*)

spec.term.none.idempotent spec.term.none.inf(2) spec.term.none-all spec.term.all.rel)

then show *?thesis2*
by (*simp add: ac-simps*)
qed

lemma *inf-rel*:

shows *spec.term.none* $P \sqcap \text{spec.rel } r = \text{spec.term.none } (P \sqcap \text{spec.rel } r)$ (**is** *?thesis1*)
and *spec.rel* $r \sqcap \text{spec.term.none } P = \text{spec.term.none } (\text{spec.rel } r \sqcap P)$ (**is** *?thesis2*)
by (*simp-all add: ac-simps spec.term.none.inf(2) spec.term.none.inf-none-rel(2)*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path return} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

lemma *act-le*:

shows *spec.return* $() \leq \text{spec.rel.act } r$
by (*simp add: spec.rel.act.mono flip: spec.rel.act.empty*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *rel-le*:

shows *spec.return* $v \leq \text{spec.rel } r$
by (*simp add: spec.rel-def spec.term.none.return spec.idle.kleene.star-le flip: spec.term.galois*)

lemma *Sup-rel-le*:

shows $\bigsqcup \text{range spec.return} \leq \text{spec.rel } r$
by (*simp add: spec.return.rel-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path idle} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

lemmas *act-le[spec.idle-le]* = *order.trans[OF spec.idle.return-le spec.return.rel.act-le]*

setup $\langle \text{Sign.parent-path} \rangle$

lemmas *rel-le[spec.idle-le]* = *order.trans[OF spec.idle.return-le spec.return.rel-le]*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

setup $\langle \text{Sign.mandatory-path act} \rangle$

lemma *le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.rel.act } r \iff \text{trace.steps } \sigma = \{\} \vee (\exists x \in r. \text{trace.steps } \sigma = \{x\})$

by (*auto simp: spec.rel.act-def spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv split: option.split*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path monomorphic} \rangle$

lemma *le-steps*:

assumes $\text{trace.steps } \sigma \subseteq r$

shows $\langle \sigma \rangle \leq \text{spec.rel.monomorphic } r$

using *assms*

proof(*induct trace.rest σ arbitrary: σ rule: rev-induct*)

case *Nil* **then show** *?case*

by (*simp add: spec.singleton.rel.act.le-conv order.trans[OF - spec.kleene.expansive-star]*)

next

case (*snoc x xs σ*)

from *snoc(2,3)*

have $*$: $\langle \text{trace.init } \sigma, xs, \text{Some } () \rangle \leq \text{spec.rel.monomorphic } r$

by (*cases σ*) (*fastforce intro: snoc(1) simp: trace.steps'.append*)

have $**$: $\langle \text{trace.final}' (\text{trace.init } \sigma) xs, [x], \text{trace.term } \sigma \rangle \leq \text{spec.rel.act } r$

proof(*cases trace.final' (trace.init σ) xs = snd x*)

case *True* **with** *snoc.premis snoc.hyps(2)* **show** *?thesis*

by (*simp add: spec.singleton.le-conv*)

next

case *False* **with** *snoc.premis snoc.hyps(2)* **show** *?thesis*

by (*cases σ*) (*clarsimp simp: spec.singleton.le-conv trace.steps'.append*)

qed

show *?case*

by (*rule order.trans[OF spec.bind.continueI[OF * **, simplified snoc.hyps(2) trace.t.collapse] spec.kleene.fold-starR]*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path rel.act} \rangle$

lemmas *mono-le = spec.kleene.expansive-star*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path rel.monomorphic} \rangle$

lemma *alt-def*:

shows $\text{spec.rel.monomorphic } r = \bigsqcup (\text{spec.singleton } \{ \sigma. \text{trace.steps } \sigma \subseteq r \})$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

show *?lhs \leq ?rhs*

proof(*induct rule: spec.kleene.star.fixp-induct[case-names adm bot step]*)

case (*step R*)

have $\text{spec.return } () \leq ?rhs$

by (*force intro: spec.singleton-le-extI simp: spec.singleton.le-conv dest: trace.steps'.sims(5)*)

moreover

have $\text{spec.rel.act } r \gg ?rhs \leq ?rhs$

proof(*induct rule: spec.bind-le*)

case *incomplete* **show** *?case*

by (*rule spec.singleton-le-extI*)

(*clarsimp simp: spec.singleton.le-conv;*

metis order.refl empty-subsetI insert-subsetI trace.steps'.empty-conv(1))

next

```

case (continue  $\sigma_f \sigma_g v$ ) then show ?case
  by (fastforce intro!: exI[where  $x = \text{trace}.T (\text{trace}.init \sigma_f) (\text{trace}.rest \sigma_f @ \text{trace}.rest \sigma_g) (\text{trace}.term \sigma_g)$ ]
    simp: trace.steps'.append spec.singleton.rel.act.le-conv
    dest: trace.steps.mono[OF iffD1[OF spec.singleton-le-conv], simplified,
      simplified trace.steps'.natural'])

qed
ultimately show ?case
  by - (strengthen ord-to-strengthen(1)[OF step]; simp)
qed simp-all
show ?rhs  $\leq$  ?lhs
  by (simp add: spec.singleton.rel.monomorphic.le-steps)
qed

setup <Sign.parent-path>

setup <Sign.mandatory-path singleton>

setup <Sign.mandatory-path rel>

lemma monomorphic-le-conv[spec.singleton.le-conv]:
  shows  $\langle \sigma \rangle \leq \text{spec}.rel.monomorphic\ r \iff \text{trace}.steps\ \sigma \subseteq r$ 
by (fastforce simp: spec.rel.monomorphic.alt-def spec.singleton-le-conv trace.steps'.natural'
  dest: trace.steps.mono)

setup <Sign.parent-path>

lemma rel-le-conv[spec.singleton.le-conv]:
  shows  $\langle \sigma \rangle \leq \text{spec}.rel\ r \iff \text{trace}.steps\ \sigma \subseteq r$ 
by (cases  $\sigma$ ) (auto simp add: spec.rel-def spec.singleton.le-conv)

setup <Sign.parent-path>

interpretation rel: galois.complete-lattice-class spec.steps spec.rel
proof(rule galois.upper-preserves-InfI)
  show mono spec.rel
  by (simp add: monoD monotoneI spec.kleene.monotone-star spec.rel.act.mono spec.rel-def)
  show  $(\prod x \in X. \text{spec}.rel\ x) \leq \text{spec}.rel\ (\bigcap X)$  for  $X :: ('a, 'b)\ steps\ set$ 
  by (fastforce intro: spec.singleton-le-extI simp: le-INF-iff spec.singleton.le-conv)
qed (simp add: spec.steps-def)

lemma rel-alt-def:
  shows  $\text{spec}.rel\ r = \bigsqcup (\text{spec}.singleton\ \{ \sigma. \text{trace}.steps\ \sigma \subseteq r \})$ 
by (simp flip: spec.singleton.rel-le-conv)

setup <Sign.mandatory-path vmap>

lemma unit-rel:
  shows  $\text{spec}.vmap\ \langle () \rangle (\text{spec}.rel\ r) = \text{spec}.rel\ r$ 
by (simp add: spec.rel-def spec.term.all.vmap-unit-absorb)

setup <Sign.parent-path>

setup <Sign.mandatory-path rel>

lemma monomorphic-conv: — if the return type is unit
  shows  $\text{spec}.rel\ r = \text{spec}.rel.monomorphic\ r$ 
by (simp add: spec.rel-def
  flip: spec.term.all.closed-conv[OF spec.term.closed.kleene.star[OF spec.term.closed.rel.act]])

```

lemma *monomorphic-act-le*: — unit return type

shows $\text{spec.rel.act } r \leq \text{spec.rel } r$

by (*simp add: spec.rel.monomorphic-conv spec.rel.act.mono-le*)

lemma *empty*:

shows $\text{spec.rel } \{\} = (\bigsqcup v. \text{spec.return } v)$

by (*simp add: spec.rel-def spec.kleene.star-epsilon spec.rel.act.empty spec.term.all.return*)

lemmas $\text{UNIV} = \text{spec.rel.upper-top}$

lemmas $\text{top} = \text{spec.rel.UNIV}$

lemmas $\text{INF} = \text{spec.rel.upper-INF}$

lemmas $\text{Inf} = \text{spec.rel.upper-Inf}$

lemmas $\text{inf} = \text{spec.rel.upper-inf}$

lemmas $\text{Sup-le} = \text{spec.rel.Sup-upper-le}$

lemmas $\text{sup-le} = \text{spec.rel.sup-upper-le}$ — Converse does not hold: the RHS allows interleaving of r and s steps

lemma *reflcl*:

shows $\text{spec.rel } (r \cup A \times \text{Id}) = \text{spec.rel } r$

and $\text{spec.rel } (A \times \text{Id} \cup r) = \text{spec.rel } r$

by (*simp-all add: spec.rel-def spec.rel.act-def ac-simps flip: Times-Un-distrib1*)

lemma *minus-Id*:

shows $\text{spec.rel } (r - A \times \text{Id}) = \text{spec.rel } r$

by (*metis Un-Diff-cancel spec.rel.reflcl(2)*)

lemma *Id*:

shows $\text{spec.rel } (A \times \text{Id}) = (\bigsqcup v. \text{spec.return } v)$

by (*subst spec.rel.minus-Id[where A=A, symmetric]*) (*simp add: spec.rel.empty*)

lemmas $\text{monotone} = \text{spec.rel.monotone-upper}$

lemmas $\text{mono} = \text{monotoneD}[OF \text{spec.rel.monotone}, \text{of } r \text{ } r' \text{ for } r \text{ } r']$

lemma *mono-reflcl*:

assumes $r \subseteq s \cup \text{UNIV} \times \text{Id}$

shows $\text{spec.rel } r \leq \text{spec.rel } s$

by (*metis assms spec.rel.mono spec.rel.reflcl(1)*)

lemma *unfoldL*:

shows $\text{spec.rel } r = \text{spec.rel.act } r \gg \text{spec.rel } r$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

by (*rule order.trans[OF spec.bind.returnL-le spec.bind.mono[OF spec.return.rel.act-le order.refl]]*)

show $?rhs \leq ?lhs$

by (*subst (2) spec.rel-def, subst spec.kleene.star-unfoldL*)

(*simp add: spec.term.all.sup spec.term.all.bind le-supI1 flip: spec.rel-def*)

qed

lemma *foldR*: — arbitrary interstitial return type

shows $\text{spec.rel } r \gg \text{spec.rel.act } r = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)

proof —

have $?lhs = \text{spec.rel.monomorphic } r \gg \text{spec.rel.act } r$

by (*subst spec.vmap.unitL*) (*simp add: spec.vmap.unit-rel spec.rel.monomorphic-conv*)

also have $\dots = ?rhs$

proof(*rule antisym*)

show $\text{spec.rel.monomorphic } r \gg \text{spec.rel.act } r \leq ?rhs$

by (*simp add: spec.kleene.fold-starR spec.rel.monomorphic-conv*)
 show $?rhs \leq \text{spec.rel.monomorphic } r \gg \text{spec.rel.act } r$
 by (*simp add: spec.rel.monomorphic-conv*)
 (*rule spec.bind.mono[OF order.refl spec.return.rel.act-le, where 'c=unit, simplified]*)
 qed
 finally show *?thesis* .
 qed

lemma *wind-bind*: — arbitrary interstitial return type
 shows $\text{spec.rel } r \gg \text{spec.rel } r = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
 show $?lhs \leq ?rhs$
proof(*induct rule: spec.bind-le*)
 case *incomplete* show *?case*
 by (*simp add: spec.term.all.rel spec.term.galois*)
next
 case (*continue* $\sigma_f \sigma_g v$) **then** show *?case*
 by (*simp add: spec.singleton.rel-le-conv trace.steps'.append*)
 qed
 show $?rhs \leq ?lhs$
 by (*meson order.trans order.refl spec.bind.mono spec.bind.returnL-le spec.return.rel-le*)
 qed

lemma *wind-bind-leading*: — arbitrary interstitial return type
 assumes $r' \subseteq r$
 shows $\text{spec.rel } r' \gg \text{spec.rel } r = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
from *assms* show $?lhs \leq ?rhs$
 by (*metis order.refl spec.bind.mono spec.rel.mono spec.rel.wind-bind*)
 show $?rhs \leq ?lhs$
 by (*meson order.trans spec.eq-iff spec.bind.mono spec.bind.returnL-le spec.return.rel-le*)
 qed

lemma *wind-bind-trailing*: — arbitrary interstitial return type
 assumes $r' \subseteq r$
 shows $\text{spec.rel } r \gg \text{spec.rel } r' = \text{spec.rel } r$ (**is** $?lhs = ?rhs$)
proof(*rule antisym[OF - spec.singleton-le-extI]*)
from *assms* show $?lhs \leq ?rhs$
 by (*metis order.refl spec.bind.mono spec.rel.mono spec.rel.wind-bind*)
show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ
 using *that*
 by (*cases* σ)
 (*force simp: spec.singleton.le-conv spec.singleton.bind-le-conv spec.singleton.continue-le-conv*)
 qed

Interstitial unit, for unfolding

lemmas *unwind-bind* = *spec.rel.wind-bind*[**where** $'d=\text{unit}$, *symmetric*]
lemmas *unwind-bind-leading* = *spec.rel.wind-bind-leading*[**where** $'d=\text{unit}$, *symmetric*]
lemmas *unwind-bind-trailing* = *spec.rel.wind-bind-trailing*[**where** $'d=\text{unit}$, *symmetric*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *rel*:

shows $\text{spec.invmap } af \text{ sf } vf (\text{spec.rel } r) = \text{spec.rel } (\text{map-prod } af (\text{map-prod } sf \text{ sf}) - ' (r \cup \text{UNIV} \times \text{Id}))$
by (*fastforce intro: antisym spec.singleton-le-extI simp: spec.singleton.invmap-le-conv spec.singleton.rel-le-conv trace.steps'.map*)

lemma *range*:

shows $\text{spec.invmap } af \text{ } sf \text{ } vf \text{ } P = \text{spec.invmap } af \text{ } sf \text{ } vf \text{ } (P \sqcap \text{spec.rel } (\text{range } af \times \text{range } sf \times \text{range } sf))$
by (*rule antisym*[*OF spec.singleton-le-extI*])
(*auto simp: spec.singleton.le-conv trace.steps'.map spec.invmap.mono*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path map} \rangle$

lemma *inf-rel*:

shows $\text{spec.map } af \text{ } sf \text{ } vf \text{ } P \sqcap \text{spec.rel } r$
 $= \text{spec.map } af \text{ } sf \text{ } vf \text{ } (P \sqcap \text{spec.rel } (\text{map-prod } af \text{ } (\text{map-prod } sf \text{ } sf) - ' (r \cup \text{UNIV} \times \text{Id})))$
and $\text{spec.rel } r \sqcap \text{spec.map } af \text{ } sf \text{ } vf \text{ } P$
 $= \text{spec.map } af \text{ } sf \text{ } vf \text{ } (\text{spec.rel } (\text{map-prod } af \text{ } (\text{map-prod } sf \text{ } sf) - ' (r \cup \text{UNIV} \times \text{Id})) \sqcap P)$
by (*simp-all add: spec.invmap.rel spec.map.inf-distr ac-simps*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma *rel-le*:

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$
fixes $r :: ('a, 's) \text{ steps}$
assumes $\bigwedge v \ a \ s \ s'. (v, a, s, s') \in F \implies (a, s, s') \in r \vee s = s'$
shows $\text{spec.action } F \leq \text{spec.rel } r$
unfolding *spec.rel-def*
by (*strengthen ord-to-strengthen(2)*[*OF spec.kleene.expansive-star*])
(*fastforce simp: spec.rel.act-def spec.term.all.action intro: le-supI1 spec.action.mono dest: assms*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path kleene} \rangle$

lemma *star-le*:

assumes $S \leq \text{spec.rel } r$
shows $\text{spec.kleene.star } S \leq \text{spec.rel } r$
by (*strengthen ord-to-strengthen(1)*[*OF assms*])
(*simp add: spec.rel-def spec.kleene.idempotent-star*
flip: spec.term.all.closed-conv[*OF spec.term.closed.kleene.star*][*OF spec.term.closed.rel.act*])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemma *relL-le*:

shows $g \ x \leq \text{spec.rel } r \ggg g$
by (*rule order.trans*[*OF spec.bind.returnL-le spec.bind.mono*][*OF spec.return.rel-le order.refl*])

lemma *relR-le*:

shows $f \leq f \ggg \text{spec.rel } r$
by (*rule order.trans*[*OF eq-refl*][*OF spec.bind.returnR*][*symmetric*]
spec.bind.mono[*OF order.refl spec.return.rel-le*])

lemma *inf-rel*:

shows $(f \ggg g) \sqcap \text{spec.rel } r = (\text{spec.rel } r \sqcap f) \ggg (\lambda x. \text{spec.rel } r \sqcap g \ x)$ (**is** *?thesis1*)
and $\text{spec.rel } r \sqcap (f \ggg g) = (\text{spec.rel } r \sqcap f) \ggg (\lambda x. \text{spec.rel } r \sqcap g \ x)$ (**is** *?lhs = ?rhs*)
proof –

```

show ?lhs = ?rhs
proof(rule antisym[OF spec.singleton-le-extI])
  fix  $\sigma$  assume lhs:  $\langle \sigma \rangle \leq ?lhs$ 
  then have  $\langle \sigma \rangle \leq f \ggg g$  by simp
  then show  $\langle \sigma \rangle \leq ?rhs$ 
  proof(cases rule: spec.singleton.bind-le)
    case incomplete with lhs show ?thesis
      by (cases  $\sigma$ ) (simp add: spec.singleton.le-conv spec.bind.incompleteI)
  next
    case (continue  $\sigma_f \sigma_g v_f$ ) with lhs show ?thesis
      by (cases  $\sigma_f$ ) (simp add: spec.singleton.le-conv trace.steps'.append spec.bind.continueI)
  qed
next
  show ?rhs  $\leq$  ?lhs
  proof(induct rule: spec.bind-le)
    case incomplete show ?case
      by (auto simp: spec.term.none.inf spec.term.galois spec.term.all.rel intro: le-infI1 le-infI2)
  next
    case (continue  $\sigma_f \sigma_g v$ ) then show ?case
      by (cases  $\sigma_f$ ; cases  $\sigma_g$ ) (simp add: spec.singleton.rel-le-conv spec.bind.continueI trace.steps'.append)
  qed
qed
then show ?thesis1
  by (simp add: ac-simps)
qed

```

lemma inf-rel-distr-le:

```

shows  $(f \sqcap \text{spec.rel } r) \ggg (\lambda v. g_1 v \sqcap g_2) \leq (f \ggg g_1) \sqcap (\text{spec.rel } r \ggg (\lambda :: \text{unit}. g_2))$ 
by (rule spec.bind-le;
  force simp: trace.split-all spec.singleton.le-conv spec.term.galois spec.term.none.inf
    spec.term.all.bind spec.term.all.rel
    intro: le-infI1 le-infI2 spec.bind.continueI)

```

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma inf-rel:

```

shows  $\langle \sigma \rangle \sqcap \text{spec.rel } r = \bigsqcup (\text{spec.singleton } \{ \sigma'. \sigma' \leq \sigma \wedge \text{trace.steps } \sigma' \subseteq r \})$  (is ?lhs = ?rhs)
and  $\text{spec.rel } r \sqcap \langle \sigma \rangle = \bigsqcup (\text{spec.singleton } \{ \sigma'. \sigma' \leq \sigma \wedge \text{trace.steps } \sigma' \subseteq r \})$  (is ?thesis2)

```

proof –

```

show ?lhs = ?rhs
proof(rule antisym[OF spec.singleton-le-extI])
  show  $\langle \sigma' \rangle \leq ?rhs$  if  $\langle \sigma' \rangle \leq ?lhs$  for  $\sigma'$ 
    using that
    by (fastforce simp: spec.singleton.le-conv spec.singleton-le-conv
      elim: trace.natural.less-eqE[where  $u = \sigma$  and  $u' = \sigma$ , simplified]
      dest: trace.stuttering.equiv.steps)
  show ?rhs  $\leq$  ?lhs
    by (clarsimp simp: spec.singleton.le-conv spec.singleton.mono)
qed
then show ?thesis2
  by (rule inf-commute-conv)

```

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma *inf-rel*:
fixes $F :: ('v \times 'a \times 's \times 's)$ *set*
fixes $r :: ('a, 's)$ *steps*
assumes $\bigwedge a. \text{refl } (r \text{ `` } \{a\})$
shows $\text{spec.action } F \sqcap \text{spec.rel } r = \text{spec.action } (F \cap \text{UNIV} \times r)$ (**is** $?lhs = ?rhs$)
and $\text{spec.rel } r \sqcap \text{spec.action } F = \text{spec.action } (F \cap \text{UNIV} \times r)$ (**is** $?thesis1$)
proof –
show $?lhs = ?rhs$
proof(*rule antisym[OF spec.singleton-le-extI]*)
from $\text{reflD}[OF \text{assms}]$ **show** $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ
using *that*
by (*auto 0 2 simp: spec.singleton.le-conv spec.singleton.action-le-conv split: option.split-asm*)
show $?rhs \leq ?lhs$
by (*rule order.trans[OF spec.action.inf-le inf.mono[OF order.refl spec.action.rel-le]]*) *simp*
qed
then show $?thesis1$
by (*rule inf-commute-conv*)
qed

lemma *inf-rel-reflcl*:
shows $\text{spec.action } F \sqcap \text{spec.rel } r = \text{spec.action } (F \cap \text{UNIV} \times (r \cup \text{UNIV} \times \text{Id}))$
and $\text{spec.rel } r \sqcap \text{spec.action } F = \text{spec.action } (F \cap \text{UNIV} \times (r \cup \text{UNIV} \times \text{Id}))$
by (*simp-all add: refl-on-def spec.rel.reflcl ac-simps flip: spec.action.inf-rel*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path return} \rangle$

lemma *inf-rel*:
shows $\text{spec.rel } r \sqcap \text{spec.return } v = \text{spec.return } v$
and $\text{spec.return } v \sqcap \text{spec.rel } r = \text{spec.return } v$
by (*simp-all add: spec.return-def ac-simps spec.action.inf-rel-reflcl Sigma-Un-distrib2 Int-Un-distrib Times-Int-Times flip: Sigma-Un-distrib2*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path kleene.star} \rangle$

lemma *inf-rel*:
shows $\text{spec.kleene.star } P \sqcap \text{spec.rel } r = \text{spec.kleene.star } (P \sqcap \text{spec.rel } r)$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*induct rule: spec.kleene.star.fixp-induct*)
(simp-all add: ac-simps inf-sup-distrib1 spec.bind.inf-rel le-supI1 le-supI2 spec.bind.mono)
show $?rhs \leq ?lhs$
by (*induct rule: spec.kleene.star.fixp-induct*)
(simp-all add: ac-simps le-supI2 inf-sup-distrib1 spec.bind.inf-rel spec.bind.mono spec.return.inf-rel)
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path steps} \rangle$

lemma *simps[simp]*:

shows $(a, s, s) \notin \text{spec.steps } P$
by (*simp add: spec.steps-def exI*[**where** $x = \text{UNIV} \times -\text{Id}$]
spec.rel.minus-Id[**where** $r = \text{UNIV}$ **and** $A = \text{UNIV}$, *simplified*] *spec.rel.UNIV*)

lemma member-conv:

shows $x \in \text{spec.steps } P \longleftrightarrow (\exists \sigma. \langle \sigma \rangle \leq P \wedge x \in \text{trace.steps } \sigma)$
by (*meson spec.rel.galois spec.singleton.rel-le-conv spec.singleton-le-ext-conv subset-Compl-singleton*)

setup $\langle \text{Sign.mandatory-path term} \rangle$

lemma none:

shows $\text{spec.steps } (\text{spec.term.none } P) = \text{spec.steps } P$
by (*metis order.eq-iff spec.rel.galois spec.term.all.rel spec.term.galois*)

lemma all:

shows $\text{spec.steps } (\text{spec.term.all } P) = \text{spec.steps } P$
by (*metis spec.steps.term.none spec.term.none-all*)

setup $\langle \text{Sign.parent-path} \rangle$

lemmas $\text{bot} = \text{spec.rel.lower-bot}$

lemmas $\text{monotone} = \text{spec.rel.monotone-lower}$

lemmas $\text{mono} = \text{monotoneD}[OF \text{spec.steps.monotone}]$

lemmas $\text{Sup} = \text{spec.rel.lower-Sup}$

lemmas $\text{sup} = \text{spec.rel.lower-sup}$

lemmas $\text{Inf-le} = \text{spec.rel.lower-Inf-le}$

lemmas $\text{inf-le} = \text{spec.rel.lower-inf-le}$

lemma singleton:

shows $\text{spec.steps } \langle \sigma \rangle = \text{trace.steps } \sigma$
by (*meson subset-antisym order.refl spec.rel.galois spec.singleton.rel-le-conv*)

lemma idle:

shows $\text{spec.steps } \text{spec.idle} = \{\}$
by (*simp add: spec.steps-def spec.idle.rel-le*)

lemma action:

shows $\text{spec.steps } (\text{spec.action } F) = \text{snd } ' F - \text{UNIV} \times \text{Id}$
by (*force simp: spec.action-def split-def*
spec.steps.Sup spec.steps.sup spec.steps.singleton spec.steps.idle)

lemma return:

shows $\text{spec.steps } (\text{spec.return } v) = \{\}$
by (*simp add: spec.return-def spec.steps.action*)

lemma bind-le: — see *spec.steps.bind*

shows $\text{spec.steps } (f \ggg g) \subseteq \text{spec.steps } f \cup (\bigcup v. \text{spec.steps } (g v))$
by (*force simp: spec.steps.member-conv spec.singleton.le-conv trace.split-all trace.steps'.append*
elim!: spec.singleton.bind-le)

lemma kleene-star:

shows $\text{spec.steps } (\text{spec.kleene.star } P) = \text{spec.steps } P$ (**is** *?lhs = ?rhs*)
proof(*rule antisym*)
show *?lhs* \leq *?rhs*
proof(*induct rule: spec.kleene.star.fixp-induct*[*case-names adm bot step*])
case (*step S*) **then show** *?case*

by (*simp add: spec.steps.sup spec.steps.return order.trans[OF spec.steps.bind-le]*)
qed (*simp-all add: spec.steps.bot*)
show $?rhs \subseteq ?lhs$
by (*simp add: spec.steps.mono spec.kleene.expansive-star*)
qed

lemma map:

shows $spec.steps (spec.map\ af\ sf\ vf\ P)$
 $= map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ 'spec.steps\ P - UNIV \times Id$
by (*rule spec.singleton.exhaust[of P]*)
(force simp: spec.map.Sup spec.map.singleton spec.steps.Sup spec.steps.singleton trace.steps'.map image-Union)

lemma invmap-le:

shows $spec.steps (spec.invmap\ af\ sf\ vf\ P)$
 $\subseteq map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ - '(spec.steps (P \sqcap spec.rel (range\ af \times range\ sf \times range\ sf)) \cup UNIV \times Id) - UNIV \times Id$
by (*simp add: spec.rel.galois spec.rel.minus-Id*
order.trans[OF - spec.invmap.mono[OF spec.rel.upper-lower-expansive]]
flip: vimage-Un spec.invmap.rel[where vf=vf] spec.invmap.range)

setup $\langle Sign.mandatory\text{-}path\ rel \rangle$

lemma monomorphic:

fixes $r :: ('a, 's)\ steps$
shows $spec.steps (spec.rel.monomorphic\ r) = r - UNIV \times Id$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \subseteq ?rhs$
by (*simp add: spec.rel.galois spec.rel.minus-Id flip: spec.rel.monomorphic-conv*)
show $?rhs \subseteq ?lhs$
by (*force simp: spec.rel.monomorphic.alt-def spec.term.all.Sup spec.term.all.singleton*
spec.steps.Sup spec.steps.singleton
dest: spec[where x=trace.T s [(a, s')] None for a s']
split: if-splits)

qed

setup $\langle Sign.parent\text{-}path \rangle$

lemma rel:

fixes $r :: ('a, 's)\ steps$
shows $spec.steps (spec.rel\ r) = r - UNIV \times Id$
by (*simp add: spec.rel-def spec.steps.term.all spec.steps.rel.monomorphic*)

lemma top:

shows $spec.steps \top = UNIV \times - Id$
using $spec.steps.rel[where\ r=UNIV]$ **by** (*simp add: spec.rel.UNIV*)

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle Sign.parent\text{-}path \rangle$

8.11 Sequential assertions

We specify sequential behavior with preconditions and postconditions.

8.11.1 Preconditions

setup $\langle Sign.mandatory\text{-}path\ spec \rangle$

definition $pre :: 's \text{ pred} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**
 $pre \ P = \sqcup (\text{spec.singleton } \{ \sigma. P (\text{trace.init } \sigma) \})$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma $pre\text{-le-conv}[\text{spec.singleton.le-conv}]$:

shows $\langle \sigma \rangle \leq \text{spec.pre } P \longleftrightarrow P (\text{trace.init } \sigma)$

by $(\text{auto simp add: spec.pre-def spec.singleton-le-conv trace.natural-def elim: trace.less-eqE})$

lemma $inf\text{-pre}$:

shows $\text{spec.pre } P \sqcap \langle \sigma \rangle = (\text{if } P (\text{trace.init } \sigma) \text{ then } \langle \sigma \rangle \text{ else } \perp)$ **(is ?thesis1)**

and $\langle \sigma \rangle \sqcap \text{spec.pre } P = (\text{if } P (\text{trace.init } \sigma) \text{ then } \langle \sigma \rangle \text{ else } \perp)$ **(is ?thesis2)**

proof –

show $?thesis1$

by $(\text{cases } \sigma; \text{rule spec.singleton.antisym})$

$(\text{auto simp: spec.singleton.le-conv spec.singleton-le-conv spec.singleton.not-bot trace.natural.trace-conv}$

$\text{split: if-split-asm}$

$\text{elim: trace.less-eqE})$

then show $?thesis2$

by $(\text{rule inf-commute-conv})$

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path idle} \rangle$

lemma $pre\text{-le-conv}[\text{spec.idle-le}]$:

shows $\text{spec.idle} \leq (\text{spec.pre } P :: ('a, 's, 'v) \text{ spec}) \longleftrightarrow P = \top$ **(is ?lhs \longleftrightarrow ?rhs)**

proof(rule iffI)

show $?lhs \implies ?rhs$

by (rule ccontr)

$(\text{simp add: fun-eq-iff spec.pre-def spec.idle-def trace.split-Ex}$

$\text{spec.singleton-le-conv trace.less-eq-None trace.natural.simps})$

show $?rhs \implies ?lhs$

by $(\text{rule spec.singleton-le-extI}) (\text{simp add: spec.singleton.le-conv})$

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma pre :

shows $\text{spec.term.all } (\text{spec.pre } P) = \text{spec.pre } P$

by $(\text{rule spec.singleton.antisym; simp add: spec.singleton.le-conv})$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma pre :

shows $\text{spec.pre } P \in \text{spec.term.closed}$ -

by $(\text{rule spec.term.closed-upper}[\text{of spec.pre } P, \text{simplified spec.term.all.pre}])$

lemma $none\text{-inf-pre}$:

fixes $P :: 's \text{ pred}$

fixes $Q :: ('a, 's, 'v) \text{ spec}$

shows $\text{spec.term.none } (Q \sqcap \text{spec.pre } P) = (\text{spec.term.none } Q \sqcap \text{spec.pre } P :: ('a, 's, 'w) \text{ spec})$ (**is** $?lhs = ?rhs$)
and $\text{spec.term.none } (\text{spec.pre } P \sqcap Q) = (\text{spec.pre } P \sqcap \text{spec.term.none } Q :: ('a, 's, 'w) \text{ spec})$ (**is** $?thesis2$)

proof –

show $?lhs = ?rhs$
apply ($\text{subst spec.term.none-all[symmetric]}$)
apply ($\text{subst spec.term.all.inf}$)
apply ($\text{subst spec.term.closed.none-inf-monomorphic}(2)[\text{symmetric}]$)
apply ($\text{simp-all add: spec.term.all.pre spec.term.closed.pre}$)
done
then show $?thesis2$
by ($\text{simp add: ac-simps}$)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path pre} \rangle$

lemma bot[iff] :

shows $\text{spec.pre } \langle \text{False} \rangle = \perp$
and $\text{spec.pre } \perp = \perp$
by ($\text{simp-all add: spec.pre-def}$)

lemma top[iff] :

shows $\text{spec.pre } \langle \text{True} \rangle = \top$
and $\text{spec.pre } \top = \top$
by ($\text{simp-all add: spec.pre-def full-SetCompr-eq spec.singleton.top}$)

lemma top-conv :

shows $\text{spec.pre } P = (\top :: ('a, 's, 'v) \text{ spec}) \longleftrightarrow P = \top$
by ($\text{auto intro: iffDI[OF spec.idle.pre-le-conv[where 'a='a and 's='s and 'v='v]]}$)

lemma K :

shows $\text{spec.pre } \langle P \rangle = (\text{if } P \text{ then } \top \text{ else } \perp)$
by ($\text{simp add: spec.pre-def full-SetCompr-eq spec.singleton.top}$)

lemma monotone :

shows mono spec.pre
by ($\text{fastforce simp: spec.pre-def intro: monoI}$)

lemmas $\text{strengthen[strg]} = \text{st-monotone[OF spec.pre.monotone]}$

lemmas $\text{mono} = \text{monotoneD[OF spec.pre.monotone]}$

lemma SUP :

shows $\text{spec.pre } (\bigsqcup x \in X. P x) = (\bigsqcup x \in X. \text{spec.pre } (P x))$
by ($\text{auto simp: spec.pre-def spec.eq-iff intro: rev-SUPI}$)

lemma Sup :

shows $\text{spec.pre } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.pre } x)$
by ($\text{metis image-ident image-image spec.pre.SUP}$)

lemma Bex :

shows $\text{spec.pre } (\lambda s. \exists x \in X. P x s) = (\bigsqcup x \in X. \text{spec.pre } (P x))$
by ($\text{simp add: Sup-fun-def flip: spec.pre.SUP}$)

lemma Ex :

shows $\text{spec.pre } (\lambda s. \exists x. P x s) = (\bigsqcup x. \text{spec.pre } (P x))$

by (*simp add: Sup-fun-def flip: spec.pre.SUP*)

lemma

shows *disj: spec.pre* $(P \vee Q) = \text{spec.pre } P \sqcup \text{spec.pre } Q$

and *sup: spec.pre* $(P \sqcup Q) = \text{spec.pre } P \sqcup \text{spec.pre } Q$

using *spec.pre.Sup*[**where** $X=\{P, Q\}$] **by** (*simp-all add: sup-fun-def*)

lemma *INF:*

shows *spec.pre* $(\prod_{x \in X}. P x) = (\prod_{x \in X}. \text{spec.pre } (P x))$

by (*auto simp: spec.eq-iff spec.singleton.pre-le-conv le-INF-iff intro: spec.singleton-le-extI*)

lemma *Inf:*

shows *spec.pre* $(\prod X) = (\prod_{x \in X}. \text{spec.pre } x)$

by (*metis image-ident image-image spec.pre.INF*)

lemma *Ball:*

shows *spec.pre* $(\lambda s. \forall x \in X. P x s) = (\prod_{x \in X}. \text{spec.pre } (P x))$

by (*simp add: Inf-fun-def flip: spec.pre.INF*)

lemma *All:*

shows *spec.pre* $(\lambda s. \forall x. P x s) = (\prod x. \text{spec.pre } (P x))$

by (*simp add: Inf-fun-def flip: spec.pre.INF*)

lemma *inf:*

shows *conj: spec.pre* $(P \wedge Q) = \text{spec.pre } P \sqcap \text{spec.pre } Q$

and *spec.pre* $(P \sqcap Q) = \text{spec.pre } P \sqcap \text{spec.pre } Q$

using *spec.pre.Inf*[**where** $X=\{P, Q\}$] **by** (*simp-all add: inf-fun-def*)

lemma *inf-action-le:* — Converse does not hold

shows *spec.pre* $P \sqcap \text{spec.action } F \leq \text{spec.action } (UNIV \times UNIV \times \text{Collect } P \times UNIV \cap F)$ (**is** *?lhs ≤ ?rhs*)

and *spec.action* $F \sqcap \text{spec.pre } P \leq \text{spec.action } (F \cap UNIV \times UNIV \times \text{Collect } P \times UNIV)$ (**is** *?thesis2*)

proof –

show *?lhs ≤ ?rhs*

proof(*rule spec.singleton-le-extI*)

show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ

using *that*[*simplified, unfolded spec.singleton.action-le-conv spec.singleton.le-conv*]

by (*cases* σ ;

safe; clarsimp simp: trace.steps'.step-conv spec.action.idleI spec.action.stutter-stepsI

split: option.split-asm;

subst spec.singleton.Cons; blast intro: spec.action.stepI)

qed

then show *?thesis2*

by (*simp add: ac-simps*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *pre:*

shows *spec.invmmap af sf vf* $(\text{spec.pre } P) = \text{spec.pre } (\lambda s. P (sf s))$

by (*rule spec.singleton.antisym*) (*simp-all add: spec.singleton.le-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemma *inf-pre:*

shows $spec.pre P \sqcap (f \ggg g) = (spec.pre P \sqcap f) \ggg g$ (**is** ?lhs = ?rhs)
and $(f \ggg g) \sqcap spec.pre P = (f \sqcap spec.pre P) \ggg g$ (**is** ?thesis1)
proof –
show ?lhs = ?rhs
proof(rule antisym)
show ?lhs \leq ?rhs
by (fastforce simp: spec.bind-def spec.continue-def inf-sup-distrib1 inf-Sup spec.singleton.inf-pre
simp flip: spec.term.closed.none-inf-pre spec.singleton.pre-le-conv)
show ?rhs \leq ?lhs
by (fastforce simp: spec.bind-def spec.continue-def inf-sup-distrib1 spec.singleton.pre-le-conv
simp flip: spec.term.closed.none-inf-pre)
qed
then show ?thesis1
by (simp add: ac-simps)
qed

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path steps \rangle$

lemma pre:

assumes $P s_0$

shows $spec.steps (spec.pre P :: ('a, 's, 'v) spec) = UNIV \times - Id$

proof –

have $(a, s, s') \in spec.steps (spec.pre P)$ **if** $s \neq s'$ **for** $a :: 'a$ **and** $s s'$

using *assms that*

by (simp add: spec.singleton.le-conv spec.steps.member-conv trace.steps'.Cons-eq-if
exI[**where** $x=trace.T s_0 [(undefined, s), (a, s')] None]$)

then show ?thesis

by *auto*

qed

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

8.11.2 Postconditions

Unlike $spec.pre$ $spec.post$ can be expressed in terms of other constants.

setup $\langle Sign.mandatory-path spec \rangle$

setup $\langle Sign.mandatory-path post \rangle$

definition $act :: ('v \Rightarrow 's pred) \Rightarrow ('v \times 'a \times 's \times 's) set$ **where**

$act Q = \{(v, a, s, s') \mid v a s s'. Q v s'\}$

setup $\langle Sign.mandatory-path act \rangle$

lemma $simps[simp]$:

shows $spec.post.act \langle \langle False \rangle \rangle = \{\}$

and $spec.post.act \langle \perp \rangle = \{\}$

and $spec.post.act \perp = \{\}$

and $spec.post.act \langle \langle True \rangle \rangle = UNIV$

and $spec.post.act \langle \top \rangle = UNIV$

and $spec.post.act \top = UNIV$

and $spec.post.act (Q \sqcup Q') = spec.post.act Q \cup spec.post.act Q'$

and $spec.post.act (\bigsqcup X) = (\bigcup x \in X. spec.post.act x)$

and $spec.post.act (\lambda v. \bigsqcup x \in Y. R x v) = (\bigcup x \in Y. spec.post.act (R x))$

by (auto 0 2 simp: spec.post.act-def)

lemma monotone:

shows mono spec.post.act

proof(rule monotoneI)

show spec.post.act $Q \leq$ spec.post.act Q' if $Q \leq Q'$ for $Q Q' :: 'v \Rightarrow 's$ pred

using that unfolding spec.post.act-def by blast

qed

lemmas strengthen[strg] = st-monotone[OF spec.post.act.monotone]

lemmas mono = monotoneD[OF spec.post.act.monotone]

setup <Sign.parent-path>

setup <Sign.parent-path>

definition post :: ('v \Rightarrow 's pred) \Rightarrow ('a, 's, 'v) spec **where**

post $Q = \top \gg (\lambda::unit. \text{spec.action } (\text{spec.post.act } Q))$

setup <Sign.mandatory-path singleton>

lemma post-le-conv[spec.singleton.le-conv]:

fixes $Q :: 'v \Rightarrow 's$ pred

shows $\langle \sigma \rangle \leq$ spec.post Q

\longleftrightarrow (case trace.term σ of None \Rightarrow True | Some $v \Rightarrow Q v$ (trace.final σ)) (is ?lhs \longleftrightarrow ?rhs)

proof(rule iffI)

show ?lhs \Longrightarrow ?rhs

by (fastforce simp: spec.post-def spec.post.act-def

spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv

split: option.split

elim: spec.singleton.bind-le)

show ?rhs \Longrightarrow ?lhs

by (cases σ)

(simp add: spec.post-def spec.post.act-def spec.action.stutterI

spec.bind.incompleteI spec.bind.continueI[**where** ys=[], simplified]

split: option.splits)

qed

setup <Sign.parent-path>

setup <Sign.mandatory-path idle>

lemma post-le[spec.idle-le]:

shows spec.idle \leq spec.post Q

by (rule spec.singleton-le-extI) (simp add: spec.singleton.le-conv)

setup <Sign.parent-path>

setup <Sign.mandatory-path term>

setup <Sign.mandatory-path none>

lemma post-le:

shows spec.term.none $P \leq$ spec.post Q

by (rule spec.singleton-le-extI) (simp add: spec.singleton.le-conv)

lemma post:

shows spec.term.none (spec.post $Q :: ('a, 's, 'v)$ spec)

$= \text{spec.term.none } (\top :: ('a, 's, \text{unit}) \text{ spec})$
by (*metis spec.eq-iff spec.term.galois spec.term.none.post-le spec.term.none-all top-greatest*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *post*:

shows $\text{spec.term.all } (\text{spec.post } Q) = \top$
by (*metis spec.term.all-none spec.term.none.post spec.term.upper-top*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path post} \rangle$

lemma *bot[iff]*:

shows $\text{spec.post } \langle \langle \text{False} \rangle \rangle = \text{spec.term.none } (\top :: (-, -, \text{unit}) \text{ spec})$
and $\text{spec.post } \langle \perp \rangle = \text{spec.term.none } (\top :: (-, -, \text{unit}) \text{ spec})$
and $\text{spec.post } \perp = \text{spec.term.none } (\top :: (-, -, \text{unit}) \text{ spec})$
by (*simp-all add: spec.post-def spec.action.empty spec.bind.idleR spec.bind.botR*)

lemma *monotone*:

shows *mono spec.post*
by (*simp add: spec.post-def monoI spec.action.mono spec.bind.mono spec.post.act.mono*)

lemmas *strengthen[strg] = st-monotone[OF spec.post.monotone]*

lemmas *mono = monotoneD[OF spec.post.monotone]*

lemma *SUP-not-empty*:

fixes $X :: 'a \text{ set}$
fixes $Q :: 'a \Rightarrow 'v \Rightarrow 's \text{ pred}$
assumes $X \neq \{\}$
shows $\text{spec.post } (\lambda v. \bigsqcup_{x \in X}. Q \ x \ v) = (\bigsqcup_{x \in X}. \text{spec.post } (Q \ x))$
by (*simp add: assms spec.post-def flip: spec.bind.SUPR-not-empty spec.action.SUP-not-empty*)

lemma *disj*:

shows $\text{spec.post } (Q \sqcup Q') = \text{spec.post } Q \sqcup \text{spec.post } Q'$
and $\text{spec.post } (\lambda rv. Q \ rv \sqcup Q' \ rv) = \text{spec.post } Q \sqcup \text{spec.post } Q'$
and $\text{spec.post } (\lambda rv. Q \ rv \vee Q' \ rv) = \text{spec.post } Q \sqcup \text{spec.post } Q'$
using *spec.post.SUP-not-empty[where X=UNIV and Q= $\lambda x. \text{if } x \text{ then } Q' \text{ else } Q$]*
by (*simp-all add: UNIV-bool sup-fun-def*)

lemma *INF*:

shows $\text{spec.post } (\prod_{x \in X}. Q \ x) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
and $\text{spec.post } (\lambda v. \prod_{x \in X}. Q \ x \ v) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
and $\text{spec.post } (\lambda v \ s. \prod_{x \in X}. Q \ x \ v \ s) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
by (*fastforce intro: antisym spec.singleton-le-extI simp: spec.singleton.le-conv le-Inf-iff split: option.split*)⁺

lemma *Inf*:

shows $\text{spec.post } (\prod X) = (\prod_{x \in X}. \text{spec.post } x)$
by (*metis image-ident image-image spec.post.INF(1)*)

lemma *Ball*:

shows $\text{spec.post } (\lambda v \ s. \forall x \in X. Q \ x \ v \ s) = (\prod_{x \in X}. \text{spec.post } (Q \ x))$
by (*simp add: Inf-fun-def flip: spec.post.INF*)

lemma *All*:

shows $\text{spec.post } (\lambda v s. \forall x. Q x v s) = (\prod x. \text{spec.post } (Q x))$
by (*simp add: Inf-fun-def flip: spec.post.INF*)

lemma *inf*:

shows $\text{spec.post } (Q \sqcap Q') = \text{spec.post } Q \sqcap \text{spec.post } Q'$
and $\text{spec.post } (\lambda rv. Q rv \sqcap Q' rv) = \text{spec.post } Q \sqcap \text{spec.post } Q'$
and *conj*: $\text{spec.post } (\lambda rv. Q rv \wedge Q' rv) = \text{spec.post } Q \sqcap \text{spec.post } Q'$
by (*simp-all add: inf-fun-def flip: spec.post.INF[where X=UNIV and Q= $\lambda x. \text{if } x \text{ then } Q' \text{ else } Q$, simplified UNIV-bool, simplified]*)

lemma *top[iff]*:

shows $\text{spec.post } \langle\langle \text{True} \rangle\rangle = \top$
and $\text{spec.post } \langle\top\rangle = \top$
and $\text{spec.post } \top = \top$
by (*simp-all add: top-fun-def flip: spec.post.INF[where X={}, simplified]*)

lemma *top-conv*:

shows $\text{spec.post } Q = (\top :: ('a, 's, 'v) \text{spec}) \longleftrightarrow Q = \top$
by (*fastforce simp: spec.singleton.post-le-conv*
dest: arg-cong[where f= $\lambda x. \forall \sigma. \langle\sigma\rangle \leq x$] spec[where x= $\text{trace.T } s \ []$ (Some v) for s v]))

lemma *K*:

shows $\text{spec.post } (\lambda - . Q) = (\text{if } Q \text{ then } \top \text{ else } \top \ggg (\lambda :: \text{unit. } \perp))$
by (*auto simp flip: spec.bind.botR bot-fun-def*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *bind-post-pre*:

shows $f \sqcap \text{spec.post } Q \ggg g = f \ggg (\lambda v. g v \sqcap \text{spec.pre } (Q v))$ (*is ?lhs = ?rhs*)
and $\text{spec.post } Q \sqcap f \ggg g = f \ggg (\lambda v. \text{spec.pre } (Q v) \sqcap g v)$ (*is ?thesis1*)

proof –

show *?lhs = ?rhs*
proof(*rule spec.singleton.antisym*)
show $\langle\sigma\rangle \leq ?rhs$ **if** $\langle\sigma\rangle \leq ?lhs$ **for** σ
using *that*
by (*induct rule: spec.singleton.bind-le*)
(cases σ ; force simp: trace.split-all spec.singleton.le-conv
intro: spec.bind.incompleteI spec.bind.continueI)+
show $\langle\sigma\rangle \leq ?lhs$ **if** $\langle\sigma\rangle \leq ?rhs$ **for** σ
using *that*
by (*induct rule: spec.singleton.bind-le*)
(cases σ ; force simp: trace.split-all spec.singleton.le-conv
intro: spec.bind.incompleteI spec.bind.continueI)+

qed

then show *?thesis1*
by (*simp add: ac-simps*)

qed

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *post*:

shows $\text{spec.invmap } af \ sf \ vf \ (\text{spec.post } Q) = \text{spec.post } (\lambda v s. Q (vf v) (sf s))$
by (*rule antisym[OF spec.singleton-le-extI spec.singleton-le-extI]*)
(simp-all add: spec.singleton.invmap-le-conv spec.singleton.post-le-conv trace.final'.map split: option.split-asm)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma *post-le-conv*:

shows $\text{spec.action } F \leq \text{spec.post } Q \iff (\forall v a s s'. (v, a, s, s') \in F \longrightarrow Q v s')$
by (*fastforce simp: spec.action-def split-def spec.singleton.le-conv spec.idle.post-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemma *post-le*:

assumes $\bigwedge v. g v \leq \text{spec.post } Q$
shows $f \ggg g \leq \text{spec.post } Q$
by (*strengthen ord-to-strengthen(1)[OF assms]*)
(*simp add: spec.post-def spec.bind.mono flip: spec.bind.bind*)

lemma *inf-post*:

shows $(f \ggg g) \sqcap \text{spec.post } Q = f \ggg (\lambda v. g v \sqcap \text{spec.post } Q)$ (**is** *?lhs = ?rhs*)
and $\text{spec.post } Q \sqcap (f \ggg g) = f \ggg (\lambda v. \text{spec.post } Q \sqcap g v)$ (**is** *?thesis2*)

proof –

show *?lhs = ?rhs*
proof(*rule antisym[OF spec.singleton-le-extI]*)
fix σ
assume $\text{lhs}: \langle \sigma \rangle \leq \text{?lhs}$
from $\text{lhs}[\text{simplified}, \text{THEN } \text{conjunct1}] \text{lhs}[\text{simplified}, \text{THEN } \text{conjunct2}]$ **show** $\langle \sigma \rangle \leq \text{?rhs}$
proof(*induct rule: spec.singleton.bind-le*)
case (*continue* $\sigma_f \sigma_g v_f$) **then show** *?case*
by (*cases* σ_f)
(*force intro: spec.bind.continueI simp: spec.singleton.le-conv split: option.split*)
qed (*simp add: spec.singleton.bind-le-conv*)
qed (*simp add: spec.bind.mono spec.bind.post-le*)
then show *?thesis2*
by (*simp add: ac-simps*)
qed

lemma *mono-stronger*:

assumes $f: f \leq f' \sqcap \text{spec.post } Q$
assumes $g: \bigwedge v. g v \sqcap \text{spec.pre } (Q v) \leq g' v$
shows $\text{spec.bind } f g \leq \text{spec.bind } f' g'$
by (*strengthen ord-to-strengthen(1)[OF f]*) (*simp add: g spec.bind.mono spec.bind-post-pre*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.11.3 Strongest postconditions

setup $\langle \text{Sign.mandatory-path spec.post} \rangle$

definition *strongest* :: $(\text{'a}, \text{'s}, \text{'v}) \text{spec} \Rightarrow \text{'v} \Rightarrow \text{'s} \text{pred}$ **where**
 $\text{strongest } P = \bigsqcap \{ Q. P \leq \text{spec.post } Q \}$

interpretation *strongest*: *galois.complete-lattice-class spec.post.strongest spec.post*

by (*simp add: spec.post.strongest-def galois.upper-preserves-InfI spec.post.Inf spec.post.monotone*)

lemma *strongest-alt-def*:

shows $\text{spec.post.strongest } P = (\lambda v s. \exists \sigma. \langle \sigma \rangle \leq P \wedge \text{trace.term } \sigma = \text{Some } v \wedge \text{trace.final } \sigma = s)$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*rule spec.singleton.exhaust[of P]*)
(*fastforce simp: spec.post.strongest-def spec.singleton.le-conv*
dest: spec[where $x=\lambda v s. \exists \sigma \in X. trace.term \sigma = Some v \wedge trace.final \sigma = s$ for X]
split: option.split)
show $?rhs \leq ?lhs$
by (*fastforce simp: spec.post.strongest-def spec.singleton.le-conv dest: order.trans*)
qed

setup $\langle Sign.mandatory-path strongest \rangle$

lemma *singleton*:

shows *spec.post.strongest* $\langle \sigma \rangle$
 $= (\lambda v s. case trace.term \sigma of None \Rightarrow False \mid Some v' \Rightarrow v' = v \wedge trace.final \sigma = s)$
by (*auto simp: spec.post.strongest-alt-def fun-eq-iff trace.split-all*
cong: trace.final'.natural'-cong
split: option.split)

lemmas *monotone* = *spec.post.strongest.monotone-lower*

lemmas *mono* = *monoD[OF spec.post.strongest.monotone]*

lemmas *Sup* = *spec.post.strongest.lower-Sup*

lemmas *sup* = *spec.post.strongest.lower-sup*

lemma *top[iff]*:

shows *spec.post.strongest* $\top = \top$
by (*simp add: spec.post.strongest-def spec.post.top-conv top.extremum-unique*)

lemma *action*:

shows *spec.post.strongest* (*spec.action F*) = $(\lambda v s'. \exists a s. (v, a, s, s') \in F)$
by (*simp add: spec.post.strongest-def spec.action.post-le-conv*) *fast*

lemma *return*:

shows *spec.post.strongest* (*spec.return v*) = $(\lambda v' s. v' = v)$
by (*simp add: spec.return-def spec.post.strongest.action*)

setup $\langle Sign.mandatory-path term \rangle$

lemma *none*:

shows *spec.post.strongest* (*spec.term.none P*) = \perp
by (*clarsimp simp: spec.term.none-def spec.post.strongest.Sup spec.post.strongest.singleton fun-eq-iff*)

lemma *all*:

assumes *spec.idle* $\leq P$
shows *spec.post.strongest* (*spec.term.all P*) = \top
by (*rule top-le[OF order.trans[OF - spec.post.strongest.mono[OF spec.term.all.mono[OF assms]]]]*)
(*simp add: spec.term.all.idle spec.post.strongest.Sup spec.post.strongest.return Sup-fun-def top-fun-def*)

lemma *closed*:

assumes *spec.idle* $\leq P$
assumes $P \in spec.term.closed$ -
shows *spec.post.strongest* $P = \top$
by (*metis spec.post.strongest.term.all[OF assms(1)] spec.term.all.closed-conv[OF assms(2)]*)

setup $\langle Sign.parent-path \rangle$

lemma *bind*:

shows *spec.post.strongest* ($f \ggg g$)

$= \text{spec.post.strongest } (\sqcup v. \text{spec.pre } (\text{spec.post.strongest } f v) \sqcap g v) \text{ (is ?lhs = ?rhs)}$
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*auto 0 4 simp: spec.post.strongest-alt-def spec.singleton.le-conv*
elim!: spec.singleton.bind-le)
show $?rhs \leq ?lhs$
by (*force simp: spec.post.strongest-alt-def spec.singleton.le-conv trace.split-all*
dest: spec.bind.continueI)
qed

lemma rel:
shows $\text{spec.post.strongest } (\text{spec.rel } r) = \top$
by (*simp add: spec.rel-def spec.post.strongest.term.all spec.idle.kleene.star-le*)

lemma pre:
shows $\text{spec.post.strongest } (\text{spec.pre } P) = (\lambda v s'. \exists s. P s)$
by (*auto simp: spec.post.strongest-alt-def spec.singleton.le-conv trace.split-Ex fun-eq-iff*
intro!: exI[where x=[(undefined, s)] for s])

lemma post:
shows $\text{spec.post.strongest } (\text{spec.post } Q) = Q$
by (*auto simp: spec.post.strongest-alt-def spec.singleton.le-conv trace.split-Ex fun-eq-iff*
intro!: exI[where x=[]])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.12 Initial steps

The initial transition of a process.

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition *initial-steps* :: $(\prime a, \prime s, \prime v) \text{ spec} \Rightarrow (\prime a, \prime s) \text{ steps}$ **where**
initial-steps $P = \{(a, s, s'). \downarrow s, [(a, s)], \text{None}\} \leq P$

setup $\langle \text{Sign.mandatory-path initial-steps} \rangle$

lemma steps-le:
shows $\text{spec.initial-steps } P \subseteq \text{spec.steps } P \cup \text{UNIV} \times \text{Id}$
by (*fastforce simp: spec.initial-steps-def spec.steps.member-conv split: if-splits*)

lemma galois:
shows $r \subseteq \text{spec.initial-steps } P \wedge \text{spec.idle} \leq P \iff \text{spec.action } (\{()\} \times r) \ggg \perp \leq P$ **(is ?lhs \iff ?rhs)**
proof(*rule iffI*)

show $?lhs \implies ?rhs$
by (*auto simp: spec.action-def spec.initial-steps-def spec.bind.SupL spec.bind.supL*
spec.bind.singletonL spec.singleton.not-bot spec.term.none.singleton)
show $?rhs \implies ?lhs$
by (*auto simp: spec.initial-steps-def spec.idle.action-le spec.idle.bind-le-conv*
elim!: order.trans[rotated]
intro: spec.action.stepI spec.bind.incompleteI)

qed

lemma bot:
shows $\text{spec.initial-steps } \perp = \{\}$
by (*auto simp: spec.initial-steps-def spec.singleton.not-bot*)

lemma *top*:

shows $\text{spec.initial-steps } \top = \text{UNIV}$

by (*auto simp: spec.initial-steps-def spec.singleton.not-bot*)

lemma *monotone*:

shows $\text{mono spec.initial-steps}$

by (*force intro: monoI simp: spec.initial-steps-def*)

lemmas $\text{strengthen}[strg] = \text{st-monotone}[OF \text{ spec.initial-steps.monotone}]$

lemmas $\text{mono} = \text{monotoneD}[OF \text{ spec.initial-steps.monotone}]$

lemma *Sup*:

shows $\text{spec.initial-steps } (\bigsqcup X) = \bigcup (\text{spec.initial-steps } ` X)$

by (*auto simp: spec.initial-steps-def*)

lemma *Inf*:

shows $\text{spec.initial-steps } (\bigsqcap X) = \bigcap (\text{spec.initial-steps } ` X)$

by (*auto simp: spec.initial-steps-def le-Inf-iff*)

lemma *idle*:

shows $\text{spec.initial-steps spec.idle} = \text{UNIV} \times \text{Id}$

by (*auto simp: spec.initial-steps-def spec.singleton.le-conv*)

lemma *action*:

shows $\text{spec.initial-steps } (\text{spec.action } F) = \text{snd } ` F \cup \text{UNIV} \times \text{Id}$

by (*auto simp: spec.initial-steps-def spec.singleton.action-le-conv trace.steps'.step-conv*)

lemma *return*:

shows $\text{spec.initial-steps } (\text{spec.return } v) = \text{UNIV} \times \text{Id}$

by (*auto simp: spec.initial-steps-def spec.singleton.le-conv*)

lemma *bind*:

shows $\text{spec.initial-steps } (f \gg= g)$

$= \text{spec.initial-steps } f$

$\cup \text{spec.initial-steps } (\bigsqcup v. \text{spec.pre } (\text{spec.post.strongest } (f \sqcap \text{spec.return } v) v) \sqcap g v)$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

show *?lhs* \leq *?rhs*

by (*fastforce simp: spec.initial-steps-def spec.post.strongest-alt-def spec.singleton.le-conv*

trace.split-all Cons-eq-append-conv trace.natural.simps

elim!: *spec.singleton.bind-le*)

show *?rhs* \leq *?lhs*

by (*auto simp: spec.initial-steps-def spec.post.strongest-alt-def spec.singleton.le-conv*

trace.split-all spec.bind.incompleteI order.trans[OF - spec.bind.continueI, rotated])

qed

lemma *rel*:

shows $\text{spec.initial-steps } (\text{spec.rel } r) = r \cup \text{UNIV} \times \text{Id}$

by (*auto simp: spec.initial-steps-def spec.singleton.le-conv*)

lemma *pre*:

shows $\text{spec.initial-steps } (\text{spec.pre } P) = \text{UNIV} \times \text{Pre } P$

by (*auto simp: spec.initial-steps-def spec.singleton.le-conv*)

lemma *post*:

shows $\text{spec.initial-steps } (\text{spec.post } Q) = \text{UNIV}$

by (*auto simp: spec.initial-steps-def spec.singleton.le-conv*)

setup $\langle \text{Sign.mandatory-path term} \rangle$

lemma *none*:

shows *spec.initial-steps* (*spec.term.none* P) = *spec.initial-steps* P
by (*auto simp: spec.initial-steps-def spec.singleton.le-conv*)

lemma *all*:

shows *spec.initial-steps* (*spec.term.all* P) = *spec.initial-steps* P
by (*auto simp: spec.initial-steps-def spec.singleton.le-conv order.trans[rotated]*
spec.singleton.mono trace.less-eq-None)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.parent-path* \rangle

8.13 Heyting implication

setup \langle *Sign.mandatory-path spec* \rangle

setup \langle *Sign.mandatory-path singleton* \rangle

lemma *heyting-le-conv*:

shows $\langle\sigma\rangle \leq P \longrightarrow_H Q \longleftrightarrow (\forall \sigma' \leq \sigma. \langle\sigma'\rangle \leq P \longrightarrow \langle\sigma'\rangle \leq Q)$ (**is** *?lhs* \longleftrightarrow *?rhs*)
proof(*rule iffI*)

show *?lhs* \implies *?rhs*

by (*meson order.trans heyting.mp spec.singleton.mono*)

show *?rhs* \implies *?lhs*

by (*rule spec.singleton.exhaust[of P]*)

(*clarsimp simp: heyting inf-Sup spec.singleton.inf trace.less-eq-take-def spec.singleton-le-conv;*
metis spec.singleton.simps(1) trace.take.naturalE(2))

qed

setup \langle *Sign.parent-path* \rangle

Connect the generic definition of Heyting implication to a concrete one in the model.

lift-definition *heyting* :: $(\prime a, \prime s, \prime v)$ *spec* \Rightarrow $(\prime a, \prime s, \prime v)$ *spec* \Rightarrow $(\prime a, \prime s, \prime v)$ *spec* **is**
downwards.imp

by (*rule raw.spec.closed.downwards-imp*)

lemma *heyting-alt-def*:

shows $(\longrightarrow_H) = (\text{spec.heyting} :: \dashv\vdash \Rightarrow (\prime a, \prime s, \prime v)$ *spec*)

proof –

have $P \leq \text{spec.heyting } Q R \longleftrightarrow P \sqcap Q \leq R$ **for** $P Q R :: (\prime a, \prime s, \prime v)$ *spec*

by *transfer (simp add: raw.spec.closed.heyting-downwards-imp)*

with *heyting show ?thesis by (intro fun-eqI antisym; fast)*

qed

declare *spec.heyting.transfer[transfer-rule del]*

setup \langle *Sign.mandatory-path heyting* \rangle

lemma *transfer-alt[transfer-rule]*:

shows *rel-fun* (*pcr-spec* (=) (=) (=)) (*rel-fun* (*pcr-spec* (=) (=) (=)) (*pcr-spec* (=) (=) (=))) *downwards.imp*
 (\longrightarrow_H)

by (*simp add: spec.heyting.transfer spec.heyting-alt-def*)

An example due to [Abadi and Merz \(1995, p504\)](#) where the (TLA) model validates a theorem that is not intuitionistically valid. This is “some kind of linearity” and intuitively encodes disjunction elimination.

lemma *linearity*:

fixes $Q :: (-, -, -)$ *spec*

shows $((P \longrightarrow_H Q) \longrightarrow_H R) \sqcap ((Q \longrightarrow_H P) \longrightarrow_H R) \leq R$

by *transfer*

(*clarsimp simp: downwards.imp-def*;

meson downwards.closed-in[OF - - raw.spec.closed.downwards-closed] trace.less-eq-same-cases order.refl)

lemma *SupR*:

fixes $P :: (-, -, -)$ *spec*

assumes $X \neq \{\}$

shows $P \longrightarrow_H (\bigsqcup_{x \in X}. Q\ x) = (\bigsqcup_{x \in X}. P \longrightarrow_H Q\ x)$ (**is** *?lhs = ?rhs*)

proof(*rule antisym[OF spec.singleton-le-extI heyting.SUPR-le]*)

show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ

proof(*cases* $\langle \sigma \rangle \leq P$)

case *True* **with** $\langle \sigma \rangle \leq ?lhs$ **show** *?thesis* **by** (*simp add: heyting.inf.absorb1*)

next

case *False* **show** *?thesis*

proof(*cases* $\langle \text{trace.init } \sigma, [], \text{None} \rangle \leq P$)

case *True* **with** $\langle \neg \langle \sigma \rangle \leq P \rangle$

obtain j **where** $\forall i \leq j. \langle \text{trace.take } i\ \sigma \rangle \leq P$

and $\neg \langle \text{trace.take } (\text{Suc } j)\ \sigma \rangle \leq P$

using *ex-least-nat-less*[**where** $P = \lambda i. \neg \langle \text{trace.take } i\ \sigma \rangle \leq P$ **and** $n = \text{Suc } (\text{length } (\text{trace.rest } \sigma))$]

by (*clarsimp simp: less-Suc-eq-le simp flip: trace.take.Ex-all*)

with $\langle \sigma \rangle \leq ?lhs$ **show** *?thesis*

by (*clarsimp simp: spec.singleton.heyting-le-conv dest!: spec*[**where** $x = \text{trace.take } j\ \sigma$]

(*metis not-less-eq-eq order-refl spec.singleton.mono spec.singleton-le-ext-conv*

trace.less-eq-takeE trace.take.mono)

next

case *False* **with** $\langle X \neq \{\} \rangle$ $\langle \sigma \rangle \leq ?lhs$ **show** *?thesis*

by (*clarsimp simp: spec.singleton.heyting-le-conv simp flip: ex-in-conv*)

(*metis trace.take.0 spec.singleton.takeI trace.less-eq-take-def trace.take.sel(1)*)

qed

qed

qed

lemma *cont*:

fixes $P :: (-, -, -)$ *spec*

shows *cont Sup* (\leq) *Sup* (\leq) $((\longrightarrow_H) P)$

by (*rule contI*) (*simp add: spec.heyting.SupR*[**where** $Q = \text{id}$, *simplified*])

lemma *mcont*:

fixes $P :: (-, -, -)$ *spec*

shows *mcont Sup* (\leq) *Sup* (\leq) $((\longrightarrow_H) P)$

by (*simp add: mcontI*[*OF - spec.heyting.cont*])

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF spec.heyting.mcont*, *of luba orda Q P*] **for** *luba orda Q P*

lemma *non-triv*:

shows $P \longrightarrow_H \perp \leq P \longleftrightarrow \text{spec.idle} \leq P$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

by (*rule spec.singleton.exhaust*[*of P*])

(*fastforce dest: spec*[**where** $x = \langle x, [], \text{None} \rangle$ **for** x]

simp: spec.idle-def heyting-def heyting.inf-Sup-distrib trace.split-all

spec.singleton.inf spec.singleton-le-conv trace.less-eq-None trace.natural.simps)

have *spec.idle* $\longrightarrow_H \perp \leq \text{spec.idle}$

by (*fastforce intro: spec.singleton-le-extI*

dest: spec[**where** $x = \text{trace.T } (\text{trace.init } \sigma) [] \text{None}$ **for** σ]

$\text{simp: spec.singleton.le-conv spec.singleton.heyting-le-conv spec.singleton.not-bot trace.less-eq-None}$
then show $?lhs$ **if** $?rhs$
by $-$ ($\text{strengthen ord-to-strengthen}(2)[OF \langle ?rhs \rangle]$)
qed

lemma *post*:

shows $\text{spec.post } Q \longrightarrow_H \text{spec.post } Q' = \text{spec.post } (\lambda v s. Q v s \longrightarrow Q' v s)$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

by ($\text{auto intro: spec.singleton-le-extI simp: spec.singleton.heyting-le-conv spec.singleton.le-conv split: option.splits}$)

show $?rhs \leq ?lhs$

by ($\text{auto simp add: heyting simp flip: spec.post.conj intro: spec.post.mono}$)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *heyting*:

shows $\text{spec.invmap af sf vf } (P \longrightarrow_H Q) = \text{spec.invmap af sf vf } P \longrightarrow_H \text{spec.invmap af sf vf } Q$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$ **by** ($\text{simp add: heyting heyting.detachment spec.invmap.mono flip: spec.invmap.inf}$)

show $?rhs \leq ?lhs$

by ($\text{simp add: heyting heyting.detachment spec.map.inf-distr flip: spec.map-invmap.galois spec.invmap.inf}$)
 $(\text{simp add: spec.invmap.mono spec.map-invmap.galois})$

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

lemma *heyting-noneL-allR-mono*:

fixes $P :: (-, -, 'v)$ *spec*

fixes $Q :: (-, -, 'v)$ *spec*

shows $\text{spec.term.none } P \longrightarrow_H Q = P \longrightarrow_H \text{spec.term.all } Q$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

by ($\text{simp add: heyting spec.term.none.inf flip: spec.term.galois}$) ($\text{simp add: heyting.uncurry}$)

show $?rhs \leq ?lhs$

by ($\text{simp add: heyting heyting.discharge spec.term.closed.none-inf-monomorphic spec.term.galois}$)

qed

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *heyting*: — polymorphic *spec.term.all*

fixes $P :: (-, -, 'v)$ *spec*

fixes $Q :: (-, -, 'v)$ *spec*

shows $(\text{spec.term.all } (P \longrightarrow_H Q)) :: (-, -, 'w)$ *spec*

$= \text{spec.term.none } P \longrightarrow_H \text{spec.term.all } Q$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

by ($\text{simp add: heyting spec.term.none.inf flip: spec.term.galois}$)

$(\text{metis heyting.detachment}(2) \text{ le-inf-iff spec.term.none.contractive spec.term.none.inf}(2))$

have $\text{spec.term.none } (\text{spec.term.none } P \longrightarrow_H \text{spec.term.all } Q :: (-, -, 'w)$ *spec*)

$\sqcap \text{spec.term.none } (\text{spec.term.none } P :: (-, -, 'w)$ *spec*)

$\leq Q$

by ($\text{metis heyting.detachment}(2) \text{ inf-sup-ord}(2) \text{ spec.term.galois spec.term.none.inf}(2)$)

then show $?rhs \leq ?lhs$
by (*simp add: heyting flip: spec.term.galois*)
(metis spec.term.cl-def spec.term.all.monomorphic spec.term.none-all
heyting.detachment(2) spec.term.heyting-noneL-allR-mono)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

lemma *heyting-le:*
shows $\text{spec.term.none } (P \longrightarrow_H Q) \leq \text{spec.term.all } P \longrightarrow_H \text{spec.term.none } Q$
by (*simp add: spec.term.galois spec.term.all.heyting heyting.mono spec.term.all.expansive*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *heyting:*
assumes $Q \in \text{spec.term.closed}$ -
shows $P \longrightarrow_H Q \in \text{spec.term.closed}$ -
by (*rule spec.term.closed-clI*)
(simp add: spec.term.all.heyting spec.term.heyting-noneL-allR-mono spec.term.all.monomorphic
flip: spec.term.all.closed-conv[OF assms])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

8.14 Miscellaneous algebra

setup $\langle \text{Sign.mandatory-path spec} \rangle$

setup $\langle \text{Sign.mandatory-path steps} \rangle$

lemma *bind:*
shows $\text{spec.steps } (f \ggg g)$
 $= \text{spec.steps } f \cup (\bigcup v. \text{spec.steps } (\text{spec.pre } (\text{spec.post.strongest } f \ v) \sqcap g \ v))$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)
show $?lhs \leq ?rhs$
unfolding *spec.rel.galois*
by (*rule spec.singleton-le-extI*)
(fastforce elim!: spec.singleton.bind-le
simp: spec.singleton.le-conv spec.steps.member-conv trace.steps'.append
spec.post.strongest-alt-def)

show $?rhs \leq ?lhs$
by (*fastforce simp: spec.post.strongest-alt-def spec.bind-def spec.continue-def*
spec.steps.term.none spec.steps.Sup spec.steps.sup spec.steps.singleton
spec.steps.member-conv spec.singleton.le-conv trace.split-Ex trace.steps'.append)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path map} \rangle$

lemma *idle:*

shows $spec.map\ af\ sf\ vf\ spec.idle = spec.pre\ (\lambda s. s \in range\ sf) \sqcap spec.idle$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*auto simp: spec.idle-def spec.map.Sup spec.map.singleton spec.singleton.pre-le-conv*)
show $?rhs \leq ?lhs$
by (*auto simp: spec.idle-def spec.pre-def trace.split-all image-image inf-Sup Sup-inf spec.map.Sup spec.map.singleton spec.singleton.inf elim!: trace.less-eqE*)

qed

lemma *return:*

fixes $F :: ('v \times 'a \times 's \times 's)\ set$
shows $spec.map\ af\ sf\ vf\ (spec.return\ v)$
 $= spec.pre\ (\lambda s. s \in range\ sf) \sqcap spec.return\ (vf\ v)$ (**is** $?lhs = ?rhs$)

proof(*rule antisym[OF - spec.singleton-le-extI]*)

show $?lhs \leq ?rhs$
by (*force simp: spec.return-def spec.action-def spec.idle-def spec.map.Sup spec.map.sup spec.map.singleton spec.singleton.pre-le-conv*)

fix σ

assume $\langle \sigma \rangle \leq ?rhs$

then obtain s **where** $trace.init\ \sigma = sf\ s$ **by** (*clarsimp simp: spec.singleton.le-conv*)

with $\langle \sigma \rangle \leq ?rhs$ **show** $\langle \sigma \rangle \leq ?lhs$

by (*simp add: spec.singleton.le-conv exI[where $x=trace.T\ s$] (Some v)] spec.singleton-le-conv trace.natural-def trace.less-eq-None flip: trace.natural'.eq-Nil-conv split: option.split-asm*)

qed

lemma *kleene-star-le:*

fixes $P :: ('a, 's, unit)\ spec$
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: unit \Rightarrow unit$
shows $spec.map\ af\ sf\ vf\ (spec.kleene.star\ P) \leq spec.kleene.star\ (spec.map\ af\ sf\ vf\ P)$ (**is** $- \leq ?rhs$)
proof(*induct rule: spec.kleene.star.fixp-induct[where $P=\lambda R. spec.map\ af\ sf\ vf\ (R\ P) \leq ?rhs$, case-names adm bot step]*)

case (*step R*) **show** $?case$

apply (*simp add: spec.map.sup spec.map.return order.trans[OF - spec.kleene.epsilon-star-le]*)

apply (*subst spec.kleene.star.simps*)

apply (*strengthen ord-to-strengthen(1)[OF spec.map.bind-le]*)

apply (*strengthen ord-to-strengthen(1)[OF step]*)

apply (*simp add: fun-unit-id[where $f=vf$]*)

done

qed (*simp-all add: spec.map.bot*)

lemma *rel-le:*

shows $spec.map\ af\ sf\ vf\ (spec.rel\ r) \leq spec.rel\ (map-prod\ af\ (map-prod\ sf\ sf)\ 'r)$
apply (*simp add: spec.rel-def spec.term.none.map flip: spec.term.galois*)
apply (*simp add: spec.rel.act-def flip: spec.term.none.map[where $vf=id$]*)
apply (*strengthen ord-to-strengthen(1)[OF spec.map.kleene-star-le]*)
apply (*strengthen ord-to-strengthen(1)[OF spec.map.action-le]*)
apply (*strengthen ord-to-strengthen(1)[OF spec.term.none.contractive]*)
apply (*auto intro!: monotoneD[OF spec.kleene.monotone-star] spec.action.mono*)
done

General lemmas for $spec.map$ are elusive. We relate it to $spec.rel$, $spec.pre$ and $spec.post$ under a somewhat weak constraint. Intuitively we ask that, for distinct representations (s_0 and s_0') of an abstract state ($sf\ s_0$ where $sf\ s_0' = sf\ s_0$), if agent a can evolve s_0 to s_1 according to r ($(a, s_0, s_1) \in r$) then there is an agent a' where $af\ a' = af$

a that can evolve s_0' to an s_1' which represents the same abstract state ($sf\ s_1' = sf\ s_1$).
All injective sf satisfy this condition.

context

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$

begin

context

fixes $r :: ('a, 's)\ steps$
assumes $step\text{-}cong: \forall a\ s_0\ s_1\ s_0'. (a, s_0, s_1) \in r \wedge sf\ s_1 \neq sf\ s_0 \wedge sf\ s_0' = sf\ s_0$
 $\longrightarrow (\exists a'\ s_1'. af\ a' = af\ a \wedge sf\ s_1' = sf\ s_1 \wedge (a', s_0', s_1') \in r)$

begin

private lemma $map\text{-}relE[consumes\ 1]$:

fixes $xs :: ('b \times 't)\ list$
assumes $trace.steps'\ s\ xs \subseteq map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ 'r$
obtains $(Idle)\ snd\ 'set\ xs \subseteq \{s\}$
 $| (Step)\ s'\ xs'$
where $sf\ s' = s$
and $trace.natural'\ s\ xs = map\ (map\text{-}prod\ af\ sf)\ xs'$
and $trace.steps'\ s'\ xs' \subseteq r$

using $assms$

proof($atomize\text{-}elim, induct\ xs\ rule: prefix\text{-}induct$)

case $(snoc\ xs\ x)\ show\ ?case$
proof($cases\ snd\ x = trace.final'\ s\ xs$)
case $True\ with\ snoc(2,3)\ show\ ?thesis$
by ($fastforce\ simp: trace.steps'.append\ trace.natural'.append$)

next

case $False$
with $snoc(2,3)\ consider$
 $(idle)\ snd\ 'set\ xs \subseteq \{s\}$
 $| (step)\ s'\ xs'$
where $sf\ s' = s$
and $trace.natural'\ s\ xs = map\ (map\text{-}prod\ af\ sf)\ xs'$
and $trace.steps'\ s'\ xs' \subseteq r$
by ($auto\ 0\ 0\ simp: trace.steps'.append$)

then show $?thesis$

proof $cases$

case $idle\ with\ snoc(3)\ show\ ?thesis$
by ($cases\ x$)
 $(clarsimp\ simp: trace.steps'.append\ trace.natural'.append\ Cons\text{-}eq\text{-}map\text{-}conv$
 $\quad simp\ flip: trace.natural'.eq\text{-}Nil\text{-}conv\ ex\text{-}simps$
 $\quad split: if\text{-}splits;$
 $\quad metis)$

next

case $(step\ s\ xs')\ with\ False\ snoc(3)\ step\text{-}cong\ show\ ?thesis$
by ($cases\ x$)
 $(clarsimp\ simp: trace.steps'.append\ trace.natural'.append\ append\text{-}eq\text{-}map\text{-}conv\ Cons\text{-}eq\text{-}map\text{-}conv$
 $\quad simp\ flip: ex\text{-}simps$
 $\quad intro!: exI[\mathbf{where}\ x=s]\ exI[\mathbf{where}\ x=xs'];$
 $\quad metis\ trace.final'.map\ trace.final'.natural')$

qed

qed

qed $simp$

lemma rel :

shows $spec.map\ af\ sf\ vf\ (spec.rel\ r)$
 $=\ spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ 'r)$
 $\sqcap\ spec.pre\ (\lambda s. s \in range\ sf)$
 $\sqcap\ spec.post\ (\lambda v\ s. v \in range\ vf)\ (is\ ?lhs = ?rhs)$

proof(*rule antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*])
show $\langle\sigma\rangle \leq ?rhs$ **if** $\langle\sigma\rangle \leq ?lhs$ **for** σ
proof(*intro le-infI*)
from *that show* $\langle\sigma\rangle \leq spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ 'r)$
by (*force simp: spec.singleton.le-conv spec.steps.singleton trace.steps'.map*
dest: spec.steps.mono)
from *that show* $\langle\sigma\rangle \leq spec.pre\ (\lambda s. s \in range\ sf)$
by (*fastforce simp: spec.singleton.le-conv spec.singleton-le-conv trace.natural-def*
elim: trace.less-eqE)
from *that show* $\langle\sigma\rangle \leq spec.post\ (\lambda v\ s. v \in range\ vf)$
by (*cases* σ) (*clarsimp simp: spec.singleton.le-conv split: option.split*)
qed
show $\langle\sigma\rangle \leq ?lhs$ **if** $\langle\sigma\rangle \leq ?rhs$ **for** σ
using *that*[*simplified, simplified spec.singleton.le-conv, THEN conjunct1*]
that[*simplified, simplified spec.singleton.le-conv, THEN conjunct2*]
proof(*induct rule: map-relE*)
case *Idle* **then show** *?case*
by (*cases* σ)
(*clarsimp simp: spec.singleton.le-conv;*
force simp: trace.natural.idle trace.natural.simps f-inv-into-f order-le-less
split: option.split-asm
intro!: *exI*[**where** $x=trace.T\ s\ []\ (map\text{-}option\ (inv\ vf)\ (trace.term\ \sigma))$ **for** s])
next
case (*Step* $s\ xs$)
from *Step*(1,3,4) *Step*(2)[*symmetric*] **show** *?case*
by (*cases* σ)
(*clarsimp simp: spec.singleton.le-conv f-inv-into-f*[*OF rangeI*] *trace.natural'.natural'*
exI[**where** $x=trace.T\ s\ xs\ (map\text{-}option\ (inv\ vf)\ (trace.term\ \sigma))$]
split: option.split-asm)
qed
qed

lemma *pre*:
fixes $P :: 't\ pred$
shows $spec.map\ af\ sf\ vf\ (spec.pre\ (\lambda s. P\ (sf\ s)))$
 $=\ spec.pre\ (\lambda s. P\ s \wedge s \in range\ sf) \sqcap spec.post\ (\lambda v\ s. s \in range\ sf \longrightarrow v \in range\ vf)$
 $\sqcap\ spec.rel\ (range\ af \times range\ sf \times range\ sf)\ (is\ ?lhs = ?rhs)$

proof(*rule antisym*[*OF - spec.singleton-le-extI*])
show $?lhs \leq ?rhs$
by (*simp add: spec.map-invmap.galois spec.invmap.pre spec.invmap.post spec.invmap.rel*
map-prod-vimage-Times vimage-range spec.rel.UNIV)
fix σ
assume $\langle\sigma\rangle \leq ?rhs$
then obtain $s\ xs$
where $P\ (sf\ s)$
and $trace.init\ \sigma = sf\ s$
and $case\ trace.term\ \sigma\ of\ None \Rightarrow True$
 $\mid\ Some\ v \Rightarrow trace.final'\ (trace.init\ \sigma)\ (trace.rest\ \sigma) \in range\ sf \longrightarrow v \in range\ vf$
and $map\ (map\text{-}prod\ af\ sf)\ xs = trace.natural'\ (sf\ s)\ (trace.rest\ \sigma)$
by (*clarsimp simp: spec.singleton.le-conv trace.steps'.map-range-conv*)
then show $\langle\sigma\rangle \leq ?lhs$
by (*cases* σ)
(*fastforce intro: exI*[**where** $x=trace.T\ s\ xs\ (Some\ (inv\ vf)\ (the\ (trace.term\ \sigma)))$]
range-eqI[**where** $x=trace.final'\ s\ xs$])

dest: *arg-cong*[**where** $f = \text{trace.final}' (sf\ s)$]
simp: *spec.singleton.le-conv* *trace.final'.map* *f-inv-into-f* *trace.natural'.natural'*
order.trans[*OF spec.singleton.less-eq-None spec.singleton.simps*(2)]
split: *option.split-asm*)

qed

lemma *post*:

fixes $Q :: 'w \Rightarrow 't\ \text{pred}$

shows *spec.map* *af sf vf* (*spec.post* ($\lambda v\ s.\ Q\ (vf\ v)\ (sf\ s)$))

= *spec.pre* ($\lambda s.\ s \in \text{range}\ sf$) \sqcap *spec.post* ($\lambda v\ s.\ s \in \text{range}\ sf \longrightarrow Q\ v\ s \wedge v \in \text{range}\ vf$)
 \sqcap *spec.rel* ($\text{range}\ af \times \text{range}\ sf \times \text{range}\ sf$) (**is** ?lhs = ?rhs)

proof(*rule antisym*[*OF - spec.singleton-le-extI*])

show ?lhs \leq ?rhs

by (*simp add*: *spec.map-invmap.galois spec.invmap.pre spec.invmap.post spec.invmap.rel*
map-prod-vimage-Times vimage-range spec.rel.UNIV)

fix σ

assume $\langle \sigma \rangle \leq ?rhs$

then obtain $s\ xs$

where *trace.init* $\sigma = sf\ s$

and *case* *trace.term* σ of *None* \Rightarrow *True* | *Some* $v \Rightarrow \text{trace.final}' (\text{trace.init}\ \sigma)\ (\text{trace.rest}\ \sigma) \in \text{range}\ sf \longrightarrow$
 $Q\ v\ (\text{trace.final}' (\text{trace.init}\ \sigma)\ (\text{trace.rest}\ \sigma)) \wedge v \in \text{range}\ vf$

and *map* (*map-prod* *af sf*) $xs = \text{trace.natural}' (sf\ s)\ (\text{trace.rest}\ \sigma)$

by (*clarsimp simp*: *spec.singleton.le-conv trace.steps'.map-range-conv*)

then show $\langle \sigma \rangle \leq ?lhs$

by (*cases* σ)

(*clarsimp simp*: *spec.singleton.le-conv trace.natural'.natural'*

intro!: *exI*[**where** $x = \text{trace.T}\ s\ xs\ (\text{map-option}\ (inv\ vf)\ (\text{trace.term}\ \sigma))$])

split: *option.split-asm*;

clarsimp dest!: *arg-cong*[**where** $f = \text{trace.final}' (sf\ s)$] *simp*: *trace.final'.map*;

metis *f-inv-into-f rangeI*)

qed

end

end

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *idle*:

shows *spec.invmap* *af sf vf* *spec.idle*

= *spec.term.none* (*spec.rel* ($UNIV \times \text{map-prod}\ sf\ sf - 'Id$) :: $('a,\ 's,\ unit)\ \text{spec}$) (**is** ?lhs = ?rhs)

proof(*rule antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*])

have $sf\ s = sf\ s'$

if $(a,\ s,\ s') \in \text{trace.steps}'\ s_0\ xs$

and $(\lambda x.\ sf\ (\text{snd}\ x))\ 'set\ xs \subseteq \{sf\ s_0\}$

for $a :: 'a$ and $s\ s'\ s_0 :: 's$ and $xs :: ('a \times 's)\ \text{list}$

using **that** by (*induct* xs *arbitrary*: s_0) (*auto simp*: *trace.steps'.Cons-eq-if split: if-split-asm*)

then show $\langle \sigma \rangle \leq ?rhs$ if $\langle \sigma \rangle \leq ?lhs$ for σ

using **that** by (*clarsimp simp*: *spec.singleton.le-conv image-image*)

have $sf\ s' = sf\ s$

if $\text{trace.steps}'\ s\ xs \subseteq UNIV \times \text{map-prod}\ sf\ sf - 'Id$

and $(a,\ s') \in \text{set}\ xs$

for $a\ s\ s'$ and $xs :: ('a \times 's)\ \text{list}$

using **that**

by (*induct* xs *arbitrary*: s)

(*auto simp*: *Diff-subset-conv trace.steps'.Cons-eq-if split: if-split-asm*)

then show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ
using that by (*clarsimp simp: spec.singleton.le-conv*)
qed

lemma *inf-rel*:

shows *spec.rel* (*map-prod af* (*map-prod sf sf*) -' ($r \cup UNIV \times Id$)) \sqcap *spec.invmap af sf vf P* = *spec.invmap af sf vf* (*spec.rel r* \sqcap *P*)
and *spec.invmap af sf vf P* \sqcap *spec.rel* (*map-prod af* (*map-prod sf sf*) -' ($r \cup UNIV \times Id$)) = *spec.invmap af sf vf* (*P* \sqcap *spec.rel r*)
by (*simp-all add: inf-commute spec.invmap.rel spec.invmap.inf*)

lemma *action*: — (* could restrict the stuttering expansion to *range af* or an arbitrary element of that

fixes *af* :: 'a \Rightarrow 'b
fixes *sf* :: 's \Rightarrow 't
fixes *vf* :: 'v \Rightarrow 'w
fixes *F* :: ('w \times 'b \times 't \times 't) set
defines $F' \equiv \text{map-prod id (map-prod af (map-prod sf sf))}$
 $-' (F \cup \{(v, a', s, s) \mid v \ a \ a' \ s. (v, a, s, s) \in F \wedge \neg \text{surj } af\})$
shows *spec.invmap af sf vf* (*spec.action F*)
 $= \text{spec.rel } (UNIV \times \text{map-prod sf sf } -' Id)$
 $\gg (\lambda::\text{unit. spec.action } F')$
 $\gg (\lambda v. \text{spec.rel } (UNIV \times \text{map-prod sf sf } -' Id)$
 $\gg (\lambda::\text{unit. } \sqcup_{v' \in vf} -' \{v'\}. \text{spec.return } v'))$ (**is** *?lhs = ?rhs*)

proof(*rule antisym[OF spec.singleton-le-extI], unfold spec.singleton.invmap-le-conv*)

have *: *sf x = sf y*
if $(\lambda x. sf (snd x))$ ' set $xs \subseteq \{sf s\}$
and $(a, x, y) \in \text{trace.steps}' s xs$
for $s \ a \ x \ y$ **and** $v :: 'v$ **and** $xs :: ('a \times 's)$ list
using that
by (*induct xs arbitrary: s; clarsimp simp: trace.steps'.Cons-eq-if split: if-split-asm; metis*)
show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \text{trace.map af sf vf } \sigma \rangle \leq \text{spec.action } F$ **for** σ
proof(*cases* $\natural (\text{trace.map af sf vf } \sigma) = \text{trace.T } (\text{trace.init } (\text{trace.map af sf vf } \sigma)) \sqcup \text{None}$)
case *True* **then show** *?thesis*
by (*cases* σ)
 $(\text{force simp: spec.singleton.le-conv trace.natural-def trace.natural'.eq-Nil-conv image-image}$
 $\text{dest: } *)$
 $\text{intro: spec.bind.incompleteI}$)

next

case *False* **with that show** *?thesis*
proof(*cases rule: spec.singleton.action-not-idle-le-splitE*)
case (*return v a*) **then show** *?thesis*
by (*cases* σ ; *clarsimp simp: image-image*)
 $(\text{rule spec.action.stutterI}[\text{where } v=v \ \text{and } a=\text{inv af } a]$
 $\text{spec.bind.continueI}[\text{where } ys=[], \text{simplified}]$
 $| (\text{fastforce simp: spec.singleton.le-conv } F'\text{-def trace.final'.map-idle surj-f-inv-f dest: } *)$
 $)+$

next

case (*step v a ys zs*) **then show** *?thesis*
by (*cases* σ ; *clarsimp simp: map-eq-append-conv image-image split: option.split-asm*)
 $(\text{rule spec.bind.continueI}$
 $\text{spec.bind.continueI}[\text{where } xs=[x] \ \text{for } x, \text{simplified}]$
 $\text{spec.bind.incompleteI}[\text{where } g=\langle \text{Sup } X \rangle \ \text{for } X]$
 $\text{spec.bind.continueI}[\text{where } ys=[], \text{simplified}]$
 $| (\text{rule spec.action.stepI; force simp: } F'\text{-def trace.final'.map-idle })$
 $| (\text{fastforce simp: spec.singleton.le-conv trace.final'.map-idle dest: } *)$
 $)+$

qed

qed

have *: *map-prod af (map-prod sf sf) ‘ (UNIV × map-prod sf sf – ‘ Id) – UNIV × Id = {}* **by** *blast*
have $(v, a, s, s') \in F' \implies \langle sf\ s, [(af\ a, sf\ s')], None \rangle \leq spec.action\ F$ **for** $v\ a\ s\ s'$
by *(auto simp: F'-def spec.action.stepI intro: order.trans[OF spec.idle.minimal-le spec.idle.action-le])*
moreover
have $[(vf\ v, a, s, s') \in F'; sf\ s' = trace.init\ \sigma; \langle \sigma \rangle \leq spec.pre\ (\lambda s. s \in range\ sf); \langle \sigma \rangle \leq spec.return\ (vf\ v)]$
 $\implies \langle sf\ s, (af\ a, trace.init\ \sigma) \# trace.rest\ \sigma, trace.term\ \sigma \rangle \leq spec.action\ F$ **for** $v\ a\ s\ s'\ \sigma$
by *(auto simp: F'-def spec.action.stepI spec.action.stutterI spec.singleton.le-conv*
spec.singleton.Cons[where as=trace.rest \sigma]
intro: order.trans[OF spec.idle.minimal-le spec.idle.action-le]
split: option.split-asm)
ultimately have *spec.action (map-prod id (map-prod af (map-prod sf sf)) ‘ F')*
 $\ggg (\lambda v. \bigsqcup_{x \in vf} - \{v\}. spec.pre\ (\lambda s. s \in range\ sf) \sqcap spec.return\ v)$
 $\leq spec.action\ F$
by *(subst spec.action-def)*
(auto simp: spec.bind.SupL spec.bind.supL spec.bind.singletonL spec.idle-le split-def spec.term.none.singleton)
then show $?rhs \leq ?lhs$
apply *(fold spec.map-invmap.galois)*
apply *(strengthen ord-to-strengthen(1)[OF spec.map.bind-le])+*
apply *(strengthen ord-to-strengthen[OF spec.map.rel-le])*
apply *(strengthen ord-to-strengthen[OF spec.map.action-le])*
apply *(subst (1 2) spec.rel.minus-Id[where A=UNIV, symmetric])*
apply *(simp add: image-image * spec.map.return spec.rel.empty spec.bind.SupL spec.bind.returnL*
spec.idle.action-le spec.idle.bind-le-conv spec.bind.SUPR-not-empty spec.bind.supR
spec.bind.return spec.map.Sup)
done
qed

lemma *return:*

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
fixes $F :: ('w \times 'b \times 't \times 't)\ set$
shows $spec.invmap\ af\ sf\ vf\ (spec.return\ v)$
 $= spec.rel\ (UNIV \times map-prod\ sf\ sf\ - \{v\}) \ggg (\lambda :: unit. \bigsqcup_{v' \in vf} - \{v\}. spec.return\ v')$

proof –

have *: *spec.action ({} × UNIV × map-prod sf sf – ‘ Id) = spec.rel.act (UNIV × map-prod sf sf – ‘ Id)*
by *(auto simp: spec.rel.act-def intro: spec.action.cong)*
show *?thesis*
apply *(subst spec.return-def)*
apply *(simp add: spec.invmap.action map-prod-vimage-Times)*
apply *(subst sup.absorb1, force)*
apply *(simp add: spec.action.return-const[where V={v} and W={}] spec.bind.bind spec.bind.return *)*
apply *(simp add: spec.rel.wind-bind flip: spec.bind.bind spec.rel.unfoldL)*
done

qed

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

9 Constructions in the ($'a$, $'s$, $'v$) *spec* lattice

9.1 Constrains-at-most

Abadi and Plotkin (1993, §3.1) require that processes to be composed in parallel *constrain at most* (CAM) distinct sets of agents: intuitively each process cannot block other processes from taking steps after any of its transitions. We model this as a closure.

See §9.2 for a discussion of their composition rules.

Observations:

- the sense of the relation r here is inverted wrt Abadi/Plotkin
- this is a key ingredient in interference closure (§9.3)
- this closure is antimatroidal

setup $\langle \text{Sign.mandatory-path spec} \rangle$

setup $\langle \text{Sign.mandatory-path cam} \rangle$

definition $cl :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**
 $cl \ r \ P = P \sqcup \text{spec.term.none} (\text{spec.term.all } P \ggg (\lambda\text{-::unit. spec.rel } r :: (-, -, \text{unit}) \text{ spec}))$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path none.cam} \rangle$

lemma cl :

shows $\text{spec.term.none} (\text{spec.cam.cl } r \ P) = \text{spec.cam.cl } r (\text{spec.term.none } P)$
by ($\text{simp add: spec.cam.cl-def spec.bind.supL spec.bind.bind spec.term.all.bind ac-simps}$
 $\text{flip: spec.bind.botR bot-fun-def}$)

lemma $cl\text{-rel-wind}$:

fixes $P :: ('a, 's, 'v) \text{ spec}$
shows $\text{spec.cam.cl } r \ P \gg \text{spec.term.none} (\text{spec.rel } r :: ('a, 's, 'w) \text{ spec})$
 $= \text{spec.term.none} (\text{spec.cam.cl } r \ P)$
by ($\text{simp add: spec.cam.cl-def spec.term.none.sup spec.term.none.bind spec.bind.supL spec.bind.bind}$
 $\text{bot-fun-def sup.absorb2}$
 $\text{spec.vmap.unitL[where } f=P] \text{ spec.vmap.unitL[where } f=\text{spec.term.all } P]$
 $\text{spec.vmap.unitL[where } f=\text{spec.rel } r :: ('a, 's, 'w) \text{ spec}]$
 $\text{spec.term.all.vmap-unit spec.vmap.unit-rel spec.bind.mono spec.term.all.expansive}$
 $\text{flip: spec.bind.botR}$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all.cam} \rangle$

lemma $cl\text{-le}$: — Converse does not hold

shows $\text{spec.cam.cl } r (\text{spec.term.all } P) \leq \text{spec.term.all} (\text{spec.cam.cl } r \ P)$
by ($\text{simp add: spec.term.none.cam.cl flip: spec.term.galois} (\text{simp flip: spec.term.none.cam.cl})$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

interpretation cam : closure-complete-distrib-lattice-distributive-class $\text{spec.cam.cl } r$ **for** $r :: ('a, 's) \text{ steps}$

proof *standard*

show $P \leq \text{spec.cam.cl } r \ Q \iff \text{spec.cam.cl } r \ P \leq \text{spec.cam.cl } r \ Q$ (**is** $?lhs \iff ?rhs$) **for** $P \ Q :: ('a, 's, 'v) \text{ spec}$

proof(*rule iffI*)

assume $?lhs$ **show** $?rhs$

apply ($\text{subst spec.cam.cl-def}$)

apply ($\text{strengthen ord-to-strengthen}(1)[OF \langle ?lhs \rangle]$)

apply ($\text{simp add: spec.cam.cl-def spec.term.galois spec.term.all.sup spec.term.all.bind}$)

spec.bind.supL spec.term.all.rel spec.bind.bind spec.rel.wind-bind)

done

next

show $?rhs \implies ?lhs$

by (*simp add: spec.cam.cl-def*)

qed

show *spec.cam.cl* r $(\sqcup P) \leq \sqcup (spec.cam.cl\ r \ 'P) \sqcup spec.cam.cl\ r \ \perp$ **for** $P :: ('a, 's, 'v)$ *spec set*

by (*simp add: spec.cam.cl-def spec.term.none.bind spec.term.all.Sup spec.bind.SupL spec.term.none.Sup SUP-upper2*)

qed

setup $\langle Sign.mandatory-path\ cam \rangle$

setup $\langle Sign.mandatory-path\ cl \rangle$

lemma *bot[*simp*]*:

shows *spec.cam.cl* $r \ \perp = \perp$

by (*simp add: spec.cam.cl-def*)

lemma *mono*:

fixes $r :: ('a, 's)$ *steps*

assumes $r \subseteq r'$

assumes $P \leq P'$

shows *spec.cam.cl* $r\ P \leq spec.cam.cl\ r'\ P'$

unfolding *spec.cam.cl-def*

apply (*strengthen ord-to-strengthen(1)[OF $\langle r \leq r' \rangle$]*)

apply (*strengthen ord-to-strengthen(1)[OF $\langle P \leq P' \rangle$]*)

apply *blast*

done

declare *spec.cam.strengthen-cl[*strg del*]*

lemma *strengthen[*strg*]*:

assumes *st-ord* $F\ r\ r'$

assumes *st-ord* $F\ P\ P'$

shows *st-ord* $F\ (spec.cam.cl\ r\ P)\ (spec.cam.cl\ r'\ P')$

using *assms* **by** (*cases F; simp add: spec.cam.cl.mono*)

lemma *Sup*:

shows *spec.cam.cl* $r\ (\sqcup X) = (\sqcup P \in X.\ spec.cam.cl\ r\ P)$

by (*simp add: spec.cam.cl-Sup*)

lemmas *sup = spec.cam.cl.Sup[where $X = \{P, Q\}$ for $P\ Q$, simplified]*

lemma *rel-empty*:

shows *spec.cam.cl* $\{\} P = P$

by (*simp add: spec.cam.cl-def spec.rel.empty sup.absorb1 UNIV-unit*)

lemma *rel-reflcl*:

shows *spec.cam.cl* $(r \cup A \times Id)\ P = spec.cam.cl\ r\ P$

and *spec.cam.cl* $(A \times Id \cup r)\ P = spec.cam.cl\ r\ P$

by (*simp-all add: spec.cam.cl-def spec.rel.reflcl*)

lemma *rel-minus-Id*:

shows *spec.cam.cl* $(r - UNIV \times Id)\ P = spec.cam.cl\ r\ P$

by (*metis Un-Diff-cancel2 spec.cam.cl.rel-reflcl(1)*)

lemma *Inf*:

shows $\text{spec.cam.cl } r \ (\sqcap X) = \sqcap (\text{spec.cam.cl } r \text{ ' } X)$ (**is** $?lhs = ?rhs$)
proof(*rule antisym[OF spec.cam.cl-Inf-le spec.singleton-le-extI]*)
show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ
proof (*cases trace.term* σ)
case *None*
have $\langle \sigma \rangle \leq \sqcap (\text{spec.term.all ' } X) \ggg (\lambda v. \text{spec.term.none } (\text{spec.rel } r))$
if $x \in X$ **and** $\neg \langle \sigma \rangle \leq x$
for x
proof –
from $\langle \sigma \rangle \leq ?rhs$ **that**
have $\langle \sigma \rangle \leq \text{spec.term.all } x \ggg (\lambda :: \text{unit. spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec}))$
by (*auto simp: spec.cam.cl-def le-Inf-iff spec.term.none.bind*)
then show $?thesis$
proof(*induct rule: spec.singleton.bind-le*)
case incomplete with $\langle \neg \langle \sigma \rangle \leq x \rangle$ **show** $?case$
using *order-trans* **by** *auto*
next
case (*continue* $\sigma_f \sigma_g v_f$)
from *None* **obtain** $xs\ ys$
where $*$: $\forall xs' zs. \text{trace.rest } \sigma = xs' @ zs \wedge \text{trace.steps}' (\text{trace.final}' (\text{trace.init } \sigma) xs') zs \subseteq r$
 $\longrightarrow \text{length } xs \leq \text{length } xs'$
 $xs @ ys = \text{trace.rest } \sigma$
 $\text{trace.steps}' (\text{trace.final}' (\text{trace.init } \sigma) xs) ys \subseteq r$
using *ex-has-least-nat*[**where** $P = \lambda xs. \exists ys. \text{trace.rest } \sigma = xs @ ys$
 $\wedge \text{trace.steps}' (\text{trace.final}' (\text{trace.init } \sigma) xs) ys \subseteq r$
and $k = \text{trace.rest } \sigma$
and $m = \text{length}$]
by *clarsimp*
show $?case$
proof(*induct rule: spec.bind.continueI*[**where** $s = \text{trace.init } \sigma$ **and** $xs = xs$ **and** $ys = ys$
and $v = \text{undefined}$ **and** $w = \text{trace.term } \sigma$,
simplified $\langle xs @ ys = \text{trace.rest } \sigma \rangle \text{trace.t.collapse}$,
case-names $f\ g$])
case f
have $\langle \text{trace.init } \sigma, xs, \text{None} \rangle \leq x$
if $x \in X$
and $\langle \sigma \rangle \leq \text{spec.cam.cl } r\ x$
for x
using *that(2)*[*unfolded spec.cam.cl-def, simplified*]
proof(*induct rule: disjE*[*consumes 1, case-names expansive cam*])
case expansive with $\langle xs @ ys = \text{trace.rest } \sigma \rangle$ **show** $?case$
by (*cases* σ)
(fastforce elim: order.trans[*rotated*]
simp: spec.singleton.mono trace.less-eq-same-append-conv)
next
case *cam* **from** *cam*[*unfolded spec.term.none.bind*] **show** $?case$
proof(*induct rule: spec.singleton.bind-le*)
case incomplete with $\langle xs @ ys = \text{trace.rest } \sigma \rangle$ **show** $?case$
by *clarsimp*
(metis prefixI spec.singleton.mono spec.singleton-le-ext-conv
spec.term.none.contractive trace.less-eq-None(2))
next
case (*continue* $\sigma_f \sigma_g v_f$) **with** $*(1,2)$ **show** $?case$
by (*clarsimp simp: spec.singleton.le-conv trace.less-eq-None*
elim!: order.trans[*rotated*]
intro!: spec.singleton.mono)
(metis prefixI prefix-length-prefix)

```

    qed
  qed
  with  $\langle \langle \sigma \rangle \leq ?rhs \rangle$  show ?case
  by (simp add: le-Inf-iff spec.singleton.le-conv exI[where  $x=None$ ])
next
  case  $g$  with  $None$   $*(\beta)$  show ?case
  by (simp add: spec.singleton.le-conv)
  qed
  qed
  then show  $\langle \langle \sigma \rangle \leq ?lhs \rangle$ 
  by (auto simp: spec.cam.cl-def spec.term.none.bind spec.term.all.Inf le-Inf-iff)
next
  case  $Some$  with  $\langle \langle \sigma \rangle \leq ?rhs \rangle$  show ?thesis
  by (simp add: le-Inf-iff spec.cam.cl-def spec.singleton.term.none-le-conv)
  qed
qed

```

lemmas $inf = spec.cam.cl.Inf$ [**where** $X=\{P, Q\}$ **for** $P Q$, *simplified*]

lemma *idle*:

shows $spec.cam.cl r spec.idle = spec.term.none (spec.rel r :: (-, -, unit) spec)$
by (simp add: spec.cam.cl-def spec.term.all.idle UNIV-unit spec.bind.returnL
spec.idle-le sup-absorb2)

lemma *bind*:

shows $spec.cam.cl r (f \ggg g) = spec.cam.cl r f \ggg (\lambda v. spec.cam.cl r (g v))$
by (simp add: spec.cam.cl-def spec.bind.supL spec.bind.supR spec.bind.bind ac-simps spec.term.all.bind
flip: spec.bind.botR bot-fun-def)

lemma *action*:

fixes $r :: ('a, 's) steps$
fixes $F :: ('v \times 'a \times 's \times 's) set$
shows $spec.cam.cl r (spec.action F)$
 $= spec.action F$
 $\sqcup spec.term.none (spec.action F \ggg (spec.rel r :: (-, -, unit) spec))$
 $\sqcup spec.term.none (spec.rel r :: (-, -, unit) spec)$
by (simp add: spec.cam.cl-def spec.term.all.action spec.term.none.bind spec.term.none.sup
spec.bind.botR spec.bind.supL spec.bind.returnL spec.idle-le
spec.vmap.unitL[**where** $f=spec.action F$] spec.map.surj-sf-action
UNIV-unit map-prod-const-image ac-simps
flip: spec.return-def)

lemma *return*:

shows $spec.cam.cl r (spec.return v) = spec.return v \sqcup spec.term.none (spec.rel r :: (-, -, unit) spec)$
unfolding $spec.return-def spec.cam.cl.action$
by (simp add: spec.bind.returnL spec.idle-le bot-fun-def
flip: spec.return-def bot-fun-def)

lemma *rel-le*:

assumes $r \subseteq r' \vee r' \subseteq r$
shows $spec.cam.cl r (spec.rel r') \leq spec.rel (r \cup r')$
using *assms*
by (auto simp: spec.cam.cl-def spec.rel.mono spec.term.all.rel
spec.rel.wind-bind-leading spec.rel.wind-bind-trailing spec.term.galois)

lemma *rel*:

assumes $r \subseteq r'$

shows $\text{spec.cam.cl } r \text{ (spec.rel } r') = \text{spec.rel } r'$
by (*simp add: assms spec.eq-iff spec.cam.expansive*
order.trans[OF spec.cam.cl.rel-le[OF disjI1] spec.rel.mono])

lemma *inf-rel*:

fixes $r :: ('a, 's) \text{ steps}$
fixes $s :: ('a, 's) \text{ steps}$
fixes $P :: ('a, 's, 'v) \text{ spec}$
shows $\text{spec.rel } r \sqcap \text{spec.cam.cl } r' P = \text{spec.cam.cl } (r \sqcap r') (\text{spec.rel } r \sqcap P)$ (**is** *?thesis1*)
and $\text{spec.cam.cl } r' P \sqcap \text{spec.rel } r = \text{spec.cam.cl } (r \sqcap r') (\text{spec.rel } r \sqcap P)$ (**is** *?thesis2*)
proof –
show *?thesis1*
by (*simp add: spec.cam.cl-def ac-simps inf-sup-distrib*
spec.term.none.bind spec.term.all.inf spec.term.all.rel
spec.bind.inf-rel spec.rel.inf spec.term.none.inf spec.term.none.inf-none-rel(1))
then show *?thesis2*
by (*rule inf-commute-conv*)
qed

lemma *bind-return*:

shows $\text{spec.cam.cl } r (f \gg \text{spec.return } v) = \text{spec.cam.cl } r f \gg \text{spec.return } v$
by (*simp add: spec.cam.cl.bind spec.cam.cl.return spec.bind.supR sup.absorb1 spec.term.none.cam.cl-rel-wind*)

lemma *heyting-le*:

shows $\text{spec.cam.cl } r (P \longrightarrow_H Q) \leq P \longrightarrow_H \text{spec.cam.cl } r Q$
by (*force intro!: SupI*
dest: spec.cam.mono-cl[where r=r]
elim: order.trans[rotated]
simp: heyting-def spec.cam.cl.Sup spec.cam.cl.inf le-infI1 spec.cam.expansive)

lemma *pre*:

shows $\text{spec.cam.cl } r (\text{spec.pre } P) = \text{spec.pre } P$
by (*simp add: spec.cam.cl-def spec.term.none.bind spec.term.all.pre sup-iff-le spec.bind.inf-pre*
flip: inf-iff-le)

lemma *post*:

shows $\text{spec.cam.cl } r (\text{spec.post } Q) = \text{spec.post } Q$
by (*simp add: spec.cam.cl-def spec.term.none.post-le sup-iff-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *empty*:

shows $\text{spec.cam.closed } \{\} = \text{UNIV}$
by (*simp add: order.eq-iff spec.cam.cl.rel-empty spec.cam.closed-clI subsetI*)

lemma *antimonotone*:

shows *antimono spec.cam.closed*
by (*rule monotoneI*) (*auto intro: spec.cam.closed-clI elim: spec.cam.le-closedE[OF spec.cam.cl.mono]*)

lemmas *strengthen[strg] = st-ord-antimono[OF spec.cam.closed.antimonotone]*

lemmas *antimono = antimonoD[OF spec.cam.closed.antimonotone, of r r' for r r']*

lemma *reflcl*:

shows $\text{spec.cam.closed } (r \cup A \times \text{Id}) = \text{spec.cam.closed } r$
by (*simp add: spec.cam.cl.rel-reflcl(1) spec.cam.closed-def*)

setup $\langle \text{Sign.mandatory-path term} \rangle$

lemma *none*:

assumes $P \in \text{spec.cam.closed } r$

shows $\text{spec.term.none } P \in \text{spec.cam.closed } r$

by (*simp add: assms spec.cam.closed-clI flip: spec.term.none.cam.cl spec.cam.closed-conv*[*OF assms*])

setup $\langle \text{Sign.parent-path} \rangle$

lemma *bind*:

fixes $f :: ('a, 's, 'v) \text{ spec}$

fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{ spec}$

assumes $f \in \text{spec.cam.closed } r$

assumes $\bigwedge x. g \ x \in \text{spec.cam.closed } r$

shows $f \ggg g \in \text{spec.cam.closed } r$

by (*simp add: assms spec.cam.closed-clI spec.cam.least spec.cam.cl.bind spec.bind.mono*)

lemma *rel*[*intro*]:

assumes $r \subseteq r'$

shows $\text{spec.rel } r' \in \text{spec.cam.closed } r$

by (*simp add: assms spec.cam.closed-clI spec.cam.cl.rel*)

lemma *pre*[*intro*]:

shows $\text{spec.pre } P \in \text{spec.cam.closed } r$

by (*simp add: spec.cam.closed-clI spec.cam.cl.pre*)

lemma *post*[*intro*]:

shows $\text{spec.post } Q \in \text{spec.cam.closed } r$

by (*simp add: spec.cam.closed-clI spec.cam.cl.post*)

lemma *heyting*[*intro*]:

assumes $Q \in \text{spec.cam.closed } r$

shows $P \longrightarrow_H Q \in \text{spec.cam.closed } r$

by (*rule spec.cam.closed-clI*)

(*simp add: assms order.trans*[*OF spec.cam.cl.heyting-le*] *flip: spec.cam.closed-conv*)

lemma *snoc-conv*:

fixes $P :: ('a, 's, 'v) \text{ spec}$

assumes $P \in \text{spec.cam.closed } r$

assumes $(fst \ x, trace.final' \ s \ xs, snd \ x) \in r \cup UNIV \times Id$

shows $\langle s, xs \ @ \ [x], None \rangle \leq P \longleftrightarrow \langle s, xs, None \rangle \leq P$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof(*rule iffI*)

show $?lhs \implies ?rhs$

by (*erule order.trans*[*rotated*]) (*simp add: spec.singleton.mono trace.less-eq-same-append-conv*)

from *assms*(2) **show** $?rhs \implies ?lhs$

by (*subst spec.cam.closed-conv*[*OF* $\langle P \in \text{spec.cam.closed } r \rangle$])

(*auto simp: spec.cam.cl-def spec.singleton.term.none-le-conv*

spec.term.none.singleton spec.steps.singleton

simp flip: spec.rel.galois spec.term.galois

intro: spec.bind.continueI)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap.cam} \rangle$

lemma *cl*:

fixes *af* :: 'a ⇒ 'b

fixes *sf* :: 's ⇒ 't

fixes *vf* :: 'v ⇒ 'w

fixes *r* :: ('b, 't) *steps*

fixes *P* :: ('b, 't, 'w) *spec*

shows *spec.invmap af sf vf (spec.cam.cl r P)*

= spec.cam.cl (map-prod af (map-prod sf sf) - ' (r ∪ UNIV × Id)) (spec.invmap af sf vf P)

by (*simp add: spec.cam.cl-def spec.invmap.sup spec.invmap.bind spec.invmap.rel spec.term.all.invmap flip: spec.term.none.invmap-gen[where vf=id]*)

setup ⟨*Sign.parent-path*⟩

setup ⟨*Sign.mandatory-path map.cam*⟩

lemma *cl-le*:

fixes *af* :: 'a ⇒ 'b

fixes *sf* :: 's ⇒ 't

fixes *vf* :: 'v ⇒ 'w

fixes *r* :: ('a, 's) *steps*

fixes *P* :: ('a, 's, 'v) *spec*

shows *spec.map af sf vf (spec.cam.cl r P)*

≤ spec.cam.cl (map-prod af (map-prod sf sf) ' r) (spec.map af sf vf P)

by (*simp add: spec.map-invmap.galois spec.map-invmap.upper-lower-expansive spec.invmap.cam.cl spec.cam.cl.mono subset-vimage-iff le-supI1*)

lemma *cl-inj-sf*:

fixes *af* :: 'a ⇒ 'b

fixes *sf* :: 's ⇒ 't

fixes *vf* :: 'v ⇒ 'w

fixes *r* :: ('a, 's) *steps*

fixes *P* :: ('a, 's, 'v) *spec*

assumes *inj sf*

shows *spec.map af sf vf (spec.cam.cl r P)*

= spec.cam.cl (map-prod af (map-prod sf sf) ' r) (spec.map af sf vf P)

apply (*simp add: spec.cam.cl-def spec.map.sup spec.map.bind-inj-sf[OF <inj sf>] spec.term.all.map flip: spec.term.none.map-gen[where vf=id]*)

apply (*subst spec.map.rel, blast dest: injD[OF <inj sf>]*)

apply (*simp add: inf.absorb1 spec.map-invmap.galois spec.invmap.post flip: spec.bind-post-pre*)

done

setup ⟨*Sign.parent-path*⟩

setup ⟨*Sign.parent-path*⟩

9.2 Abadi and Plotkin's composition principle

Abadi and Plotkin (1991, 1993) develop a theory of circular reasoning about Heyting implication for safety properties under the mild condition that each is CAM-closed with respect to the other.

setup ⟨*Sign.mandatory-path spec*⟩

abbreviation *ap-cam-cl* :: 'a *set* ⇒ ('a, 's, 'v) *spec* ⇒ ('a, 's, 'v) *spec* **where**

ap-cam-cl as ≡ *spec.cam.cl ((-as) × UNIV)*

abbreviation (*input*) *ap-cam-closed* :: 'a *set* ⇒ ('a, 's, 'v) *spec set* **where**

ap-cam-closed as ≡ *spec.cam.closed ((-as) × UNIV)*

lemma *composition-principle-1*:

fixes $P :: ('a, 's, 'v) \text{ spec}$
assumes $P \in \text{spec.ap-cam-closed } as$
assumes $P \in \text{spec.term.closed}$ -
assumes $\text{spec.idle} \leq P$
shows $\text{spec.ap-cam-cl } (-as) P \longrightarrow_H P \leq P$ (**is** $?lhs \leq ?rhs$)
proof(*rule spec.term.closed.singleton-le-extI*)
show $\langle s, xs, None \rangle \leq ?rhs$ **if** $\langle s, xs, None \rangle \leq ?lhs$ **for** $s \ xs$
using *that*
proof(*induct xs rule: rev-induct*)
case *Nil*
from $\langle \text{spec.idle} \leq P \rangle$ **show** $?case$
by (*simp add: order.trans[OF spec.idle.minimal-le]*)
next
case (*snoc x xs*)
from snoc.premis **have** $\langle s, xs, None \rangle \leq \text{spec.ap-cam-cl } (-as) P \longrightarrow_H P$
by (*simp add: order.trans[OF spec.singleton.mono, rotated] trace.less-eq-None*)
with snoc.hyps **have** $\langle s, xs, None \rangle \leq P$ **by** *blast*
show $?case$
proof(*cases fst x \in as*)
case *True*
with $\langle \langle s, xs, None \rangle \leq P \rangle$ **have** $\langle s, xs @ [x], None \rangle \leq \text{spec.ap-cam-cl } (-as) P$
by (*subst spec.cam.closed.snoc-conv*) (*auto simp: order.trans[OF - spec.cam.expansive]*)
with snoc.premis **show** $?thesis$ **by** (*blast intro: heyting.mp*)
next
case *False* **with** $\langle P \in \text{spec.ap-cam-closed } as \rangle$ $\langle \langle s, xs, None \rangle \leq P \rangle$ **show** $?thesis$
by (*simp add: spec.cam.closed.snoc-conv*)
qed
qed
qed fact

lemma *composition-principle-half*: — [Abadi and Plotkin \(1993, §3.1\(4\)\)](#) – cleaner than in [Abadi and Plotkin \(1991, §3.1\)](#)

assumes $M_1 \in \text{spec.ap-cam-closed } a_1$
assumes $M_2 \in \text{spec.ap-cam-closed } a_2$
assumes $M_1 \in \text{spec.term.closed}$ -
assumes $\text{spec.idle} \leq M_1$
assumes $a_1 \cap a_2 = \{\}$
shows $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1$
proof -
have $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq (\text{spec.ap-cam-cl } (-a_1) M_1 \longrightarrow_H \text{spec.ap-cam-cl } (-a_1) M_2) \sqcap (M_2 \longrightarrow_H M_1)$
by (*rule inf-mono[OF heyting.closure-imp-distrib-le[OF closure.axioms(2)[OF spec.cam.closure-axioms]] order.refl*)
(simp add: spec.cam.cl.inf)
also have $\dots \leq \text{spec.ap-cam-cl } (-a_1) M_1 \longrightarrow_H M_1$
proof -
from $\langle M_2 \in \text{spec.ap-cam-closed } a_2 \rangle$ $\langle a_1 \cap a_2 = \{\} \rangle$ **have** $\text{spec.ap-cam-cl } (-a_1) M_2 \leq M_2$
by (*fastforce intro: spec.cam.least elim: subsetD[OF spec.cam.closed.antimono, rotated]*)
then show $?thesis$
by (*simp add: heyting.trans order-antisym-conv spec.cam.expansive*)
qed
also have $\dots \leq M_1$
by (*rule spec.composition-principle-1[OF \langle M_1 \in \text{spec.ap-cam-closed } a_1 \rangle \langle M_1 \in \text{spec.term.closed} \rightarrow \langle \text{spec.idle} \leq M_1 \rangle)
finally show $?thesis$.
qed*

theorem *composition-principle*: — [Abadi and Plotkin \(1993, §3.1\(3\)\)](#)

```

assumes  $M_1 \in \text{spec.ap-cam-closed } a_1$ 
assumes  $M_2 \in \text{spec.ap-cam-closed } a_2$ 
assumes  $M_1 \in \text{spec.term.closed -}$ 
assumes  $M_2 \in \text{spec.term.closed -}$ 
assumes  $\text{spec.idle} \leq M_1$ 
assumes  $\text{spec.idle} \leq M_2$ 
assumes  $a_1 \cap a_2 = \{\}$ 
shows  $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1 \sqcap M_2$ 
using assms by (metis spec.composition-principle-half inf.bounded-iff inf commute)

```

An infinitary variant can be established in essentially the same way as *spec.composition-principle-1*.

lemma *ag-circular*:

```

fixes  $P_s :: 'a \Rightarrow ('a, 's, 'v) \text{ spec}$ 
assumes cam-closed:  $\bigwedge a. a \in as \implies P_s a \in \text{spec.ap-cam-closed } \{a\}$ 
assumes term-closed:  $\bigwedge a. a \in as \implies P_s a \in \text{spec.term.closed -}$ 
assumes idle:  $\bigwedge a. a \in as \implies \text{spec.idle} \leq P_s a$ 
shows  $(\bigsqcap a \in as. (\bigsqcap a' \in as - \{a\}. P_s a')) \longrightarrow_H P_s a \leq (\bigsqcap a \in as. P_s a)$  (is ?lhs  $\leq$  ?rhs)
proof(rule spec.term.closed.singleton-le-extI)
show  $\langle s, xs, \text{None} \rangle \leq ?rhs$  if  $\langle s, xs, \text{None} \rangle \leq ?lhs$  for  $s \ xs$ 
  using that
proof(induct xs rule: rev-induct)
  case Nil from idle show ?case
    by (simp add: le-INF-iff order.trans[OF spec.idle.minimal-le])
next
case (snoc x xs)
have  $*$ :  $\langle s, xs, \text{None} \rangle \leq ?rhs$ 
  by (simp add: snoc(1) order.trans[OF spec.singleton.mono snoc(2)] trace.less-eq-same-append-conv)
have  $\langle s, xs @ [x], \text{None} \rangle \leq P_s a$  if  $a \in as$  for  $a$ 
proof(cases fst x = a)
  case True
with cam-closed * have  $\langle s, xs @ [x], \text{None} \rangle \leq \bigsqcap (P_s ' (as - \{a\}))$ 
  by (subst spec.cam.closed.snoc-conv[where r= $\bigsqcap a' \in as - \{a\}. (- \{a\}) \times UNIV$ ])
  (auto simp: le-INF-iff intro: subsetD[OF spec.cam.closed.antimono, rotated])
with snoc.premis(1)  $\langle a \in as \rangle$  show ?thesis
  by (meson heyting.mp le-INF-iff)
next
case False with cam-closed  $*$   $\langle a \in as \rangle$  show ?thesis
  by (fastforce simp: spec.cam.closed.snoc-conv le-INF-iff)
qed
then show ?case by (blast intro: INFI)
qed
from term-closed show ?rhs  $\in \text{spec.term.closed -}$ 
  by (fastforce simp: spec.term.all.monomorphic)
qed

```

setup $\langle \text{Sign.parent-path} \rangle$

9.3 Interference closure

We add environment interference to the beginnings and ends of behaviors for two reasons:

- it ensures the wellformedness of parallel composition as conjunction (see §9.5)
- it guarantees the monad laws hold (see §13.3.1)
 - *spec.cam.cl* by itself is too weak to justify these

We use this closure to build the program sublattice of the $('a, 's, 'v) \text{ spec}$ lattice (see §13).

Observations:

- if processes are made out of actions then it is not necessary to apply *spec.cam.cl*

setup $\langle \text{Sign.mandatory-path spec} \rangle$

setup $\langle \text{Sign.mandatory-path interference} \rangle$

definition $cl :: ('a, 's) \text{ steps} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

$cl \ r \ P = \text{spec.rel } r \ggg (\lambda :: \text{unit. spec.cam.cl } r \ P) \ggg (\lambda v. \text{spec.rel } r \ggg (\lambda :: \text{unit. spec.return } v))$

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *interference: closure-complete-distrib-lattice-distributive-class spec.interference.cl r*
for $r :: ('a, 's) \text{ steps}$

proof

show $P \leq \text{spec.interference.cl } r \ Q \longleftrightarrow \text{spec.interference.cl } r \ P \leq \text{spec.interference.cl } r \ Q$ (**is** $?lhs \longleftrightarrow ?rhs$)

for $P \ Q :: ('a, 's, 'v) \text{ spec}$

proof(*rule iffI*)

assume $?lhs$ **show** $?rhs$

apply (*subst spec.interference.cl-def*)

apply (*strengthen ord-to-strengthen(1)[OF $\langle ?lhs \rangle$]*)

apply (*simp add: spec.interference.cl-def spec.cam.cl.bind spec.cam.cl.return spec.cam.cl.rel spec.bind.bind spec.bind.supL spec.bind.supR spec.bind.returnL spec.idle-le flip: bot-fun-def spec.bind.botR*)

apply (*simp add: spec.rel.wind-bind flip: spec.bind.bind*)

apply (*simp add: spec.bind.bind spec.bind.mono*)

done

next

assume $?rhs$ **show** $?lhs$

apply (*strengthen ord-to-strengthen(2)[OF $\langle ?rhs \rangle$]*)

apply (*simp add: spec.interference.cl-def spec.bind.bind*)

apply (*strengthen ord-to-strengthen(2)[OF spec.cam.expansive]*)

apply (*strengthen ord-to-strengthen(2)[OF spec.return.rel-le]*)

apply (*auto simp: spec.bind.return intro: spec.bind.returnL-le*)

done

qed

show $\text{spec.interference.cl } r \ (\bigsqcup P) \leq \bigsqcup (\text{spec.interference.cl } r \ ' P) \sqcup \text{spec.interference.cl } r \ \perp$

for $P :: ('a, 's, 'v) \text{ spec set}$

by (*simp add: spec.interference.cl-def spec.cam.cl.Sup image-image spec.bind.SupL spec.bind.supL spec.bind.SUPR flip: bot-fun-def*)

qed

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

setup $\langle \text{Sign.mandatory-path interference} \rangle$

lemma *cl*:

shows $\text{spec.term.none } (\text{spec.interference.cl } r \ P) = \text{spec.interference.cl } r \ (\text{spec.term.none } P)$

by (*simp add: spec.interference.cl-def spec.term.none.bind spec.term.none.return*

spec.bind.bind spec.bind.idleR spec.bind.botR spec.term.none.cam.cl-rel-wind

flip: spec.term.none.cam.cl)

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *rel-le*:

assumes $P \in \text{spec.interference.closed } r$
shows $\text{spec.term.none } (\text{spec.rel } r) \leq P$
by (*simp* $\text{spec.interference.closed-conv}[OF \text{ assms}]$)
(simp add: spec.interference.cl-def spec.term.galois spec.term.all.bind spec.term.all.rel ac-simps)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *cl-le*: — Converse does not hold

shows $\text{spec.interference.cl } r (\text{spec.term.all } P) \leq \text{spec.term.all } (\text{spec.interference.cl } r P)$
by (*simp* $\text{add: spec.interference.cl-def spec.bind.bind spec.bind.idleR spec.bind.botR}$
 $\text{spec.term.none.bind spec.term.none.return}$
 $\text{spec.term.none.cam.cl-rel-wind spec.term.none.cam.cl}$
 $\text{flip: spec.term.galois}$)
(simp add: spec.bind.mono flip: spec.term.none.cam.cl)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path cam.closed.interference} \rangle$

lemma *cl*:

shows $\text{spec.interference.cl } r P \in \text{spec.cam.closed } r$
by (*metis* $\text{spec.cam.closed-clI spec.interference.cl-def spec.interference.expansive}$
 $\text{spec.interference.idempotent}(1) \text{ spec.cam.idempotent}(1)$)

lemma *closed-subseteq*:

shows $\text{spec.interference.closed } r \subseteq \text{spec.cam.closed } r$
by (*metis* $\text{spec.cam.closed.interference.cl spec.interference.closed-conv subsetI}$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path interference} \rangle$

setup $\langle \text{Sign.mandatory-path cl} \rangle$

lemma *mono*:

assumes $r \subseteq r'$
assumes $P \leq P'$
shows $\text{spec.interference.cl } r P \leq \text{spec.interference.cl } r' P'$
unfolding $\text{spec.interference.cl-def}$
apply (*strengthen ord-to-strengthen*(1)[*OF* $\langle r \subseteq r' \rangle$])
apply (*strengthen ord-to-strengthen*(1)[*OF* $\langle P \leq P' \rangle$])
apply *simp*
done

declare $\text{spec.interference.strengthen-cl}[strg \text{ del}]$

lemma *strengthen[strg]*:

assumes $\text{st-ord } F r r'$
assumes $\text{st-ord } F P P'$
shows $\text{st-ord } F (\text{spec.interference.cl } r P) (\text{spec.interference.cl } r' P')$

using *assms* by (cases *F*; *simp* add: *spec.interference.cl.mono*)

lemma *bot*:

shows *spec.interference.cl* $r \perp = \text{spec.term.none}$ (*spec.rel* $r :: (-, -, \text{unit})$ *spec*)
by (*simp* add: *spec.interference.cl-def* *spec.bind.bind* *flip: bot-fun-def* *spec.bind.botR*)

lemmas *Sup* = *spec.interference.cl-Sup*

lemmas *sup* = *spec.interference.cl-sup*

lemma *idle*:

shows *spec.interference.cl* $r \text{ spec.idle} = \text{spec.term.none}$ (*spec.rel* $r :: (-, -, \text{unit})$ *spec*)
by (*simp* add: *spec.interference.cl-def* *spec.cam.cl.idle* *spec.bind.bind* *spec.rel.wind-bind*
flip: spec.term.none.bind)

lemma *rel-empty*:

assumes *spec.idle* $\leq P$
shows *spec.interference.cl* $\{\}$ $P = P$
by (*simp* add: *spec.interference.cl-def* *spec.rel.empty* *spec.cam.cl.rel-empty* *spec.bind.return*
spec.bind.returnL *assms UNIV-unit*)

lemma *rel-reflcl*:

shows *spec.interference.cl* $(r \cup A \times \text{Id}) P = \text{spec.interference.cl}$ $r P$
and *spec.interference.cl* $(A \times \text{Id} \cup r) P = \text{spec.interference.cl}$ $r P$
by (*simp-all* add: *spec.interference.cl-def* *spec.cam.cl.rel-reflcl* *spec.rel.reflcl*)

lemma *rel-minus-Id*:

shows *spec.interference.cl* $(r - \text{UNIV} \times \text{Id}) P = \text{spec.interference.cl}$ $r P$
by (*metis Un-Diff-cancel2* *spec.interference.cl.rel-reflcl(1)*)

lemma *inf-rel*:

shows *spec.interference.cl* $s P \sqcap \text{spec.rel}$ $r = \text{spec.interference.cl}$ $(r \sqcap s)$ (*spec.rel* $r \sqcap P$)
and *spec.rel* $r \sqcap \text{spec.interference.cl}$ $s P = \text{spec.interference.cl}$ $(r \sqcap s)$ (*spec.rel* $r \sqcap P$)
by (*simp-all* add: *spec.interference.cl-def* *spec.bind.inf-rel* *spec.return.inf-rel* *spec.cam.cl.inf-rel*
flip: spec.rel.inf)

lemma *bindL*:

assumes $f \in \text{spec.interference.closed}$ r
shows *spec.interference.cl* $r (f \ggg g) = f \ggg (\lambda v. \text{spec.interference.cl}$ $r (g v))$
apply (*subst* (1 2) *spec.interference.closed-conv[OF assms]*)
apply (*simp* add: *spec.interference.cl-def* *spec.bind.bind* *spec.cam.cl.bind* *spec.cam.cl.rel*
spec.cam.cl.return *spec.bind.supL* *spec.bind.return*)
apply (*simp* add: *spec.rel.wind-bind* *flip: spec.bind.bind*)
done

lemma *bindR*:

assumes $\bigwedge v. g v \in \text{spec.interference.closed}$ r
shows *spec.interference.cl* $r (f \ggg g) = \text{spec.interference.cl}$ $r f \ggg g$ (**is** *?lhs* = *?rhs*)
proof –
from *assms* have *?lhs* = *spec.interference.cl* $r (f \ggg (\lambda v. \text{spec.interference.cl}$ $r (g v)))$
by (*meson* *spec.interference.closed-conv*)
also have $\dots = \text{spec.interference.cl}$ $r f \ggg (\lambda v. \text{spec.interference.cl}$ $r (g v))$
apply (*simp* add: *spec.interference.cl-def* *spec.bind.bind* *spec.cam.cl.bind* *spec.cam.cl.rel*
spec.cam.cl.return *spec.bind.supL* *spec.bind.supR* *spec.bind.return*
sup.absorb1 *spec.bind.mono*
flip: spec.bind.botR)
apply (*simp* add: *spec.rel.wind-bind* *flip: spec.bind.bind*)

done
also from *assms* **have** ... = ?*rhs*
by (*simp flip: spec.interference.closed-conv*)
finally show ?*thesis* .
qed

lemma *bind-conv*:
assumes $f \in \text{spec.interference.closed } r$
assumes $\forall x. g \ x \in \text{spec.interference.closed } r$
shows $\text{spec.interference.cl } r \ (f \ggg g) = f \ggg g$
using *assms* **by** (*simp add: spec.interference.cl.bindR flip: spec.interference.closed-conv*)

lemma *action*:
shows $\text{spec.interference.cl } r \ (\text{spec.action } F)$
 $= \text{spec.rel } r \ggg (\lambda::\text{unit. spec.action } F \ggg (\lambda v. \text{spec.rel } r \ggg (\lambda::\text{unit. spec.return } v)))$
by (*simp add: spec.interference.cl-def spec.cam.cl.action spec.bind.supL spec.bind.supR*
flip: spec.bind.botR spec.bind.bind spec.rel.unwind-bind)
(simp add: spec.bind.bind sup.absorb1 spec.bind.mono)

lemma *return*:
shows $\text{spec.interference.cl } r \ (\text{spec.return } v) = \text{spec.rel } r \ggg (\lambda::\text{unit. spec.return } v)$
by (*simp add: spec.return-def spec.interference.cl.action spec.bind.bind*)
(simp add: spec.bind.return spec.rel.wind-bind flip: spec.return-def spec.bind.bind)

lemma *bind-return*:
shows $\text{spec.interference.cl } r \ (f \gg \text{spec.return } v) = \text{spec.interference.cl } r \ f \gg \text{spec.return } v$
by (*simp add: spec.interference.cl-def spec.bind.bind spec.bind.return spec.cam.cl.bind-return*)

lemma *rel*: — complicated by polymorphic *spec.rel*
assumes $r \subseteq r' \vee r' \subseteq r$
shows $\text{spec.interference.cl } r \ (\text{spec.rel } r') = \text{spec.rel } (r \cup r')$ (**is** ?*lhs* = ?*rhs*)
using *assms*

proof
show ?*thesis* **if** $r \subseteq r'$
apply (*simp add: <r ⊆ r'> sup.absorb2 spec.eq-iff spec.interference.expansive*)
apply (*strengthen ord-to-strengthen(1)[OF <r ⊆ r'>]*)
apply (*metis spec.interference.cl.bot spec.interference.idempotent(1) spec.term.all.rel*
spec.term.all-none spec.term.none.interference.all.cl-le)
done
show ?*thesis* **if** $r' \subseteq r$
proof(*rule antisym*)
from $\langle r' \subseteq r \rangle$ **show** ?*lhs* \leq ?*rhs*
by (*simp add: inf.absorb-iff1 spec.interference.cl.inf-rel flip: spec.rel.inf*)
from $\langle r' \subseteq r \rangle$ **show** ?*rhs* \leq ?*lhs*
by (*simp add: sup.absorb1 spec.interference.cl-def spec.cam.cl-def*
spec.rel.wind-bind-trailing le-supI1 spec.bind.supR spec.bind.return
order.trans[OF - spec.bind.mono[OF order.refl spec.bind.mono[OF spec.return.rel-le order.refl]]])

qed
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path idle.interference} \rangle$

lemma *cl-le[spec.idle-le]*:
shows $\text{spec.idle} \leq \text{spec.interference.cl } r \ P$

by (simp add: spec.interference.cl-def spec.idle-le)

lemma closed-le[spec.idle-le]:

assumes $P \in \text{spec.interference.closed } r$

shows $\text{spec.idle} \leq P$

by (subst spec.interference.closed-conv[OF assms]) (simp add: spec.idle.interference.cl-le)

setup <Sign.parent-path>

setup <Sign.mandatory-path map.interference>

lemma cl-sf-id:

shows $\text{spec.map af id vf } (\text{spec.interference.cl } r P)$

$= \text{spec.interference.cl } (\text{map-prod af id } 'r) (\text{spec.map af id vf } P)$

apply (simp add: spec.interference.cl-def spec.map.return

$\text{spec.map.bind-inj-sf}[OF \text{inj-on-id}] \text{spec.map.cam.cl-inj-sf}[OF \text{inj-on-id}])$

apply (subst (1 2) spec.map.rel, force, force)

apply (simp add: spec.vmap.eq-return(2) spec.bind.bind

$\text{spec.bind.returnL spec.idle-le}$

$\text{flip: spec.map.cam.cl-inj-sf}[\mathbf{where\ af=id\ and\ sf=id\ and\ vf=vf\ and\ P=spec.amap\ af\ P,}$
 $\text{simplified spec.map.comp, simplified, folded id-def}]$

done

setup <Sign.parent-path>

setup <Sign.mandatory-path invmap.interference>

lemma cl:

fixes $as :: 'b \text{ set}$

fixes $af :: 'a \Rightarrow 'b$

fixes $sf :: 's \Rightarrow 't$

fixes $vf :: 'v \Rightarrow 'w$

fixes $r :: ('b, 't) \text{ steps}$

fixes $P :: ('b, 't, 'w) \text{ spec}$

shows $\text{spec.invmap af sf vf } (\text{spec.interference.cl } r P)$

$= \text{spec.interference.cl } (\text{map-prod af } (\text{map-prod sf sf}) - ' (r \cup \text{UNIV} \times \text{Id})) (\text{spec.invmap af sf vf } P)$

apply (simp add: spec.interference.cl-def map-prod-vimage-Times spec.rel.wind-bind-trailing

$\text{spec.invmap.bind spec.invmap.cam.cl spec.invmap.rel spec.invmap.return}$

$\text{flip: spec.bind.bind})$

apply (subst (2) spec.invmap.split-vinvmap)

apply (simp add: spec.cam.cl.bind spec.cam.cl.return spec.cam.cl.Sup spec.term.none.cam.cl-rel-wind

$\text{spec.bind.mono spec.bind.bind spec.bind.SupL spec.bind.supL}$

$\text{spec.bind.SUPR spec.bind.supR spec.bind.returnL spec.idle-le spec.bind.botR}$

$\text{image-image sup.absorb1})$

done

setup <Sign.parent-path>

setup <Sign.mandatory-path interference.closed>

lemma antimonotone:

shows $\text{antimono spec.interference.closed}$

proof(rule antimonoI)

show $\text{spec.interference.closed } r' \subseteq \text{spec.interference.closed } r$ if $r \subseteq r'$ for $r r' :: ('a, 's) \text{ steps}$

unfolding $\text{spec.interference.closed-def}$ by (strengthen ord-to-strengthen(2)[OF < $r \subseteq r'$ >]) simp

qed

lemmas $\text{strengthen}[strg] = \text{st-ord-antimono}[OF \text{spec.interference.closed.antimonotone}]$

lemmas *antimono* = *antimonoD*[*OF spec.interference.closed.antimonotone*]

lemma *Sup'*:

assumes $X \subseteq \text{spec.interference.closed } r$

shows $\sqcup X \sqcup \text{spec.term.none } (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec}) \in \text{spec.interference.closed } r$

by (*metis assms spec.interference.cl.bot spec.interference.closed-Sup*)

lemma *Sup-not-empty*:

assumes $X \subseteq \text{spec.interference.closed } r$

assumes $X \neq \{\}$

shows $\sqcup X \in \text{spec.interference.closed } r$

using *spec.interference.closed-Sup*[*OF assms(1)*] *assms*

by (*simp add: assms spec.interference.closed-Sup*[*OF assms(1)*] *less-eq-Sup spec.interference.least subsetD sup.absorb1*)

lemma *rel*:

assumes $r' \subseteq r$

shows $\text{spec.rel } r \in \text{spec.interference.closed } r'$

by (*metis assms spec.eq-iff inf.absorb-iff2 spec.interference.cl.inf-rel(2) spec.interference.closed-clI*)

lemma *bind-relL*:

fixes $P :: ('a, 's, 'v) \text{ spec}$

assumes $P \in \text{spec.interference.closed } r$

shows $\text{spec.rel } r \ggg (\lambda :: \text{unit}. P) = P$

by (*subst (1 2) spec.interference.closed-conv*[*OF assms*])

(*simp add: spec.interference.cl-def spec.rel.wind-bind flip: spec.bind.bind*)

lemma *bind-relR*:

assumes $P \in \text{spec.interference.closed } r$

shows $P \ggg (\lambda v. \text{spec.rel } r \ggg (\lambda :: \text{unit}. Q v)) = P \ggg Q$

by (*subst (1 2) spec.interference.closed-conv*[*OF assms*])

(*simp add: spec.interference.cl-def spec.bind.bind spec.bind.return; simp add: spec.rel.wind-bind flip: spec.bind.bind*)

lemma *bind-rel-unitR*:

assumes $P \in \text{spec.interference.closed } r$

shows $P \ggg (\text{spec.rel } r :: (-, -, \text{unit}) \text{ spec}) = P$

by (*subst (1 2) spec.interference.closed-conv*[*OF assms*])

(*simp add: spec.interference.cl-def spec.bind.bind spec.rel.wind-bind*)

lemma *bind-rel-botR*:

assumes $P \in \text{spec.interference.closed } r$

shows $P \ggg (\lambda v. \text{spec.rel } r \ggg (\lambda :: \text{unit}. \perp)) = P \ggg \perp$

by (*subst (1 2) spec.interference.closed-conv*[*OF assms*])

(*simp add: spec.interference.cl-def spec.bind.bind spec.bind.return; simp add: spec.rel.wind-bind flip: spec.bind.bind*)

lemma *bind[intro]*:

fixes $f :: ('a, 's, 'v) \text{ spec}$

fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{ spec}$

assumes $f \in \text{spec.interference.closed } r$

assumes $\bigwedge x. g x \in \text{spec.interference.closed } r$

shows $(f \ggg g) \in \text{spec.interference.closed } r$

using *assms* **by** (*simp add: spec.interference.closed-clI spec.interference.cl.bindL flip: spec.interference.closed-conv*)

lemma *kleene-star*:

assumes $P \in \text{spec.interference.closed } r$

assumes $spec.return () \leq P$
shows $spec.kleene.star P \in spec.interference.closed r$
proof(*rule spec.interference.closed-clI*,
*induct rule: spec.kleene.star.fixp-induct[where P= $\lambda R. spec.interference.cl r (R P) \leq spec.kleene.star P$,
case-names adm bot step]*)
case bot from $\langle P \in spec.interference.closed r \rangle$ **show** *?case*
by (*simp add: order.trans[OF - spec.kleene.expansive-star] spec.interference.cl.bot*
spec.term.none.interference.closed.rel-le)

next
case (*step R*) **show** *?case*
apply (*simp add: spec.interference.cl-sup spec.interference.cl.bindL[OF assms(1)]*)
apply (*strengthen ord-to-strengthen(1)[OF step]*)
apply (*strengthen ord-to-strengthen(1)[OF $\langle spec.return () \leq P \rangle$]*)
apply (*simp add: spec.kleene.fold-starL spec.kleene.expansive-star*
flip: spec.interference.closed-conv[OF assms(1)])
done
qed *simp-all*

lemma *map-sf-id*:
fixes $af :: 'a \Rightarrow 'b$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P \in spec.interference.closed r$
shows $spec.map af id vf P \in spec.interference.closed (map-prod af id ' r)$
by (*rule spec.interference.closed-clI*)
(subst (2) spec.interference.closed-conv[OF assms];
simp add: spec.map.interference.cl-sf-id map-prod-image-Times)

lemma *invmap*:
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P \in spec.interference.closed r$
shows $spec.invmap af sf vf P \in spec.interference.closed (map-prod af (map-prod sf sf) - ' r)$
by (*rule spec.interference.closed-clI*)
(subst (2) spec.interference.closed-conv[OF assms];
fastforce simp: spec.invmap.interference.cl intro: spec.interference.cl.mono)

setup $\langle Sign.mandatory-path term \rangle$

lemma *none*:
assumes $P \in spec.interference.closed r$
shows $spec.term.none P \in spec.interference.closed r$
by (*rule spec.interference.closed-clI*)
(subst (2) spec.interference.closed-conv[OF assms];
simp add: spec.term.none.interference.cl)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

9.4 The 'a agent datatype

For compositionality we often wish to designate a specific agent as the environment.

datatype $'a agent = proc (the-agent: 'a) | env$
type-synonym $sequential = unit agent$ — Sequential programs (§13)
abbreviation $self :: sequential$ **where** $self \equiv proc ()$

declare *agent.map-id*[*simp*]
declare *agent.map-id0*[*simp*]
declare *agent.map-id0*[*unfolded id-def, simp*]
declare *agent.map-comp*[*unfolded comp-def, simp*]

lemma *env-not-in-range-proc*[*iff*]:
shows $env \notin \text{range } proc$
by *fastforce*

lemma *range-proc-conv*[*simp*]:
shows $x \in \text{range } proc \longleftrightarrow x \neq env$
by (*cases x*) *simp-all*

lemma *inj-proc*[*iff*]:
shows *inj proc*
by (*simp add: inj-def*)

lemma *surj-the-inv-proc*[*iff*]:
shows *surj (the-inv proc)*
by (*meson inj-proc surjI the-inv-f-f*)

lemma *the-inv-proc*[*simp*]:
shows $the\text{-}inv\ proc (proc\ a) = a$
by (*simp add: the-inv-f-f*)

lemma *uminus-env-range-proc*[*simp*]:
shows $-\{env\} = \text{range } proc$
by (*auto intro: agent.exhaust*)

lemma *env-range-proc-UNIV*[*simp*]:
shows $insert\ env\ (\text{range } proc) = UNIV$
by (*auto intro: agent.exhaust*)

setup $\langle Sign.mandatory\text{-}path\ sequential \rangle$

lemma *not-conv*[*simp*]:
shows $a \neq env \longleftrightarrow a = self$
and $a \neq self \longleftrightarrow a = env$
by (*cases a; simp*)+

lemma *range-proc-self*[*simp*]:
shows $\text{range } proc = \{self\}$
by *fastforce*

lemma *UNIV*:
shows $UNIV = \{env, self\}$
by *fastforce*

lemma *rev-UNIV*[*simp*]:
shows $\{env, self\} = UNIV$
and $\{self, env\} = UNIV$
by *fastforce*+

lemma *uminus-self-env*[*simp*]:
shows $-\{self\} = \{env\}$
by *fastforce*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path map-agent} \rangle$

lemma *eq-conv*:

shows $\text{map-agent } f \ x = \text{env} \longleftrightarrow x = \text{env}$

and $\text{env} = \text{map-agent } f \ x \longleftrightarrow x = \text{env}$

and $\text{map-agent } f \ x = \text{proc } a \longleftrightarrow (\exists a'. x = \text{proc } a' \wedge a = f \ a')$

and $\text{proc } a = \text{map-agent } f \ x \longleftrightarrow (\exists a'. x = \text{proc } a' \wedge a = f \ a')$

by $(\text{cases } x; \text{auto})+$

lemma *surj*:

fixes $\pi :: 'a \Rightarrow 'b$

assumes *surj* π

shows *surj* $(\text{map-agent } \pi)$

by $(\text{metis } \text{assms } \text{surj-def } \text{agent.exhaust } \text{agent.map}(1,2))$

lemma *bij*:

fixes $\pi :: 'a \Rightarrow 'b$

assumes *bij* π

shows *bij* $(\text{map-agent } \pi)$

by $(\text{rule } \text{bijI}[\text{OF } \text{agent.inj-map}[\text{OF } \text{bij-is-inj}[\text{OF } \text{assms}]] \ \text{map-agent.surj}[\text{OF } \text{bij-is-surj}[\text{OF } \text{assms}]])$

setup $\langle \text{Sign.parent-path} \rangle$

definition *swap-env-self-fn* $:: \text{sequential} \Rightarrow \text{sequential}$ **where**

$\text{swap-env-self-fn } a = (\text{case } a \text{ of } \text{proc } () \Rightarrow \text{env} \mid \text{env} \Rightarrow \text{self})$

lemma *swap-env-self-fn-simps*:

shows $\text{swap-env-self-fn } \text{self} = \text{env}$

$\text{swap-env-self-fn } \text{env} = \text{self}$

unfolding *swap-env-self-fn-def* **by** *simp-all*

lemma *bij-swap-env-self-fn*:

shows *bij* swap-env-self-fn

unfolding *swap-env-self-fn-def* *bij-def* *inj-def* *surj-def* **by** $(\text{auto } \text{split: } \text{agent.split})$

lemma *swap-env-self-fn-vimage-singleton*:

shows $\text{swap-env-self-fn} \ -' \ \{\text{env}\} = \{\text{self}\}$

and $\text{swap-env-self-fn} \ -' \ \{\text{self}\} = \{\text{env}\}$

unfolding *swap-env-self-fn-def* **by** $(\text{auto } \text{split: } \text{agent.splits})$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

abbreviation *swap-env-self* $:: (\text{sequential}, 's, 'v) \text{ spec} \Rightarrow (\text{sequential}, 's, 'v) \text{ spec}$ **where**

$\text{swap-env-self} \equiv \text{spec.amap } \text{swap-env-self-fn}$

setup $\langle \text{Sign.parent-path} \rangle$

9.5 Parallel composition

We compose a collection of programs $(\text{sequential}, 's, 'v) \text{ spec}$ in parallel by mapping these into the $('a \text{ agent}, 's, 'v) \text{ spec}$ lattice, taking the infimum, and mapping back.

definition *toConcurrent-fn* $:: 'a \Rightarrow 'a \Rightarrow \text{sequential}$ **where**

$\text{toConcurrent-fn} = (\lambda a \ a'. \text{if } a' = a \text{ then } \text{self} \text{ else } \text{env})$

definition *toSequential-fn* $:: 'a \text{ agent} \Rightarrow \text{sequential}$ **where**

$\text{toSequential-fn} = \text{map-agent } \langle () \rangle$

lemma *toSequential-fn-alt-def*:

shows $toSequential\text{-}fn = (\lambda x. \text{case } x \text{ of } \text{proc } x \Rightarrow \text{self} \mid \text{env} \Rightarrow \text{env})$
by (*simp add: toSequential-fn-def fun-eq-iff split: agent.split*)

setup $\langle \text{Sign.mandatory-path spec} \rangle$

abbreviation $toConcurrent :: 'a \Rightarrow (\text{sequential}, 's, 'v) \text{ spec} \Rightarrow ('a \text{ agent}, 's, 'v) \text{ spec}$ **where**
 $toConcurrent\ a \equiv \text{spec.ainvmap } (toConcurrent\text{-}fn \text{ (proc } a))$

abbreviation $toSequential :: ('a \text{ agent}, 's, 'v) \text{ spec} \Rightarrow (\text{sequential}, 's, 'v) \text{ spec}$ **where**
 $toSequential \equiv \text{spec.amap } toSequential\text{-}fn$

definition $Parallel :: 'a \text{ set} \Rightarrow ('a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}) \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$ **where**
 $Parallel\ as\ Ps = \text{spec.toSequential } (\text{spec.rel } (\text{insert } \text{env } (\text{proc } 'as) \times UNIV) \sqcap (\prod a \in as. \text{spec.toConcurrent } a\ (Ps\ a)))$

definition $parallel :: (\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$ **where**
 $parallel\ P\ Q = \text{spec.Parallel } UNIV\ (\lambda a::\text{bool}. \text{if } a \text{ then } P \text{ else } Q)$

adhoc-overloading

$Parallel \equiv \text{spec.Parallel}$

adhoc-overloading

$parallel \equiv \text{spec.parallel}$

lemma *parallel-alt-def*:

shows $\text{spec.parallel } P\ Q = \text{spec.toSequential } (\text{spec.toConcurrent } \text{True } P \sqcap \text{spec.toConcurrent } \text{False } Q)$
by (*simp add: spec.parallel-def spec.Parallel-def INF-UNIV-bool-expand spec.rel.UNIV*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path } toConcurrent\text{-}fn \rangle$

lemma *simps[simp]*:

shows $toConcurrent\text{-}fn \text{ (proc } a) \text{ (proc } a) = \text{self}$
and $toConcurrent\text{-}fn \text{ (proc } a) \text{ env} = \text{env}$
and $toConcurrent\text{-}fn\ a'\ a'' = \text{self} \longleftrightarrow a'' = a'$
and $\text{self} = toConcurrent\text{-}fn\ a'\ a'' \longleftrightarrow a'' = a'$
and $toConcurrent\text{-}fn\ a'\ a'' = \text{env} \longleftrightarrow a'' \neq a'$
and $\text{env} = toConcurrent\text{-}fn\ a'\ a'' \longleftrightarrow a'' \neq a'$
and $toConcurrent\text{-}fn \text{ (proc } a) \text{ (map-agent } \langle a \rangle x) = \text{map-agent } \langle () \rangle x$
by (*auto simp: toConcurrent-fn-def map-agent.eq-conv intro: agent.exhaust*)

lemma *inj-map-agent*:

assumes $\text{inj-on } f \text{ (insert } x \text{ (set-agent } a))$
shows $toConcurrent\text{-}fn \text{ (proc } (f\ x)) \text{ (map-agent } f\ a) = toConcurrent\text{-}fn \text{ (proc } x) a$
by (*cases a (auto simp: toConcurrent-fn-def intro: inj-onD[OF assms])*)

lemma *inv-into-map-agent*:

fixes $f :: 'a \Rightarrow 'b$
fixes $a :: 'b \text{ agent}$
fixes $x :: 'a$
assumes $\text{inj-on } f\ as$
assumes $x \in as$
assumes $a \in \text{insert } \text{env } ((\lambda x. \text{proc } (f\ x)) 'as)$
shows $toConcurrent\text{-}fn \text{ (proc } x) \text{ (map-agent } (\text{inv-into } as\ f) a) = toConcurrent\text{-}fn \text{ (proc } (f\ x)) a$
using *assms* **by** (*auto simp: toConcurrent-fn-def*)

lemma *vimage-sequential*[simp]:

shows *toConcurrent-fn* (proc a) - ' {self} = {proc a}
and *toConcurrent-fn* (proc a) - ' {env} = -{proc a}
by (auto simp: *toConcurrent-fn-def split: if-splits*)

setup <Sign.parent-path>

setup <Sign.mandatory-path toSequential-fn>

lemma *simps*[simp]:

shows *toSequential-fn* env = env
and *toSequential-fn* (proc x) = self
and *toSequential-fn* (map-agent f a) = *toSequential-fn* a
and *trace.map toSequential-fn id id* $\sigma = \sigma$
and *trace.map toSequential-fn* ($\lambda x. x$) ($\lambda x. x$) $\sigma = \sigma$
and ($\lambda x. \text{if } x = \text{self} \text{ then self else env}$) = id
by (simp-all add: *toSequential-fn-def fun-unit-id*[**where** $f = \lambda x. ()$] *fun-eq-iff flip: id-def*)

lemma *eq-conv*:

shows *toSequential-fn* x = env \longleftrightarrow x = env
and *toSequential-fn* x = self \longleftrightarrow ($\exists a. x = \text{proc } a$)
by (simp-all add: *toSequential-fn-def map-agent.eq-conv*)

lemma *surj*:

shows *surj toSequential-fn*
proof -
have $x \in \text{range } \text{toSequential-fn}$ **for** x
by (cases x)
(simp-all add: *toSequential-fn-def range-eqI*[**where** $x = \text{proc undefined}$] *range-eqI*[**where** $x = \text{env}$])
then show ?thesis **by** blast
qed

lemma *image*[simp]:

assumes $as \neq \{\}$
shows *toSequential-fn* ' proc ' as = {self}
using *assms* **by** (auto simp: *toSequential-fn-def image-image*)

lemma *vimage-sequential*[simp]:

shows *toSequential-fn* - ' {env} = {env}
and *toSequential-fn* - ' {self} = range proc
by (auto simp: *toSequential-fn-def map-agent.eq-conv*)

setup <Sign.parent-path>

lemma *toSequential-fn-eq-toConcurrent-fn-conv*:

shows *toSequential-fn* a = *toConcurrent-fn* a' a'' \longleftrightarrow (case a of env \Rightarrow $a'' \neq a'$ | proc - \Rightarrow $a'' = a'$)
and *toConcurrent-fn* a' a'' = *toSequential-fn* a \longleftrightarrow (case a of env \Rightarrow $a'' \neq a'$ | proc - \Rightarrow $a'' = a'$)
by (simp-all *split: agent.split*)

setup <Sign.mandatory-path spec>

setup <Sign.mandatory-path toSequential>

lemma *interference*:

shows *spec.toSequential* (spec.rel ({env} \times r)) = *spec.rel* ({env} \times r)
by (simp add: *spec.map.rel map-prod-image-Times*)

lemma *interference-inf-toConcurrent*:

```

fixes  $a :: 'a$ 
fixes  $P :: (sequential, 's, 'v) spec$ 
shows  $spec.toSequential (spec.rel (\{env, proc\ a\} \times UNIV) \sqcap spec.toConcurrent a P) = P$  (is ?lhs = ?rhs)
  and  $spec.toSequential (spec.toConcurrent a P \sqcap spec.rel (\{env, proc\ a\} \times UNIV)) = P$  (is ?thesis1)
proof –
  show ?lhs = ?rhs
  proof(rule spec.singleton.antisym)
    have *:  $trace.natural' s (map (map-prod toSequential-fn id) xs)$ 
      =  $trace.natural' s (map (map-prod (toConcurrent-fn (proc a)) id) xs)$ 
      if  $trace.steps' s xs \subseteq \{env, proc\ a\} \times UNIV$ 
      for  $s$  and  $xs :: ('a agent \times 's) list$ 
      using that by (induct xs arbitrary: s) auto
  show  $\langle \sigma \rangle \leq ?rhs$  if  $\langle \sigma \rangle \leq ?lhs$  for  $\sigma$ 
    using that
    by (force simp: spec.singleton.le-conv spec.singleton-le-conv trace.natural-def *
      elim: order.trans[rotated])
  show  $\langle \sigma \rangle \leq ?lhs$  if  $\langle \sigma \rangle \leq ?rhs$  for  $\sigma$ 
    using that
    by (clarsimp intro!: exI[where  $x=trace.map (map-agent \langle a \rangle) id id \sigma$ ]
      simp: spec.singleton.le-conv trace.steps'.map map-agent.eq-conv
      fun-unit-id[where  $f=\lambda::unit. ()$ ]
      simp flip: id-def)
  qed
  then show ?thesis1
    by (simp add: ac-simps)
qed

setup  $\langle Sign.parent-path \rangle$ 

setup  $\langle Sign.mandatory-path toConcurrent \rangle$ 

lemma interference:
  shows  $spec.toConcurrent a (spec.rel (\{env\} \times UNIV)) = spec.rel ((- \{proc\ a\}) \times UNIV)$ 
  by (simp add: spec.invmap.rel map-prod-vimage-Times spec.rel.reflcl)

setup  $\langle Sign.parent-path \rangle$ 

setup  $\langle Sign.mandatory-path idle \rangle$ 

lemma Parallel-le[spec.idle-le]:
  assumes  $\bigwedge a. a \in as \implies spec.idle \leq Ps a$ 
  shows  $spec.idle \leq spec.Parallel as Ps$ 
  apply (simp add: spec.Parallel-def)
  apply (strengthen ord-to-strengthen(2)[OF assms], assumption)
  apply (strengthen ord-to-strengthen(2)[OF spec.idle.invmap-le[OF order.refl]])
  apply (simp add: le-INF-iff spec.idle-le)
  done

setup  $\langle Sign.parent-path \rangle$ 

setup  $\langle Sign.mandatory-path Parallel \rangle$ 

lemma cong:
  assumes  $as = as'$ 
  assumes  $\bigwedge a. a \in as' \implies Ps a = Ps' a$ 
  shows  $spec.Parallel as Ps = spec.Parallel as' Ps'$ 
unfolding spec.Parallel-def using assms by simp

```

lemma *no-agents*:

shows $\text{spec.Parallel } \{\} Ps = \text{spec.rel } (\{\text{env}\} \times \text{UNIV})$

by (*simp add: spec.Parallel-def spec.toSequential.interference*)

lemma *singleton-agents*:

shows $\text{spec.Parallel } \{a\} Ps = Ps a$

by (*simp add: spec.Parallel-def spec.toSequential.interference-inf-toConcurrent*)

lemma *bot*:

assumes $P_s a = \perp$

assumes $a \in as$

shows $\text{spec.Parallel } as Ps = \perp$

by (*simp add: spec.Parallel-def assms INF-unwind-index[of a] spec.invmap.bot spec.map.bot*)

lemma *top*:

shows $\text{spec.Parallel } as \top = (\text{if } as = \{\} \text{ then } \text{spec.rel } (\{\text{env}\} \times \text{UNIV}) \text{ else } \top)$

proof –

have $\text{spec.toSequential } (\text{spec.rel } (\text{insert env } (\text{proc } 'as) \times \text{UNIV})) = \top$ **if** $as \neq \{\}$

using *that by (subst spec.map.rel, force, simp add: map-prod-image-Times flip: spec.rel.UNIV)*

then show *?thesis*

by (*simp add: spec.Parallel.no-agents*) (*auto simp: spec.Parallel-def spec.invmap.top*)

qed

lemma *mono*:

assumes $\bigwedge a. a \in as \implies P_s a \leq P_{s'} a$

shows $\text{spec.Parallel } as Ps \leq \text{spec.Parallel } as Ps'$

unfolding *spec.Parallel-def* **by** (*strengthen ord-to-strengthen(1)[OF assms(1)]; simp*)

lemma *strengthen[strg]*:

assumes $\bigwedge a. a \in as \implies \text{st-ord } F (P_s a) (P_{s'} a)$

shows $\text{st-ord } F (\text{spec.Parallel } as Ps) (\text{spec.Parallel } as Ps')$

using *assms by (cases F) (auto simp: spec.Parallel.mono)*

lemma *mono2mono[cont-intro, partial-function-mono]*:

fixes $P_s :: 'a \Rightarrow 'b \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$

assumes $\bigwedge a. a \in as \implies \text{monotone } \text{orda } (\leq) (P_s a)$

shows $\text{monotone } \text{orda } (\leq) (\lambda x :: 'b. \text{spec.Parallel } as (\lambda a. P_s a x))$

using *spec.Parallel.mono assms unfolding monotone-def by meson*

lemma *invmap*: — *af = id* in *spec.invmap*

shows $\text{spec.invmap } id \text{ sf } vf (\text{spec.Parallel } \text{UNIV } Ps) = \text{spec.Parallel } \text{UNIV } (\text{spec.invmap } id \text{ sf } vf \circ Ps)$

by (*simp add: spec.Parallel-def image-image spec.invmap.inf spec.invmap.Inf spec.invmap.comp spec.rel.UNIV flip: spec.amap.surj-invmap[OF toSequential-fn.surj]*)

lemma *discard-interference*:

assumes $\bigwedge a. a \in bs \implies P_s a = \text{spec.rel } (\{\text{env}\} \times \text{UNIV})$

shows $\text{spec.Parallel } as Ps = \text{spec.Parallel } (as - bs) Ps$

proof –

have $*$: $as = (as - bs) \cup (as \cap bs)$ **by** *blast*

have $**$: $(\text{insert env } (\text{proc } 'as) \cap - \text{proc } '(as \cap bs)) = \text{insert env } (\text{proc } '(as - bs))$ **by** *blast*

from *assms* **have** $***$: $(\bigcap a \in as \cap bs. \text{spec.toConcurrent } a (P_s a))$

$= \text{spec.rel } ((- \text{proc } '(as \cap bs)) \times \text{UNIV})$

by (*force simp: assms spec.toConcurrent.interference le-Inf-iff*

simp flip: spec.rel.INF

intro: spec.rel.mono antisym)

show *?thesis*

```

apply (simp add: spec.Parallel-def)
apply (subst (2) *)
apply (simp add: image-Un Inf-union-distrib ac-simps ** *** Times-Int-Times
  flip: spec.rel.inf inf.assoc)
done
qed

lemma rename-UNIV-aux:
  fixes f :: 'a ⇒ 'b
  assumes inj-on f as
  shows spec.toSequential (spec.rel (insert env (proc ' as) × UNIV)
    □ (□ a∈as. spec.toConcurrent a (Ps a)))
    = spec.toSequential (spec.rel (insert env (proc ' f ' as) × UNIV)
    □ (□ a∈as. spec.toConcurrent (f a) (Ps a))) (is ?lhs = ?rhs)
proof(rule spec.singleton.antisym)
  show ⟨σ⟩ ≤ ?rhs if ⟨σ⟩ ≤ ?lhs for σ
    using that assms
    apply (clarsimp simp: spec.singleton.le-conv le-Inf-iff)
    apply (rule exI[where x=trace.map (map-agent f) id id σ for σ])
    apply (intro conjI)
    apply (fastforce simp: trace.steps'.map)
    apply (fastforce intro: ord-eq-le-trans[OF spec.singleton.map-cong[OF toConcurrent-fn.inj-map-agent refl refl
refl]])
      dest: inj-onD trace.steps'.asetD
      simp flip: id-def)
  apply (fastforce simp flip: id-def)
done
show ⟨σ⟩ ≤ ?lhs if ⟨σ⟩ ≤ ?rhs for σ
  using that assms
  apply (clarsimp simp: spec.singleton.le-conv le-Inf-iff image-image)
  apply (rule exI[where x=trace.map (map-agent (inv-into as f)) id id σ for σ])
  apply (auto 4 2 dest: trace.steps'.asetD
    simp: spec.singleton.map-cong[OF toConcurrent-fn.inv-into-map-agent refl refl refl]
    comp-def trace.steps'.map
    simp flip: id-def)
done
qed

lemma rename-UNIV: — expand the set of agents to UNIV
  fixes f :: 'a ⇒ 'b
  assumes inj-on f as
  shows spec.Parallel as Ps
    = spec.Parallel (UNIV :: 'b set)
      (λb. if b ∈ f ' as then Ps (inv-into as f b) else spec.rel ({env} × UNIV))
  (is ?lhs = spec.Parallel - ?f)
proof —
  have *: (□ x. spec.toConcurrent x (?f x))
    = spec.rel (insert env (proc ' f ' as) × UNIV)
      □ (□ x∈f ' as. spec.toConcurrent x (Ps (inv-into as f x)))
proof —
  have *: (□ x∈- f ' as. (- {proc x}) × UNIV) = insert env (proc ' f ' as) × UNIV
    by (auto intro: agent.exhaust)
  have (□ x. spec.toConcurrent x (?f x))
    = (□ x∈f ' as. spec.toConcurrent x (?f x)) □ (□ x∈- f ' as. spec.toConcurrent x (?f x))
    by (subst INF-union[symmetric]) simp
  also have ... = spec.rel (insert env (proc ' f ' as) × UNIV)
    □ (□ x∈f ' as. spec.toConcurrent x (Ps (inv-into as f x)))
    by (simp add: ac-simps spec.invmmap.rel map-prod-vimage-Times spec.rel.reflcl *)

```

flip: spec.rel.upper-INF)

finally show *?thesis* .

qed

show *?thesis*

by (*simp add: spec.Parallel-def * inv-into-f-f[OF assms] spec.rel.UNIV
INF-rename-bij[OF inj-on-imp-bij-betw[OF assms],
where F= λ - x. spec.toConcurrent x (Ps (inv-into as f x))]
spec.Parallel.rename-UNIV-aux[OF assms]*)

qed

lemma *rename:*

fixes $\pi :: 'a \Rightarrow 'b$

fixes $P_s :: 'b \Rightarrow (\text{sequential}, 's, \text{unit}) \text{spec}$

assumes *bij-betw* π *as* *bs*

shows *spec.Parallel as* ($P_s \circ \pi$) = *spec.Parallel bs Ps*

proof –

define π' **where** $\pi' = (\lambda x :: 'a + 'b. \text{case } x \text{ of}$
 $\text{Inl } a \Rightarrow \text{if } a \in \text{as} \text{ then } \text{Inr } (\pi \ a) \text{ else } \text{Inl } a$
 $| \text{Inr } b \Rightarrow \text{if } b \in \text{bs} \text{ then } \text{Inl } (\text{inv-into as } \pi \ b) \text{ else } \text{Inr } b)$

from *assms* **have** *inj* π'

by (*force intro: injI*
 simp: π' -def bij-betw-apply bij-betw-imp-surj-on inv-into-into
 split: sum.split-asm if-split-asm
 dest: bij-betw-inv-into-left[rotated] bij-betw-inv-into-right[rotated])

have *simps*: $\bigwedge a. \pi' (\text{Inl } a) = (\text{if } a \in \text{as} \text{ then } \text{Inr } (\pi \ a) \text{ else } \text{Inl } a)$
 $\bigwedge b. \pi' (\text{Inr } b) = (\text{if } b \in \text{bs} \text{ then } \text{Inl } (\text{inv-into as } \pi \ b) \text{ else } \text{Inr } b)$

by (*simp-all add: π' -def*)

have *inv-simps*: $\bigwedge a. a \in \text{as} \Longrightarrow \text{inv } \pi' (\text{Inl } a) = \text{Inr } (\pi \ a)$

by (*simp add: inv-f-eq[OF <inj π' >] bij-betw-inv-into-left[OF assms] bij-betw-apply[OF assms] simps(2)*)

show *?thesis*

apply (*simp add: spec.Parallel.rename-UNIV[where as=as and f=Inl :: 'a \Rightarrow 'a + 'b]
spec.Parallel.rename-UNIV[where as=bs and f=Inr :: 'b \Rightarrow 'a + 'b] comp-def*)

apply (*subst (2) spec.Parallel.rename-UNIV[where as=UNIV, OF <inj π' >]*)

apply (*fastforce intro: arg-cong[where f=spec.Parallel UNIV]
simp: fun-eq-iff split-sum-all image-iff simps inv-simps
inv-f-f[OF <inj π' >] bij-betw-apply[OF bij-betw-inv-into[OF assms]]
bij-betw-apply[OF assms] bij-betw-inv-into-left[OF assms]*)

done

qed

lemma *rename-cong:*

fixes $\pi :: 'a \Rightarrow 'b$

fixes $P_s :: 'a \Rightarrow (-, -, -) \text{spec}$

fixes $P_s' :: 'b \Rightarrow (-, -, -) \text{spec}$

assumes *bij-betw* π *as* *bs*

assumes $\bigwedge a. a \in \text{as} \Longrightarrow P_s \ a = P_s' (\pi \ a)$

shows *spec.Parallel as Ps* = *spec.Parallel bs Ps'*

by (*simp add: assms(2) flip: spec.Parallel.rename[OF assms(1)] cong: spec.Parallel.cong*)

lemma *inf-pre:*

assumes $\text{as} \neq \{\}$

shows *spec.Parallel as Ps* \sqcap *spec.pre P* = ($\|i \in \text{as}. P_s \ i \sqcap \text{spec.pre } P$) (**is** *?thesis1*)

and *spec.pre P* \sqcap *spec.Parallel as Ps* = ($\|i \in \text{as}. \text{spec.pre } P \sqcap P_s \ i$) (**is** *?thesis2*)

proof –

show *?thesis1*

by (*simp add: spec.Parallel-def assms spec.invmap.inf spec.invmap.pre spec.map.inf-distr
inf.assoc INF-inf-const2*)

then show *?thesis2*

by (simp add: ac-simps)

qed

lemma inf-post:

assumes $as \neq \{\}$

shows $spec.Parallel\ as\ Ps \sqcap spec.post\ Q = spec.Parallel\ as\ (\lambda i. Ps\ i \sqcap spec.post\ Q)$ (is ?thesis1)

and $spec.post\ Q \sqcap spec.Parallel\ as\ Ps = spec.Parallel\ as\ (spec.post\ Q \sqcap Ps\ i)$ (is ?thesis2)

proof –

show ?thesis1

by (simp add: spec.Parallel-def assms spec.invmap.inf spec.invmap.post spec.map.inf-distr
inf.assoc INF-inf-const2)

then show ?thesis2

by (simp add: ac-simps)

qed

lemma unwind:

— All other processes begin with interference

assumes $b: \bigwedge b. b \in as - \{a\} \implies spec.rel\ (\{env\} \times UNIV) \ggg (\lambda::unit. Ps\ b) \leq Ps\ b$

assumes $a: f \ggg g \leq Ps\ a$ — The selected process starts with f

assumes $a \in as$

shows $f \ggg (\lambda v. spec.Parallel\ as\ (Ps(a := g\ v))) \leq spec.Parallel\ as\ Ps$

proof –

have *: $spec.toConcurrent\ a\ f \sqcap spec.rel\ (\bigcap x \in as - \{a\}. (-\{proc\ x\}) \times UNIV)$

$\ggg (\lambda v. \bigcap b \in as. spec.toConcurrent\ b\ ((Ps(a:=g\ v))\ b))$

$\leq (\bigcap a \in as. spec.toConcurrent\ a\ (Ps\ a))$ (is ?lhs \leq ?rhs)

proof –

from $\langle a \in as \rangle$

have ?lhs = $spec.toConcurrent\ a\ f \sqcap spec.rel\ (\bigcap x \in as - \{a\}. (-\{proc\ x\}) \times UNIV)$

$\ggg (\lambda v. spec.toConcurrent\ a\ (g\ v) \sqcap (\bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (Ps\ b)))$

by (simp add: INF-unwind-index)

also have ... $\leq (spec.toConcurrent\ a\ f \ggg (\lambda x. spec.toConcurrent\ a\ (g\ x)))$

$\sqcap (spec.rel\ (\bigcap x \in as - \{a\}. (-\{proc\ x\}) \times UNIV)$

$\ggg (\lambda::unit. \bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (Ps\ b)))$

by (strengthen ord-to-strengthen(2)[OF spec.bind.inf-rel-distr-le]) simp

also have ... = $(spec.toConcurrent\ a\ f \ggg (\lambda x. spec.toConcurrent\ a\ (g\ x)))$

$\sqcap ((\bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (spec.rel\ (\{env\} \times UNIV)))$

$\ggg (\lambda::unit. \bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (Ps\ b)))$

by (simp add: spec.invmap.rel map-prod-vimage-Times spec.rel.reflcl flip: spec.rel.INF)

also have ... $\leq (spec.toConcurrent\ a\ f \ggg (\lambda x. spec.toConcurrent\ a\ (g\ x)))$

$\sqcap (\bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (spec.rel\ (\{env\} \times UNIV)))$

$\ggg (\lambda::unit. spec.toConcurrent\ b\ (Ps\ b)))$

by (strengthen ord-to-strengthen(2)[OF spec.bind.Inf-le]) simp

also have ... = $spec.toConcurrent\ a\ (f \ggg g)$

$\sqcap (\bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (spec.rel\ (\{env\} \times UNIV) \ggg (\lambda::unit. Ps\ b)))$

by (simp add: spec.invmap.bind)

also have ... $\leq spec.toConcurrent\ a\ (Ps\ a) \sqcap (\bigcap b \in as - \{a\}. spec.toConcurrent\ b\ (Ps\ b))$

by (strengthen ord-to-strengthen(2)[OF a], strengthen ord-to-strengthen(2)[OF b], assumption, rule order.refl)

also from $\langle a \in as \rangle$ have ... = ?rhs by (simp add: INF-unwind-index)

finally show ?thesis .

qed

from $\langle a \in as \rangle$

have **: $(insert\ env\ (proc\ 'as) \times UNIV \cap (\bigcap x \in as - \{a\}. (-\{proc\ x\}) \times UNIV)) = \{env, proc\ a\} \times UNIV$

by blast

show ?thesis

unfolding spec.Parallel-def

by (strengthen ord-to-strengthen(2)[OF *])

(simp add: ac-simps spec.bind.inf-rel spec.map.bind-inj-sf **

spec.toSequential.interference-inf-toConcurrent

flip: spec.rel.inf)

qed

lemma *inf-rel*:

fixes *as* :: 'a set

fixes *r* :: 's rel

shows *spec.rel* ($\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r$) \sqcap *spec.Parallel as Ps*

= *spec.Parallel as* ($\lambda a. \text{spec.rel} (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap Ps a$) (is ?lhs = ?rhs)

and *spec.Parallel as Ps* \sqcap *spec.rel* ($\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r$)

= *spec.Parallel as* ($\lambda a. Ps a \sqcap \text{spec.rel} (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r)$) (is ?thesis1)

proof –

show ?lhs = ?rhs

proof(cases *as* = {})

case *True* then show ?thesis

by (simp add: *spec.Parallel.no-agents flip: spec.rel.inf*)

next

case *False* show ?thesis

proof(rule *antisym*)

have *: *insert env* (*proc* ' *as*) $\times \text{UNIV} \sqcap$ *map-prod toSequential-fn id* –' ($\text{UNIV} \times \text{Id} \cup (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r)$)

\subseteq *insert env* (*proc* ' *as*) $\times \text{UNIV} \sqcap$ *map-prod* (*toConcurrent-fn* (*proc a*)) *id* –' ($\text{UNIV} \times \text{Id} \cup ((\{\text{env}\} \times \text{UNIV}) \cup \{\text{self}\} \times r)$) for *a*

by *auto*

from *False*

show ?lhs \leq ?rhs

apply (simp add: *spec.Parallel-def ac-simps spec.map.inf-rel*

flip: spec.rel.inf spec.invmap.inf-rel INF-inf-const1 INF-inf-const2

del: vimage-Un)

apply (*strengthen ord-to-strengthen*(1)[*OF* *])

apply (rule *order.refl*)

done

have *spec.toSequential* ($\prod x \in as. \text{spec.toConcurrent } x (Ps x) \sqcap \text{spec.rel} (\text{insert env} (\text{proc } ' as) \times \text{UNIV} \sqcap \text{map-prod} (\text{toConcurrent-fn} (\text{proc } x)) \text{ id } -' (\text{UNIV} \times \text{Id} \cup (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r)))$)

$\leq \text{spec.toSequential} (\prod x \in as. \text{spec.toConcurrent } x (Ps x) \sqcap \text{spec.rel} (\text{insert env} (\text{proc } ' as) \times \text{UNIV} \sqcap \text{map-prod } \text{toSequential-fn id } -' (\text{UNIV} \times \text{Id} \cup (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r)))$)

apply (rule *spec.singleton-le-extI*)

apply (*clarsimp simp: spec.singleton.le-conv le-INF-iff*)

apply (*rename-tac* $\sigma \sigma'$)

apply (rule-tac $x = \sigma'$ in *exI*)

apply (*clarsimp simp: toConcurrent-fn-def toSequential-fn-def trace.split-all*)

apply (*rename-tac* $\sigma \sigma' a s s' a'$)

apply (*case-tac a*)

apply (*case-tac the-agent a* $\in as$; *force*)

apply *simp*

done

with *False*

show ?rhs \leq ?lhs

by (simp add: *spec.Parallel-def ac-simps spec.map.inf-rel*

flip: INF-inf-const1 INF-inf-const2 spec.invmap.inf-rel spec.rel.inf)

qed

qed

then show ?thesis1

by (simp add: *ac-simps*)

qed

lemma *flatten*:

fixes *as* :: 'a set

fixes *a* :: 'a

```

fixes  $bs :: 'b \text{ set}$ 
fixes  $P_s :: 'a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$ 
fixes  $P_{s'} :: 'b \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$ 
assumes  $P_s a = \text{spec.Parallel } bs P_{s'}$ 
assumes  $a \in as$ 
shows  $\text{spec.Parallel } as P_s = \text{spec.Parallel } ((as - \{a\}) <+> bs) (\text{case-sum } P_s P_{s'})$  (is  $?lhs = ?rhs$ )
proof(rule spec.singleton.antisym)
  have simps:
     $\bigwedge a'. a' \neq a \implies (\lambda x :: ('a + 'b) \text{ agent. toConcurrent-fn } (\text{proc } a') (\text{case } x \text{ of proc } (\text{Inl } a) \Rightarrow \text{proc } a \mid \text{proc } (\text{Inr } -) \Rightarrow \text{proc } a \mid \text{env} \Rightarrow \text{env})) = \text{toConcurrent-fn } (\text{proc } (\text{Inl } a'))$ 
     $\bigwedge a'. (\lambda x :: ('a + 'b) \text{ agent. toConcurrent-fn } (\text{proc } a') (\text{case } x \text{ of proc } (\text{Inl } -) \Rightarrow \text{env} \mid \text{proc } (\text{Inr } a) \Rightarrow \text{proc } a \mid \text{env} \Rightarrow \text{env})) = \text{toConcurrent-fn } (\text{proc } (\text{Inr } a'))$ 
  by (auto simp: fun-eq-iff toConcurrent-fn-def split: agent.split sum.split)
  have *:
     $\exists \sigma''' :: (('a + 'b) \text{ agent}, 's, \text{unit}) \text{ trace.t.}$ 
     $\langle \sigma' \rangle = \langle \text{trace.map } (\text{case-agent } (\text{case-sum } \text{proc } \langle \text{proc } a \rangle) \text{ env}) \text{ id id } \sigma''' \rangle$ 
     $\wedge \langle \text{trace.map } (\text{case-agent } (\text{case-sum } \langle \text{env} \rangle \text{proc}) \text{ env}) \text{ id id } \sigma''' \rangle \leq \langle \sigma' \rangle$ 
     $\wedge \text{proc } (\text{Inl } a) \notin \text{trace.aset } (\text{h}\sigma''')$ 
    if  $\langle \text{trace.map } (\text{toConcurrent-fn } (\text{proc } a)) \text{ id id } \sigma' \rangle \leq \langle \text{trace.map } \text{toSequential-fn } \text{id id } \sigma' \rangle$ 
  for  $\sigma' :: ('a \text{ agent}, 's, \text{unit}) \text{ trace.t}$ 
  and  $\sigma'' :: ('b \text{ agent}, 's, \text{unit}) \text{ trace.t}$ 
proof(cases trace.term  $\sigma'$ )
  case None
  have  $\exists zs :: (('a + 'b) \text{ agent} \times 's) \text{ list.}$ 
     $xs = \text{map } (\text{map-prod } (\text{case-agent } (\text{case-sum } \text{proc } (\lambda s. \text{proc } a)) \text{ env}) \text{ id}) \text{ zs}$ 
     $\wedge \text{prefix } (\text{map } (\text{map-prod } (\text{case-agent } (\text{case-sum } (\lambda s. \text{env})) \text{proc}) \text{ env}) \text{ id}) \text{ zs} \text{ ys}$ 
     $\wedge (\text{proc } (\text{Inl } a) \notin \text{fst } ' \text{set } zs) (\text{is } \exists zs. ?\text{goal } xs \text{ zs})$ 
  if  $\text{prefix } (\text{map } (\text{map-prod } (\text{toConcurrent-fn } (\text{proc } a)) \text{ id}) \text{ xs}) (\text{map } (\text{map-prod } \text{toSequential-fn } \text{id}) \text{ ys})$ 
  for  $xs :: ('a \text{ agent} \times 's) \text{ list}$  and  $ys :: ('b \text{ agent} \times 's) \text{ list}$ 
  using that
proof(induct xs rule: rev-induct)
  case (snoc x xs)
  then obtain  $zs$  where  $?\text{goal } xs \text{ zs}$  by (auto dest: append-prefixD)
  with snoc.prems show  $?\text{case}$ 
  apply (clarsimp simp: map-prod.comp map-prod-conv simp flip: id-def elim!: prefixE)
  subgoal for  $a_y$ 
  by (rule exI[where x=zs @ [(if fst x = proc a then map-agent Inr a_y else map-agent Inl (fst x), (snd x)]]])
    (auto simp: toSequential-fn.eq-conv map-agent.eq-conv simp flip: all-simps split: agent.split)
  done
qed simp
from this[of (trace.natural' (trace.init  $\sigma'$ ) (trace.rest  $\sigma'$ ))
  trace.natural' (trace.init  $\sigma'$ ) (trace.rest  $\sigma''$ )] that None
show  $?\text{thesis}$ 
apply (simp add: spec.singleton-le-conv trace.natural.map-inj-on-sf)
apply (clarsimp simp: trace.natural-def trace.aset.simps trace.split-Ex image-iff trace.less-eq-None)
subgoal for  $zs$ 
by (clarsimp dest!: trace.natural'.amap-noop intro!: exI[where x=zs])
done
next
case (Some v)
have *:  $\exists zs :: (('a + 'b) \text{ agent} \times 's) \text{ list.}$ 
     $xs = \text{map } (\text{map-prod } (\text{case-agent } (\text{case-sum } \text{proc } (\lambda s. \text{proc } a)) \text{ env}) \text{ id}) \text{ zs}$ 
     $\wedge \text{ys} = \text{map } (\text{map-prod } (\text{case-agent } (\text{case-sum } (\lambda s. \text{env})) \text{proc}) \text{ env}) \text{ id}) \text{ zs}$ 
     $\wedge (\text{proc } (\text{Inl } a) \notin \text{fst } ' \text{set } zs)$ 
  if  $\text{map } (\text{map-prod } (\text{toConcurrent-fn } (\text{proc } a)) \text{ id}) \text{ xs} = \text{map } (\text{map-prod } \text{toSequential-fn } \text{id}) \text{ ys}$ 
  for  $xs :: ('a \text{ agent} \times 's) \text{ list}$  and  $ys :: ('b \text{ agent} \times 's) \text{ list}$ 
proof –

```

```

from that have  $\text{length } xs = \text{length } ys$ 
  using map-eq-imp-length-eq by blast
from this that show ?thesis
proof(induct rule: list-induct2)
  case (Cons x xs y ys) then show ?case
    by (cases x, cases y, cases fst x)
      (auto 8 0 simp: Cons-eq-map-conv comp-def toSequential-fn-eq-toConcurrent-fn-conv
        simp flip: id-def ex-simps
        split: if-splits agent.splits sum.split)
  qed simp
qed
from that Some
  *[where  $xs = \text{trace.natural}' (\text{trace.init } \sigma') (\text{trace.rest } \sigma')$ 
    and  $ys = \text{trace.natural}' (\text{trace.init } \sigma') (\text{trace.rest } \sigma'')$ 
  show ?thesis
  apply (simp add: spec.singleton-le-conv trace.natural.map-inj-on-sf)
  apply (clarsimp simp: trace.natural-def)
  subgoal for zs
    by (clarsimp simp: trace.split-Ex trace.aset.simps
      dest!: trace.natural'.amap-noop
      intro!: exI[where x=zs])
  done
qed
{
  fix  $\sigma$ 
  assume  $\langle \sigma \rangle \leq ?lhs$ 
  then obtain  $\sigma_a \sigma_b \sigma_c$ 
    where 1:  $\text{trace.steps}' (\text{trace.init } \sigma_a) (\text{trace.rest } \sigma_a) \subseteq \text{insert env } (\text{proc } 'as) \times \text{UNIV}$ 
      and 2:  $\forall x \in as. \langle \text{trace.map } (\text{toConcurrent-fn } (\text{proc } x)) \text{ id id } \sigma_a \rangle \leq Ps \ x$ 
      and 3:  $\langle \sigma \rangle \leq \langle \text{trace.map } \text{toSequential-fn id id } \sigma_a \rangle$ 
      and 4:  $\text{trace.steps}' (\text{trace.init } \sigma_b) (\text{trace.rest } \sigma_b) \subseteq \text{insert env } (\text{proc } 'bs) \times \text{UNIV}$ 
      and 5:  $\forall x \in bs. \langle \text{trace.map } (\text{toConcurrent-fn } (\text{proc } x)) \text{ id id } \sigma_b \rangle \leq Ps' \ x$ 
      and 6:  $\sigma_a \simeq_S \text{trace.map } (\text{case-agent } (\text{case-sum } \text{proc } \langle \text{proc } a \rangle) \text{env}) \text{ id id } \sigma_c$ 
      and 7:  $\langle \text{trace.map } (\text{case-agent } (\text{case-sum } \langle \text{env} \rangle \text{proc}) \text{env}) \text{ id id } \sigma_c \rangle \leq \langle \sigma_b \rangle$ 
      and 8:  $\text{proc } (\text{Inl } a) \notin \text{trace.aset } (\ddagger \sigma_c)$ 
    apply (clarsimp simp: spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
    apply (frule bspec[OF - ⟨a ∈ as⟩])
    apply (clarsimp simp: assms(1) spec.Parallel-def spec.singleton.le-conv le-Inf-iff dest!: *)
    done
  show  $\langle \sigma \rangle \leq ?rhs$ 
    unfolding spec.Parallel-def spec.singleton.le-conv inf.bounded-iff le-Inf-iff ball-simps
  proof(intro exI[where x=σc] conjI ballI)
    show  $\text{trace.steps}' (\text{trace.init } \sigma_c) (\text{trace.rest } \sigma_c)$ 
       $\subseteq \text{insert env } (\text{proc } '((as - \{a\}) \langle + \rangle bs)) \times \text{UNIV}$  (is  $?lhs \subseteq ?rhs$ )
  proof(rule subsetI, unfold split-paired-all)
    show  $(x, s, s') \in ?rhs$  if  $(x, s, s') \in ?lhs$  for  $x \ s \ s'$ 
    proof(cases x)
      case (proc y) then show ?thesis
      proof(cases y)
        case (Inl a)
          with that proc 1 arg-cong[OF 6, where f=trace.steps] 8
          show ?thesis
            by (fastforce simp: trace.steps'.natural' trace.steps'.map trace.aset.natural-conv)
        next
          case (Inr b)
          with that proc 4 spec.steps.mono[OF 7]
          show ?thesis
            by (fastforce simp: trace.steps'.natural' trace.steps'.map spec.steps.singleton)

```

```

qed
qed simp
qed
show  $\langle \text{trace.map (toConcurrent-fn (proc x)) id id } \sigma_c \rangle \leq (\text{case } x \text{ of } \text{Inl } x \Rightarrow \text{Ps } x \mid \text{Inr } x \Rightarrow \text{Ps}' x)$ 
  if  $x \in (as - \{a\}) <+> bs$ 
  for  $x$ 
proof(cases  $x$ )
  case (Inl  $l$ ) with 2 6 that show ?thesis
    by (force dest: trace.stuttering.equiv.map[where  $af = \text{toConcurrent-fn (proc } l)$  and  $sf = \text{id}$  and  $vf = \text{id}$ ]
      simp: sum-In-conv_simps fun-unit-id[where  $f = \lambda::\text{unit. ()}$ ]
      simp flip: id-def
      cong: spec.singleton-cong)
  next
  case (Inr  $r$ ) with 2 5 6 7 that show ?thesis
    by (fastforce dest: spec.singleton.map-le[where  $af = \text{toConcurrent-fn (proc } r)$  and  $sf = \text{id}$  and  $vf = \text{id}$ ]
      simp:_simps fun-unit-id[where  $f = \lambda::\text{unit. ()}$ ]
      simp flip: id-def
      cong: spec.singleton-cong)
qed
from 3 6
show  $\langle \sigma \rangle \leq \langle \text{trace.map toSequential-fn id id } \sigma_c \rangle$ 
  by (fastforce dest: trace.stuttering.equiv.map[where  $af = \text{toSequential-fn}$  and  $sf = \text{id}$  and  $vf = \text{id}$ ]
    simp: spec.singleton-le-conv id-def agent.case-distrib sum.case-distrib
    simp flip: toSequential-fn-alt-def
    cong: agent.case-cong sum.case-cong)
qed
}
}
fix  $\sigma$ 
assume  $\langle \sigma \rangle \leq ?rhs$ 
then obtain  $\sigma'$ 
  where 1:  $\text{trace.steps}' (\text{trace.init } \sigma') (\text{trace.rest } \sigma') \subseteq \text{insert env (proc ' ((as - \{a\}) <+> bs)) \times UNIV}$ 
  and 2:  $\forall x \in (as - \{a\}) <+> bs. \langle \text{trace.map (toConcurrent-fn (proc } x)) id id } \sigma' \rangle \leq (\text{case } x \text{ of } \text{Inl } x \Rightarrow \text{Ps}$ 
 $x \mid \text{Inr } x \Rightarrow \text{Ps}' x)$ 
  and 3:  $\langle \sigma \rangle \leq \langle \text{trace.map toSequential-fn id id } \sigma' \rangle$ 
  by (clarsimp simp: spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
show  $\langle \sigma \rangle \leq ?lhs$ 
proof -
  from  $\langle a \in as \rangle 1$ 
  have  $\text{trace.steps}' (\text{trace.init } \sigma') (\text{map (map-prod (case-agent (case-sum proc } \langle \text{proc } a \rangle) \text{env}) id) (\text{trace.rest}$ 
 $\sigma'))$ 
     $\subseteq \text{insert env (proc ' as) \times UNIV}$ 
  by (auto simp: trace.steps'.map)
  moreover
  have  $\langle \text{trace.map } (\lambda y. \text{toConcurrent-fn (proc } x) (\text{case } y \text{ of } \text{proc (Inl } a) \Rightarrow \text{proc } a \mid \text{proc (Inr } b) \Rightarrow \text{proc } a \mid$ 
 $\text{env} \Rightarrow \text{env})) id id } \sigma' \rangle \leq \text{Ps } x$ 
  if  $x \in as$ 
  for  $x$ 
proof(cases  $x = a$ )
  case True show ?thesis
  proof -
    from  $\langle a \in as \rangle 1$ 
    have  $\text{trace.steps}' (\text{trace.init } \sigma') (\text{map (map-prod (case-agent (case-sum } \langle \text{env} \rangle \text{proc}) \text{env}) id) (\text{trace.rest}$ 
 $\sigma'))$ 
       $\subseteq \text{insert env (proc ' bs) \times UNIV}$ 
    by (auto simp: trace.steps'.map)
  moreover
  from 2

```

```

have  $\langle \text{trace.map } (\lambda y. \text{toConcurrent-fn } (\text{proc } b) (\text{case } y \text{ of proc } (\text{Inl } a) \Rightarrow \text{env} \mid \text{proc } (\text{Inr } b) \Rightarrow \text{proc } b \mid \text{env} \Rightarrow \text{env})) \text{ id id } \sigma' \rangle \leq Ps' b$ 
  if  $b \in bs$ 
  for  $b$ 
  using that by (fastforce simp:_simps dest: bspec[where x=Inr b])
moreover
from 1
have  $\langle \text{trace.map } (\lambda y. \text{toConcurrent-fn } (\text{proc } a) (\text{case } y \text{ of proc } (\text{Inl } a) \Rightarrow \text{proc } a \mid \text{proc } (\text{Inr } b) \Rightarrow \text{proc } a \mid \text{env} \Rightarrow \text{env})) \text{ id id } \sigma' \rangle$ 
   $\leq \langle \text{trace.map } (\lambda y. \text{toSequential-fn } (\text{case } y \text{ of proc } (\text{Inl } a) \Rightarrow \text{env} \mid \text{proc } (\text{Inr } b) \Rightarrow \text{proc } b \mid \text{env} \Rightarrow \text{env})) \text{ id id } \sigma' \rangle$ 
  by (subst spec.singleton.map-cong[where af'= $\lambda y. \text{toSequential-fn } (\text{case } y \text{ of proc } (\text{Inl } s) \Rightarrow \text{env} \mid \text{proc } (\text{Inr } b) \Rightarrow \text{proc } b \mid \text{env} \Rightarrow \text{env})$ , OF - refl refl refl];
    fastforce simp: trace.aset.natural-conv split: agent.split sum.split)
ultimately show ?thesis
  apply (simp add:  $\langle x = a \rangle$  assms(1) spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
  apply (rule exI[where x= $\text{trace.map } (\text{case-agent } (\text{case-sum } \langle \text{env} \rangle \text{proc}) \text{env}) \text{ id id } \sigma'$ ] conjI ballI)
  apply (intro conjI ballI)
  apply (simp-all add: fun-unit-id[where f= $\lambda :: \text{unit. } ()$ ] flip: id-def)
done
qed
next
case False with 2 that show ?thesis
  by (fastforce simp:_simps dest: bspec[where x=Inl x])
qed
moreover
from 3
have  $\langle \sigma \rangle \leq \langle \text{trace.map } (\lambda x. \text{toSequential-fn } (\text{case } x \text{ of proc } (\text{Inl } a) \Rightarrow \text{proc } a \mid \text{proc } (\text{Inr } b) \Rightarrow \text{proc } a \mid \text{env} \Rightarrow \text{env})) \text{ id id } \sigma' \rangle$ 
  by (simp add: agent.case-distrib sum.case-distrib
    flip: toSequential-fn-alt-def
    cong: agent.case-cong sum.case-cong)
ultimately show ?thesis
  apply (simp add: spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
  apply (rule exI[where x= $\text{trace.map } (\text{case-agent } (\text{case-sum } \text{proc } \langle \text{proc } a \rangle) \text{env}) \text{ id id } \sigma'$ ] conjI ballI)
  apply (simp add: fun-unit-id[where f= $\lambda :: \text{unit. } ()$ ] flip: id-def)
done
qed
}
qed

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.mandatory-path term} \rangle$ 

setup  $\langle \text{Sign.mandatory-path none} \rangle$ 

lemma Parallel-some-agents:
  assumes  $\bigwedge a. a \in bs \implies Ps a = \text{spec.term.none } (Ps' a)$ 
  assumes  $as \cap bs \neq \{\}$ 
  shows  $\text{spec.Parallel } as Ps = \text{spec.term.none } (\|a \in as. \text{if } a \in as \cap bs \text{ then } Ps' a \text{ else } Ps a)$ 
using assms(1)[symmetric] assms(2)
  INF-union[where A= $as - bs$  and B= $as \cap bs$ 
    and M= $\lambda a. \text{spec.toConcurrent } a (\text{if } a \in bs \text{ then } Ps' a \text{ else } Ps a)$ ])
by (simp add: spec.Parallel-def Un-Diff-Int inf-assoc image-image
  spec.term.none.map spec.term.none.invmap spec.term.none.inf-unit(2) spec.term.none.Inf-not-empty
  flip: INF-union)

```

lemma *Parallel-not-empty*:

assumes $as \neq \{\}$

shows $spec.term.none (Parallel\ as\ Ps) = Parallel\ as\ (spec.term.none \circ Ps)$

using $assms\ spec.term.none.Parallel-some-agents[where\ as=as\ and\ bs=as\ and\ Ps=spec.term.none \circ Ps\ and\ Ps'=Ps]$

by (*simp cong: spec.Parallel.cong*)

lemma *parallel*:

shows $spec.term.none (P \parallel Q) = spec.term.none P \parallel spec.term.none Q$

by (*simp add: spec.parallel-def spec.term.none.Parallel-not-empty comp-def if-distrib*)

lemma

shows *parallelL*: $spec.term.none P \parallel Q = spec.term.none (P \parallel Q)$

and *parallelR*: $P \parallel spec.term.none Q = spec.term.none (P \parallel Q)$

using

$spec.term.none.Parallel-some-agents[where$

$as=UNIV\ and\ bs=\{True\}$

and $Ps=\lambda a. if\ a\ then\ spec.term.none\ P\ else\ Q\ and\ Ps'=\lambda a. if\ a\ then\ P\ else\ Q]$

$spec.term.none.Parallel-some-agents[where$

$as=UNIV\ and\ bs=\{False\}$

and $Ps=\lambda a. if\ a\ then\ P\ else\ spec.term.none\ Q\ and\ Ps'=\lambda a. if\ a\ then\ P\ else\ Q]$

by (*simp-all add: spec.parallel-def cong: if-cong*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ all \rangle$

lemma *Parallel*:

shows $spec.term.all (spec.Parallel\ as\ Ps) = spec.Parallel\ as\ (spec.term.all \circ Ps)$

by (*simp add: spec.Parallel-def image-image*

$spec.term.all.Inf\ spec.term.all.inf\ spec.term.all.invmap\ spec.term.all.map\ spec.term.all.rel)$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ idle \rangle$

lemma *parallel-le*:

assumes $spec.idle \leq P$

assumes $spec.idle \leq Q$

shows $spec.idle \leq P \parallel Q$

by (*simp add: assms spec.parallel-def spec.idle-le*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ invmap \rangle$

lemma *parallel*: — $af = id$ in $spec.invmap$

shows $spec.invmap\ id\ sf\ vf (spec.parallel\ P\ Q)$

$= spec.parallel (spec.invmap\ id\ sf\ vf\ P) (spec.invmap\ id\ sf\ vf\ Q)$

by (*simp add: spec.parallel-def spec.Parallel.invmap comp-def if-distrib*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ parallel \rangle$

lemma *bot*:

shows *botL*: *spec.parallel* \perp $P = \perp$
and *botR*: *spec.parallel* $P \perp = \perp$
by (*simp-all add: spec.parallel-def spec.Parallel.bot*[**where** $a=False$] *spec.Parallel.bot*[**where** $a=True$])

lemma *commute*:

shows *spec.parallel* $P Q = \text{spec.parallel } Q P$
unfolding *spec.parallel-def* **by** (*subst spec.Parallel.rename*[*symmetric, OF bij-Not*]) (*simp add: comp-def*)

lemma *mono*:

assumes $P \leq P'$
assumes $Q \leq Q'$
shows *spec.parallel* $P Q \leq \text{spec.parallel } P' Q'$
by (*simp add: assms spec.parallel-def spec.Parallel.mono*)

lemma *strengthen*[*strg*]:

assumes *st-ord* $F P P'$
assumes *st-ord* $F Q Q'$
shows *st-ord* $F (\text{spec.parallel } P Q) (\text{spec.parallel } P' Q')$
using *assms* **by** (*cases F; simp add: spec.parallel.mono*)

lemma *mono2mono*[*cont-intro, partial-function-mono*]:

assumes *monotone orda* $(\leq) F$
assumes *monotone orda* $(\leq) G$
shows *monotone orda* $(\leq) (\lambda f. \text{spec.parallel } (F f) (G f))$
using *assms* **by** (*simp add: spec.parallel-def spec.Parallel.mono2mono*)

lemma *Sup*:

fixes $P_s :: (\text{sequential}, 's, \text{unit}) \text{ spec set}$
shows *SupL*: $\bigsqcup P_s \parallel Q = (\bigsqcup P \in P_s. P \parallel Q)$
and *SupR*: $Q \parallel \bigsqcup P_s = (\bigsqcup P \in P_s. Q \parallel P)$
by (*simp-all add: spec.parallel-alt-def spec.invmap.Sup spec.map.Sup heyting.inf-SUP-distrib image-image*)

lemma *sup*:

fixes $P :: (\text{sequential}, 's, \text{unit}) \text{ spec}$
shows *supL*: $(P \sqcup Q) \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel (Q \sqcup R) = (P \parallel Q) \sqcup (P \parallel R)$
using *spec.parallel.Sup*[**where** $P_s=\{P, Q\}$ **for** $P Q$, *simplified*] **by** *fast+*

We can residuate (\parallel) but not *prog.parallel* (see §13.3) as the latter is not strict. Intuitively any realistic modelling of parallel composition will be non-strict as the divergence of one process should not block the progress of others, and incorporating such interference may preclude the implementation of a specification via this residuation.

References:

- [Hayes \(2016, Law 23\)](#): residuate parallel
- [van Staden \(2015, Lemma 6\)](#) who cites [Armstrong, Gomes, and Struth \(2014\)](#)

definition *res* :: $(\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$ **where**
res $S i = \bigsqcup \{P. P \parallel i \leq S\}$

interpretation *res*: *galois.complete-lattice-class* $\lambda S. \text{spec.parallel } S i \lambda S. \text{spec.parallel.res } S i$ **for** i — [Hayes \(2016, Law 23 \(rely refinement\)\)](#)

proof

have $*$: *spec.parallel.res* $S i \parallel i \leq S$ **for** S — [Hayes \(2016, Law 22 \(rely quotient\)\)](#)
by (*simp add: spec.parallel.res-def spec.parallel.SupL*)
show $x \parallel i \leq S \iff x \leq \text{spec.parallel.res } S i$ **for** $x S$
by (*fastforce simp: spec.parallel.res-def Sup-upper spec.parallel.mono intro: order.trans*[*OF - **])

qed

lemma *mcont2mcont*[*cont-intro*]:

assumes *mcont luba orda Sup* (\leq) *P*

assumes *mcont luba orda Sup* (\leq) *Q*

shows *mcont luba orda Sup* (\leq) $(\lambda x. \text{spec.parallel } (P \ x) \ (Q \ x))$

proof(*rule cppo.mcont2mcont*'[*OF complete-lattice-ccpo - - assms(1)*])

show *mcont Sup* (\leq) *Sup* (\leq) $(\lambda y. \text{spec.parallel } y \ (Q \ x))$ **for** *x*

by (*intro mcontI contI monotoneI*) (*simp-all add: spec.parallel.mono spec.parallel.SupL*)

show *mcont luba orda Sup* (\leq) $(\lambda x. \text{spec.parallel } y \ (Q \ x))$ **for** *y*

by (*simp add: mcontI monotoneI contI mcont-monoD*[*OF assms(2)*] *spec.parallel.mono mcont-contD*[*OF assms(2)*] *spec.parallel.SupR image-image*)

qed

lemma *inf-rel*:

shows *spec.rel* $(\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap (P \parallel Q)$

$= (\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap P) \parallel (\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap Q)$

and $(P \parallel Q) \sqcap \text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r)$

$= (\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap P) \parallel (\text{spec.rel } (\{\text{env}\} \times \text{UNIV} \cup \{\text{self}\} \times r) \sqcap Q)$

by (*simp-all add: ac-simps spec.parallel-def spec.Parallel.inf-rel if-distrib*[**where** $f = \lambda x. x \sqcap y$ **for** *y*])

lemma *assoc*:

shows *spec.parallel* *P* (*spec.parallel* *Q* *R*) = *spec.parallel* (*spec.parallel* *P* *Q*) *R* (**is** *?lhs = ?rhs*)

by (*auto simp: spec.parallel-def bij-betw-def Plus-def UNIV-bool*

spec.Parallel.flatten[**where** $a = \text{False}$] *spec.Parallel.flatten*[**where** $a = \text{True}$]

intro!: *spec.Parallel.rename-cong*[**where** $\pi = \text{case-sum } (\lambda a. \text{if } a \text{ then } \text{Inr } \text{True} \text{ else } \text{undefined})$
 $(\lambda a. \text{if } a \text{ then } \text{Inr } \text{False} \text{ else } \text{Inl } \text{False})$])

lemma *bind-botR*:

shows *spec.parallel* $(P \gg \perp) Q = \text{spec.parallel } P Q \gg \perp$

and *spec.parallel* $P (Q \gg \perp) = \text{spec.parallel } P Q \gg \perp$

by (*simp-all add: spec.bind.botR spec.term.none.parallelL spec.term.none.parallelR*)

lemma *interference*:

shows *interferenceL*: *spec.rel* $(\{\text{env}\} \times \text{UNIV}) \parallel c = c$

and *interferenceR*: $c \parallel \text{spec.rel } (\{\text{env}\} \times \text{UNIV}) = c$

by (*simp-all add: spec.parallel-def spec.Parallel.singleton-agents*

flip: spec.Parallel.rename-UNIV[**where** $as = \{\text{False}\}$ **and** $f = \text{id}$ **and** $P_s = \langle c \rangle$, *simplified*]

spec.Parallel.rename-UNIV[**where** $as = \{\text{True}\}$ **and** $f = \text{id}$ **and** $P_s = \langle c \rangle$, *simplified*])

lemma *unwindL*:

assumes *spec.rel* $(\{\text{env}\} \times \text{UNIV}) \gg (\lambda :: \text{unit}. Q) \leq Q$ — All other processes begin with interference

assumes $f \gg g \leq P$ — The selected process starts with action *f*

shows $f \gg (\lambda v. g \ v \parallel Q) \leq P \parallel Q$

unfolding *spec.parallel-def*

by (*strengthen ord-to-strengthen*[*OF spec.Parallel.unwind*[**where** $a = \text{True}$]])

(*auto simp: spec.Parallel.mono spec.bind.mono intro: assms*)

lemma *unwindR*:

assumes *spec.rel* $(\{\text{env}\} \times \text{UNIV}) \gg (\lambda :: \text{unit}. P) \leq P$ — All other processes begin with interference

assumes $f \gg g \leq Q$ — The selected process starts with action *f*

shows $f \gg (\lambda v. P \parallel g \ v) \leq P \parallel Q$

by (*subst* (1 2) *spec.parallel commute*) (*rule spec.parallel.unwindL*[*OF assms*])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path interference.closed} \rangle$

lemma *toConcurrent-gen*:

```

fixes  $P :: (\text{sequential}, 's, 'v) \text{ spec}$ 
fixes  $a :: 'a$ 
assumes  $P: P \in \text{spec.interference.closed } (\{\text{env}\} \times r)$ 
shows  $\text{spec.toConcurrent } a P \in \text{spec.interference.closed } ((-\{\text{proc } a\}) \times r)$ 
proof –
have  $*$ :  $\text{map-prod } (\text{toConcurrent-fn } (\text{proc } a)) \text{ id} -' (\{\text{env}\} \times r) = (-\{\text{proc } a\}) \times r$ 
by  $(\text{force simp: toConcurrent-fn-def})$ 
show  $?thesis$ 
apply  $(\text{rule spec.interference.closed-clI})$ 
apply  $(\text{subst } (2) \text{ spec.interference.closed-conv}[OF P])$ 
apply  $(\text{force intro: spec.interference.cl.mono simp: } * \text{ spec.invmap.interference.cl})$ 
done
qed

```

```

lemma toConcurrent:
fixes  $P :: (\text{sequential}, 's, 'v) \text{ spec}$ 
fixes  $a :: 'a$ 
assumes  $P: P \in \text{spec.interference.closed } (\{\text{env}\} \times r)$ 
shows  $\text{spec.toConcurrent } a P \in \text{spec.interference.closed } (\{\text{env}\} \times r)$ 
by  $(\text{blast intro: subsetD}[OF \text{spec.interference.closed.antimono}$ 
 $\text{spec.interference.closed.toConcurrent-gen}[OF \text{assms}]])$ 

```

```

lemma toSequential:
fixes  $P :: ('a \text{ agent}, 's, 'v) \text{ spec}$ 
assumes  $P \in \text{spec.interference.closed } (\{\text{env}\} \times r)$ 
shows  $\text{spec.toSequential } P \in \text{spec.interference.closed } (\{\text{env}\} \times r)$ 
proof –
have  $*$ :  $\text{map-prod } \text{toSequential-fn id} -' (\{\text{env}\} \times r) = \{\text{env}\} \times r$ 
by  $(\text{force simp: toSequential-fn-def})$ 
show  $?thesis$ 
apply  $(\text{rule spec.interference.closed-clI})$ 
apply  $(\text{subst } (2) \text{ spec.interference.closed-conv}[OF \text{assms}])$ 
apply  $(\text{simp add: } * \text{ spec.map.interference.cl-sf-id})$ 
done
qed

```

```

lemma Parallel:
assumes  $\bigwedge a. Ps \ a \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$ 
shows  $\text{spec.Parallel } as Ps \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$ 
unfolding  $\text{spec.Parallel-def}$ 
by  $(\text{fastforce intro: spec.interference.closed.rel spec.interference.closed-Inf spec.interference.closed.toSequential}$ 
 $\text{simp: assms image-subset-iff spec.interference.closed.toConcurrent})$ 

```

```

lemma parallel:
assumes  $P \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$ 
assumes  $Q \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$ 
shows  $P \parallel Q \in \text{spec.interference.closed } (\{\text{env}\} \times UNIV)$ 
by  $(\text{simp add: assms spec.parallel-def spec.interference.closed.Parallel})$ 

```

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

9.6 Specification Inhabitation

Given that \perp satisfies any specification S , we may wish to show that a specific trace σ is allowed by S .

The strategy is to compute the allowed transitions from a given initial state and possibly a return value. We almost always discard the closures we've added for various kinds of compositionality.

References:

- Similar to how [van Staden \(2014, §3.3\)](#) models a small-step operational semantics.
 - i.e., we can (semantically) define something like an LTS, which is compositional wrt parallel
 - a bit like a resumption or a residual
- Similar to [Hoare, He, and Sampaio \(2000\)](#)

TODO:

- often want transitive variants of these rules
- automate: only stop when there's a scheduling decision to be made

definition *inhabits* :: ('a, 's, 'w) spec ⇒ 's ⇒ ('a × 's) list ⇒ ('a, 's, 'w) spec ⇒ bool (⟨-/ --, -->/ -> [50, 0, 0, 50] 50) **where**

$S -s, xs \rightarrow T \longleftrightarrow \langle s, xs, \text{Some } () \rangle \gg T \leq S$

setup ⟨*Sign.mandatory-path* *inhabits*⟩

lemma *incomplete*:

assumes $S -s, xs \rightarrow S'$

shows $\langle s, xs, \text{None} \rangle \leq S$

by (*strengthen ord-to-strengthen*[*OF assms*[*unfolded inhabits-def*]])
(*simp add: spec.bind.incompleteI*)

lemma *complete*:

assumes $S -s, xs \rightarrow \text{spec.return } v$

shows $\langle s, xs, \text{Some } v \rangle \leq S$

by (*strengthen ord-to-strengthen*[*OF assms*[*unfolded inhabits-def*]])
(*simp add: spec.bind.continueI*[**where** *ys*=[], *simplified*] *spec.singleton.le-conv*)

lemmas $I = \text{inhabits.complete inhabits.incomplete}$

lemma *mono*:

assumes $S \leq S'$

assumes $T' \leq T$

assumes $S -s, xs \rightarrow T$

shows $S' -s, xs \rightarrow T'$

unfolding *inhabits-def*

apply (*strengthen ord-to-strengthen*[*OF assms*(1)])

apply (*strengthen ord-to-strengthen*[*OF assms*(2)])

apply (*rule assms*(3)[*unfolded inhabits-def*])

done

lemma *strengthen*[*strg*]:

assumes *st-ord* $F S S'$

assumes *st-ord* $(\neg F) T T'$

shows *st* $F (\longrightarrow) (S -s, xs \rightarrow T) (S' -s, xs \rightarrow T')$

using *assms* **by** (*cases* *F*; *simp add: inhabits.mono*)

lemma *pre*:

assumes $S -s, xs' \rightarrow T$

assumes $T' \leq T$

assumes $xs = xs'$

shows $S -s, xs \rightarrow T'$

using *assms* **by** (*blast intro: inhabits.mono*[*OF order.refl assms*(2)])

lemma tau:

assumes $spec.idle \leq S$

shows $S -s, [] \rightarrow S$

unfolding *inhabits-def*

by (*strengthen ord-to-strengthen*[*OF spec.action.stutterI*[**where** $F=\{()\} \times UNIV \times Id$]])

(*simp-all add: assms spec.bind.returnL flip: spec.return-def*)

lemma trans:

assumes $R -s, xs \rightarrow S$

assumes $S -trace.final' s xs, ys \rightarrow T$

shows $R -s, xs @ ys \rightarrow T$

unfolding *inhabits-def*

apply (*strengthen ord-to-strengthen*(2)[*OF assms*(1)[*unfolded inhabits-def*]])

apply (*strengthen ord-to-strengthen*(2)[*OF assms*(2)[*unfolded inhabits-def*]])

apply (*simp add: spec.bind.continueI spec.bind.mono flip: spec.bind.bind*)

done

lemma Sup:

assumes $P -s, xs \rightarrow P'$

assumes $P \in X$

shows $\bigsqcup X -s, xs \rightarrow P'$

using *assms* **by** (*simp add: inhabits-def Sup-upper2*)

lemma supL:

assumes $P -s, xs \rightarrow P'$

shows $P \sqcup Q -s, xs \rightarrow P'$

using *assms* **by** (*simp add: inhabits-def le-supI1*)

lemma supR:

assumes $Q -s, xs \rightarrow Q'$

shows $P \sqcup Q -s, xs \rightarrow Q'$

using *assms* **by** (*simp add: inhabits-def le-supI2*)

lemma inf:

assumes $P -s, xs \rightarrow P'$

assumes $Q -s, xs \rightarrow Q'$

shows $P \sqcap Q -s, xs \rightarrow P' \sqcap Q'$

using *assms* **by** (*meson inf.cobounded1 inf.cobounded2 le-inf-iff inhabits.pre inhabits-def*)

lemma infL:

assumes $P -s, xs \rightarrow R$

assumes $Q -s, xs \rightarrow R$

shows $P \sqcap Q -s, xs \rightarrow R$

using *assms* **by** (*meson le-inf-iff inhabits-def*)

setup $\langle Sign.mandatory-path spec \rangle$

lemma bind:

assumes $f -s, xs \rightarrow f'$

shows $f \ggg g -s, xs \rightarrow f' \ggg g$

using *assms* **by** (*simp add: inhabits-def spec.bind.mono flip: spec.bind.bind*)

lemmas $bind' = inhabits.trans$ [*OF inhabits.spec.bind*]

lemma parallelL:

assumes $P -s, xs \rightarrow P'$

assumes $spec.rel (\{env\} \times UNIV) \ggg (\lambda::unit. Q) \leq Q$

shows $P \parallel Q -s, xs \rightarrow P' \parallel Q$
by (*rule inhabits.mono*[*OF spec.parallel.unwindL*[*OF assms*(2) *assms*(1)[*unfolded inhabits-def*]]
order.refl])
(*simp add: inhabits-def*)

lemma parallelR:

assumes $Q -s, xs \rightarrow Q'$
assumes $spec.rel (\{env\} \times UNIV) \gg (\lambda::unit. P) \leq P$
shows $P \parallel Q -s, xs \rightarrow P \parallel Q'$
by (*subst* (1 2) *spec.parallel commute*) (*rule inhabits.spec.parallelL assms*)+

lemmas parallelL' = inhabits.trans[*OF inhabits.spec.parallelL*]

lemmas parallelR' = inhabits.trans[*OF inhabits.spec.parallelR*]

setup $\langle Sign.mandatory-path \text{ action} \rangle$

lemma step:

assumes $(v, a, s, s') \in F$
shows $spec.action F -s, [(a, s')] \rightarrow spec.return v$
by (*clarsimp simp: inhabits-def trace.split-all spec.bind.singletonL spec.term.none.singleton*
spec.singleton.le-conv spec.action.stepI[*OF assms*]
intro!: *ord-eq-le-trans*[*OF spec.singleton.Cons spec.action.stepI*[*OF assms*]])

lemma stutter:

assumes $(v, a, s, s) \in F$
shows $spec.action F -s, [] \rightarrow spec.return v$
using *inhabits.spec.action.step*[*OF assms*] **by** (*simp add: inhabits-def*)

setup $\langle Sign.parent-path \rangle$

lemma map:

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P -s, xs \rightarrow spec.return v$
shows $spec.map af sf vf P -sf s, map (map-prod af sf) xs \rightarrow spec.return (vf v)$

proof –

have $\langle sf s, map (map-prod af sf) xs, Some () \rangle \gg spec.return (vf v)$
 $\leq spec.map af sf vf (\langle s, xs, Some () \rangle \gg spec.return v)$
by (*subst* (1) *spec.bind.singletonL*)
(*fastforce intro: spec.bind.incompleteI*
spec.bind.continueI[**where** $ys=[]$ **and** $w=Some v$, *simplified*]
simp: spec.singleton.le-conv spec.term.none.singleton
split: option.split-asm)
then show *?thesis*
using *assms* **by** (*auto simp: inhabits-def dest: spec.map.mono*[**where** $af=af$ **and** $sf=sf$ **and** $vf=vf$])
qed

lemma invmap:

fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P -sf s, map (map-prod af sf) xs \rightarrow P'$
shows $spec.invmap af sf vf P -s, xs \rightarrow spec.invmap af sf vf P'$
by (*strengthen ord-to-strengthen*(2)[*OF assms*(1)[*unfolded inhabits-def*]])
(*simp add: inhabits-def spec.invmap.bind spec.map.singleton spec.bind.mono*
flip: spec.map-invmap.galois)

setup $\langle \text{Sign.mandatory-path term.none} \rangle$

lemma step:

assumes $P -s, xs \rightarrow P'$

shows $\text{spec.term.none } P -s, xs \rightarrow \text{spec.term.none } P'$

by (*simp add: inhabits.spec.bind[OF assms] flip: spec.bind.botR*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term.all} \rangle$

lemma step:

assumes $P -s, xs \rightarrow P'$

shows $\text{spec.term.all } P -s, xs \rightarrow \text{spec.term.all } P'$

by (*strengthen ord-to-strengthen(2)[OF assms(1)[unfolded inhabits-def]]*)
(*simp add: inhabits-def spec.term.all.bind*)

lemma term:

assumes $\text{spec.idle} \leq P$

shows $\text{spec.term.all } P -s, [] \rightarrow \text{spec.return } v$

by (*strengthen ord-to-strengthen(2)[OF assms(1)]*)
(*auto simp: spec.term.all.idle intro: spec.idle-le inhabits.tau inhabits.Sup*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path kleene.star} \rangle$

lemma step:

assumes $P -s, xs \rightarrow P'$

shows $\text{spec.kleene.star } P -s, xs \rightarrow P' \gg \text{spec.kleene.star } P$

by (*subst spec.kleene.star.simps*) (*simp add: assms inhabits.supL inhabits.spec.bind*)

lemma term:

shows $\text{spec.kleene.star } P -s, [] \rightarrow \text{spec.return } ()$

by (*metis inhabits.tau inhabits.supR spec.kleene.star.simps spec.idle.return-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path rel} \rangle$

lemma rel:

assumes $\text{trace.steps}' s xs \subseteq r$

shows $\text{spec.rel } r -s, xs \rightarrow \text{spec.rel } r$

proof –

from *assms*

have $\langle s, xs, \text{Some } () \rangle \gg \text{spec.rel } r \leq \text{spec.rel } r \gg (\lambda::\text{unit. spec.rel } r)$

by (*simp add: spec.bind.mono spec.singleton.le-conv*)

then show *?thesis*

by (*simp add: inhabits-def spec.rel.wind-bind*)

qed

lemma rel-term:

assumes $\text{trace.steps}' s xs \subseteq r$

shows $\text{spec.rel } r -s, xs \rightarrow \text{spec.return } v$

by (*rule inhabits.pre[OF inhabits.spec.rel.rel[OF assms] spec.return.rel-le refl]*)

lemma step:

assumes $(a, s, s') \in r$

shows $spec.rel\ r -s, [(a, s')] \rightarrow spec.rel\ r$
by (*rule inhabits.pre*)
 (*auto intro: assms inhabits.spec.action.step[where s'=s'] inhabits.spec.kleene.star.step*
inhabits.spec.term.all.step
simp: spec.rel-def spec.rel.act-def spec.bind.returnL spec.idle-le)

lemma *term*:

shows $spec.rel\ r -s, [] \rightarrow spec.return\ v$
by (*simp add: inhabits.pre[OF inhabits.tau[OF spec.idle.rel-le] spec.return.rel-le]*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

10 “Next step” implication ala Abadi and Merz (and Lamport)

As was apparently well-known in the mid-1990s (see, e.g., [Xu, Cau, and Collette \(1994, §4\)](#) and the references therein), Heyting implication is inadequate for a general refinement story. (We show it is strong enough for a relational assume/guarantee program logic; see §9.2, §12.2 and §13.5.2. In our setting it fails to generalize (at least) because the composition theorem for Heyting implication (§9.2) is not termination sensitive.)

We therefore follow [Abadi and Lamport \(1995\)](#) by developing a stronger implication $P \longrightarrow_+ Q$ with the intuitive semantics that the consequent Q holds for at least one step beyond the antecedent P . This is some kind of step indexing.

Here we sketch the relevant parts of [Abadi and Merz \(1995, 1996\)](#), the latter of which has a fuller story, including a formal account of the logical core of TLA and the (implicit) observation that infinitary parallel composition poses no problem for safety properties (see the comments under Theorem 5.2 and §5.5). [Abadi and Lamport \(1995\)](#); [Cau and Collette \(1996\)](#); [Xu et al. \(1994\)](#) provide further background; [Jonsson and Tsay \(1996, Appendix B\)](#) provide a more syntactic account.

Observations:

- The hypothesis P is always a safety property here
- TLA does not label transitions or have termination markers
- Abadi/Cau/Collette/Lamport/Merz/Xu/... avoid naming this operator

Further references:

- [Maier \(2001\)](#)

definition *next-imp* :: $'a::preorder\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where** — [Abadi and Merz \(1995, §2\)](#)
 $next-imp\ P\ Q = \{\sigma. \forall \sigma' \leq \sigma. (\forall \sigma'' < \sigma'. \sigma'' \in P) \longrightarrow \sigma' \in Q\}$

setup $\langle Sign.mandatory-path\ next-imp \rangle$

lemma *downwards-closed*:

assumes $P \in downwards.closed$
shows $next-imp\ P\ Q \in downwards.closed$
unfolding *next-imp-def* **by** (*blast elim: downwards.clE intro: order-trans*)

lemma *mono*:

assumes $x' \leq x$
assumes $y \leq y'$
shows $next-imp\ x\ y \leq next-imp\ x'\ y'$
unfolding *next-imp-def* **using** *assms* **by** *fast*

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F) X X'$
assumes *st-ord* $F Y Y'$
shows *st-ord* $F (next\text{-}imp X Y) (next\text{-}imp X' Y')$
using *assms* **by** (*cases* F) (*auto simp: next-imp.mono*)

lemma *minimal*:
assumes *trace*. $T s xs v \in next\text{-}imp P Q$
shows *trace*. $T s [] None \in Q$
using *assms* **by** (*simp add: next-imp-def trace.less trace.less-eq-None*)

lemma *alt-def*: — This definition coincides with [Cau and Collette \(1996\)](#), [Abadi and Lamport \(1995, §3.5.3\)](#)

assumes $P \in downwards.closed$
shows *next-imp* $P Q$
 $= \{\sigma. trace.T (trace.init \sigma) [] None \in Q$
 $\wedge (\forall i. trace.take i \sigma \in P \longrightarrow trace.take (Suc i) \sigma \in Q)\}$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

have *trace.take* $(Suc i) (trace.T s xs v) \in Q$
if *trace.T s xs v* $\in ?lhs$ **and** *trace.take i* $(trace.T s xs v) \in P$
for $s xs v i$
using *that* $\langle P \in downwards.closed \rangle$
by (*force simp: next-imp-def trace.less-take-less-eq*
dest: spec[where x=trace.take (Suc i) (trace.T s xs v)]
elim: downwards.closed-in)
then show *?lhs* $\subseteq ?rhs$
by (*clarsimp simp: trace.split-all next-imp.minimal*)

next

have *trace.T s xs v* $\in ?lhs$
if *minimal: trace.T s [] None* $\in Q$
and *imp: $\forall i. trace.take i (trace.T s xs v) \in P \longrightarrow trace.take (Suc i) (trace.T s xs v) \in Q$*
for $s xs v$

proof —

have *trace.take i* $(trace.T s xs v) \in Q$
if $\forall \sigma'' < trace.take i (trace.T s xs v). \sigma'' \in P$
for i
using *that*
proof(*induct i*)
case $(Suc i)$ **with** *imp* **show** *?case*
by (*metis le-add2 order-le-less plus-1-eq-Suc trace.take.mono*)
qed (*simp add: minimal*)
then show *trace.T s xs v* $\in ?lhs$
by (*clarsimp simp: next-imp-def trace.less-eq-take-def*)
qed
then show *?rhs* $\subseteq ?lhs$
by (*clarsimp simp: trace.split-all next-imp.minimal*)

qed

[Abadi and Lamport \(1995, §3.5.3\)](#) assert but do not prove the following connection with Heyting implication. [Abadi and Merz \(1995\)](#) do. See also [Abadi and Merz \(1996, §5.3 and §5.5\)](#).

lemma *Abadi-Merz-Prop-1-subseteq*: — First half of [Abadi and Merz \(1995, Proposition 1\)](#)

fixes $P :: 'a::preorder\ set$
assumes $P \in downwards.closed$
assumes *wf*: $wfP ((<) :: 'a\ relp)$
shows *next-imp* $P Q \subseteq downwards.imp (downwards.imp Q P) Q$ (**is** *?lhs* \subseteq *?rhs*)

proof(*rule subsetI*)

fix σ **assume** $\sigma \in ?lhs$ **with** *wf* **show** $\sigma \in ?rhs$
proof(*induct rule: wfp-induct-rule*)

```

case (less  $\sigma$ )
have  $\tau \in Q$  if  $\tau \leq \sigma$  and  $YYY: \forall \sigma' \leq \tau. \sigma' \in Q \longrightarrow \sigma' \in P$  for  $\tau$ 
proof –
  have  $\varrho \in P$  if  $\varrho < \tau$  for  $\varrho$ 
  proof –
    from  $\langle \varrho < \tau \rangle \langle \tau \leq \sigma \rangle \langle P \in \text{downwards.closed} \rangle$  have  $\varrho \in \text{next-imp } P \ Q$ 
    by (meson downwards.closed-in next-imp.downwards.closed less.premis less-imp-le)
    with  $\langle \varrho < \tau \rangle \langle \tau \leq \sigma \rangle$  have  $\varrho \in \text{downwards.imp } (\text{downwards.imp } Q \ P) \ Q$ 
    using less.hyps less-le-trans by blast
    moreover from  $\langle \varrho < \tau \rangle \ YYY$  have  $\varrho \in \text{downwards.imp } Q \ P$ 
    by (simp add: downwards.imp-def) (meson order.trans order-less-imp-le)
    ultimately show ?thesis by (meson downwards.imp-mp')
  qed
  with that less.premis show ?thesis
  unfolding next-imp-def by blast
qed
then show ?case
  unfolding downwards.imp-def by blast
qed
qed

```

The converse holds if either Q is a safety property or the order is partial.

lemma *Abadi-Merz-Prop1*: — [Abadi and Merz \(1995, Proposition 1\)](#) and [Abadi and Merz \(1996, Proposition 5.2\)](#)

```

fixes  $P :: 'a::preorder \text{ set}$ 
assumes  $P \in \text{downwards.closed}$ 
assumes  $Q \in \text{downwards.closed}$ 
assumes  $wf: wfP ((<) :: 'a \text{ relp})$ 
shows  $\text{next-imp } P \ Q = \text{downwards.imp } (\text{downwards.imp } Q \ P) \ Q$  (is ?lhs = ?rhs)
proof(rule antisym[OF next-imp.Abadi-Merz-Prop-1-subseteq[OF assms(1,3)]])
  from  $\langle Q \in \text{downwards.closed} \rangle$  show  $?rhs \subseteq ?lhs$ 
  by (auto simp: next-imp-def downwards.imp-def order.strict-iff-not dest: downwards.closed-in)
qed

```

lemma *Abadi-Lamport-Lemma6*: — [Abadi and Lamport \(1995, Lemma 6\)](#) (no proof given there)

```

fixes  $P :: 'a::order \text{ set}$ 
assumes  $P \in \text{downwards.closed}$ 
assumes  $wf: wfP ((<) :: 'a \text{ relp})$ 
shows  $\text{next-imp } P \ Q = \text{downwards.imp } (\text{downwards.imp } Q \ P) \ Q$  (is ?lhs = ?rhs)
proof(rule Set.equalityI[OF next-imp.Abadi-Merz-Prop-1-subseteq[OF assms]])
  show  $?rhs \subseteq ?lhs$ 
  unfolding next-imp-def downwards.imp-def by (fastforce simp: le-less elim: downwards.closed-in)
qed

```

lemmas $\text{downwards.imp} = \text{next-imp.Abadi-Lamport-Lemma6}[OF - \text{trace.wfP-less}]$

lemma *boolean-implication-le*:

```

assumes  $P \in \text{downwards.closed}$ 
shows  $\text{next-imp } P \ Q \subseteq P \longrightarrow_B Q$ 
using downwards.closed-conv[OF assms]
by (fastforce simp: next-imp-def boolean-implication.member
  intro: order-less-imp-le)

```

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

lift-definition $\text{next-imp} :: ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ (**infixr** $\langle \longrightarrow_+ \rangle$ 61) **is**
 Next-Imp.next-imp

by (simp add: next-imp.downwards-imp raw.spec.closed.downwards-closed raw.spec.closed.downwards-imp)

setup <Sign.mandatory-path next-imp>

lemma heyting: — fundamental

shows $P \longrightarrow_+ Q = (Q \longrightarrow_H P) \longrightarrow_H Q$

by transfer (simp add: next-imp.downwards-imp raw.spec.closed.downwards-closed)

setup <Sign.parent-path>

setup <Sign.mandatory-path singleton>

lemma next-imp-le-conv:

fixes $P :: ('a, 's, 'v) \text{ spec}$

shows $\langle \sigma \rangle \leq P \longrightarrow_+ Q \iff (\forall \sigma' \leq \sigma. (\forall \sigma'' < \sigma'. \langle \sigma'' \rangle \leq P) \longrightarrow \langle \sigma' \rangle \leq Q)$ (is ?lhs \iff ?rhs)

proof(rule iffI)

show ?lhs \implies ?rhs

by (force simp: spec.next-imp.heyting spec.singleton.heyting-le-conv)

note spec.singleton.transfer[transfer-rule]

show ?rhs \implies ?lhs

proof(transfer, unfold raw.singleton-def, rule raw.spec.least)

show $\{\sigma\} \subseteq \text{next-imp } P \ Q$

if $P \in \text{raw.spec.closed}$

and $\forall \sigma' \leq \sigma. (\forall \sigma'' < \sigma'. \text{raw.spec.cl } \{\sigma''\} \subseteq P) \longrightarrow \text{raw.spec.cl } \{\sigma'\} \subseteq Q$

for $P \ Q :: ('a, 's, 'v) \text{ trace.t set}$ and σ

using that by (auto simp: next-imp-def raw.spec.least-conv
dest: order.trans[OF raw.spec.expansive])

show next-imp $P \ Q \in \text{raw.spec.closed}$

if $P \in \text{raw.spec.closed}$

and $Q \in \text{raw.spec.closed}$

for $P \ Q :: ('a, 's, 'v) \text{ trace.t set}$

using that

by (simp add: next-imp.downwards-imp raw.spec.closed.downwards-closed raw.spec.closed.downwards-imp)

qed

qed

setup <Sign.parent-path>

setup <Sign.mandatory-path next-imp>

lemma mono:

assumes $x' \leq x$

assumes $y \leq y'$

shows $x \longrightarrow_+ y \leq x' \longrightarrow_+ y'$

by (simp add: assms heyting.mono spec.next-imp.heyting)

lemma strengthen[strg]:

assumes $\text{st-ord } (\neg F) \ X \ X'$

assumes $\text{st-ord } F \ Y \ Y'$

shows $\text{st-ord } F \ (X \longrightarrow_+ Y) \ (X' \longrightarrow_+ Y')$

using assms by (cases F) (auto simp: spec.next-imp.mono)

lemma idempotentR:

shows $P \longrightarrow_+ (P \longrightarrow_+ Q) = P \longrightarrow_+ Q$

by (simp add: spec.next-imp.heyting heyting.heyting.detachement(1) heyting.discharge(2) inf.absorb2
flip: heyting.curry-conv)

lemma contains:

assumes $X \leq Q$
shows $X \leq P \longrightarrow_+ Q$
by (*simp add: assms spec.next-imp.heyting heyting.curry le-infI1*)

interpretation *closure: closure-complete-lattice-class* (\longrightarrow_+) ***P*** for ***P***
by *standard*

*(metis (no-types, lifting) order.refl order.trans
spec.next-imp.idempotentR spec.next-imp.contains spec.next-imp.mono)*

lemma *InfR:*

shows $x \longrightarrow_+ \bigsqcap X = \bigsqcap ((\longrightarrow_+) x \text{ ' } X)$
by *transfer (auto simp: next-imp-def)*

lemma *SupR-not-empty:*

fixes $P :: (-, -, -)$ *spec*
assumes $X \neq \{\}$
shows $P \longrightarrow_+ (\bigsqcup x \in X. Q x) = (\bigsqcup x \in X. P \longrightarrow_+ Q x)$ (**is** *?lhs = ?rhs*)

proof(*rule antisym[OF spec.singleton-le-extI
spec.next-imp.closure.Sup-cl-le[where X=Q ' X, simplified image-image]]*)

show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ

proof(*cases* $\langle \sigma \rangle \leq P$)

case *True* **with** $\langle \sigma \rangle \leq ?lhs$ **show** *?thesis*

by (*fastforce simp: spec.singleton.next-imp-le-conv
intro: order.trans[OF spec.singleton.mono]
elim!: order-less-imp-le
dest: spec[where x= σ]*)

next

case *False* **show** *?thesis*

proof(*cases* $\langle \text{trace.init } \sigma, [], \text{None} \rangle \leq P$)

case *True* **with** $\neg \langle \sigma \rangle \leq P$

obtain j **where** $*$: $\forall \sigma'' < \text{trace.take } (Suc j) \sigma. \langle \sigma'' \rangle \leq P$

and $**$: $\neg \langle \text{trace.take } (Suc j) \sigma \rangle \leq P$

using *ex-least-nat-less[where P= $\lambda i. \neg \langle \text{trace.take } i \sigma \rangle \leq P$ and $n = Suc (\text{length } (\text{trace.rest } \sigma))$]*

by (*force dest: trace.less-take-less-eq*)

simp: less-Suc-eq-le order.trans[OF spec.singleton.mono]

simp flip: trace.take.Ex-all)

from $\langle \sigma \rangle \leq ?lhs$ ****** **show** *?thesis*

by (*clarsimp simp: spec.singleton.next-imp-le-conv*)

dest!: spec[where x= $\text{trace.take } (Suc j) \sigma$] rev-mp[OF $$]*

elim!: rev-bexI)

(*meson order.trans less-le-not-le spec.singleton.mono trace.less-eq-same-cases trace.less-eq-take*)

next

case *False* **with** $\langle X \neq \{\} \rangle \langle \sigma \rangle \leq ?lhs$ **show** *?thesis*

by (*clarsimp simp: spec.singleton.next-imp-le-conv simp flip: ex-in-conv*)

(*metis trace.take.0 trace.less-eq-take-def trace.less-t-def trace.take.sel(1)*)

qed

qed

qed

lemma *cont:*

shows *cont* $Sup (\leq) Sup (\leq) ((\longrightarrow_+) P)$

by (*rule contI*) (*simp add: spec.next-imp.SupR-not-empty[where Q=id, simplified]*)

lemma *mcont:*

shows *mcont* $Sup (\leq) Sup (\leq) ((\longrightarrow_+) P)$

by (*simp add: monotoneI mcontI[OF - spec.next-imp.cont]*)

lemmas *mcont2mcont[cont-intro] = mcont2mcont[OF spec.next-imp.mcont, of luba orda Q P] for luba orda Q P*

lemma *botL*:

assumes *spec.idle* $\leq P$

shows $\perp \longrightarrow_+ P = \top$

by (*simp add: assms spec.next-imp.heyting spec.eq-iff Heyting.heyting spec.heyting.non-triv*)

lemma *topL[simp]*:

shows $\top \longrightarrow_+ P = P$

by (*simp add: spec.next-imp.heyting*)

lemmas *topR[simp]* = *spec.next-imp.closure.cl-top*

lemma *refl*:

shows $P \longrightarrow_+ P \leq P$

by (*simp add: spec.next-imp.heyting*)

lemma *heyting-le*:

shows $P \longrightarrow_+ Q \leq P \longrightarrow_H Q$

by (*simp add: spec.next-imp.heyting heyting.ignore heyting.mono*)

lemma *discharge*:

shows $P \sqcap (P \sqcap Q \longrightarrow_+ R) = P \sqcap (Q \longrightarrow_+ R)$ (**is** *?thesis1 P Q*)

and $(P \sqcap Q \longrightarrow_+ R) \sqcap P = P \sqcap (Q \longrightarrow_+ R)$ (**is** *?thesis2*)

and $Q \sqcap (P \sqcap Q \longrightarrow_+ R) = Q \sqcap (P \longrightarrow_+ R)$ (**is** *?thesis3*)

and $(P \sqcap Q \longrightarrow_+ R) \sqcap Q = Q \sqcap (P \longrightarrow_+ R)$ (**is** *?thesis4*)

proof –

show *?thesis1 P Q* **for** $P Q$

by (*simp add: spec.next-imp.heyting heyting.infR heyting.curry-conv heyting.ignore heyting.discharge*)

then show *?thesis2* **by** (*rule inf-commute-conv*)

from $\langle ?thesis1 Q P \rangle$ **show** *?thesis3* **by** (*simp add: ac-simps*)

then show *?thesis4* **by** (*rule inf-commute-conv*)

qed

lemma *detachment*:

shows $x \sqcap (x \longrightarrow_+ y) \leq y$

and $(x \longrightarrow_+ y) \sqcap x \leq y$

by (*simp-all add: spec.next-imp.heyting heyting.ignore heyting.discharge*)

lemma *infR*:

shows $P \longrightarrow_+ Q \sqcap R = (P \longrightarrow_+ Q) \sqcap (P \longrightarrow_+ R)$

by (*rule antisym[OF spec.next-imp.closure.cl-inf-le]*)

(*rule spec.singleton-le-extI; clarsimp simp: spec.singleton.next-imp-le-conv*)

lemma *supL-le*:

shows $x \sqcup y \longrightarrow_+ z \leq (x \longrightarrow_+ z) \sqcup (y \longrightarrow_+ z)$

by (*simp add: le-supI1 spec.next-imp.mono*)

lemma *heytingL*:

shows $(P \longrightarrow_H Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$

by (*simp add: spec.next-imp.heyting heyting ac-simps*)

(*simp add: heyting.rev-trans heyting.ignore flip: inf.assoc*)

lemma *heytingR*:

shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_H R) \leq P \longrightarrow_+ R$

by (*simp add: spec.next-imp.heyting heyting ac-simps*)

(*simp add: heyting.ignore heyting.trans heyting.uncurry flip: inf.assoc*)

lemma *heytingL-distrib*:

shows $P \longrightarrow_H (Q \longrightarrow_+ R) = (P \sqcap Q) \longrightarrow_+ (P \longrightarrow_H R)$
by (*metis* (*no-types*, *opaque-lifting*) *heyting.curry-conv* *heyting.detachment*(2) *heyting.infR*
heyting.refl *heyting.swap* *inf-top-left* *spec.next-imp.heyting*)

lemma *trans*:

shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
by (*meson* *order.trans* *Heyting.heyting* *spec.next-imp.heytingL* *spec.next-imp.heyting-le*)

lemma *rev-trans*:

shows $(Q \longrightarrow_+ R) \sqcap (P \longrightarrow_+ Q) \leq P \longrightarrow_+ R$
by (*simp* *add: inf.commute* *spec.next-imp.trans*)

lemma

assumes $x' \leq x$
shows *discharge-leL*: $x' \sqcap (x \longrightarrow_+ y) = x' \sqcap y$ (**is** *?thesis1*)
and *discharge-leR*: $(x \longrightarrow_+ y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)

proof –

from *assms* **show** *?thesis1*
by (*metis* *inf.absorb-iff2* *inf-top.right-neutral* *spec.next-imp.discharge*(4) *spec.next-imp.topL*)
then show *?thesis2* **by** (*simp* *add: ac-simps*)

qed

lemma *invmap*:

shows *spec.invmap* *af sf vf* $(P \longrightarrow_+ Q) = \text{spec.invmap } af \text{ sf vf } P \longrightarrow_+ \text{spec.invmap } af \text{ sf vf } Q$
by (*simp* *add: spec.next-imp.heyting* *spec.invmap.heyting*)

lemma *Abadi-Lamport-Lemma7*:

assumes $Q \sqcap R \leq P$
shows $P \longrightarrow_+ Q \leq R \longrightarrow_+ Q$
by (*simp* *add: assms* *spec.next-imp.heyting* *Heyting.heyting* *heyting.detachment*(2) *heyting.discharge*(2))

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

lemma *next-imp*:

shows *spec.term.none* $(P \longrightarrow_+ Q) \leq \text{spec.term.all } P \longrightarrow_+ \text{spec.term.none } Q$
by (*simp* *add: spec.next-imp.heyting* *order.trans*[*OF spec.term.none.heyting-le*] *spec.term.all.heyting*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *next-imp*:

shows *spec.term.all* $(P \longrightarrow_+ Q) = \text{spec.term.all } P \longrightarrow_+ \text{spec.term.all } Q$
by (*simp* *add: spec.next-imp.heyting*)
(*metis* *spec.term.all.heyting* *spec.term.all-none* *spec.term.heyting-noneL-allR-mono* *spec.term.none-all*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *next-imp*:

assumes $Q \in \text{spec.term.closed}$ -
shows $P \longrightarrow_+ Q \in \text{spec.term.closed}$ -
using *assms*

by (simp add: spec.next-imp.heyting spec.term.closed.heyting)

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.mandatory-path pre>

lemma next-imp-eq-heyting:

assumes spec.idle $\leq R$

shows $Q \sqcap \text{spec.pre } P \longrightarrow_+ R = \text{spec.pre } P \longrightarrow_H (Q \longrightarrow_+ R)$ (is ?lhs = ?rhs)

and $\text{spec.pre } P \sqcap Q \longrightarrow_+ R = \text{spec.pre } P \longrightarrow_H (Q \longrightarrow_+ R)$ (is ?thesis1)

proof –

show ?lhs = ?rhs

proof(rule antisym[OF - spec.singleton-le-extI])

show ?lhs \leq ?rhs

by (simp add: heyting spec.next-imp.discharge)

show $\langle \sigma \rangle \leq ?lhs$ if $\langle \sigma \rangle \leq ?rhs$ for σ

using assms that

by (clarsimp simp: spec.singleton.next-imp-le-conv spec.singleton.heyting-le-conv
spec.singleton.le-conv)

(metis order.refl append-self-conv2 spec.singleton.idle-le-conv spec.singleton-le-ext-conv
trace.less(1) trace.less-eqE trace.steps'.simps(1) trace.t.sel(1))

qed

then show ?thesis1

by (simp add: ac-simps)

qed

setup <Sign.parent-path>

setup <Sign.parent-path>

10.1 Compositionality ala Abadi and Merz (and Lamport)

The main theorem for this implication (Abadi and Merz (1995, Theorem 4) and Abadi and Merz (1996, Corollary 5.1)) shows how to do assumption/commitment proofs for TLA considered as an algebraic logic. See also Cau and Collette (1996).

setup <Sign.mandatory-path spec>

lemma Abadi-Lamport-Lemma5:

shows $(\prod_{i \in I}. P \ i \longrightarrow_+ Q \ i) \leq (\prod_{i \in I}. P \ i) \longrightarrow_+ (\prod_{i \in I}. Q \ i)$

by (simp add: spec.next-imp.InfR INF-lower INF-superset-mono image-image spec.next-imp.mono)

lemma Abadi-Merz-Prop2-1:

shows $(P \longrightarrow_+ Q) \sqcap (P \longrightarrow_+ (Q \longrightarrow_H R)) \leq P \longrightarrow_+ R$

by (metis heyting.detachment(1) inf-sup-ord(2) spec.next-imp.infR)

lemma Abadi-Merz-Theorem3-5:

shows $P \longrightarrow_H (Q \longrightarrow_H R) \leq (R \longrightarrow_+ Q) \longrightarrow_H (P \longrightarrow_+ Q)$

by (simp add: heyting.order.trans[OF spec.next-imp.heytingL] spec.next-imp.Abadi-Lamport-Lemma7
flip: heyting.curry-conv)

theorem Abadi-Merz-Theorem4:

shows $(A \sqcap (\prod_{i \in I}. C \ i) \longrightarrow_H (\prod_{i \in I}. A \ i))$

$\sqcap (A \longrightarrow_+ ((\prod_{i \in I}. C \ i) \longrightarrow_H C))$

$\sqcap (\prod_{i \in I}. A \ i \longrightarrow_+ C \ i)$

$\leq A \longrightarrow_+ C$ (is ?lhs \leq ?rhs)

proof –

```

have ?lhs ≤ A →H (∏i∈I. Cs i) →H (∏i∈I. As i)
  by (simp add: heyting.curry-conv inf.coboundedI1)
then have 2: ?lhs ≤ ((∏i∈I. As i) →+ (∏i∈I. Cs i)) →H (A →+ (∏i∈I. Cs i))
  by (simp add: heyting.curry-conv inf.coboundedI1 spec.Abadi-Merz-Theorem3-5)
have 3: ?lhs ≤ (∏i∈I. As i) →+ (∏i∈I. Cs i)
  using spec.Abadi-Lamport-Lemma5 le-infI2 by blast
from 2 3 have ?lhs ≤ A →+ (∏i∈I. Cs i)
  using heyting.mp by blast
then show ?thesis
  by - (rule order.trans[OF - spec.Abadi-Merz-Prop2-1[where Q=∏ (Cs ‘ I)]]; simp add: inf.coboundedI1)
qed

```

```

setup ‹Sign.parent-path›

```

11 Stability

The essence of rely/guarantee reasoning is that sequential assertions must be *stable* with respect to interfering transitions as expressed in a *rely* relation. Formally an assertion P is stable if it becomes no less true for each transition in the rely r . This is essentially monotonicity, or that the extension of P is r -closed.

References:

- Vafeiadis (2008, §3.1.3) has a def for stability in terms of separation logic

```

definition stable :: 'a rel ⇒ 'a pred ⇒ bool where
  stable r P = monotone (λx y. (x, y) ∈ r) (≤) P

```

```

setup ‹Sign.mandatory-path stable›

```

```

named-theorems intro stability intro rules

```

```

lemma singleton-conv:

```

```

  shows stable {(s, s')} P ↔ (P s → P s')
by (simp add: stable-def monotone-def le-bool-def)

```

```

lemma closed:

```

```

  shows stable r P ↔ r “Collect P ⊆ Collect P
unfolding stable-def monotone-def le-bool-def by auto

```

```

lemma rtrancl-conv:

```

```

  shows stable (r*) = stable r
by (auto simp: stable-def monotone-def le-bool-def fun-eq-iff elim!: rtrancl-induct)

```

```

lemma reflcl-conv:

```

```

  shows stable (r⌊) = stable r
unfolding stable-def monotone-def by simp

```

```

lemma empty[stable.intro]:

```

```

  shows stable {} P
unfolding stable-def by simp

```

```

lemma [stable.intro]:

```

```

  shows Id: stable Id P
  and Id-fst: ∧P. stable (Id ×R A) (λs. P (fst s))
  and Id-fst-fst-snd: ∧P. stable (Id ×R Id ×R A) (λs. P (fst s) (fst (snd s)))
by (simp-all add: stable-def monotone-def)

```

```

lemma UNIV:

```

shows *stable UNIV* $P \longleftrightarrow (\exists c. P = \langle c \rangle)$
unfolding *stable-def monotone-def le-bool-def* **by** *simp meson*

lemma *antimono-rel*:

shows *antimono* $(\lambda r. \text{stable } r P)$

unfolding *stable-def monotone-def* **using** *subset-iff* **by** $(\text{fastforce intro: antimonoI})$

lemmas *strengthen-rel*[*strg*] = *st-ord-antimono*[*OF stable.antimono-rel, unfolded le-bool-def*]

lemma *infI*:

assumes *stable* $r P$

shows *infI1*: *stable* $(r \cap s) P$

and *infI2*: *stable* $(s \cap r) P$

using *assms unfolding stable-def monotone-def* **by** *simp-all*

lemma *UNION-conv*:

shows *stable* $(\bigcup_{x \in X}. r x) P \longleftrightarrow (\forall x \in X. \text{stable } (r x) P)$

unfolding *stable-def monotone-def* **by** *blast*

lemmas *UNIONI*[*stable.intro*] = *iffD2*[*OF stable.UNION-conv, rule-format*]

lemma *Union-conv*:

shows *stable* $(\bigcup X) P \longleftrightarrow (\forall x \in X. \text{stable } x P)$

unfolding *stable-def monotone-def* **by** *blast*

lemma *union-conv*:

shows *stable* $(r \cup s) P \longleftrightarrow \text{stable } r P \wedge \text{stable } s P$

unfolding *stable-def monotone-def* **by** *blast*

lemmas *UnionI*[*stable.intro*] = *iffD2*[*OF stable.Union-conv, rule-format*]

lemmas *unionI*[*stable.intro*] = *iffD2*[*OF stable.union-conv, rule-format, unfolded conj-explode*]

Properties of stable with respect to the predicate **lemma** *const*[*stable.intro*]:

shows *stable* $r \langle c \rangle$

and *stable* $r \perp$

and *stable* $r \top$

by $(\text{simp-all add: stable-def monotone-def})$

lemma *conjI*[*stable.intro*]:

assumes *stable* $r P$

assumes *stable* $r Q$

shows *stable* $r (P \wedge Q)$

using *assms by (simp add: stable-def)*

lemma *disjI*[*stable.intro*]:

assumes *stable* $r P$

assumes *stable* $r Q$

shows *stable* $r (P \vee Q)$

using *assms by (simp add: stable-def monotone-def le-bool-def)*

lemma *implies-constI*[*stable.intro*]:

assumes $P \implies \text{stable } r Q$

shows *stable* $r (\lambda s. P \longrightarrow Q s)$

using *assms by (auto simp: stable-def monotone-def le-bool-def)*

lemma *allI*[*stable.intro*]:

assumes $\bigwedge x. \text{stable } r (P x)$

shows *stable* $r (\forall x. P x)$

using *assms* **by** (*simp add: stable-def monotone-def le-bool-def*)

lemma *ballI[stable.intro]*:

assumes $\bigwedge x. x \in X \implies \text{stable } r (P x)$

shows $\text{stable } r (\lambda s. \forall x \in X. P x s)$

using *assms* **by** (*simp add: stable-def monotone-def le-bool-def*)

lemma *stable-relprod-fstI[stable.intro]*:

assumes $\text{stable } r P$

shows $\text{stable } (r \times_R s) (\lambda s. P (fst s))$

using *assms* **by** (*clarsimp simp: stable-def monotone-def*)

lemma *stable-relprod-sndI[stable.intro]*:

assumes $\text{stable } s P$

shows $\text{stable } (r \times_R s) (\lambda s. P (snd s))$

using *assms* **by** (*clarsimp simp: stable-def monotone-def*)

lemma *local-only*: — for predicates that are insensitive to the shared state

assumes $\bigwedge ls s s'. P (ls, s) \longleftrightarrow P (ls, s')$

shows $\text{stable } (Id \times_R UNIV) P$

using *assms* **by** (*fastforce simp: stable-def monotone-def le-bool-def*)

lemma *Id-on-proj*:

assumes $\bigwedge v. \text{stable } Id_f (\lambda s. P v s)$

shows $\text{stable } Id_f (\lambda s. P (f s) s)$

using *assms* **unfolding** *stable-def* **by** (*rule monotone-Id-on-proj*)

lemma *if-const-conv*:

shows $\text{stable } r (\text{if } c \text{ then } P \text{ else } Q) \longleftrightarrow \text{stable } r (\lambda s. c \longrightarrow P s) \wedge \text{stable } r (\lambda s. \neg c \longrightarrow Q s)$

by (*simp add: stable-def*)

lemma *ifI[stable.intro]*:

assumes $\text{stable } r (\lambda s. c s \longrightarrow P s)$

assumes $\text{stable } r (\lambda s. \neg c s \longrightarrow Q s)$

shows $\text{stable } r (\lambda s. \text{if } c \text{ then } P s \text{ else } Q s)$

using *assms* **by** (*simp add: stable.intro*)

lemma *ifI2[stable.intro]*:

assumes $\text{stable } r (\lambda s. c s \longrightarrow P s s)$

assumes $\text{stable } r (\lambda s. \neg c s \longrightarrow Q s s)$

shows $\text{stable } r (\lambda s. (\text{if } c \text{ then } P s \text{ else } Q s) s)$

using *assms* **by** (*simp add: stable.intro*)

lemma *case-optionI[stable.intro]*:

assumes $\text{stable } r (\lambda s. \text{opt } s = \text{None} \longrightarrow \text{none } s)$

assumes $\bigwedge v. \text{stable } r (\lambda s. \text{opt } s = \text{Some } v \longrightarrow \text{some } v s)$

shows $\text{stable } r (\lambda s. \text{case opt } s \text{ of } \text{None} \Rightarrow \text{none } s \mid \text{Some } v \Rightarrow \text{some } v s)$

using *assms* **by** (*simp add: stable.intro split: option.split*)

lemma *case-optionI2[stable.intro]*:

assumes $\text{opt} = \text{None} \implies \text{stable } r \text{ none}$

assumes $\bigwedge v. \text{opt} = \text{Some } v \implies \text{stable } r (\text{some } v)$

shows $\text{stable } r (\text{case opt of } \text{None} \Rightarrow \text{none} \mid \text{Some } v \Rightarrow \text{some } v)$

using *assms* **by** (*simp add: stable.intro split: option.split*)

In practice the following rules are often too weak

lemma *impliesI*:

assumes $\text{stable } r (\neg P)$

assumes *stable r Q*
shows *stable r (P → Q)*
using *assms by (auto simp: stable-def monotone-def le-bool-def)*

lemma *exI*:
assumes $\bigwedge x. \text{stable } r (P x)$
shows *stable r (∃ x. P x)*
using *assms by (auto simp: stable-def monotone-def le-bool-def)*

lemma *beX*:
assumes $\bigwedge x. x \in X \implies \text{stable } r (P x)$
shows *stable r (λs. ∃ x ∈ X. P x s)*
using *assms by (auto simp: stable-def monotone-def le-bool-def)*

setup $\langle \text{Sign.parent-path} \rangle$

12 Refinement

We develop a refinement story for the $(\prime a, \prime s, \prime v)$ *spec* lattice.

References:

- Vafeiadis (2008) (RGsep, program logic) and Liang, Feng, and Fu (2014) (RGsim, refinement)
- Armstrong et al. (2014)
- van Staden (2015)

definition *refinement* :: $\prime s \text{ pred} \Rightarrow (\prime a, \prime s, \prime v) \text{ spec} \Rightarrow (\prime a, \prime s, \prime v) \text{ spec} \Rightarrow (\prime v \Rightarrow \prime s \text{ pred}) \Rightarrow (\prime a, \prime s, \prime v) \text{ spec}$ ($\langle \{-\} \rangle$, \vdash , $\{-\}$) $[0,0,0,0]$ 100) **where**
 $\{\{P\}\}, A \Vdash G, \{\{Q\}\} = \text{spec.pre } P \sqcap A \longrightarrow_+ G \sqcap \text{spec.post } Q$

An intuitive gloss on the proposition $c \leq \{\{P\}\}, A \Vdash G, \{\{Q\}\}$ is: assuming the precondition P holds and all steps conform to the process A , then c is a refinement of G and satisfies the postcondition Q .

Observations:

- We use *next-imp* here; (\longrightarrow_H) is (only) enough for an assume/guarantee program logic (see §12.2)
- A is arbitrary but is intended to constrain only *env* steps
 - similarly termination can depend on A : a parallel composition can only terminate if all of its constituent processes terminate
- As $P \longrightarrow_+ Q$ implies $\text{idle} \leq Q$, in practice $\text{idle} \leq G$
- see §13.4.1 for some introduction rules

setup $\langle \text{Sign.mandatory-path refinement} \rangle$

lemma *E*:
assumes $c \leq \{\{P\}\}, A \Vdash G, \{\{Q\}\}$
obtains $c \leq \text{spec.pre } P \sqcap A \longrightarrow_+ G$
and $c \leq \text{spec.pre } P \sqcap A \longrightarrow_+ \text{spec.post } Q$
using *assms by (simp add: refinement-def spec.next-imp.infR)*

lemma *pre-post-cong*:
assumes $P = P'$
assumes $Q = Q'$
shows $\{\{P\}\}, A \Vdash G, \{\{Q\}\} = \{\{P'\}\}, A \Vdash G, \{\{Q'\}\}$
using *assms by simp*

lemma *top*:

shows $\{\!|P|\!\}, A \Vdash \top, \{\!\top\!\} = \top$
and $\{\!|P|\!\}, A \Vdash \top, \{\!\langle \top \rangle\!\} = \top$
and $\{\!|P|\!\}, A \Vdash \top, \{\!\lambda -. True\!\} = \top$

by (*simp-all add: refinement-def*)

lemma *mcont2mcont[cont-intro]*:

assumes *mcont luba orda Sup* (\leq) *G*
shows *mcont luba orda Sup* (\leq) ($\lambda x. \{\!|P|\!\}, A \Vdash G x, \{\!|Q|\!\}$)
by (*simp add: assms refinement-def*)

lemma *mono*:

assumes $P' \leq P$
assumes $A' \leq A$
assumes $G \leq G'$
assumes $Q \leq Q'$
shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$

unfolding *refinement-def*

apply (*strengthen ord-to-strengthen(1)[OF assms(1)]*)
apply (*strengthen ord-to-strengthen(1)[OF assms(2)]*)
apply (*strengthen ord-to-strengthen(1)[OF assms(3)]*)
apply (*strengthen ord-to-strengthen(1)[OF assms(4)]*)
apply *simp*
done

lemma *strengthen[strg]*:

assumes *st-ord* ($\neg F$) *P P'*
assumes *st-ord* ($\neg F$) *A A'*
assumes *st-ord* *F G G'*
assumes *st-ord* *F Q Q'*
shows *st-ord* *F* ($\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$) ($\{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$)

using *assms* **by** (*cases F; simp add: refinement.mono*)

lemma *mono-stronger*:

assumes $P' \leq P$
assumes *spec.pre* $P' \sqcap A' \leq A$
assumes *spec.pre* $P' \sqcap G \leq A' \longrightarrow_+ G'$
assumes $Q \leq Q'$
assumes *spec.idle* $\leq G'$
shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$

unfolding *refinement-def*

apply (*strengthen ord-to-strengthen(2)[OF assms(1)]*)
apply (*strengthen ord-to-strengthen(2)[OF assms(2)]*)
apply (*strengthen ord-to-strengthen(2)[OF assms(4)]*)
apply (*simp add: spec.next-imp.infR*)
apply (*metis assms(3) heyting commute le-infI1*
spec.next-imp.closure.cl spec.pre.next-imp-eq-heyting(2)[OF assms(5)])

done

lemmas *pre-ag = order.trans[OF - refinement.mono[OF order.refl - - order.refl], of c]* **for** *c*

lemmas *pre-a = refinement.pre-ag[OF - - order.refl]*

lemmas *pre-g = refinement.pre-ag[OF - order.refl]*

lemma *pre*:

assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
assumes $\bigwedge s. P' s \implies P s$
assumes $A' \leq A$

assumes $G \leq G'$
assumes $\bigwedge s v. Q s v \implies Q' s v$
shows $c \leq \{\!\{P}\!\}, A' \Vdash G', \{\!\{Q'}\!\}$
using *assms(2-)* **by** (*blast intro: order.trans[OF assms(1) refinement.mono]*)

lemmas *pre-pre-post* = *refinement.pre*[*OF - - order.refl order.refl, of c*] **for** *c*

lemma *pre-imp*:
assumes $\bigwedge s. P s \implies P' s$
assumes $c \leq \{\!\{P'}\!\}, A \Vdash G, \{\!\{Q}\!\}$
shows $c \leq \{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\}$
using *assms refinement.pre* **by** *blast*

lemmas *pre-pre* = *refinement.pre-imp*[*rotated*]

lemma *post-imp*:
assumes $\bigwedge v s. Q v s \implies R v s$
assumes $c \leq \{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\}$
shows $c \leq \{\!\{P}\!\}, A \Vdash G, \{\!\{R}\!\}$
using *assms refinement.pre* **by** *blast*

lemmas *pre-post* = *refinement.post-imp*[*rotated*]

lemmas *strengthen-post* = *refinement.pre-post*

lemma *pre-inf-assume*:
shows $\{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\} = \{\!\{P}\!\}, A \sqcap \text{spec.pre } P \Vdash G, \{\!\{Q}\!\}$
by (*simp add: refinement-def ac-simps*)

lemma *pre-assume-absorb*:
assumes $A \leq \text{spec.pre } P$
shows $\{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\} = \{\!\{\top}\!\}, A \Vdash G, \{\!\{Q}\!\}$
by (*simp add: assms refinement-def inf-absorb2*)

lemmas *sup* = *sup-least*[**where** $x = \{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\}$] **for** $A G P Q$

lemma
shows *supRL*: $c \leq \{\!\{P}\!\}, A \Vdash G_1, \{\!\{Q}\!\} \implies c \leq \{\!\{P}\!\}, A \Vdash G_1 \sqcup G_2, \{\!\{Q}\!\}$
and *supRR*: $c \leq \{\!\{P}\!\}, A \Vdash G_2, \{\!\{Q}\!\} \implies c \leq \{\!\{P}\!\}, A \Vdash G_1 \sqcup G_2, \{\!\{Q}\!\}$
by (*simp-all add: refinement.pre-g*)

lemma *infR-conv*:
shows $\{\!\{P}\!\}, A \Vdash G_1 \sqcap G_2, \{\!\{Q_1 \sqcap Q_2}\!\} = \{\!\{P}\!\}, A \Vdash G_1, \{\!\{Q_1}\!\} \sqcap \{\!\{P}\!\}, A \Vdash G_2, \{\!\{Q_2}\!\}$
by (*simp add: refinement-def ac-simps spec.next-imp.infR spec.post.inf*)

lemma *inf-le*:
shows $X \sqcap \{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\} \leq \{\!\{P}\!\}, X \sqcap A \Vdash X \sqcap G, \{\!\{Q}\!\}$
by (*simp add: refinement-def le-infI1 le-infI2*
spec.next-imp.infR spec.next-imp.mono spec.next-imp.contains)

lemma *heyting-le*:
shows $\{\!\{P}\!\}, A \sqcap B \Vdash B \longrightarrow_H G, \{\!\{Q}\!\} \leq B \longrightarrow_H \{\!\{P}\!\}, A \Vdash G, \{\!\{Q}\!\}$
by (*simp add: refinement-def ac-simps heyting.infR heyting commute*
spec.next-imp.heytingL-distrib spec.next-imp.mono)

lemma *heyting-pre*:
assumes *spec.idle* $\leq G$
shows *spec.pre* $P \longrightarrow_H \{\!\{P'}\!\}, A \Vdash G, \{\!\{Q}\!\} = \{\!\{P \wedge P'}\!\}, A \Vdash G, \{\!\{Q}\!\}$
by (*simp add: ac-simps refinement-def spec.pre.conj assms spec.idle.post-le*)

flip: spec.pre.next-imp-eq-heyting)

lemma *sort-of-refl*:

assumes $c \leq \langle P \rangle$, $A \Vdash G$, $\langle Q \rangle$

shows $c \leq \langle P \rangle$, $A \Vdash c$, $\langle Q \rangle$

using *assms* **by** (*simp add: refinement-def spec.next-imp.infR spec.next-imp.closure.expansive*)

lemma *gen-asm-base*:

assumes $P \implies c \leq \langle P' \wedge P'' \rangle$, $A \Vdash G$, $\langle Q \rangle$

assumes *spec.idle* $\leq G$

shows $c \leq \langle P' \wedge \langle P \rangle \wedge P'' \rangle$, $A \Vdash G$, $\langle Q \rangle$

using *assms* **by** (*simp add: refinement-def spec.pre.conj spec.pre.K spec.next-imp.botL spec.idle.post-le*)

lemmas *gen-asm* =

refinement.gen-asm-base[**where** $P' = \langle \text{True} \rangle$ **and** $P'' = \langle \text{True} \rangle$, *simplified*]

refinement.gen-asm-base[**where** $P' = \langle \text{True} \rangle$, *simplified*]

refinement.gen-asm-base[**where** $P'' = \langle \text{True} \rangle$, *simplified*]

refinement.gen-asm-base

lemma *post-conj*:

assumes $c \leq \langle P \rangle$, $A \Vdash G$, $\langle Q \rangle$

assumes $c \leq \langle P \rangle$, $A \Vdash G$, $\langle Q' \rangle$

shows $c \leq \langle P \rangle$, $A \Vdash G$, $\langle \lambda rv. Q \text{ } rv \wedge Q' \text{ } rv \rangle$

using *assms* **unfolding** *refinement-def* **by** (*simp add: spec.post.conj spec.next-imp.infR*)

lemma *conj-lift*:

assumes $c \leq \langle P \rangle$, $A \Vdash G$, $\langle Q \rangle$

assumes $c \leq \langle P' \rangle$, $A \Vdash G$, $\langle Q' \rangle$

shows $c \leq \langle P \wedge P' \rangle$, $A \Vdash G$, $\langle \lambda rv. Q \text{ } rv \wedge Q' \text{ } rv \rangle$

using *assms* **by** (*blast intro: refinement.post-conj refinement.pre*)

lemma *drop-imp*:

assumes $c \leq \langle P \rangle$, $A \Vdash G$, $\langle Q \rangle$

shows $c \leq \langle P \rangle$, $A \Vdash G$, $\langle \lambda rv. Q' \text{ } rv \longrightarrow Q \text{ } rv \rangle$

using *assms* *refinement.strengthen-post* **by** *fastforce*

lemma *prop*:

shows $c \leq \langle \langle P \rangle \rangle$, $A \Vdash c$, $\langle \lambda v. \langle P \rangle \rangle$

by (*simp add: refinement.sort-of-refl*[**where** $G = \top$] *refinement.gen-asm refinement.top*)

lemma *name-pre-state*:

assumes $\bigwedge s. P \text{ } s \implies c \leq \langle (=) \text{ } s \rangle$, $A \Vdash G$, $\langle Q \rangle$

assumes *spec.idle* $\leq G$

shows $c \leq \langle P \rangle$, $A \Vdash G$, $\langle Q \rangle$ (**is** $?lhs \leq ?rhs$)

proof –

have $\langle \sigma \rangle \leq G \wedge \langle \sigma \rangle \leq \text{spec.post } Q$

if $\langle \sigma \rangle \leq c$

and $\forall \sigma'' < \sigma. \langle \sigma'' \rangle \leq \text{spec.pre } P \wedge \langle \sigma'' \rangle \leq A$

for σ

proof(*cases trace.rest* $\sigma = [] \wedge \text{trace.term } \sigma = \text{None}$)

case *True* **with** $\langle \text{spec.idle} \leq G \rangle$ **show** *?thesis*

by (*cases* σ) (*simp add: spec.singleton.le-conv order.trans*[*OF spec.idle.minimal-le*])

next

case *False* **with** *order.trans*[*OF* $\langle \langle \sigma \rangle \leq c \rangle$] *assms*(1)[**where** $s = \text{trace.init } \sigma$] *that*(2)

show *?thesis*

by (*cases* σ)

(*clarsimp simp: refinement-def spec.singleton.next-imp-le-conv spec.singleton.le-conv*;

fastforce simp: trace.less dest: spec[**where** $x = \text{trace.T } (\text{trace.init } \sigma) [] \text{None}$])

```

qed
then show ?thesis
  by – (rule spec.singleton-le-extI;
        auto simp: refinement-def spec.singleton.next-imp-le-conv
            intro: order.trans[OF spec.singleton.mono])
qed

```

```

setup <Sign.parent-path>

```

12.1 Geenral rules for the ('a, 's, 'v) spec lattice

```

setup <Sign.mandatory-path spec>

```

```

setup <Sign.mandatory-path term>

```

```

setup <Sign.mandatory-path all>

```

lemma *refinement*:

```

  shows spec.term.all (⟦P⟧, A ⊨ G, ⟦Q⟧) = ⟦P⟧, spec.term.all A ⊨ spec.term.all G, ⟦⊤⟧
  by (simp add: refinement-def spec.term.all.next-imp spec.term.all.inf spec.term.all.pre spec.term.all.post)

```

```

setup <Sign.parent-path>

```

```

setup <Sign.mandatory-path none>

```

lemma *refinement-le*:

```

  shows spec.term.none (⟦P⟧, A ⊨ G, ⟦Q⟧) ≤ ⟦P⟧, spec.term.all A ⊨ spec.term.all G, ⟦⊥⟧
  by (simp add: spec.term.galois spec.term.all.refinement order.trans[OF spec.term.all.expansive])

```

```

setup <Sign.parent-path>

```

```

setup <Sign.parent-path>

```

```

setup <Sign.mandatory-path invmap>

```

lemma *refinement*:

```

  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: 'v ⇒ 'w
  fixes A :: ('b, 't, 'w) spec
  fixes G :: ('b, 't, 'w) spec
  fixes P :: 't pred
  fixes Q :: 'w ⇒ 't pred
  shows spec.invmap af sf vf (⟦P⟧, A ⊨ G, ⟦Q⟧)
    = (⟦λs. P (sf s)⟧, spec.invmap af sf vf A ⊨ spec.invmap af sf vf G, ⟦λv s. Q (vf v) (sf s)⟧)

```

unfolding *refinement-def*

```

  by (simp only: spec.next-imp.invmap spec.invmap.inf spec.invmap.pre spec.invmap.post)

```

```

setup <Sign.parent-path>

```

```

setup <Sign.parent-path>

```

12.1.1 Actions

Actions are anchored at the start of a trace, and therefore must be an initial step of the assume A . However by the semantics of (\longrightarrow_+) we may only know that that initial state of the trace is acceptable to A when showing that F -steps are F' -steps (the second assumption). This hypothesis is vacuous when $idle \leq A$.

```

setup <Sign.mandatory-path refinement.spec>

```

lemma *action*:

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$

assumes $\bigwedge v a s s'. \llbracket P \ s; (v, a, s, s') \in F; (a, s, s') \in \text{spec.initial-steps } A \rrbracket \implies Q \ v \ s'$

assumes $\bigwedge v a s s'. \llbracket P \ s; (v, a, s, s') \in F; (a, s, s) \in \text{spec.initial-steps } A \rrbracket \implies (v, a, s, s') \in F'$

shows $\text{spec.action } F \leq \llbracket P \rrbracket, A \Vdash \text{spec.action } F', \llbracket Q \rrbracket$

proof –

have $\text{spec.action } (F \cap \text{UNIV} \times \text{UNIV} \times \text{Pre } P) \leq A \longrightarrow_+ \text{spec.action } F' \sqcap \text{spec.post } Q$

proof(*induct rule: spec.action-le*)

case *idle* **show** *?case*

by (*simp add: spec.next-imp.contains spec.idle.action-le spec.idle.post-le*)

next

case (*step v a s s'*) **then show** *?case*

by (*fastforce simp: spec.singleton.next-imp-le-conv trace.split-all spec.initial-steps-def*

trace.less Cons-eq-append-conv spec.singleton.post-le-conv

order.trans[OF spec.idle.minimal-le spec.idle.action-le]

elim: assms trace.less-eqE prefixE

intro: spec.action.stepI)

qed

then show *?thesis*

by (*simp add: refinement-def spec.pre.next-imp-eq-heyting spec.idle.post-le spec.idle.action-le*

heyting order.trans[OF spec.pre.inf-action-le(2)])

qed

lemma *return*:

shows $\text{spec.return } v \leq \llbracket Q \ v \rrbracket, A \Vdash \text{spec.return } v, \llbracket Q \rrbracket$

by (*auto simp: spec.return-def intro: refinement.spec.action*)

lemma *action-rel*:

fixes $F :: ('v \times 'a \times 's \times 's) \text{ set}$

assumes $\bigwedge v a s s'. \llbracket P \ s; (v, a, s, s') \in F; (a, s, s') \in \text{spec.initial-steps } A \rrbracket \implies Q \ v \ s'$

assumes $\bigwedge v a s s'. \llbracket P \ s; (v, a, s, s') \in F; (a, s, s) \in \text{spec.initial-steps } A; s \neq s' \rrbracket \implies (a, s, s') \in r$

shows $\text{spec.action } F \leq \llbracket P \rrbracket, A \Vdash \text{spec.rel } r, \llbracket Q \rrbracket$

by (*force simp: spec.rel-def spec.rel.act-def spec.term.all.action*

intro: assms refinement.supRL refinement.spec.action

refinement.pre-g[OF - spec.term.all.mono[OF spec.kleene.expansive-star]])

setup $\langle \text{Sign.parent-path} \rangle$

12.1.2 Bind

Consider showing $f \ggg g \leq f' \ggg g'$ under the assume A and pre/post conditions P/Q .

The tricky part is to residuate the assume A wrt the process f for use in the refinement proof of g .

- we want to preserve as much of the structure of A as possible
- intuition: we want all the ways a trace can continue in A having started with a terminating trace in f
- intuitively a right residual for (\ggg) should do the job
 - however unlike [Hoare and He \(1987\)](#) we have no chance of a right residual for (\ggg) as we use traces (they use relations)
 - * i.e., if it is not the case that $f \ggg \perp \leq A$ then there is no continuation h such that $f \ggg h \leq A$.
 - * also such a residual h has arbitrary behavior starting from states that f cannot reach
 - i.e., for traces $\neg\sigma \leq f \langle \sigma \rangle \ggg h \leq A$ need not hold
 - and the user-provided assertions may be too weak to correct for this
- in practice the termination information in the assume A is not useful

We therefore define an ad hoc residual that does the trick.

See [Emerson \(1983, §4\)](#) for some related concerns.

setup $\langle \text{Sign.mandatory-path refinement.spec.bind} \rangle$

definition $\text{res} :: ('a, 's, 'v) \text{spec} \Rightarrow ('a, 's, 'w) \text{spec} \Rightarrow 'v \Rightarrow ('a, 's, 'w) \text{spec}$ **where**
 $\text{res } f \ A \ v = \sqcup \{ \langle \text{trace.final}' \ s \ xs, \ ys, \ w \rangle \mid s \ xs \ ys \ w. \langle s, \ xs, \ \text{Some } v \rangle \leq f \wedge \langle s, \ xs \ @ \ ys, \ \text{None} \rangle \leq A \}$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.singleton.refinement.bind} \rangle$

lemma $\text{res-le-conv}[\text{spec.singleton.le-conv}]$:

shows $\langle \sigma \rangle \leq \text{refinement.spec.bind.res } f \ A \ v$

$\longleftrightarrow (\exists s \ xs. \langle s, \ xs, \ \text{Some } v \rangle \leq f$

$\wedge \text{trace.init } \sigma = \text{trace.final}' \ s \ xs$

$\wedge \langle s, \ xs \ @ \ \text{trace.rest } \sigma, \ \text{None} \rangle \leq A)$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof(*rule iffI*)

show $?lhs \Longrightarrow ?rhs$

by (*fastforce simp: refinement.spec.bind.res-def trace.split-Ex spec.singleton-le-conv*

trace.less-eq-None trace.natural'.append trace.natural-def

elim: trace.less-eqE order.trans[rotated])

show $?rhs \Longrightarrow ?lhs$

by (*cases* σ) (*clarsimp simp: refinement.spec.bind.res-def; blast*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.term.none.refinement.bind} \rangle$

lemma resL :

shows $\text{refinement.spec.bind.res } (\text{spec.term.none } f) \ A \ v = \perp$

by (*simp add: refinement.spec.bind.res-def spec.singleton.le-conv*)

lemma resR :

shows $\text{refinement.spec.bind.res } f \ (\text{spec.term.none } A) \ v = \text{refinement.spec.bind.res } f \ A \ v$

by (*simp add: refinement.spec.bind.res-def spec.singleton.le-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.term.all.refinement.bind} \rangle$

lemma resR-mono :

shows $\text{refinement.spec.bind.res } f \ (\text{spec.term.all } A) \ v = \text{refinement.spec.bind.res } f \ A \ v$

by (*simp add: refinement.spec.bind.res-def spec.singleton.le-conv*)

(*meson dual-order.trans spec.singleton.less-eq-None*)

lemma res :

shows $\text{spec.term.all } (\text{refinement.spec.bind.res } f \ A \ v) = \text{refinement.spec.bind.res } f \ A \ v$

by (*rule spec.singleton.antisym*) (*auto simp: spec.singleton.le-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.term.closed.refinement.bind} \rangle$

lemma res :

shows $\text{refinement.spec.bind.res } f \ A \ v \in \text{spec.term.closed}$

by (*subst spec.term.all.refinement.bind.res[symmetric]*) (*rule spec.term.all.closed*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path refinement.spec.bind.res} \rangle$

lemma *bot*:

shows *botL*: $\text{refinement.spec.bind.res } \perp = \perp$

and *botR*: $\text{refinement.spec.bind.res } f \perp = \perp$

by (*simp-all add: refinement.spec.bind.res-def fun-eq-iff spec.singleton.not-bot*)

lemma *mono*:

assumes $f \leq f'$

assumes $A \leq A'$

shows $\text{refinement.spec.bind.res } f A v \leq \text{refinement.spec.bind.res } f' A' v$

using *assms unfolding refinement.spec.bind.res-def* **by** (*fastforce intro!: Sup-subset-mono*)

lemma *strengthen[stg]*:

assumes *st-ord* $F f f'$

assumes *st-ord* $F A A'$

shows *st-ord* $F (\text{refinement.spec.bind.res } f A v) (\text{refinement.spec.bind.res } f' A' v)$

using *assms* **by** (*cases F; simp add: refinement.spec.bind.res.mono*)

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone orda* $(\leq) f$

assumes *monotone orda* $(\leq) A$

shows *monotone orda* $(\leq) (\lambda x. \text{refinement.spec.bind.res } (f x) (A x) v)$

using *assms* **by** (*simp add: monotone-def refinement.spec.bind.res.mono*)

lemma *SupL*:

shows $\text{refinement.spec.bind.res } (\bigsqcup X) A v = (\bigsqcup x \in X. \text{refinement.spec.bind.res } x A v)$

by (*rule antisym; simp add: refinement.spec.bind.res-def; blast*)

lemma *SupR*:

shows $\text{refinement.spec.bind.res } f (\bigsqcup X) v = (\bigsqcup x \in X. \text{refinement.spec.bind.res } f x v)$

by (*rule antisym; simp add: refinement.spec.bind.res-def; blast*)

lemma *InfL-le*:

shows $\text{refinement.spec.bind.res } (\bigsqcap X) A v \leq (\bigsqcap x \in X. \text{refinement.spec.bind.res } x A v)$

by (*simp add: refinement.spec.bind.res-def le-Inf-iff*) *fast*

lemma *InfR-le*:

shows $\text{refinement.spec.bind.res } f (\bigsqcap X) v \leq (\bigsqcap x \in X. \text{refinement.spec.bind.res } f x v)$

by (*simp add: refinement.spec.bind.res-def le-Inf-iff*) *fast*

lemma *mcont2mcont[cont-intro]*:

assumes *mcont luba orda Sup* $(\leq) f$

assumes *mcont luba orda Sup* $(\leq) A$

shows *mcont luba orda Sup* $(\leq) (\lambda x. \text{refinement.spec.bind.res } (f x) (A x) v)$

proof(*rule ccpo.mcont2mcont'[OF complete-lattice-ccpo - - assms(1)]*)

show *mcont Sup* $(\leq) \text{Sup } (\leq) (\lambda f. \text{refinement.spec.bind.res } f (A x) v)$ **for** x

by (*intro mcontI contI monotoneI*)

(*simp-all add: refinement.spec.bind.res.mono refinement.spec.bind.res.SupL*)

show *mcont luba orda Sup* $(\leq) (\lambda x. \text{refinement.spec.bind.res } f (A x) v)$ **for** f

by (*intro mcontI monotoneI contI*)

(*simp-all add: mcont-monoD[OF assms(2)] refinement.spec.bind.res.mono contD[OF mcont-cont[OF assms(2)]]*)

refinement.spec.bind.res.SupR image-image)

qed

lemma *returnL*:

assumes $\text{spec.idle} \leq A$
shows $\text{refinement.spec.bind.res} (\text{spec.return } v) A v = \text{spec.term.all } A$ (**is** $?lhs = ?rhs$)
proof(*rule antisym[OF - spec.singleton-le-extI]*)
show $?lhs \leq ?rhs$
by (*auto simp: refinement.spec.bind.res-def trace.split-all spec.singleton-le-conv*)
show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ
using *that*
by (*auto simp: spec.singleton-le-conv*
intro!: exI[where x=trace.init σ] exI[where x=[]]
elim: order.trans[rotated])

qed

lemma *rel-le*:

assumes $r \subseteq r'$
shows $\text{refinement.spec.bind.res } f (\text{spec.rel } r) v \leq \text{spec.rel } r'$
using *assms* **by** (*force intro: spec.singleton-le-extI simp: spec.singleton-le-conv trace.steps'.append*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.steps.refinement.spec.bind} \rangle$

lemma *res-le*: — we can always discard the extra structure

shows $\text{spec.steps} (\text{refinement.spec.bind.res } f A v) \subseteq \text{spec.steps } A$
by (*meson order-trans refinement.spec.bind.res.mono refinement.spec.bind.res.rel-le*
spec.rel.galois spec.rel.upper-lower-expansive)

setup $\langle \text{Sign.parent-path} \rangle$

A refinement rule for (\gg). The function *vf* abstracts interstitial monadic return values.

setup $\langle \text{Sign.mandatory-path refinement.spec} \rangle$

lemma *bind-abstract*:

fixes $f :: ('a, 's, 'v) \text{ spec}$
fixes $f' :: ('a, 's, 'v') \text{ spec}$
fixes $g :: 'v \Rightarrow ('a, 's, 'w) \text{ spec}$
fixes $g' :: 'v' \Rightarrow ('a, 's, 'w) \text{ spec}$
fixes $vf :: 'v \Rightarrow 'v'$
assumes $g: \bigwedge v. g v \leq \{Q' (vf v)\}$, $\text{refinement.spec.bind.res} (\text{spec.pre } P \sqcap \text{spec.term.all } A \sqcap f') A (vf v) \Vdash g'$
 $(vf v), \{Q\}$
assumes $f: f \leq \{P\}$, $\text{spec.term.all } A \Vdash \text{spec.vinvm} vf f', \{\lambda v. Q' (vf v)\}$
shows $f \gg g \leq \{P\}$, $A \Vdash f' \gg g', \{Q\}$
proof (*rule order.trans[OF spec.bind.mono[OF f g]]*,
unfold refinement-def spec.bind.inf-post,
induct rule: spec.bind-le)
case incomplete show *?case*
apply (*simp add: spec.term.galois spec.term.all.next-imp spec.term.all.bind spec.term.all.inf*
spec.term.all.post spec.term.all.pre)
apply (*simp add: spec.next-imp.mono[OF order.refl] le-supI1 le-infI1 spec.term.none.invm*
spec.invm.id
flip: spec.term.galois)
done
next
case (*continue* $\sigma_f \sigma_g v$)
have $\langle s, xs, w \rangle \leq f' \gg (\lambda v. g' v \sqcap \text{spec.post } Q)$
if $le: \text{trace.T } s \ xs \ w \leq \text{trace.T} (\text{trace.init } \sigma_f) (\text{trace.rest } \sigma_f @ \text{trace.rest } \sigma_g) (\text{trace.term } \sigma_g)$
and $pre: \forall \sigma'' < \text{trace.T } s \ xs \ w. \langle \sigma'' \rangle \leq \text{spec.pre } P \sqcap A$

```

for s xs w
using le
proof(induct rule: trace.less-eq-extE)
case prefix
from prefix(3) show ?case
proof(induct rule: prefix-append-not-NilE[case-names incomplete1 continue1])
case incomplete1 with pre continue(1) prefix(1,2) show ?case
apply (clarsimp simp: spec.singleton.next-imp-le-conv)
apply (drule spec[where x=trace.T s xs None],
      drule mp[where P=trace.T s xs None ≤ σf])
apply (force simp: spec.singleton.le-conv spec.map.singleton
        le-inf-iff trace.less trace.split-All trace.less-eq-None
        simp flip: spec.map-invmap.galois
        intro!: spec.bind.incompleteI)+
done
next
case (continue1 us)
from continue(1,3) prefix(2) continue1(1,2)
spec[OF pre, where x=trace.T (trace.init σf) (trace.rest σf) None]
have ⟨trace.init σf, trace.rest σf, Some (vf v)⟩ ≤ spec.pre P ⊓ spec.term.all A ⊓ f' ⊓ spec.post Q'
apply (cases σf)
apply (clarsimp simp: spec.singleton.le-conv spec.singleton.next-imp-le-conv
        trace.less le-inf-iff exI[where x=None]
        split: option.split-asm
        dest!: spec[where x=σf])
apply (metis append-is-Nil-conv le-inf-iff pre trace.less-same-append-conv)
done
with pre continue(1,2,5) prefix(1,2) continue1
spec[OF continue(4)[unfolded spec.singleton.next-imp-le-conv],
      where x=trace.T (trace.init σg) us None]
show ?case
apply clarsimp
apply (rule spec.bind.continueI[where v=vf v], assumption)
apply (clarsimp simp: spec.singleton.le-conv trace.split-All trace.less-eq-None trace.less)
apply (metis append.assoc)
done
qed
next
case (maximal w')
from maximal(1-3) continue(1,3)
spec[OF pre, where x=trace.T (trace.init σf) (trace.rest σf) None]
have ⟨trace.init σf, trace.rest σf, Some (vf v)⟩ ≤ spec.pre P ⊓ spec.term.all A ⊓ f' ⊓ spec.post Q'
apply (cases σf)
apply (clarsimp simp: spec.singleton.le-conv spec.singleton.next-imp-le-conv le-inf-iff
        split: option.split-asm)
apply (force simp: trace.less spec.singleton.mono trace.less-eq-same-append-conv
        elim: notE order.trans[rotated]
        dest!: spec[where x=σf] spec[where x=None])
done
with maximal(2-4) pre continue(2)
spec[OF continue(4)[unfolded spec.singleton.next-imp-le-conv], where x=σg]
show ?case
by (cases σg)
(auto 0 2 intro!: spec.bind.continueI[where v=vf v] exI[where x=s]
    simp: spec.singleton.le-conv trace.split-All trace.less)
qed
then show ?case
by (clarsimp simp: spec.singleton.next-imp-le-conv trace.split-all)

```

qed

lemmas *bind* = *refinement.spec.bind-abstract*[**where** *vf=id, simplified spec.invmap.id, simplified*]

12.1.3 Interference

lemma *rel-mono*:

assumes $r \subseteq r'$

assumes *stable* (*snd* ' (*spec.steps* $A \cap r$)) *P*

shows *spec.rel* $r \leq \{\!\{P\}\!\}$, $A \Vdash \text{spec.rel } r'$, $\{\!\{\lambda\!-\!::\text{unit. } P\}\!\}$

apply (*subst* (1) *spec.rel.monomorphic-conv*)

using *assms*(2)

proof(*induct arbitrary: A rule: spec.kleene.star.fixp-induct*[*case-names adm bot step*])

case (*step R A*)

have *: *spec.rel.act* $r \leq \{\!\{P\}\!\}$, *spec.term.all* $A \Vdash \text{spec.rel } r'$, $\{\!\{P\}\!\}$

unfolding *spec.rel.act-def*

proof(*rule refinement.spec.action-rel*)

show $P s'$

if $P s$

and $(v, a, s, s') \in \{\!\{()\}\!\} \times (r \cup \text{UNIV} \times \text{Id})$

and $(a, s, s') \in \text{spec.initial-steps} (\text{spec.term.all } A)$

for $v a s s'$

using *that monotoneD*[*OF stable.antimono-rel, unfolded le-bool-def, rule-format, OF - step.premis,*
where $x=\{(s, s')\}$]

by (*cases* $s = s'$;

force simp: spec.initial-steps.term.all stable.singleton-conv

dest: subsetD[*OF spec.initial-steps.steps-le*])

show $(a, s, s') \in r'$ if $(v, a, s, s') \in \{\!\{()\}\!\} \times (r \cup \text{UNIV} \times \text{Id})$ and $s \neq s'$ for $v a s s'$

using *that assms*(1) by *fast*

qed

show ?*case*

apply (*rule refinement.sup*[*OF - refinement.pre-g*[*OF refinement.spec.return spec.return.rel-le*]])

apply (*subst spec.rel.unwind-bind*)

apply (*rule refinement.spec.bind*[*OF step.hyps* *])

apply (*force intro: monotoneD*[*OF stable.antimono-rel, unfolded le-bool-def, rule-format, OF - step.premis*]
dest: subsetD[*OF spec.steps.refinement.spec.bind.res-le*])

done

qed *simp-all*

setup $\langle \text{Sign.parent-path} \rangle$

12.1.4 Parallel

Our refinement rule for *Parallel* does not constrain the constituent processes in any way, unlike Abadi and Plotkin's proposed rule (see §9.2).

setup $\langle \text{Sign.mandatory-path refinement.spec} \rangle$

definition — roughly the *Parallel* construction with roles reversed

env-hyp :: $('a \Rightarrow 's \text{ pred}) \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec} \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}) \Rightarrow 'a \Rightarrow$
 $(\text{sequential}, 's, \text{unit}) \text{ spec}$

where

env-hyp $P A \text{ as } Ps a =$

spec.pre ($\prod (P \text{ ' as})$)

$\sqcap \text{spec.amap} (\text{toConcurrent-fn} (\text{proc } a))$

$(\text{spec.rel} ((\{\!\{env\}\!\} \cup \text{proc ' as}) \times \text{UNIV}))$

$\sqcap (\prod i \in \text{as. spec.toConcurrent } i (Ps i))$

$\sqcap \text{spec.ainvmap toSequential-fn } A$

setup $\langle \text{Sign.mandatory-path env-hyp} \rangle$

lemma *mono*:

assumes $\bigwedge a. a \in as \implies P a \leq P' a$

assumes $A \leq A'$

assumes $\bigwedge a. a \in as \implies Ps a \leq Ps' a$

shows $\text{refinement.spec.env-hyp } P A as Ps a \leq \text{refinement.spec.env-hyp } P' A' as Ps' a$

unfolding $\text{refinement.spec.env-hyp-def}$

apply ($\text{strengthen ord-to-strengthen}(2)[OF \text{ assms}(1)], \text{assumption}$)

apply ($\text{strengthen ord-to-strengthen}(1)[OF \text{ assms}(2)]$)

apply ($\text{strengthen ord-to-strengthen}(1)[OF \text{ assms}(3)], \text{assumption}$)

apply *simp*

done

lemma $\text{strengthen}[strg]$:

assumes $\bigwedge a. a \in as \implies \text{st-ord } F (P a) (P' a)$

assumes $\text{st-ord } F A A'$

assumes $\bigwedge a. a \in as \implies \text{st-ord } F (Ps a) (Ps' a)$

shows $\text{st-ord } F (\text{refinement.spec.env-hyp } P A as Ps a) (\text{refinement.spec.env-hyp } P' A' as Ps' a)$

using *assms* **by** ($\text{cases } F$; $\text{simp add: refinement.spec.env-hyp.mono}$)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *Parallel*:

fixes $A :: (\text{sequential}, 's, \text{unit}) \text{ spec}$

fixes $Q :: 'a \Rightarrow 's \Rightarrow \text{bool}$

fixes $Ps :: 'a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$

fixes $Ps' :: 'a \Rightarrow (\text{sequential}, 's, \text{unit}) \text{ spec}$

assumes $\bigwedge a. a \in as \implies Ps a \leq \{\!\{ P a \}\!\}, \text{refinement.spec.env-hyp } P A as Ps' a \Vdash Ps' a, \{\!\{ \lambda v. Q a \}\!\}$

shows $\text{spec.Parallel } as Ps \leq \{\!\{ \bigwedge a \in as. P a \}\!\}, A \Vdash \text{spec.Parallel } as Ps', \{\!\{ \lambda v. \bigwedge a \in as. Q a \}\!\}$

proof($\text{cases } as = \{\!\{ \}\!\}$)

case *True* **then show** *?thesis*

by ($\text{simp add: spec.Parallel.no-agents refinement.sort-of-refl}[\text{where } G=\top] \text{refinement.top}$)

next

case *False* **then show** *?thesis*

unfolding refinement-def

apply ($\text{subst } (1) \text{spec.Parallel-def}$)

apply ($\text{simp only: spec.map-invmap.galois spec.invmap.inf spec.next-imp.invmap spec.invmap.post spec.invmap.pre}$)

apply ($\text{subst } (1) \text{spec.Parallel-def}$)

apply ($\text{strengthen ord-to-strengthen}(2)[OF \text{ spec.map-invmap.upper-lower-expansive}]$)

apply (subst inf.assoc)

apply ($\text{subst spec.next-imp.infR}$)

apply ($\text{simp only: spec.next-imp.contains inf.bounded-iff inf-le1}$)

apply ($\text{strengthen ord-to-strengthen}[OF \text{ assms}], \text{assumption}$)

apply ($\text{simp only: spec.invmap.refinement id-apply simp-thms}$)

apply ($\text{rule order.trans}[\text{rotated}, OF \text{ spec.Abadi-Merz-Theorem}_4][\text{where } I=as$

and $As=\lambda a. \text{spec.pre } (P a) \sqcap \text{spec.toConcurrent } a (\text{refinement.spec.env-hyp } P A as Ps' a)$

and $Cs=\lambda a. \text{spec.toConcurrent } a (Ps' a) \sqcap \text{spec.post } \langle Q a \rangle$]])

apply ($\text{simp only: inf.bounded-iff}$)

apply (intro conjI)

— the meat of $\text{refinement.spec.env-hyp}$

apply ($\text{simp only: heyting}$)

apply (rule INFI)

apply ($\text{simp only: inf.bounded-iff flip: INF-inf-distrib}$)

apply (intro conjI)

```

apply (force simp: ac-simps spec.pre.INF
        intro: le-infI1 le-infI2)
apply (simp add: refinement.spec.env-hyp-def ac-simps
        flip: spec.map-invmap.galois)
apply (rule conjI)
apply (simp add: spec.map-invmap.galois spec.invmap.pre le-infI2
        del: Inf-apply INF-apply; fail)
apply (simp add: spec.map.mono le-infI2; fail)
apply (simp add: spec.next-imp.contains heyting spec.post.INF flip: INF-inf-distrib; fail)
apply (force simp: refinement-def)
done
qed

setup ‹Sign.parent-path›

```

12.2 A relational assume/guarantee program logic for the *(sequential, 's, 'v)* spec lattice

Here we develop an assume/guarantee story based on abstracting processes (represented as safety properties) to binary relations.

Observations:

- this can be seen as a reconstruction of the algebraic account given by [van Staden \(2015\)](#) in our setting
- we show Heyting implication suffices for relations (see *ag.refinement*)
 - the processes' agent type is required to be *sequential*
- we use predicates and not relations for pre/post assertions
 - we can use the metalanguage to do some relational reasoning; see, for example, *ag.name-pre-state*
- *Id* is the smallest significant assume and guarantee relation here; processes can always stutter any state

```

setup ‹Sign.mandatory-path ag›

```

```

abbreviation (input) assm :: 's rel  $\Rightarrow$  (sequential, 's, 'v) spec where
  assm A  $\equiv$  spec.rel ({env}  $\times$  A  $\cup$  {self}  $\times$  UNIV)

```

```

abbreviation (input) guar :: 's rel  $\Rightarrow$  (sequential, 's, 'v) spec where
  guar G  $\equiv$  spec.rel ({env}  $\times$  UNIV  $\cup$  {self}  $\times$  G)

```

```

setup ‹Sign.parent-path›

```

```

definition ag :: 's pred  $\Rightarrow$  's rel  $\Rightarrow$  's rel  $\Rightarrow$  ('v  $\Rightarrow$  's pred)  $\Rightarrow$  (sequential, 's, 'v) spec
  (‹{·}, -/  $\vdash$  -, {·}› [0,0,0,0] 100) where

```

```

  {P}, A  $\vdash$  G, {Q} = spec.pre P  $\sqcap$  ag.assm A  $\longrightarrow_H$  ag.guar G  $\sqcap$  spec.post Q

```

```

setup ‹Sign.mandatory-path spec.invmap›

```

```

lemma ag: — Note af = id

```

```

  fixes sf :: 's  $\Rightarrow$  't

```

```

  fixes vf :: 'v  $\Rightarrow$  'w

```

```

  fixes A :: 't rel

```

```

  fixes G :: 't rel

```

```

  fixes P :: 't pred

```

```

  fixes Q :: 'w  $\Rightarrow$  't pred

```

```

  shows spec.invmap id sf vf ({P}, A  $\vdash$  G, {Q}) = { $\lambda$ s. P (sf s)}, inv-image (A=) sf  $\vdash$  inv-image (G=) sf, { $\lambda$ v
  s. Q (vf v) (sf s)}

```

```

proof —

```

have $\{self\} \times UNIV \cup (\{env\} \times inv\text{-image } A \text{ sf} \cup UNIV \times inv\text{-image } Id \text{ sf}) = \{self\} \times UNIV \cup \{env\} \times inv\text{-image } (A^=) \text{ sf}$
and $\{env\} \times UNIV \cup (\{self\} \times inv\text{-image } G \text{ sf} \cup UNIV \times inv\text{-image } Id \text{ sf}) = \{env\} \times UNIV \cup \{self\} \times inv\text{-image } (G^=) \text{ sf}$
by *auto*
then show *?thesis*
by (*simp add: ag-def spec.invmap.heyting spec.invmap.inf spec.invmap.rel spec.invmap.pre spec.invmap.post ac-simps map-prod-vimage-Times Sigma-Un-distrib2 flip: inv-image-alt-def*)

qed

setup $\langle Sign.parent\text{-path} \rangle$

setup $\langle Sign.mandatory\text{-path } ag \rangle$

lemma *refinement*:

shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, ag.assm A \Vdash ag.guar G, \{\!|Q|\!\}$
proof –
have *constrains-heyting-ag*: $ag.assm A \longrightarrow_+ ag.guar G = ag.assm A \longrightarrow_H ag.guar G$
apply (*rule antisym[OF spec.next-imp.heyting-le]*)
apply (*simp add: spec.next-imp.heyting heyting*)
apply (*subst inf.commute*)
apply (*rule spec.composition-principle-half[where a1={self} and a2={env}]; force simp: spec.idle-le spec.term.closed.rel*)
done
have *constrains-heyting-post*: $P \longrightarrow_+ spec.post Q = P \longrightarrow_H spec.post Q$
if $P \in spec.term.closed$ –
for $P :: (sequential, -, -) spec$
apply (*rule antisym[OF spec.next-imp.heyting-le]*)
apply (*clarsimp simp: spec.next-imp.heyting*)
apply (*metis spec.term.all.closed-conv[OF that] heyting.topL order-refl spec.term.all.post spec.term.all-none spec.term.heyting-noneL-allR-mono spec.term.lower-upper-lower(2)*)
done
show *?thesis*
by (*simp add: ag-def refinement-def spec.pre.next-imp-eq-heyting spec.idle-le constrains-heyting-ag spec.next-imp.infR spec.term.closed.rel constrains-heyting-post[OF spec.term.closed.rel] heyting.infR heyting.curry-conv*)

qed

lemma *E*:

assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
obtains $c \leq spec.pre P \sqcap ag.assm A \longrightarrow_H ag.guar G$
and $c \leq spec.pre P \sqcap ag.assm A \longrightarrow_H spec.post Q$
using *assms* **by** (*simp add: ag-def heyting.infR*)

lemma *pre-post-cong*:

assumes $P = P'$
assumes $Q = Q'$
shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$
using *assms* **by** *simp*

lemma *pre-bot*:

shows $\{\!\perp\!\}, A \vdash G, \{\!|Q|\!\} = \top$
and $\{\!\langle\perp\rangle\!\}, A \vdash G, \{\!|Q|\!\} = \top$
and $\{\!\langle False\rangle\!\}, A \vdash G, \{\!|Q|\!\} = \top$
by (*simp-all add: ag-def heyting.botL*)

lemma *post-top*:

shows $\{\!|P|\!\}, A \vdash UNIV, \{\!\top\!\} = \top$
and $\{\!|P|\!\}, A \vdash UNIV, \{\!\langle\top\rangle\!\} = \top$
and $\{\!|P|\!\}, A \vdash UNIV, \{\!\lambda-. True\!\} = \top$

by (*simp-all add: ag-def spec.rel.upper-top flip: Sigma-Un-distrib1*)

lemma *mono*:

assumes $P' \leq P$
assumes $A' \leq A$
assumes $G \leq G'$
assumes $Q \leq Q'$
shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\}$

unfolding *ag-def*

apply (*strengthen ord-to-strengthen(1)[OF assms(1)]*)

apply (*strengthen ord-to-strengthen(1)[OF assms(2)]*)

apply (*strengthen ord-to-strengthen(1)[OF assms(3)]*)

apply (*strengthen ord-to-strengthen(1)[OF assms(4)]*)

apply *simp*

done

lemma *strengthen[strg]*:

assumes *st-ord* $(\neg F) P P'$
assumes *st-ord* $(\neg F) A A'$
assumes *st-ord* $F G G'$
assumes *st-ord* $F Q Q'$
shows *st-ord* $F (\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}) (\{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\})$

using *assms* **by** (*cases F; simp add: ag.mono*)

lemma *strengthen-pre*:

assumes *st-ord* $(\neg F) P P'$
shows *st-ord* $F (\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}) (\{\!|P'|\!\}, A' \vdash G, \{\!|Q|\!\})$

using *assms* **by** (*cases F; simp add: ag.mono*)

lemmas *pre-ag* = *order.trans[OF - ag.mono[OF order.refl - - order.refl], of c]* **for** *c*

lemmas *pre-a* = *ag.pre-ag[OF - - order.refl]*

lemmas *pre-g* = *ag.pre-ag[OF - order.refl]*

lemma *pre*:

assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $\bigwedge s. P' s \implies P s$
assumes $A' \subseteq A$
assumes $G \subseteq G'$
assumes $\bigwedge v s. Q v s \implies Q' v s$
shows $c \leq \{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\}$

using *assms(2-)* **by** (*blast intro: order.trans[OF assms(1) ag.mono]*)

lemmas *pre-pre-post* = *ag.pre[OF - - order.refl order.refl, of c]* **for** *c*

lemma *pre-imp*:

assumes $\bigwedge s. P s \implies P' s$
assumes $c \leq \{\!|P'|\!\}, A \vdash G, \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$

using *assms ag.pre* **by** *blast*

lemmas *pre-pre* = *ag.pre-imp[rotated]*

lemma *post-imp*:

assumes $\bigwedge v s. Q v s \implies Q' v s$

assumes $c \leq \{P\}, A \vdash G, \{Q\}$
shows $c \leq \{P\}, A \vdash G, \{Q'\}$
using *assms ag.pre* **by** *blast*

lemmas *pre-post = ag.post-imp[rotated]*
lemmas *strengthen-post = ag.pre-post*

lemmas *reflcl-ag = spec.invmap.ag[where sf=id and vf=id, simplified spec.invmap.id, simplified]*

lemma
shows *reflcl-a*: $\{P\}, A \vdash G, \{Q\} = \{P\}, A^\perp \vdash G, \{Q\}$
and *reflcl-g*: $\{P\}, A \vdash G, \{Q\} = \{P\}, A \vdash G^\perp, \{Q\}$
by (*metis ag.reflcl-ag sup.left-idem sup-commute*)+

lemma *gen-asm-base*:
assumes $P \implies c \leq \{P' \wedge P''\}, A \vdash G, \{Q\}$
shows $c \leq \{P' \wedge \langle P \rangle \wedge P''\}, A \vdash G, \{Q\}$
using *assms* **by** (*simp add: ag-def spec.pre.conj spec.pre.K heyting.botL*)

lemmas *gen-asm =*
ag.gen-asm-base[where P'= $\langle True \rangle$ and P''= $\langle True \rangle$, simplified]
ag.gen-asm-base[where P'= $\langle True \rangle$, simplified]
ag.gen-asm-base[where P''= $\langle True \rangle$, simplified]
ag.gen-asm-base

lemma *post-conj*:
assumes $c \leq \{P\}, A \vdash G, \{Q\}$
assumes $c \leq \{P\}, A \vdash G, \{Q'\}$
shows $c \leq \{P\}, A \vdash G, \{\lambda v. Q v \wedge Q' v\}$
using *assms* **by** (*simp add: ag-def spec.post.conj heyting*)

lemma *pre-disj*:
assumes $c \leq \{P\}, A \vdash G, \{Q\}$
assumes $c \leq \{P'\}, A \vdash G, \{Q\}$
shows $c \leq \{P \vee P'\}, A \vdash G, \{Q\}$
using *assms* **by** (*simp add: ag-def spec.pre.disj inf-sup-distrib heyting*)

lemma *drop-imp*:
assumes $c \leq \{P\}, A \vdash G, \{Q\}$
shows $c \leq \{P\}, A \vdash G, \{\lambda v. Q' v \longrightarrow Q v\}$
using *assms ag.strengthen-post* **by** *fastforce*

lemma *prop*:
shows $c \leq \{\langle P \rangle\}, A \vdash UNIV, \{\lambda v. \langle P \rangle\}$
by (*simp add: ag.gen-asm(1) ag.post-top(3)*)

lemma *name-pre-state*:
assumes $\bigwedge s. P s \implies c \leq \{(\equiv) s\}, A \vdash G, \{Q\}$
shows $c \leq \{P\}, A \vdash G, \{Q\}$
by (*metis assms ag.refinement refinement.name-pre-state spec.idle.rel-le*)

lemma *conj-lift*:
assumes $c \leq \{P\}, A \vdash G, \{Q\}$
assumes $c \leq \{P'\}, A \vdash G, \{Q'\}$
shows $c \leq \{P \wedge P'\}, A \vdash G, \{\lambda v. Q v \wedge Q' v\}$
using *assms* **by** (*blast intro: ag.post-conj ag.pre*)

lemma *disj-lift*:

assumes $c \leq \{P\}, A \vdash G, \{Q\}$
assumes $c \leq \{P'\}, A \vdash G, \{Q'\}$
shows $c \leq \{P \vee P'\}, A \vdash G, \{\lambda v. Q v \vee Q' v\}$
using *assms* **by** (*simp add: ag-def spec.post.disj spec.pre.disj heyting inf-sup-distrib le-supI1 le-supI2*)

lemma *all-lift*:

assumes $\bigwedge x. c \leq \{P x\}, A \vdash G, \{Q x\}$
shows $c \leq \{\forall x. P x\}, A \vdash G, \{\lambda v. \forall x. Q x v\}$
using *assms*
by (*auto simp: ag-def spec.pre.All spec.post.All le-Inf-iff heyting simp flip: INF-inf-const1 INF-inf-const2*)

lemma *interference-le*:

shows $\text{spec.rel} (\{env\} \times UNIV) \leq \{P\}, A \vdash G, \{\top\}$
and $\text{spec.rel} (\{env\} \times UNIV) \leq \{P\}, A \vdash G, \{\lambda-. \top\}$
and $\text{spec.rel} (\{env\} \times UNIV) \leq \{P\}, A \vdash G, \{\lambda-. \text{True}\}$
by (*auto simp: ag-def heyting spec.term.all.rel intro: spec.rel.mono inf.coboundedI1*)

lemma *assm-heyting*:

fixes $Q :: 'v \Rightarrow 's \text{ pred}$
shows $\text{ag.assm } r \longrightarrow_H \{P\}, A \vdash G, \{Q\} = \{P\}, A \cap r \vdash G, \{Q\}$
by (*simp add: ag-def ac-simps Int-Un-distrib Times-Int-Times flip: heyting.curry-conv spec.rel.inf*)

lemma *augment-a*: — instantiate A'

assumes $c \leq \{P\}, A \vdash G, \{Q\}$
shows $c \leq \{P\}, A \cap A' \vdash G, \{Q\}$
by (*blast intro: ag.pre-a[OF assms]*)

lemma *augment-post*: — instantiate Q

assumes $c \leq \{P\}, A \vdash G, \{\lambda v. Q' v \wedge Q v\}$
shows $c \leq \{P\}, A \vdash G, \{Q'\}$
by (*blast intro: ag.pre-post[OF assms]*)

lemma *augment-post-imp*: — instantiate Q

assumes $c \leq \{P\}, A \vdash G, \{\lambda v. (Q v \longrightarrow Q' v) \wedge Q v\}$
shows $c \leq \{P\}, A \vdash G, \{Q'\}$
by (*blast intro: ag.pre-post[OF assms]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.term.none} \rangle$

lemma *ag-le*:

shows $\text{spec.term.none} (\{P\}, A \vdash G, \{Q\}) \leq \{P\}, A \vdash G, \{\perp\}$
by (*simp add: ag.refinement spec.term.all.rel order.trans[OF spec.term.none.refinement-le]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path ag.spec.term} \rangle$

lemmas *none-inteference* =

$\text{order.trans}[OF \text{spec.term.none.mono},$
 $OF \text{ag.interference-le}(1) \text{ ag.pre-post}[\text{where } Q'=Q \text{ for } Q, OF \text{spec.term.none.ag-le, simplified}]$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path ag.spec} \rangle$

lemma *bind*:

assumes $g: \bigwedge v. g v \leq \{\!\{Q' v\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

assumes $f: f \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q'\}\!\}$

shows $f \ggg g \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

apply (*subst ag.refinement*)

apply (*rule refinement.spec.bind[where $f'=ag.guar G$ and $g'=\langle ag.guar G \rangle$, simplified spec.rel.wind-bind]*)

apply (*rule refinement.pre-a[OF $g[\text{unfolded ag.refinement}]$]*)

apply (*simp-all add: refinement.spec.bind.res.rel-le spec.term.all.rel f[unfolded ag.refinement]*)

done

lemma *action*:

fixes $F :: ('v \times \text{sequential} \times 's \times 's)$ set

assumes $Q: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \implies Q v s'$

assumes $G: \bigwedge v s s'. \llbracket P s; (v, self, s, s') \in F; s \neq s' \rrbracket \implies (s, s') \in G$

shows *spec.action* $F \leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

proof –

from G **have** $*$: *spec.pre* $P \sqcap$ *spec.action* $F \leq$ *spec.rel* ($\{\!\{env\}\!\} \times UNIV \cup \{\!\{self\}\!\} \times G$)

by – (*rule order.trans[OF spec.pre.inf-action-le(1) spec.action.rel-le]; auto*)

show *?thesis*

by (*fastforce intro: order.trans[OF - refinement.mono-stronger[OF order.refl - - order.refl]]*
refinement.spec.action Q

*simp: ag.refinement order.trans[OF *] spec.next-imp.closure.expansive spec.idle.rel-le*)

qed

lemma *return*:

shows *spec.return* $v \leq \{\!\{Q v\}\!\}, A \vdash G, \{\!\{Q\}\!\}$

by (*auto simp: spec.return-def intro: ag.spec.action*)

lemma *Parallel-assm*:

shows *refinement.spec.env-hyp* P (*ag.assm* A) *as* (*ag.guar* $\circ G$) $a \leq$ *ag.assm* ($A \cup \bigcup (G \text{ ‘ } (as - \{a\}))$)

by (*simp add: refinement.spec.env-hyp-def spec.invmap.rel flip: spec.rel.upper-INF spec.rel.inf*)

(*force intro!: le-infI2 order.trans[OF spec.map.rel-le] spec.rel.mono-reflcl*)

lemma *Parallel-guar*:

shows *spec.Parallel* *as* (*ag.guar* $\circ G$) = *ag.guar* ($\bigcup a \in as. G a$)

proof –

have $*$: $\{\!\{self\}\!\} \times Id \cup (\text{insert } env ((\lambda x. self) \text{ ‘ } as) \times Id \cup \text{map-prod toSequential-fn id ‘ } (\text{insert } env (proc \text{ ‘ } as)$
 $\times UNIV \cap (\bigcap x \in as. \{\!\{proc x\}\!\} \times G x \cup (- \{\!\{proc x\}\!\} \times UNIV)))$

= $\{\!\{env\}\!\} \times UNIV \cup (\{\!\{self\}\!\} \times Id \cup \{\!\{self\}\!\} \times \bigcup (G \text{ ‘ } as))$

by (*rule antisym, force simp: toSequential-fn-def, (safe; force simp: map-prod-conv)*)

show *?thesis*

apply (*simp add: spec.Parallel-def spec.invmap.rel*

flip: spec.rel.INF spec.rel.inf)

apply (*subst spec.map.rel*)

apply (*clarsimp; blast*)

apply (*subst (1 2) spec.rel.reflcl[where $A=\{\!\{self\}\!\}$, symmetric]*)

apply (*clarsimp simp: ac-simps inf-sup-distrib image-Un image-image **

map-prod-image-Times map-prod-vimage-Times Times-Int-Times)

done

qed

lemma *Parallel*:

fixes $A :: 's$ rel

fixes $G :: 'a \Rightarrow 's$ rel

fixes $Q :: 'a \Rightarrow 's \Rightarrow \text{bool}$

fixes $Ps :: 'a \Rightarrow (\text{sequential}, 's, \text{unit})$ spec

assumes *proc-ag*: $\bigwedge a. a \in as \implies Ps a \leq \{\!\{P a\}\!\}, A \cup (\bigcup a' \in as - \{a\}. G a') \vdash G a, \{\!\{\lambda v. Q a\}\!\}$

shows *spec.Parallel* *as* $Ps \leq \{\!\{\bigcap a \in as. P a\}\!\}, A \vdash \bigcup a \in as. G a, \{\!\{\lambda rv. \bigcap a \in as. Q a\}\!\}$

```

unfolding ag.refinement
apply (rule order.trans[OF - refinement.mono[OF order.refl - - order.refl]])
  apply (rule refinement.spec.Parallel[where  $A=ag.assm\ A$  and  $P_s'=ag.guar\ \circ\ G$ ])
  apply (rule order.trans[OF proc-ag, unfolded ag.refinement], assumption)
  apply (rule refinement.mono[OF order.refl - - order.refl])
  apply (force intro: ag.spec.Parallel-assm simp: ag.spec.Parallel-guar) +
done

```

```

setup ⟨Sign.parent-path⟩

```

12.2.1 Stability rules

```

setup ⟨Sign.mandatory-path spec⟩

```

lemma *stable-pre-post*:

```

fixes  $S :: ('a, 's, 'v)\ spec$ 
assumes stable (snd ' r) P
assumes spec.steps  $S \subseteq r$ 
shows  $S \leq spec.pre\ P \longrightarrow_H spec.post\ \langle P \rangle$ 

```

proof –

```

have  $P\ (trace.final'\ s\ xs)$ 
  if  $P\ s$ 
and trace.steps (trace.T s xs v)  $\subseteq r$ 
for  $s :: 's$  and  $xs :: ('a \times 's)\ list$  and  $v :: 'v\ option$ 
using that

```

proof(*induct xs arbitrary: s*)

```

case (Cons x xs) with ⟨stable (snd ' r) P⟩ show ?case
  by (cases x; cases snd x = s;
    force simp: stable-def monotone-def dest: le-boolD)

```

qed *simp*

from *this* ⟨*spec.steps* $S \subseteq r$ ⟩ **show** ?*thesis*

```

by – (rule spec.singleton-le-extI;
  auto dest: order.trans[where  $b=S$ ]
  simp: spec.singleton.le-conv spec.singleton.heyting-le-conv trace.split-all spec.rel.galois
  split: option.split)

```

qed

lemma *pre-post-stable*:

```

fixes  $P :: 's \Rightarrow bool$ 
assumes stable (snd ' r) P
shows spec.rel  $r \leq spec.pre\ P \longrightarrow_H spec.post\ \langle P \rangle$ 

```

by (rule *spec.stable-pre-post*[*OF assms spec.rel.lower-upper-contractive*])

```

setup ⟨Sign.parent-path⟩

```

```

setup ⟨Sign.mandatory-path ag⟩

```

lemma *stable-lift*:

```

assumes stable ( $A \cup G$ ) P' — anything stable over  $A \cup G$  is invariant
shows  $\{\{P \wedge P'\}, A \vdash G, \{\lambda v. P' \longrightarrow Q\ v\} \leq \{\{P \wedge P'\}, A \vdash G, \{\lambda v. Q\ v \wedge P'\}$ 

```

apply (*simp add: ag-def spec.pre.conj heyting heyting.detachment le-infI2 flip: spec.heyting.post*)

apply (*simp add: ac-simps Sigma-Un-distrib2 Int-Un-distrib Times-Int-Times flip: spec.rel.inf*)

apply (rule *order.subgoal*)

apply (rule *order.trans*[*OF - spec.pre-post-stable*[**where** $r=\{env\} \times A \cup \{self\} \times G$, *simplified image-Un, simplified, OF assms*]])

apply (*simp add: le-infI2; fail*)

apply (*simp add: ac-simps spec.post.conj*)

apply (*simp add: heyting.discharge le-infI1 flip: inf.assoc*)

done

lemma *stable-augment-base*:

assumes $c \leq \{\!\{P \wedge P'\}\!\}$, $A \vdash G$, $\{\!\{\lambda v. P' \longrightarrow Q v\}\!\}$

assumes *stable* $(A \cup G) P'$ — anything stable over $A \cup G$ is invariant

shows $c \leq \{\!\{P \wedge P'\}\!\}$, $A \vdash G$, $\{\!\{\lambda v. Q v \wedge P'\}\!\}$

using *order.trans*[*OF - ag.stable-lift*] *assms* **by** *blast*

lemma *stable-augment*:

assumes $c \leq \{\!\{P'\}\!\}$, $A \vdash G$, $\{\!\{Q'\}\!\}$

assumes $\bigwedge v s. \llbracket P s; Q' v s \rrbracket \implies Q v s$

assumes *stable* $(A \cup G) P$

shows $c \leq \{\!\{P' \wedge P\}\!\}$, $A \vdash G$, $\{\!\{Q\}\!\}$

apply (*rule ag.strengthen-post*)

apply (*rule ag.stable-augment-base*[**where** $Q=Q$, *OF - assms*(3)])

apply (*auto intro: assms*(2) *ag.pre*[*OF assms*(1)])

done

lemma *stable-augment-post*:

assumes $c \leq \{\!\{P'\}\!\}$, $A \vdash G$, $\{\!\{Q'\}\!\}$ — resolve before application

assumes $\bigwedge v. \textit{stable} (A \cup G) (Q' v \longrightarrow Q v)$

shows $c \leq \{\!\{(\forall v. Q' v \longrightarrow Q v) \wedge P'\}\!\}$, $A \vdash G$, $\{\!\{Q\}\!\}$

apply (*rule ag.pre-pre-post*)

apply (*rule ag.stable-augment-base*[**where** $P=P'$ **and** $Q=Q'$ **and** $P'=(\forall v. Q' v \longrightarrow Q v)$])

apply (*rule ag.pre-pre-post*[*OF assms*(1)])

using *assms*(2) **apply** (*fast intro: stable.allI*)+

done

lemma *stable-augment-frame*: — anything stable over $A \cup G$ is invariant

assumes $c \leq \{\!\{P\}\!\}$, $A \vdash G$, $\{\!\{Q\}\!\}$

assumes *stable* $(A \cup G) P'$

shows $c \leq \{\!\{P \wedge P'\}\!\}$, $A \vdash G$, $\{\!\{\lambda v. Q v \wedge P'\}\!\}$

using *assms* **by** (*blast intro: ag.stable-augment*[*OF assms*(1)])

setup $\langle \textit{Sign.parent-path} \rangle$

setup $\langle \textit{Sign.mandatory-path ag.spec} \rangle$

lemma *stable-interference*:

assumes *stable* $(A \cap r) P$

shows *spec.rel* $(\{\!\{env\}\!\} \times r) \leq \{\!\{P\}\!\}$, $A \vdash G$, $\{\!\{\langle P \rangle\}\!\}$

using *assms*

by (*auto simp: ag-def ac-simps heyting Int-Un-distrib Times-Int-Times*

simp flip: spec.rel.inf

intro: le-infI2 spec.rel.mono spec.pre-post-stable[*simplified heyting ac-simps*])

setup $\langle \textit{Sign.parent-path} \rangle$

setup $\langle \textit{Sign.mandatory-path spec} \rangle$

setup $\langle \textit{Sign.mandatory-path cam} \rangle$

lemma *closed-ag*:

shows $\{\!\{P\}\!\}$, $A \vdash G$, $\{\!\{Q\}\!\} \in \textit{spec.cam.closed} (\{\!\{env\}\!\} \times r)$

unfolding *ag-def heyting.infR*

by (*blast intro: subsetD*[*OF spec.cam.closed.antimono, rotated*])

setup $\langle \textit{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path interference} \rangle$

lemma *cl-ag-le*:

assumes P : *stable* $(A \cap r)$ P

assumes Q : $\bigwedge v$. *stable* $(A \cap r)$ $(Q v)$

shows *spec.interference.cl* $(\{env\} \times r)$ $(\{P\}, A \vdash G, \{Q\}) \leq \{P\}, A \vdash G, \{Q\}$

unfolding *spec.interference.cl-def*

by (*rule ag.spec.bind ag.spec.return ag.spec.stable-interference spec.cam.least[OF - spec.cam.closed-ag] assms order.refl*) $+$

lemma *closed-ag*:

assumes P : *stable* $(A \cap r)$ P

assumes Q : $\bigwedge v$. *stable* $(A \cap r)$ $(Q v)$

shows $\{P\}, A \vdash G, \{Q\} \in \text{spec.interference.closed} (\{env\} \times r)$

by (*rule spec.interference.closed-clI[OF spec.interference.cl-ag-le[OF assms]]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

13 A programming language

The $(\prime a, \prime s, \prime v)$ *spec* lattice of §8.2 is adequate for logic but is deficient as a programming language. In particular we wish to interpret the parallel composition as intersection (§9.5) which requires processes to contain enough interference opportunities. Similarly we want the customary “laws of programming” (Hoare, Hayes, He, Morgan, Roscoe, Sanders, Sørensen, Spivey, and Sufrin 1987a) to hold without side conditions.

These points are discussed at some length by Zwiers (1989, §3.2) and also Foster, Baxter, Cavalcanti, Woodcock, and Zeyda (2020, Lemma 6.7).

Our $(\prime v, \prime s)$ *prog* lattice (§13.1) therefore handles the common case of the familiar constructs for sequential programming, and we lean on our $(\prime a, \prime s, \prime v)$ *spec* lattice for other constructions such as interleaving parallel composition (§9.5) and local state (§15). It allows arbitrary interference by the environment before and after every program action.

13.1 The $(\prime s, \prime v)$ *prog* lattice

According to Müller-Olm (1997, §2.1), $(\prime s, \prime v)$ *prog* is a *sub-lattice* of $(\prime a, \prime s, \prime v)$ *spec* as the corresponding (\sqcap) and (\sqcup) operations coincide. However it is not a *complete* sub-lattice as *Sup* in $(\prime s, \prime v)$ *prog* needs to account for the higher bottom of that lattice.

typedef $(\prime s, \prime v)$ *prog* = *spec.interference.closed* $(\{env\} \times UNIV)$:: $(\text{sequential}, \prime s, \prime v)$ *spec set*

morphisms $p2s$ *Abs-t*

by *blast*

hide-const (**open**) $p2s$

setup-lifting *type-definition-prog*

instantiation *prog* :: $(\text{type}, \text{type})$ *complete-distrib-lattice*

begin

lift-definition *bot-prog* :: $(\prime s, \prime v)$ *prog* **is** *spec.interference.cl* $(\{env\} \times UNIV) \perp ..$

lift-definition *top-prog* :: $(\prime s, \prime v)$ *prog* **is** $\top ..$

lift-definition *sup-prog* :: $(\prime s, \prime v)$ *prog* $\Rightarrow (\prime s, \prime v)$ *prog* $\Rightarrow (\prime s, \prime v)$ *prog* **is** *sup* ..

lift-definition *inf-prog* :: $(\prime s, \prime v)$ *prog* $\Rightarrow (\prime s, \prime v)$ *prog* $\Rightarrow (\prime s, \prime v)$ *prog* **is** *inf* ..

lift-definition *less-eq-prog* :: $(\prime s, \prime v)$ *prog* $\Rightarrow (\prime s, \prime v)$ *prog* $\Rightarrow \text{bool}$ **is** *less-eq* .

lift-definition *less-prog* :: $(\prime s, \prime v)$ *prog* $\Rightarrow (\prime s, \prime v)$ *prog* $\Rightarrow \text{bool}$ **is** *less* .

lift-definition *Inf-prog* :: ('s, 'v) prog set \Rightarrow ('s, 'v) prog is *Inf* ..

lift-definition *Sup-prog* :: ('s, 'v) prog set \Rightarrow ('s, 'v) prog is $\lambda X. \text{Sup } X \sqcup \text{spec.interference.cl } (\{\text{env}\} \times \text{UNIV})$
 \perp ..

instance

by *standard* (*transfer*; *auto simp: Inf-lower InfI SupI le-supI1 spec.interference.least*) $+$

end

13.2 Morphisms to and from the (*sequential, 's, 'v*) *spec* lattice

We can readily convert a ('s, 'v) prog into a ('a agent, 's, 'v) spec. More interestingly, on ('s, 'v) prog we have a Galois connection that embeds specifications into programs. (This connection is termed a *Galois insertion* by Melton et al. (1985) as we also have *prog.s2p.p2s*; Cousot says “Galois retraction”.)

See also §13.4.2 and §13.5.1.

setup $\langle \text{Sign.mandatory-path spec.interference.closed} \rangle$

lemmas *p2s[iff]* = *prog.p2s*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.interference.cl} \rangle$

lemmas *p2s* = *spec.interference.closed-conv*[*OF spec.interference.closed.p2s, symmetric, of P for P*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.idle} \rangle$

lemmas *p2s-le*[*spec.idle-le*]

= *spec.interference.le-closedE*[*OF spec.idle.interference.cl-le spec.interference.closed.p2s, of P for P*]

lemmas *p2s-minimal[iff]* = *order.trans*[*OF spec.idle.minimal-le spec.idle.p2s-le*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

lemma *p2s-leI*:

assumes *prog.p2s* *c* \leq *prog.p2s* *d*

shows *c* \leq *d*

by (*simp add: assms less-eq-prog.rep-eq*)

setup $\langle \text{Sign.mandatory-path p2s} \rangle$

named-theorems *simps* $\langle \text{simp rules for const } \langle \text{p2s} \rangle \rangle$

lemmas *bot* = *bot-prog.rep-eq*

lemmas *top* = *top-prog.rep-eq*

lemmas *inf* = *inf-prog.rep-eq*

lemmas *sup* = *sup-prog.rep-eq*

lemmas *Inf* = *Inf-prog.rep-eq*

lemmas *Sup* = *Sup-prog.rep-eq*

lemma *Sup-not-empty*:

assumes *X* \neq {}

shows *prog.p2s* (\sqcup *X*) = \sqcup (*prog.p2s* ‘ *X*)

using *assms* **by** *transfer* (*simp add: sup.absorb1 less-eq-Sup spec.interference.least*)

lemma *SUP-not-empty*:

assumes $X \neq \{\}$

shows $\text{prog.p2s} (\bigsqcup x \in X. f x) = (\bigsqcup x \in X. \text{prog.p2s} (f x))$

by (*simp add: asms prog.p2s.Sup-not-empty*[**where** $X=f ' X$, *simplified image-image*])

lemma *monotone*:

shows *mono prog.p2s*

by (*rule monoI*) (*transfer; simp*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.p2s.monotone*]

lemmas *mono* = *monotoneD*[*OF prog.p2s.monotone*]

lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF prog.p2s.monotone, simplified, of orda P for orda P*]

lemma *mcont*: — *Morally galois.complete-lattice.mcont-lower*

shows *mcont Sup* (\leq) *Sup* (\leq) *prog.p2s*

by (*simp add: contI mcontI prog.p2s.Sup-not-empty*)

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.p2s.mcont, of luba orda P for luba orda P*]

lemmas *Let-distrib* = *Let-distrib*[**where** $f=\text{prog.p2s}$]

lemmas [*prog.p2s.simps*] =

prog.p2s.bot

prog.p2s.top

prog.p2s.inf

prog.p2s.sup

prog.p2s.Inf

prog.p2s.Sup-not-empty

spec.interference.cl.p2s

prog.p2s.Let-distrib

lemma *interference-wind-bind*:

shows *spec.rel* ($\{\text{env}\} \times UNIV$) $\gg=$ ($\lambda::\text{unit. prog.p2s } P$) = *prog.p2s* P

by (*subst* ($1\ 2$) *spec.interference.closed-conv*[*OF prog.p2s*])

(*simp add: spec.interference.cl-def spec.rel.wind-bind flip: spec.bind.bind*)

setup $\langle \text{Sign.parent-path} \rangle$

definition *s2p* :: (*sequential, 's, 'v*) *spec* \Rightarrow (*'s, 'v*) *prog* **where** — *Morally the upper of a Galois connection*

$s2p\ P = \bigsqcup \{c. \text{prog.p2s } c \leq P\}$

setup $\langle \text{Sign.mandatory-path } s2p \rangle$

lemma *bottom*:

shows *prog.s2p* $\perp = \perp$

by (*simp add: prog.s2p-def bot.extremum-uniqueI less-eq-prog.rep-eq*)

lemma *top*:

shows *prog.s2p* $\top = \top$

by (*simp add: prog.s2p-def*)

lemma *monotone*:

shows *mono prog.s2p*

by (*fastforce simp: prog.s2p-def intro: monotoneI Sup-subset-mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.s2p.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.s2p.monotone*]
lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF prog.s2p.monotone, simplified*]

lemma *p2s*:

shows *prog.s2p* (*prog.p2s* *P*) = *P*

by (*auto simp: prog.s2p-def simp flip: less-eq-prog.rep-eq intro: Sup-eqI*)

lemma *Sup-le*:

shows $\sqcup(\text{prog.s2p } 'X) \leq \text{prog.s2p } (\sqcup X)$

by (*simp add: prog.s2p-def Collect-mono SUPE Sup-subset-mono Sup-upper2*)

lemma *sup-le*:

shows *prog.s2p* *x* \sqcup *prog.s2p* *y* \leq *prog.s2p* (*x* \sqcup *y*)

using *prog.s2p.Sup-le*[**where** *X*={*x, y*}] **by** *simp*

lemma *Inf*:

shows *prog.s2p* ($\sqcap X$) = $\sqcap(\text{prog.s2p } 'X)$ (**is** *?lhs* = *?rhs*)

proof(*rule antisym*)

show *?lhs* \leq *?rhs*

by (*simp add: prog.s2p-def SupI Sup-le-iff le-Inf-iff*)

show *?rhs* \leq *?lhs*

by (*fastforce simp: prog.s2p-def prog.p2s.mono*

Inf-Sup[**where** *A*=($\lambda x. \{c. \text{prog.p2s } c \leq x\}$) 'X, *simplified image-image*]

le-Inf-iff INF-lower2

elim: order.trans[*rotated*]

intro: Sup-subset-mono)

qed

lemma *inf*:

shows *prog.s2p* (*x* \sqcap *y*) = *prog.s2p* *x* \sqcap *prog.s2p* *y*

using *prog.s2p.Inf*[**where** *X*={*x, y*}] **by** *simp*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path } p2s\text{-}s2p \rangle$

lemma *galois*: — the Galois connection

shows *prog.p2s* *c* \leq *S*

$\longleftrightarrow c \leq \text{prog.s2p } S \wedge \text{spec.term.none } (\text{spec.rel } (\{\text{env}\} \times \text{UNIV}) :: (-, -, \text{unit}) \text{spec}) \leq S$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

by (*metis order.trans prog.s2p.mono prog.s2p.p2s spec.interference.closed.p2s*

spec.term.none.interference.closed.rel-le)

show *?rhs* \implies *?lhs*

unfolding *prog.s2p-def* **by** *transfer* (*force simp: spec.interference.cl.bot elim: order.trans*)

qed

lemma *le*:

shows *prog.p2s* (*prog.s2p* *S*) \leq *spec.interference.cl* ($\{\text{env}\} \times \text{UNIV}$) *S*

by (*metis bot-prog.rep-eq prog.p2s-s2p.galois prog.s2p.mono spec.interference.cl-bot-least spec.interference.expansive*)

lemma *insertion*:

fixes *S* :: (*sequential, 's, 'v*) *spec*

assumes *S* \in *spec.interference.closed* ($\{\text{env}\} \times \text{UNIV}$)

shows *prog.p2s* (*prog.s2p* *S*) = *S*

by (*metis* *assms* *prog.p2s-cases* *prog.s2p.p2s*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

13.3 Programming language constructs

We lift the combinators directly from the $(\prime a, \prime s, \prime v)$ *spec* lattice (§8), but need to interference-close primitive actions. Control flow is expressed via HOL's *if-then-else* construct and other case combinators where the scrutinee is a pure value. This means that the atomicity of a process is completely determined by occurrences of *prog.action*.

setup $\langle \text{Sign.mandatory-path } \text{prog} \rangle$

lift-definition $\text{bind} :: (\prime s, \prime v) \text{ prog} \Rightarrow (\prime v \Rightarrow (\prime s, \prime w) \text{ prog}) \Rightarrow (\prime s, \prime w) \text{ prog}$ is
spec.bind ..

ad hoc-overloading

Monad-Syntax.bind \equiv *prog.bind*

lift-definition $\text{action} :: (\prime v \times \prime s \times \prime s) \text{ set} \Rightarrow (\prime s, \prime v) \text{ prog}$ is
 $\lambda F. \text{spec.interference.cl } (\{\text{env}\} \times \text{UNIV}) (\text{spec.action } (\text{map-prod id } (\text{Pair self}) \prime F))$..

abbreviation (*input*) $\text{det-action} :: (\prime s \Rightarrow (\prime v \times \prime s)) \Rightarrow (\prime s, \prime v) \text{ prog}$ where
 $\text{det-action } f \equiv \text{prog.action } \{(v, s, s^\wedge). (v, s^\wedge) = f s\}$

definition $\text{return} :: \prime v \Rightarrow (\prime s, \prime v) \text{ prog}$ where
 $\text{return } v = \text{prog.action } (\{v\} \times \text{Id})$

definition $\text{guard} :: \prime s \text{ pred} \Rightarrow (\prime s, \text{unit}) \text{ prog}$ where
 $\text{guard } g \equiv \text{prog.action } (\{()\} \times \text{Diag } g)$

abbreviation (*input*) $\text{read} :: (\prime s \Rightarrow \prime v) \Rightarrow (\prime s, \prime v) \text{ prog}$ where
 $\text{read } F \equiv \text{prog.action } \{(F s, s, s) \mid s. \text{True}\}$

abbreviation (*input*) $\text{write} :: (\prime s \Rightarrow \prime s) \Rightarrow (\prime s, \text{unit}) \text{ prog}$ where
 $\text{write } F \equiv \text{prog.action } \{((), s, F s) \mid s. \text{True}\}$

lift-definition $\text{Parallel} :: \prime a \text{ set} \Rightarrow (\prime a \Rightarrow (\prime s, \text{unit}) \text{ prog}) \Rightarrow (\prime s, \text{unit}) \text{ prog}$ is *spec.Parallel*
by (*rule spec.interference.closed.Parallel*)

lift-definition $\text{parallel} :: (\prime s, \text{unit}) \text{ prog} \Rightarrow (\prime s, \text{unit}) \text{ prog} \Rightarrow (\prime s, \text{unit}) \text{ prog}$ is *spec.parallel*
by (*simp add: spec.parallel-def spec.interference.closed.Parallel*)

lift-definition $\text{vmap} :: (\prime v \Rightarrow \prime w) \Rightarrow (\prime s, \prime v) \text{ prog} \Rightarrow (\prime s, \prime w) \text{ prog}$ is *spec.vmap*
by (*rule subsetD[OF spec.interference.closed.antimono spec.interference.closed.map-sf-id, rotated]*)
auto

ad hoc-overloading

Parallel \equiv *prog.Parallel*

ad hoc-overloading

parallel \equiv *prog.parallel*

lemma *return-alt-def*:

shows $\text{prog.return } v = \text{prog.read } \langle v \rangle$

by (*auto simp: prog.return-def intro: arg-cong[where f=prog.action]*)

lemma *parallel-alt-def*:

shows $prog.parallel\ P\ Q = prog.Parallel\ UNIV\ (\lambda a::bool.\ if\ a\ then\ P\ else\ Q)$
by *transfer (simp add: spec.parallel-def)*

lift-definition $rel :: 's\ rel \Rightarrow ('s,\ 'v)\ prog\ is\ \lambda r.\ spec.rel\ (\{env\} \times UNIV \cup \{self\} \times r)$
by *(simp add: spec.interference.closed.rel)*

lift-definition $steps :: ('s,\ 'v)\ prog \Rightarrow 's\ rel\ is\ \lambda P.\ spec.steps\ P\ \{\{self\}\} .$

lift-definition $invmap :: ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('t,\ 'w)\ prog \Rightarrow ('s,\ 'v)\ prog\ is$
spec.invmap id
by *(rule subsetD[OF spec.interference.closed.antimono spec.interference.closed.invmap, rotated])*
auto

abbreviation $sinvmap :: ('s \Rightarrow 't) \Rightarrow ('t,\ 'v)\ prog \Rightarrow ('s,\ 'v)\ prog\ where$

sinvmap sf \equiv prog.invmap sf id

abbreviation $vinvmap :: ('v \Rightarrow 'w) \Rightarrow ('s,\ 'w)\ prog \Rightarrow ('s,\ 'v)\ prog\ where$

vinvmap vf \equiv prog.invmap id vf

declare *prog.bind-def[code del]*
declare *prog.action-def[code del]*
declare *prog.return-def[code del]*
declare *prog.Parallel-def[code del]*
declare *prog.parallel-def[code del]*
declare *prog.vmap-def[code del]*
declare *prog.rel-def[code del]*
declare *prog.steps-def[code del]*
declare *prog.invmap-def[code del]*

13.3.1 Laws of programming

setup $\langle Sign.mandatory-path\ p2s \rangle$

lemma *bind[prog.p2s.simps]:*
shows $prog.p2s\ (f \ggg g) = prog.p2s\ f \ggg (\lambda x.\ prog.p2s\ (g\ x))$
by *transfer simp*

lemmas $action = prog.action.rep-eq$

lemma *return:*
shows $prog.p2s\ (prog.return\ v) = spec.interference.cl\ (\{env\} \times UNIV)\ (spec.return\ v)$
by *(simp add: prog.return-def prog.p2s.action map-prod-image-Times Pair-image flip: spec.return-alt-def)*

lemma *guard:*
shows $prog.p2s\ (prog.guard\ g) = spec.interference.cl\ (\{env\} \times UNIV)\ (spec.guard\ g)$
by *(simp add: prog.guard-def prog.p2s.action map-prod-image-Times Pair-image flip: spec.guard.alt-def[where A={self}])*

lemmas $Parallel[prog.p2s.simps] = prog.Parallel.rep-eq[simplified,\ of\ as\ Ps\ for\ as\ Ps,\ unfolded\ comp-def]$
lemmas $parallel[prog.p2s.simps] = prog.parallel.rep-eq$
lemmas $invmap[prog.p2s.simps] = prog.invmap.rep-eq$
lemmas $rel[prog.p2s.simps] = prog.rel.rep-eq$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ return \rangle$

lemma *transfer[transfer-rule]:*

shows $rel\text{-}fun (=) cr\text{-}prog (\lambda v. spec.interference.cl (\{env\} \times UNIV) (spec.return v)) prog.return$
by (*simp add: rel-funI cr-prog-def prog.p2s.return*)

lemma *cong*:

fixes $F :: ('v \times 's \times 's) \text{ set}$

assumes $\bigwedge v s s'. (v, s, s') \in F \implies s' = s$

assumes $\bigwedge v s s' s''. v \in fst \text{ ` } F \implies (v, s, s) \in F$

shows $prog.action F = (\bigsqcup (v, s, s') \in F. prog.return v)$

using *assms*

by *transfer*

(*subst spec.return.cong*;

fastforce simp: spec.interference.cl.action spec.interference.cl.return

spec.interference.cl.Sup spec.interference.cl.sup spec.interference.cl.idle

spec.interference.cl.bot image-image split-def

intro: map-prod-imageI[where f=id, simplified])

lemma *rel-le*:

shows $prog.return v \leq prog.rel r$

by *transfer (simp add: spec.interference.least spec.interference.closed.rel spec.return.rel-le)*

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle Sign.mandatory\text{-}path \text{ action} \rangle$

lemma *empty*:

shows $prog.action \{\} = \perp$

by *transfer*

(*simp add: spec.action.empty spec.interference.cl.bot spec.interference.cl.idle*

flip: bot-fun-def spec.bind.botR)

lemma *monotone*:

shows $mono (prog.action :: - \implies ('s, 'v) prog)$

proof(*transfer, rule monotoneI*)

show $spec.interference.cl (\{env\} \times UNIV) (spec.action (map-prod id (Pair self) \text{ ` } F))$

$\leq spec.interference.cl (\{env\} \times UNIV) (spec.action (map-prod id (Pair self) \text{ ` } F'))$

if $F \subseteq F'$ **for** $F F' :: ('v \times 's \times 's) \text{ set}$

by (*strengthen ord-to-strengthen(1)[OF \langle F \subseteq F' \rangle]*) *simp*

qed

lemmas *strengthen[strg] = st-monotone[OF prog.action.monotone]*

lemmas *mono = monotoneD[OF prog.action.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF prog.action.monotone, simplified]*

lemma *Sup*:

shows $prog.action (\bigsqcup Fs) = (\bigsqcup F \in Fs. prog.action F)$

by *transfer*

(*simp add: spec.interference.cl.bot spec.interference.cl.Sup spec.interference.cl.sup*

spec.interference.cl.idle spec.interference.cl.action spec.action.Sup image-Union image-image

flip: bot-fun-def spec.bind.botR)

lemmas *sup = prog.action.Sup[where Fs={F, G} for F G, simplified]*

lemma *Inf-le*:

shows $prog.action (\bigcap Fs) \leq (\bigcap F \in Fs. prog.action F)$

apply *transfer*

apply (*strengthen ord-to-strengthen(1)[OF image-Inter-subseteq]*)

apply (*strengthen ord-to-strengthen(1)[OF spec.action.Inf-le]*)

apply (*strengthen ord-to-strengthen*(1)[*OF spec.interference.cl-Inf-le*])
apply (*blast intro: Inf-mono*)
done

lemma *inf-le*:

shows *prog.action* ($F \sqcap G$) \leq *prog.action* $F \sqcap$ *prog.action* G
using *prog.action.Inf-le*[**where** $Fs=\{F, G\}$] **by** *simp*

lemma *invmap-le*: — a strict refinement

shows *prog.p2s* (*prog.action* (*map-prod id* (*map-prod sf sf*) - ' F))
 \leq *spec.invmap sf* (*prog.p2s* (*prog.action F*))
by (*force intro: order.trans*[*OF - spec.interference.cl.mono*[*OF order.refl spec.action.invmap-le*]]
spec.interference.cl.mono spec.action.mono
simp: prog.p2s.action spec.invmap.interference.cl)

lemma *return-const*:

fixes $F :: 's \text{ rel}$
fixes $V :: 'v \text{ set}$
fixes $W :: 'w \text{ set}$
assumes $V \neq \{\}$
assumes $W \neq \{\}$
shows *prog.action* ($V \times F$) = *prog.action* ($W \times F$) \gg ($\bigsqcup v \in V. \text{prog.return } v$)
using *assms*(1)
by (*simp add: prog.p2s.simps prog.p2s.action prog.p2s.return image-image*
map-prod-image-Times spec.action.return-const[*OF assms*]
spec.bind.SUPR-not-empty spec.interference.cl.bind-return
spec.interference.cl.return spec.interference.cl-Sup-not-empty
spec.interference.closed.bind-relR
flip: prog.p2s-inject)

lemma *rel-le*:

assumes $\bigwedge v s s'. (v, s, s') \in F \implies (s, s') \in r \vee s = s'$
shows *prog.action F* \leq *prog.rel r*
by (*auto intro: order.trans*[*OF spec.interference.cl.mono*[*OF order.refl*
spec.action.rel-le[**where** $r=\{\text{self}\} \times r \cup \{\text{env}\} \times UNIV$]]]
dest: assms
simp: less-eq-prog.rep-eq prog.p2s.simps prog.p2s.action spec.interference.cl.rel ac-simps)

lemma *invmap-le*:

shows *prog.action* (*map-prod vf* (*map-prod sf sf*) - ' F) \leq *prog.invmap sf vf* (*prog.action F*)
by *transfer*
(*force simp: spec.invmap.interference.cl*
intro: spec.interference.cl.mono
spec.action.mono order.trans[*OF - spec.interference.cl.mono*[*OF order.refl spec.action.invmap-le*]])

lemma *action-le*:

shows *spec.action* (*map-prod id* (*Pair self*) - ' F) \leq *prog.p2s* (*prog.action F*)
by (*simp add: prog.p2s.action spec.interference.expansive*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemmas *if-distrL* = *if-distrib*[**where** $f=\lambda x. x \ggg g$ **for** $g :: - \Rightarrow (-, -) \text{ prog}$]

lemma *mono*:

assumes $f \leq f'$
assumes $\bigwedge x. g x \leq g' x$

shows $\text{prog.bind } f \ g \leq \text{prog.bind } f' \ g'$
using *assms* **by** *transfer* (*simp add: spec.bind.mono*)

lemma *strengthen*[*strg*]:
assumes *st-ord* $F \ f \ f'$
assumes $\bigwedge x. \text{st-ord } F \ (g \ x) \ (g' \ x)$
shows $\text{st-ord } F \ (\text{prog.bind } f \ g) \ (\text{prog.bind } f' \ g')$
using *assms* **by** (*cases F; clarsimp intro!: prog.bind.mono*)

lemma *mono2mono*[*cont-intro, partial-function-mono*]:
assumes *monotone* $\text{orda} \ (\leq) \ f$
assumes $\bigwedge x. \text{monotone } \text{orda} \ (\leq) \ (\lambda y. g \ y \ x)$
shows $\text{monotone } \text{orda} \ (\leq) \ (\lambda x. \text{prog.bind } (f \ x) \ (g \ x))$
using *assms* **by** *transfer simp*

The monad laws hold unconditionally in the $(\prime s, \prime v)$ *prog* lattice.

lemma *bind*:
shows $f \ggg g \ggg h = \text{prog.bind } f \ (\lambda x. g \ x \ggg h)$
by *transfer* (*simp add: spec.bind.bind*)

lemma *return*:
shows *returnL*: $(\ggg) \ (\text{prog.return } v) = (\lambda g :: \prime v \Rightarrow (\prime s, \prime w) \text{ prog. } g \ v)$ (**is** *?thesis1*)
and *returnR*: $f \ggg \text{prog.return } = f$ (**is** *?thesis2*)

proof –

have $\text{prog.return } v \ggg g = g \ v$ **for** $g :: \prime v \Rightarrow (\prime s, \prime w) \text{ prog}$
by *transfer*
(simp add: map-prod-image-Times Pair-image spec.action.read-agents
spec.interference.cl.return spec.bind.bind
spec.bind.returnL spec.idle-le prog.p2s-induct spec.interference.closed.bind-relL
flip: spec.return-def)

then show *?thesis1*

by (*simp add: fun-eq-iff*)

show *?thesis2*

by *transfer*

(simp add: map-prod-image-Times Pair-image spec.action.read-agents
flip: spec.interference.cl.bindL spec.return-def spec.interference.closed-conv)

qed

lemma *botL*:
shows $\text{prog.bind } \perp = \perp$
by (*simp add: fun-eq-iff prog.p2s.simps spec.interference.cl.bot*
flip: prog.p2s-inject)

lemma *botR-le*:
shows $\text{prog.bind } f \ \langle \perp \rangle \leq f$ (**is** *?thesis1*)
and $\text{prog.bind } f \ \perp \leq f$ (**is** *?thesis2*)

proof –

show *?thesis1*

by (*metis bot.extremum dual-order.refl prog.bind.mono prog.bind.returnR*)

then show *?thesis2*

by (*simp add: bot-fun-def*)

qed

lemma
fixes $f :: (-, -) \text{ prog}$
fixes $f_1 :: (-, -) \text{ prog}$
shows *supL*: $(f_1 \sqcup f_2) \ggg g = (f_1 \ggg g) \sqcup (f_2 \ggg g)$
and *supR*: $f \ggg (\lambda x. g_1 \ x \sqcup g_2 \ x) = (f \ggg g_1) \sqcup (f \ggg g_2)$

by (transfer; blast intro: spec.bind.supL spec.bind.supR)+

lemma SUPL:

fixes $X :: - \text{ set}$

fixes $f :: - \Rightarrow (-, -) \text{ prog}$

shows $(\bigsqcup x \in X. f x) \ggg g = (\bigsqcup x \in X. f x \ggg g)$

by transfer

(simp add: spec.interference.cl.bot spec.bind.supL spec.bind.bind spec.bind.SUPL
flip: spec.bind.botR bot-fun-def)

lemma SUPR:

fixes $X :: - \text{ set}$

fixes $f :: (-, -) \text{ prog}$

shows $f \ggg (\lambda v. \bigsqcup x \in X. g x v) = (\bigsqcup x \in X. f \ggg g x) \sqcup (f \ggg \perp)$

unfolding bot-fun-def

by transfer

(simp add: spec.bind.supL spec.bind.supR spec.bind.bind spec.bind.SUPR ac-simps le-supI2
spec.interference.closed.bind-rel-botR
sup.absorb2 spec.interference.closed.bind spec.interference.least spec.bind.mono
flip: spec.bind.botR)

lemma SupR:

fixes $X :: - \text{ set}$

fixes $f :: (-, -) \text{ prog}$

shows $f \gg (\bigsqcup X) = (\bigsqcup x \in X. f \gg x) \sqcup (f \gg \perp)$

by (simp add: prog.bind.SUPR[where $g = \lambda x v. x$, simplified])

lemma SUPR-not-empty:

fixes $f :: (-, -) \text{ prog}$

assumes $X \neq \{\}$

shows $f \ggg (\lambda v. \bigsqcup x \in X. g x v) = (\bigsqcup x \in X. f \ggg g x)$

using iffD2[OF ex-in-conv assms]

by (subst trans[OF prog.bind.SUPR sup.absorb1]; force intro: SUPI prog.bind.mono)

lemma mcont2mcont[cont-intro]:

assumes mcont luba orda Sup (\leq) f

assumes $\bigwedge v. \text{mcont luba orda Sup } (\leq) (\lambda x. g x v)$

shows mcont luba orda Sup (\leq) $(\lambda x. \text{prog.bind } (f x) (g x))$

proof(rule ccpo.mcont2mcont'[OF complete-lattice-ccpo - - assms(1)])

show mcont Sup (\leq) Sup (\leq) $(\lambda f. \text{prog.bind } f (g x))$ **for** x

by (intro mcontI contI monotoneI) (simp-all add: prog.bind.mono flip: prog.bind.SUPL)

show mcont luba orda Sup (\leq) $(\lambda x. \text{prog.bind } f (g x))$ **for** f

by (intro mcontI monotoneI contI)

(simp-all add: mcont-monoD[OF assms(2)] prog.bind.mono

flip: prog.bind.SUPR-not-empty contD[OF mcont-cont[OF assms(2)]])

qed

lemma inf-rel:

shows $\text{prog.rel } r \sqcap (f \ggg g) = \text{prog.rel } r \sqcap f \ggg (\lambda x. \text{prog.rel } r \sqcap g x)$

and $(f \ggg g) \sqcap \text{prog.rel } r = \text{prog.rel } r \sqcap f \ggg (\lambda x. \text{prog.rel } r \sqcap g x)$

by (transfer; simp add: spec.bind.inf-rel)+

setup <Sign.parent-path>

setup <Sign.mandatory-path guard>

lemma bot:

shows $\text{prog.guard } \perp = \perp$

and $prog.guard \langle False \rangle = \perp$
by (*simp-all add: prog.guard-def prog.action.empty*)

lemma top:

shows $prog.guard (\top :: 'a pred) = prog.return ()$ (**is** *?thesis1*)
and $prog.guard (\langle True \rangle :: 'a pred) = prog.return ()$ (**is** *?thesis2*)

proof –

show *?thesis1*
unfolding *prog.guard-def prog.return-def* **by** *transfer (simp add: Id-def)*
then show *?thesis2*
by (*simp add: top-fun-def*)

qed

lemma return-le:

shows $prog.guard g \leq prog.return ()$

unfolding *prog.guard-def Diag-def prog.return-def*

by *transfer (blast intro: spec.interference.cl.mono spec.action.mono)*

lemma monotone:

shows $mono (prog.guard :: 's pred \Rightarrow -)$

proof(*rule monoI*)

show $prog.guard g \leq prog.guard h$ **if** $g \leq h$ **for** $g h :: 's pred$

unfolding *prog.guard-def*

by (*strengthen ord-to-strengthen(1)[OF that]*) (*rule order.refl*)

qed

lemmas *strengthen[strg] = st-monotone[OF prog.guard.monotone]*

lemmas *mono = monotoneD[OF prog.guard.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF prog.guard.monotone, simplified]*

lemma less: — Non-triviality

assumes $g < g'$

shows $prog.guard g < prog.guard g'$

proof(*rule le-neq-trans*)

show $prog.guard g \leq prog.guard g'$

by (*strengthen ord-to-strengthen(1)[OF order-less-imp-le[OF assms]]*) *simp*

from *assms* **obtain** s **where** $g' s \neg g s$ **by** (*metis leD predicate1I*)

from $\langle \neg g s \rangle$ **have** $\langle \text{trace.T } s \ [] \ (Some ()) \rangle \leq prog.p2s (prog.guard g)$

by (*fastforce simp: trace.split-all prog.p2s.guard spec.guard-def spec.interference.cl.action spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv*)

elim: spec.singleton.bind-le)

moreover

from $\langle g' s \rangle$ **have** $\langle \text{trace.T } s \ [] \ (Some ()) \rangle \leq prog.p2s (prog.guard g')$

by (*force simp: prog.p2s.guard prog.p2s.action spec.guard-def*)

intro: order.trans[OF - spec.interference.expansive] spec.action.stutterI)

ultimately show $prog.guard g \neq prog.guard g'$ **by** *metis*

qed

lemma if:

shows $(if b then t else e) = (prog.guard \langle b \rangle \gg t) \sqcup (prog.guard \langle \neg b \rangle \gg e)$

by (*auto simp: prog.guard.bot prog.guard.top prog.bind.returnL prog.bind.botL*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path Parallel \rangle$

lemma bot:

assumes $\bigwedge a. a \in bs \implies Ps\ a = \perp$
assumes $as \cap bs \neq \{\}$
shows $prog.Parallel\ as\ Ps = prog.Parallel\ (as - bs)\ Ps \gg \perp$
by (*auto simp: assms(1)*
prog.p2s.simps spec.interference.cl.bot
spec.interference.closed.bind-rel-unitR spec.interference.closed.Parallel
spec.term.none.Parallel-some-agents[OF - assms(2), where Ps'= $\langle spec.rel\ (\{env\} \times UNIV)\rangle$]
spec.Parallel.discard-interference[where as=as and bs=as \cap bs]
simp del: Int-iff
simp flip: prog.p2s-inject spec.bind.botR spec.bind.bind
intro: arg-cong[OF spec.Parallel.cong])

lemma *mono*:

assumes $\bigwedge a. a \in as \implies Ps\ a \leq Ps'\ a$
shows $prog.Parallel\ as\ Ps \leq prog.Parallel\ as\ Ps'$
using *assms by transfer (blast intro: spec.Parallel.mono)*

lemma *strengthen-Parallel[strg]*:

assumes $\bigwedge a. a \in as \implies st\text{-ord}\ F\ (Ps\ a)\ (Ps'\ a)$
shows $st\text{-ord}\ F\ (prog.Parallel\ as\ Ps)\ (prog.Parallel\ as\ Ps')$
using *assms by (cases F) (auto simp: prog.Parallel.mono)*

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes $\bigwedge a. a \in as \implies monotone\ orda\ (\leq)\ (F\ a)$
shows $monotone\ orda\ (\leq)\ (\lambda f. prog.Parallel\ as\ (\lambda a. F\ a\ f))$
using *assms by transfer (simp add: spec.Parallel.mono2mono)*

lemma *cong*:

assumes $as = as'$
assumes $\bigwedge a. a \in as' \implies Ps\ a = Ps'\ a$
shows $prog.Parallel\ as\ Ps = prog.Parallel\ as'\ Ps'$
using *assms by transfer (rule spec.Parallel.cong; simp)*

lemma *no-agents*:

shows $prog.Parallel\ \{\}\ Ps = prog.return\ ()$
by *transfer*
(simp add: spec.Parallel.no-agents spec.interference.cl.return map-prod-image-Times Pair-image spec.action.read-agents)

lemma *singleton-agents*:

shows $prog.Parallel\ \{a\}\ Ps = Ps\ a$
by *transfer (rule spec.Parallel.singleton-agents)*

lemma *rename-UNIV*:

assumes *inj-on f as*
shows $prog.Parallel\ as\ Ps$
 $= prog.Parallel\ UNIV\ (\lambda b. if\ b \in f\ 'as\ then\ Ps\ (inv\ into\ as\ f\ b)\ else\ prog.return\ ())$
by (*simp add: prog.p2s.simps if-distrib prog.p2s.return spec.interference.cl.return*
spec.Parallel.rename-UNIV[OF assms]
flip: prog.p2s-inject
cong: spec.Parallel.cong if-cong)

lemmas *rename = spec.Parallel.rename[transferred]*

lemma *return*:

assumes $\bigwedge a. a \in bs \implies Ps\ a = prog.return\ ()$
shows $prog.Parallel\ as\ Ps = prog.Parallel\ (as - bs)\ Ps$
by (*subst (1 2) prog.Parallel.rename-UNIV[where f=id, simplified]*)

(*auto intro: arg-cong*[**where** $f = \text{prog.Parallel UNIV}$]
simp: assms fun-eq-iff f-inv-into-f[**where** $f = \text{id}$, *simplified*])

lemma *unwind*:

assumes $a: f \ggg g \leq Ps\ a$ — The selected process starts with action f

assumes $a \in as$

shows $f \ggg (\lambda v. \text{prog.Parallel}\ as\ (Ps(a := g\ v))) \leq \text{prog.Parallel}\ as\ Ps$

using *assms* **by** *transfer* (*simp add: spec.Parallel.unwind spec.interference.closed.bind-relL*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path parallel} \rangle$

lemmas *commute* = *spec.parallel.commute*[*transferred*]

lemmas *assoc* = *spec.parallel.assoc*[*transferred*]

lemmas *mono* = *spec.parallel.mono*[*transferred*]

lemma *strengthen*[*strg*]:

assumes *st-ord* $F\ P\ P'$

assumes *st-ord* $F\ Q\ Q'$

shows *st-ord* $F\ (\text{prog.parallel}\ P\ Q)\ (\text{prog.parallel}\ P'\ Q')$

using *assms* **by** (*cases F*; *simp add: prog.parallel.mono*)

lemma *mono2mono*[*cont-intro, partial-function-mono*]:

assumes *monotone orda* $(\leq)\ F$

assumes *monotone orda* $(\leq)\ G$

shows *monotone orda* $(\leq)\ (\lambda f. \text{prog.parallel}\ (F\ f)\ (G\ f))$

using *assms* **by** (*simp add: monotone-def prog.parallel.mono*)

lemma *bot*:

shows *botL*: $\text{prog.parallel}\ \perp\ P = P \ggg \perp$ (**is** *?thesis1*)

and *botR*: $\text{prog.parallel}\ P\ \perp = P \ggg \perp$ (**is** *?thesis2*)

proof —

show *?thesis1*

by (*simp add: prog.parallel-alt-def prog.Parallel.bot*[**where** $bs = \{ \text{True} \}$, *simplified*]
prog.Parallel.singleton-agents

cong: prog.Parallel.cong)

then show *?thesis2*

by (*simp add: prog.parallel.commute*)

qed

lemma *return*:

shows *returnL*: $\text{prog.return}\ () \parallel P = P$ (**is** *?thesis1*)

and *returnR*: $P \parallel \text{prog.return}\ () = P$ (**is** *?thesis2*)

proof —

show *?thesis1*

by (*simp add: prog.parallel-alt-def prog.Parallel.return*[**where** $bs = \{ \text{True} \}$, *simplified*]
prog.Parallel.singleton-agents)

then show *?thesis2*

by (*simp add: prog.parallel.commute*)

qed

lemma *Sup-not-empty*:

fixes $X :: (-, \text{unit})\ \text{prog}\ \text{set}$

assumes $X \neq \{ \}$

shows *SupL-not-empty*: $\bigsqcup X \parallel Q = (\bigsqcup P \in X. P \parallel Q)$ (**is** *?thesis1 Q*)

and *SupR-not-empty*: $P \parallel \bigsqcup X = (\bigsqcup Q \in X. P \parallel Q)$ (**is** *?thesis2*)

proof —

from *assms* **show** *?thesis1* Q **for** Q
by (*simp add: prog.p2s.parallel prog.p2s.Sup-not-empty* [*OF assms*] *image-image*
spec.parallel.Sup prog.p2s.SUP-not-empty
flip: prog.p2s-inject)
then show *?thesis2*
by (*simp add: prog.parallel commute*)
qed

lemma *sup*:

fixes $P :: (-, unit) prog$
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
using *prog.parallel.SupL-not-empty* [**where** $X = \{P, Q\}$] *prog.parallel.SupR-not-empty* [**where** $X = \{Q, R\}$] **by** *simp-all*

lemma *mcont2mcont* [*cont-intro*]:

assumes *mcont luba orda Sup* $(\leq) P$
assumes *mcont luba orda Sup* $(\leq) Q$
shows *mcont luba orda Sup* $(\leq) (\lambda x. prog.parallel (P x) (Q x))$
proof (*rule ccpo.mcont2mcont'* [*OF complete-lattice-ccpo - - assms(1)*])
show *mcont Sup* $(\leq) Sup (\leq) (\lambda y. prog.parallel y (Q x))$ **for** x
by (*intro mcontI contI monotoneI*) (*simp-all add: prog.parallel.mono prog.parallel.SupL-not-empty*)
show *mcont luba orda Sup* $(\leq) (\lambda x. prog.parallel y (Q x))$ **for** y
by (*simp add: mcontI monotoneI contI mcont-monoD* [*OF assms(2)*]
spec.parallel.mono mcont-contD [*OF assms(2)*] *prog.parallel.SupR-not-empty image-image*)
qed

qed

lemma *unwindL*:

fixes $f :: ('s, 'v) prog$
assumes $a: f \ggg g \leq P$ — The selected process starts with action f
shows $f \ggg (\lambda v. g v \parallel Q) \leq P \parallel Q$
unfolding *prog.parallel-alt-def*
by (*strengthen ord-to-strengthen* [*OF prog.Parallel.unwind* [**where** $a = True$]])
(auto simp: prog.Parallel.mono prog.bind.mono intro: assms)

lemma *unwindR*:

fixes $f :: ('s, 'v) prog$
assumes $a: f \ggg g \leq Q$ — The selected process starts with action f
shows $f \ggg (\lambda v. P \parallel g v) \leq P \parallel Q$
by (*subst* (1 2) *prog.parallel commute*) (*rule prog.parallel.unwindL* [*OF assms*])

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path bind \rangle$

lemma *parallel-le*:

fixes $P :: (-, -) prog$
shows $P \ggg Q \leq P \parallel Q$
by (*strengthen ord-to-strengthen* [*OF prog.parallel.unwindL* [**where** $g = prog.return$, *simplified prog.bind.returnR*,
OF order.refl]])
(simp add: prog.parallel.return)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path invmap \rangle$

lemma *bot*:

shows *prog.invmap sf vf* $\perp = (prog.rel (map-prod sf sf - ' Id) :: (-, unit) prog) \ggg \perp$
by (*auto simp: prog.p2s.simps spec.interference.cl.bot spec.rel.wind-bind-trailing*)

$spec.invmap.bind\ spec.invmap.rel\ spec.invmap.bot$
 $simp\ flip: prog.p2s-inject\ spec.bind.botR\ spec.bind.bind\ bot-fun-def$
 $intro: arg-cong[\mathbf{where}\ f=\lambda r. spec.rel\ r\ \gg\ \perp])$

lemma *id*:

shows $prog.invmap\ id\ id\ P = P$
and $prog.invmap\ (\lambda x. x)\ (\lambda x. x)\ P = P$
by (*transfer*; *simp add: spec.invmap.id id-def*)⁺

lemma *comp*:

shows $prog.invmap\ sf\ vf\ (prog.invmap\ sg\ vg\ P) = prog.invmap\ (\lambda s. sg\ (sf\ s))\ (\lambda s. vg\ (vf\ s))\ P$ (**is** *?thesis1 P*)
and $prog.invmap\ sf\ vf\ \circ\ prog.invmap\ sg\ vg = prog.invmap\ (sg\ \circ\ sf)\ (vg\ \circ\ vf)$ (**is** *?thesis2*)
proof –
show *?thesis1 P* **for** *P* **by** *transfer (simp add: spec.invmap.comp id-def)*
then show *?thesis2* **by** (*simp add: comp-def*)

qed

lemma *monotone*:

shows $mono\ (prog.invmap\ sf\ vf)$
unfolding *monotone-def* **by** *transfer (simp add: spec.invmap.mono)*

lemmas *strengthen[strg] = st-monotone[OF prog.invmap.monotone]*

lemmas *mono = monotoneD[OF prog.invmap.monotone]*

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes $monotone\ orda\ (\leq)\ t$
shows $monotone\ orda\ (\leq)\ (\lambda x. prog.invmap\ sf\ vf\ (t\ x))$
by (*rule monotone2monotone[OF prog.invmap.monotone assms]*) *simp-all*

lemma *Sup*:

fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
shows $prog.invmap\ sf\ vf\ (\bigsqcup X) = \bigsqcup (prog.invmap\ sf\ vf\ ' X) \sqcup prog.invmap\ sf\ vf\ \perp$
by *transfer*
(simp add: spec.invmap.bot spec.invmap.Sup spec.invmap.sup spec.invmap.bind spec.invmap.rel
spec.interference.cl.bot map-prod-vimage-Times ac-simps
sup.absorb2 spec.bind.mono[OF spec.rel.mono order.refl]
flip: spec.bind.botR spec.bind.bind spec.rel.unwind-bind-trailing bot-fun-def inv-image-alt-def)

lemma *Sup-not-empty*:

assumes $X \neq \{\}$
shows $prog.invmap\ sf\ vf\ (\bigsqcup X) = \bigsqcup (prog.invmap\ sf\ vf\ ' X)$
using *iffD2[OF ex-in-conv assms]*
by (*clarsimp simp: prog.invmap.Sup sup.absorb1 SUPI prog.invmap.mono[OF bot-least]*)

lemma *mcont*:

shows $mcont\ Sup\ (\leq)\ Sup\ (\leq)\ (prog.invmap\ sf\ vf)$
by (*simp add: contI mcontI prog.invmap.monotone prog.invmap.Sup-not-empty*)

lemmas *mcont2mcont[cont-intro] = mcont2mcont[OF prog.invmap.mcont, of luba orda P for luba orda P]*

lemma *bind*:

shows $prog.invmap\ sf\ vf\ (f\ \gg\ g) = prog.invmap\ sf\ f\ \gg\ (\lambda v. prog.invmap\ sf\ vf\ (g\ v))$
by *transfer (simp add: spec.invmap.bind)*

lemma *parallel*:

shows $prog.invmap\ sf\ vf\ (P\ \parallel\ Q) = prog.invmap\ sf\ vf\ P\ \parallel\ prog.invmap\ sf\ vf\ Q$
by *transfer (simp add: spec.invmap.parallel)*

lemma *invmap-image-vimage-commute*:

shows $\text{map-prod id (map-prod id sf)} - ' \text{map-prod id (Pair self)} ' F$
 $= \text{map-prod id (Pair self)} ' \text{map-prod id sf} - ' F$
by (*auto simp: map-prod-conv*)

lemma *action*:

shows $\text{prog.invmap sf vf (prog.action F)}$
 $= \text{prog.rel (map-prod sf sf} - ' \text{Id)}$
 $\ggg (\lambda::\text{unit. prog.action (map-prod id (map-prod sf sf)} - ' F)$
 $\ggg (\lambda v. \text{prog.rel (map-prod sf sf} - ' \text{Id)}$
 $\ggg (\lambda::\text{unit. } \sqcup v' \in \text{vf} - ' \{v\}. \text{prog.return } v'))$

proof –

have $*$: $\{env\} \times UNIV \cup \{self\} \times \text{map-prod sf sf} - ' \text{Id}$
 $= \{env\} \times UNIV \cup UNIV \times \text{map-prod sf sf} - ' \text{Id}$ **by** *auto*

show *?thesis*

by (*simp add: ac-simps prog.p2s.simps prog.p2s.action*
spec.interference.cl.action spec.invmap.bind spec.invmap.rel spec.invmap.action spec.invmap.return
spec.bind.bind spec.bind.return
*prog.invmap.invmap-image-vimage-commute map-prod-vimage-Times **
flip: prog.p2s-inject)
(*simp add: prog.p2s.Sup image-image prog.p2s.simps prog.p2s.return spec.interference.cl.return*
spec.interference.cl.bot spec.bind.supR
spec.rel.wind-bind-leading spec.rel.wind-bind-trailing
flip: spec.bind.botR spec.bind.SUPR spec.bind.bind)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path vmap} \rangle$

lemma *bot*:

shows $\text{prog.vmap vf } \perp = \perp$
by *transfer*
(*simp add: spec.interference.cl.bot spec.vmap.unit-rel*
flip: spec.term.none.map-gen[where vf=⟨()⟩])

lemma *unitL*:

shows $f \ggg g = \text{prog.vmap } \langle () \rangle f \ggg g$
by *transfer (metis spec.vmap.unitL)*

lemma *eq-return*:

shows $\text{prog.vmap vf } P = P \ggg \text{prog.return } \circ \text{vf (is ?thesis1)}$
and $\text{prog.vmap vf } P = P \ggg (\lambda v. \text{prog.return (vf } v)) \text{ (is ?thesis2)}$

proof –

show *?thesis1*

by *transfer*

(*simp add: comp-def spec.vmap.eq-return spec.interference.cl.return spec.interference.closed.bind-relR*)

then show *?thesis2*

by (*simp add: comp-def*)

qed

lemma *action*:

shows $\text{prog.vmap vf (prog.action F)} = \text{prog.action (map-prod vf id} - ' F)$
by *transfer (simp add: spec.map.interference.cl-sf-id spec.map.surj-sf-action image-comp)*

lemma *return*:

shows $\text{prog.vmap vf (prog.return } v) = \text{prog.return (vf } v)$

by (simp add: prog.return-def prog.vmap.action map-prod-image-Times)

setup <Sign.parent-path>

interpretation kleene: kleene prog.return () $\lambda x y. \text{prog.bind } x \langle y \rangle$

by standard

(simp-all add: prog.bind.bind prog.bind.return prog.bind.botL prog.bind.supL prog.bind.supR)

interpretation rel: galois.complete-lattice-class prog.steps prog.rel

proof

show $\text{prog.steps } P \subseteq r \iff P \leq \text{prog.rel } r$ for $P :: ('a, 'b) \text{ prog}$ and $r :: 'a \text{ rel}$

by transfer (auto simp flip: spec.rel.galois)

qed

setup <Sign.mandatory-path rel>

lemma empty:

shows $\text{prog.rel } \{\} = \bigsqcup \text{range prog.return}$

by (simp add: prog.p2s.simps prog.p2s.return image-image

spec.interference.cl.bot spec.interference.cl.return

spec.term.closed.bind-all-return[OF spec.term.closed.rel] spec.term.all.rel

sup.absorb1 spec.term.galois

flip: prog.p2s-inject spec.bind.SUPR-not-empty)

lemmas monotone = prog.rel.monotone-upper

lemmas strengthen[strg] = st-monotone[OF prog.rel.monotone]

lemmas mono = monotoneD[OF prog.rel.monotone]

lemmas Inf = prog.rel.upper-Inf

lemmas inf = prog.rel.upper-inf

lemma reflcl:

shows $\text{prog.rel } (r \cup \text{Id}) = (\text{prog.rel } r :: ('s, 'v) \text{ prog})$ (is ?thesis1)

and $\text{prog.rel } (\text{Id} \cup r) = (\text{prog.rel } r :: ('s, 'v) \text{ prog})$ (is ?thesis2)

proof –

show ?thesis1

by transfer

(subst (2) spec.rel.reflcl[where A=UNIV, symmetric];

auto intro: arg-cong[where f=spec.rel])

then show ?thesis2

by (simp add: ac-simps)

qed

lemma minus-Id:

shows $\text{prog.rel } (r - \text{Id}) = \text{prog.rel } r$

by (metis Un-Diff-cancel2 prog.rel.reflcl(1))

lemma Id:

shows $\text{prog.rel } \text{Id} = \bigsqcup \text{range prog.return}$

by (simp add: prog.rel.reflcl(1)[where r={}, simplified] prog.rel.empty)

lemma unfoldL:

fixes $r :: 's \text{ rel}$

assumes $\text{Id} \subseteq r$

shows $\text{prog.rel } r = \text{prog.action } (\{\} \times r) \gg \text{prog.rel } r$

proof –

have *: $\text{spec.rel } (\{\text{env}\} \times \text{UNIV})$

$\gg (\lambda v::\text{unit}. \text{spec.action } (\text{map-prod id } (\text{Pair self } '(\{\} \times r)))$

```

    >>= (λv::unit. spec.rel ({env} × UNIV ∪ {self} × r))
  = spec.rel ({env} × UNIV ∪ {self} × r) (is ?lhs = ?rhs)
  if Id ⊆ r
  for r :: 's rel
proof(rule antisym)
  let ?r' = {env} × UNIV ∪ {self} × r
  have ?lhs ≤ spec.rel ?r' >>= (λ-::unit. spec.rel ?r' >>= (λ-::unit. spec.rel ?r'))
    by (fastforce intro: spec.bind.mono spec.rel.mono spec.action.mono
        order.trans[OF - spec.rel.monomorphic-act-le]
        simp: spec.rel.act-def)
  also have ... = ?rhs
    by (simp add: spec.rel.wind-bind)
  finally show ?lhs ≤ ?rhs .
  from that show ?rhs ≤ ?lhs
  apply -
  apply (rule order.trans[OF -
    spec.bind.mono[OF spec.return.rel-le
    spec.bind.mono[OF spec.action.mono[where x={() } × {self} × Id] order.refl]])
  apply (subst spec.return.cong; simp add: image-image spec.bind.supL spec.bind.supR spec.bind.returnL
spec.idle-le)
  apply (fastforce simp: map-prod-image-Times)
  done
qed
from assms show ?thesis
  by transfer (simp add: * spec.interference.cl.action spec.bind.bind spec.rel.wind-bind-leading)
qed

```

lemma *wind-bind*: — arbitrary interstitial return type

shows $\text{prog.rel } r \gg \text{prog.rel } r = \text{prog.rel } r$
by transfer (simp add: spec.rel.wind-bind)

lemma *wind-bind-leading*: — arbitrary interstitial return type

assumes $r' \subseteq r$
shows $\text{prog.rel } r' \gg \text{prog.rel } r = \text{prog.rel } r$
using assms **by** transfer (subst spec.rel.wind-bind-leading; blast)

lemma *wind-bind-trailing*: — arbitrary interstitial return type

assumes $r' \subseteq r$
shows $\text{prog.rel } r \gg \text{prog.rel } r' = \text{prog.rel } r$ (is ?lhs = ?rhs)
using assms **by** transfer (subst spec.rel.wind-bind-trailing; blast)

Interstitial unit, for unfolding

lemmas *unwind-bind* = $\text{prog.rel.wind-bind}$ [where 'c=unit, symmetric]

lemmas *unwind-bind-leading* = $\text{prog.rel.wind-bind-leading}$ [where 'c=unit, symmetric]

lemmas *unwind-bind-trailing* = $\text{prog.rel.wind-bind-trailing}$ [where 'c=unit, symmetric]

lemma *mono-conv*:

shows $\text{prog.rel } r = \text{prog.kleene.star } (\text{prog.action } ({} \times r^=))$ (is ?lhs = ?rhs)

proof(rule antisym)

have $\text{spec.kleene.star } (\text{spec.rel.act } (\{env\} \times UNIV \cup \{self\} \times r)) \leq \text{prog.p2s } ?rhs$

proof(induct rule: spec.kleene.star.fixp-induct[case-names adm bot step])

case (step R) **show** ?case

proof(induct rule: le-supI[case-names act-step ret])

case act-step

have *: $\text{spec.rel.act } (\{env\} \times UNIV \cup \{self\} \times r) \leq \text{prog.p2s } (\text{prog.action } ({} \times r^=))$

by (auto simp: spec.rel.act-alt-def Times-Un-distrib2 spec.action.sup

prog.p2s.sup prog.p2s.return spec.interference.cl.return prog.action.sup map-prod-conv

simp flip: prog.return-def)

```

      intro: spec.action.mono le-supI2 spec.action.rel-le spec.return.rel-le
            le-supI1[OF order.trans[OF spec.action.mono prog.action.action-le]])
  show ?case
    apply (subst prog.kleene.star.simps)
    apply (strengthen ord-to-strengthen(1)[OF step])
    apply (simp add: prog.p2s.simps le-supI1[OF spec.bind.mono[OF * order.refl]])
    done
  next
    case ret show ?case
      by (simp add: order.trans[OF - prog.p2s.mono[OF prog.kleene.epsilon-star-le]]
            prog.p2s.return spec.interference.expansive)

  qed
  qed simp-all
  then show ?lhs ≤ ?rhs
    by (simp add: prog.p2s-leI prog.p2s.simps spec.rel.monomorphic-conv)
  show ?rhs ≤ ?lhs
  proof(induct rule: prog.kleene.star.fixp-induct[case-names adm bot step])
    case (step R) show ?case
      apply (strengthen ord-to-strengthen(1)[OF step])
      apply (simp add: prog.return.rel-le)
      apply (subst (2) prog.rel.unwind-bind)
      apply (auto intro: prog.bind.mono prog.action.rel-le)
      done
  qed simp-all
  qed

  setup ‹Sign.parent-path›

  setup ‹Sign.mandatory-path action›

  lemma inf-rel:
    assumes refl r
    shows prog.action F ⊓ prog.rel r = prog.action (F ∩ UNIV × r) (is ?thesis1)
      and prog.rel r ⊓ prog.action F = prog.action (F ∩ UNIV × r) (is ?thesis2)
  proof -
    from assms have refl (({env} × UNIV ∪ {self} × r) “ {a} for a
      by (fastforce dest: reflD intro: reflI)
    then show ?thesis1
      by transfer (simp add: spec.interference.cl.inf-rel spec.action.inf-rel; rule arg-cong; blast)
    then show ?thesis2
      by (rule inf-commute-conv)
  qed

  lemma inf-rel-reflcl:
    shows prog.action F ⊓ prog.rel r = prog.action (F ∩ UNIV × r=)
      and prog.rel r ⊓ prog.action F = prog.action (F ∩ UNIV × r=)
  by (simp-all add: refl-on-def prog.rel.reflcl ac-simps flip: prog.action.inf-rel)

  setup ‹Sign.parent-path›

  setup ‹Sign.mandatory-path return›

  lemma not-bot:
    shows prog.return v ≠ (⊥ :: ('s, 'v) prog)
  using prog.guard.less[where g=⊥::'s pred and g'=top]
  by (force dest: arg-cong[where f=prog.vmap (λ-::'v. ())]
      simp: prog.vmap.return prog.vmap.bot fun-eq-iff prog.guard.bot prog.guard.top
      simp flip: top.not-eq-extremum)

```

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *return*:

shows $\text{prog.invmap } sf \ vf \ (\text{prog.return } v)$
 $= \text{prog.rel } (\text{map-prod } sf \ sf \ -' \ \text{Id}) \gg \ (\lambda v. \sqcup v' \in vf \ -' \ \{v\}. \text{prog.return } v')$
apply (*simp add*: $\text{prog.return-def } \text{prog.invmap.action } \text{map-prod-vimage-Times}$)
apply (*simp add*: $\text{prog.action.return-const}$ **where** $V = \{v\}$ **and** $W = \{()\}$) $\text{prog.bind.bind } \text{prog.bind.return}$)
apply (*subst* $\text{prog.bind.bind[symmetric]}$, *subst* $\text{prog.rel.unfoldL[symmetric]}$;
force simp: $\text{prog.rel.wind-bind } \text{simp flip: prog.bind.bind}$)
done

lemma *split-vinvmap*:

fixes $P :: ('s, 'v) \text{ prog}$
shows $\text{prog.invmap } sf \ vf \ P = \text{prog.sinvmap } sf \ P \gg \ (\lambda v. \sqcup v' \in vf \ -' \ \{v\}. \text{prog.return } v')$
proof –
note $\text{sic-invmap} = \text{spec.interference.closed.invmap}$ **where** $af = id$ **and** $r = \{env\} \times UNIV$,
simplified map-prod-vimage-Times, simplified)
show *?thesis*
apply *transfer*
apply (*simp add*: $\text{spec.bind.supR } \text{sup.absorb1 } \text{spec.interference.cl.bot } \text{bot-fun-def}$
 $\text{spec.interference.closed.bind-relR } \text{sic-invmap } \text{spec.bind.mono}$
flip: spec.bind.botR)
apply (*subst* (1) $\text{spec.invmap.split-vinvmap}$)
apply (*subst* (1) $\text{spec.interference.closed.bind-relR[symmetric]}$, *erule* sic-invmap)
apply (*simp add*: $\text{spec.bind.SUPR } \text{spec.bind.supR } \text{spec.interference.cl.return}$
 $\text{sup.absorb1 } \text{bot-fun-def } \text{spec.interference.closed.bind-relR } \text{sic-invmap } \text{spec.bind.mono}$)
done
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

13.4 Refinement for $('s, 'v) \text{ prog}$

We specialize the rules of §12.1 to the $('s, 'v) \text{ prog}$ lattice. Observe that, as preconditions, postconditions and assumes are not interference closed, we apply the *prog.p2s* morphism and work in the more capacious (*sequential*, $'s, 'v$) *spec* lattice. This syntactic noise could be elided with another definition.

13.4.1 Introduction rules

Refinement is a way of showing inequalities and equalities between programs.

setup $\langle \text{Sign.mandatory-path refinement.prog} \rangle$

lemma *leI*:

assumes $\text{prog.p2s } c \leq \{\langle \text{True} \rangle\}$, $\top \Vdash \text{prog.p2s } d$, $\{\lambda -. \langle \text{True} \rangle\}$
shows $c \leq d$
using *assms* **by** (*simp add*: $\text{refinement-def } \text{prog.p2s-leI}$)

lemma *eqI*:

assumes $\text{prog.p2s } c \leq \{\langle \text{True} \rangle\}$, $\top \Vdash \text{prog.p2s } d$, $\{\lambda -. \langle \text{True} \rangle\}$
assumes $\text{prog.p2s } d \leq \{\langle \text{True} \rangle\}$, $\top \Vdash \text{prog.p2s } c$, $\{\lambda -. \langle \text{True} \rangle\}$
shows $c = d$
by (*rule antisym*; *simp add*: $\text{assms } \text{refinement.prog.leI}$)

setup $\langle \text{Sign.parent-path} \rangle$

13.4.2 Galois considerations

Refinement quadruples $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$ denote points in the $(\prime s, \prime v)$ *prog* lattice provided G is suitably interference closed.

setup $\langle \text{Sign.mandatory-path refinement.prog} \rangle$

lemma *galois*:

assumes $\text{spec.term.none } (\text{spec.rel } (\{\!|env|\!\} \times \text{UNIV}) :: (-, -, \text{unit}) \text{ spec}) \leq G$

shows $\text{prog.p2s } c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \longleftrightarrow c \leq \text{prog.s2p } (\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\})$

by (*simp add: assms prog.p2s-s2p.galois refinement-def spec.next-imp.contains spec.term.none.post-le*)

lemmas $\text{s2p-refinement} = \text{iffD1}[\text{OF refinement.prog.galois, rotated}]$

lemma *p2s-s2p*:

assumes $\text{spec.term.none } (\text{spec.rel } (\{\!|env|\!\} \times \text{UNIV}) :: (-, -, \text{unit}) \text{ spec}) \leq G$

shows $\text{prog.p2s } (\text{prog.s2p } (\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\})) \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

using *assms* by (*simp add: refinement.prog.galois*)

setup $\langle \text{Sign.parent-path} \rangle$

13.4.3 Rules

setup $\langle \text{Sign.mandatory-path refinement.prog} \rangle$

lemma *bot[iff]*:

shows $\text{prog.p2s } \perp \leq \{\!|P|\!\}, A \Vdash \text{prog.p2s } c', \{\!|Q|\!\}$

by (*simp add: refinement.prog.galois spec.term.none.interference.closed.rel-le*)

lemma *sup-conv*:

shows $\text{prog.p2s } (c_1 \sqcup c_2) \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

$\longleftrightarrow \text{prog.p2s } c_1 \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \wedge \text{prog.p2s } c_2 \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$

by (*simp add: prog.p2s.simps*)

lemmas $\text{sup} = \text{iffD2}[\text{OF refinement.prog.sup-conv, unfolded conj-explode}]$

lemma *if*:

assumes $i \implies \text{prog.p2s } t \leq \{\!|P|\!\}, A \Vdash \text{prog.p2s } t', \{\!|Q|\!\}$

assumes $\neg i \implies \text{prog.p2s } e \leq \{\!|P'|\!\}, A \Vdash \text{prog.p2s } e', \{\!|Q|\!\}$

shows $\text{prog.p2s } (\text{if } i \text{ then } t \text{ else } e) \leq \{\!|\text{if } i \text{ then } P \text{ else } P'|\!\}, A \Vdash \text{prog.p2s } (\text{if } i \text{ then } t' \text{ else } e'), \{\!|Q|\!\}$

using *assms* by *fastforce*

lemmas $\text{if}' = \text{refinement.prog.if}[\text{where } P=P \text{ and } P'=P, \text{simplified}] \text{ for } P$

lemma *case-option*:

assumes $\text{opt} = \text{None} \implies \text{prog.p2s } \text{none} \leq \{\!|P_n|\!\}, A \Vdash \text{prog.p2s } \text{none}', \{\!|Q|\!\}$

assumes $\bigwedge v. \text{opt} = \text{Some } v \implies \text{prog.p2s } (\text{some } v) \leq \{\!|P_s v|\!\}, A \Vdash \text{prog.p2s } (\text{some}' v), \{\!|Q|\!\}$

shows $\text{prog.p2s } (\text{case-option none some opt}) \leq \{\!|\text{case opt of None} \Rightarrow P_n \mid \text{Some } v \Rightarrow P_s v|\!\}, A \Vdash \text{prog.p2s } (\text{case-option none}' \text{ some}' \text{ opt}), \{\!|Q|\!\}$

using *assms* by (*simp add: option.case-eq-if*)

lemma *case-sum*:

assumes $\bigwedge v. x = \text{Inl } v \implies \text{prog.p2s } (\text{left } v) \leq \{\!|P_l v|\!\}, A \Vdash \text{prog.p2s } (\text{left}' v), \{\!|Q|\!\}$

assumes $\bigwedge v. x = \text{Inr } v \implies \text{prog.p2s } (\text{right } v) \leq \{\!|P_r v|\!\}, A \Vdash \text{prog.p2s } (\text{right}' v), \{\!|Q|\!\}$

shows $\text{prog.p2s } (\text{case-sum left right } x) \leq \{\!|\text{case-sum } P_l P_r x|\!\}, A \Vdash \text{prog.p2s } (\text{case-sum left}' \text{ right}' x), \{\!|Q|\!\}$

using *assms* by (*simp add: sum.case-eq-if*)

lemma case-list:

assumes $x = [] \implies \text{prog.p2s nil} \leq \{P_n\}, A \Vdash \text{prog.p2s nil}', \{Q\}$
assumes $\bigwedge v \text{ vs. } x = v \# \text{ vs} \implies \text{prog.p2s (cons v vs)} \leq \{P_c v \text{ vs}\}, A \Vdash \text{prog.p2s (cons' v vs)}, \{Q\}$
shows $\text{prog.p2s (case-list nil cons x)} \leq \{\text{case-list } P_n P_c x\}, A \Vdash \text{prog.p2s (case-list nil' cons' x)}, \{Q\}$
using *assms* **by** (*simp add: list.case-eq-if*)

lemma action:

fixes $F :: ('v \times 's \times 's) \text{ set}$
assumes $\bigwedge v s s'. \llbracket P s; (v, s, s') \in F; (\text{self}, s, s') \in \text{spec.steps } A \vee s = s' \rrbracket \implies Q v s'$
assumes $\bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \implies (v, s, s') \in F'$
assumes $sP: \text{stable (spec.steps } A \text{ `` } \{env\} \text{) } P$
assumes $\bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \implies \text{stable (spec.steps } A \text{ `` } \{env\} \text{) } (Q v)$
shows $\text{prog.p2s (prog.action } F) \leq \{P\}, A \Vdash \text{prog.p2s (prog.action } F'), \{Q\}$
unfolding *prog.p2s.action spec.interference.cl.action*
apply (*rule refinement.pre-a[OF - spec.rel.upper-lower-expansive]*)
apply (*rule refinement.spec.bind[rotated]*)
apply (*rule refinement.spec.rel-mono[OF order.refl]*);
fastforce simp: spec.term.all.rel spec.steps.rel
intro: sP antimonoD[OF stable.antimono-rel, unfolded le-bool-def, rule-format, rotated]
apply (*strengthen ord-to-strengthen(1)[OF inf-le2]*)
apply (*strengthen ord-to-strengthen(1)[OF refinement.spec.bind.res.rel-le[OF order.refl]]*)
apply (*rule refinement.spec.bind[rotated, where Q'= $\lambda v s. Q v s \wedge \text{stable (spec.steps } A \text{ `` } \{env\} \text{) } (Q v)$]*)
apply (*rule refinement.spec.action*);
fastforce simp: spec.initial-steps.term.all spec.initial-steps.rel
intro: assms)
apply (*rule refinement.gen-asm*)
apply (*rule refinement.spec.bind[rotated]*)
apply (*rule refinement.spec.rel-mono[OF order.refl]*);
fastforce simp: spec.steps.term.all spec.rel.lower-upper-lower
elim: antimonoD[OF stable.antimono-rel, unfolded le-bool-def, rule-format, rotated]
dest: subsetD[OF spec.steps.refinement.spec.bind.res-le]
apply (*rule refinement.spec.return*)
apply (*simp only: spec.idle-le*)
done

lemma return:

assumes $sQ: \text{stable (spec.steps } A \text{ `` } \{env\} \text{) } (Q v)$
shows $\text{prog.p2s (prog.return } v) \leq \{Q v\}, A \Vdash \text{prog.p2s (prog.return } v), \{Q\}$
unfolding *prog.return-def* **using** *assms* **by** (*blast intro: refinement.prog.action*)

lemma invmap-return:

assumes $sQ: \text{stable (spec.steps } A \text{ `` } \{env\} \text{) } (Q v)$
assumes $vf v = v'$
shows $\text{prog.p2s (prog.return } v) \leq \{Q v\}, A \Vdash \text{prog.p2s (prog.invmap sf vf (prog.return } v')), \{Q\}$
unfolding *prog.invmap.return*
by (*strengthen ord-to-strengthen(2)[OF prog.return.rel-le]*)
(simp add: assms(2) refinement.pre-g[OF refinement.prog.return[where Q=Q, OF sQ]]
SUP-upper prog.bind.return prog.p2s.mono)

lemma bind-abstract:

fixes $f :: ('s, 'v) \text{ prog}$
fixes $f' :: ('s, 'v') \text{ prog}$
fixes $g :: 'v \Rightarrow ('s, 'w) \text{ prog}$
fixes $g' :: 'v' \Rightarrow ('s, 'w) \text{ prog}$
fixes $vf :: 'v \Rightarrow 'v'$
assumes $\bigwedge v. \text{prog.p2s (g } v) \leq \{Q' (vf v)\}, \text{refinement.spec.bind.res (spec.pre } P \sqcap \text{spec.term.all } A \sqcap \text{prog.p2s } f') A (vf v) \Vdash \text{prog.p2s (g' (vf v)), \{Q\}}$

assumes $\text{prog.p2s } f \leq \{\!\{P\}\!\}$, $\text{spec.term.all } A \Vdash \text{spec.vinvmmap } \text{vf } (\text{prog.p2s } f')$, $\{\!\{\lambda v. Q' (\text{vf } v)\}\!\}$
shows $\text{prog.p2s } (f \ggg g) \leq \{\!\{P\}\!\}$, $A \Vdash \text{prog.p2s } (f' \ggg g')$, $\{\!\{Q\}\!\}$
by (*simp add: prog.p2s.simps refinement.spec.bind-abstract[OF assms]*)

lemma bind:

assumes $\bigwedge v. \text{prog.p2s } (g v) \leq \{\!\{Q' v\}\!\}$, $\text{refinement.spec.bind.res } (\text{spec.pre } P \sqcap \text{spec.term.all } A \sqcap \text{prog.p2s } f') A$
 $v \Vdash \text{prog.p2s } (g' v)$, $\{\!\{Q\}\!\}$
assumes $\text{prog.p2s } f \leq \{\!\{P\}\!\}$, $\text{spec.term.all } A \Vdash \text{prog.p2s } f'$, $\{\!\{Q'\}\!\}$
shows $\text{prog.p2s } (f \ggg g) \leq \{\!\{P\}\!\}$, $A \Vdash \text{prog.p2s } (f' \ggg g')$, $\{\!\{Q\}\!\}$
by (*simp add: prog.p2s.simps refinement.spec.bind[OF assms]*)

lemmas *rev-bind = refinement.prog.bind[rotated]*

lemma Parallel:

fixes $A :: (\text{sequential}, 's, \text{unit}) \text{spec}$
fixes $Q :: 'a \Rightarrow 's \text{pred}$
fixes $P_s :: 'a \Rightarrow ('s, \text{unit}) \text{prog}$
fixes $P_{s'} :: 'a \Rightarrow ('s, \text{unit}) \text{prog}$
assumes $\bigwedge a. a \in as \implies \text{prog.p2s } (P_s a) \leq \{\!\{P a\}\!\}$, $\text{refinement.spec.env-hyp } P A as (\text{prog.p2s} \circ P_{s'}) a \Vdash \text{prog.p2s}$
 $(P_{s'} a)$, $\{\!\{\lambda rv. Q a\}\!\}$
shows $\text{prog.p2s } (\text{prog.Parallel } as P_s) \leq \{\!\{\bigcap a \in as. P a\}\!\}$, $A \Vdash \text{prog.p2s } (\text{prog.Parallel } as P_{s'})$, $\{\!\{\lambda rv. \bigcap a \in as. Q$
 $a\}\!\}$
using *assms by transfer (simp add: refinement.spec.Parallel comp-def)*

lemma parallel:

assumes $\text{prog.p2s } c_1 \leq \{\!\{P_1\}\!\}$, $\text{refinement.spec.env-hyp } (\lambda a. \text{if } a \text{ then } P_1 \text{ else } P_2) A UNIV (\lambda a. \text{if } a \text{ then } \text{prog.p2s}$
 $c_1' \text{ else } \text{prog.p2s } c_2')$ $\text{True} \Vdash \text{prog.p2s } c_1'$, $\{\!\{Q_1\}\!\}$
assumes $\text{prog.p2s } c_2 \leq \{\!\{P_2\}\!\}$, $\text{refinement.spec.env-hyp } (\lambda a. \text{if } a \text{ then } P_1 \text{ else } P_2) A UNIV (\lambda a. \text{if } a \text{ then } \text{prog.p2s}$
 $c_1' \text{ else } \text{prog.p2s } c_2')$ $\text{False} \Vdash \text{prog.p2s } c_2'$, $\{\!\{Q_2\}\!\}$
shows $\text{prog.p2s } (\text{prog.parallel } c_1 c_2) \leq \{\!\{P_1 \wedge P_2\}\!\}$, $A \Vdash \text{prog.p2s } (\text{prog.parallel } c_1' c_2')$, $\{\!\{\lambda v. Q_1 v \wedge Q_2 v\}\!\}$
unfolding *prog.parallel-alt-def*
by (*rule refinement.pre[OF refinement.prog.Parallel[where A=A and P= $\lambda a. \text{if } a \text{ then } P_1 \text{ else } P_2$ and Ps'= $\lambda a. \text{if } a \text{ then } c_1' \text{ else } c_2'$ and Q= $\lambda a. \text{if } a \text{ then } Q_1 () \text{ else } Q_2 ()$]]])*
(use assms in <force simp: if-distrib comp-def>)+

setup *<Sign.parent-path>*

13.5 A relational assume/guarantee program logic for the (*'s*, *'v*) *prog* lattice

Similarly we specialize the assume/guarantee program logic of §12.2 to (*'s*, *'v*) *prog*.

References:

- de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001); Xu, de Roever, and He (1997)
- Prensa Nieto (2003, §7)
- Vafeiadis (2008, §3)

13.5.1 Galois considerations

For suitably stable P , Q , $\{\!\{P\}\!\}$, $A \Vdash G$, $\{\!\{Q\}\!\}$ is interference closed and hence denotes a point in (*'s*, *'v*) *prog*. In other words we can replace programs with their specifications.

setup *<Sign.mandatory-path ag.prog>*

lemma galois:

shows $\text{prog.p2s } c \leq \{\!\{P\}\!\}$, $A \Vdash G$, $\{\!\{Q\}\!\} \longleftrightarrow c \leq \text{prog.s2p } (\{\!\{P\}\!\}, A \Vdash G, \{\!\{Q\}\!\})$
by (*simp add: prog.p2s-s2p.galois ag.spec.term.none-inteference*)

lemmas $s2p\text{-}ag = \text{iff}D1[OF\ ag.\text{prog}.\text{galois}]$

lemma $p2s\text{-}s2p\text{-}ag$:

shows $\text{prog}.\text{p}2s\ (\text{prog}.\text{s}2p\ (\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\})) \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
by (*simp add: ag.prog.galois*)

lemma $p2s\text{-}s2p\text{-}ag\text{-}stable$:

assumes *stable A P*
assumes $\bigwedge v.\ \text{stable}\ A\ (Q\ v)$
shows $\text{prog}.\text{p}2s\ (\text{prog}.\text{s}2p\ (\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\})) = \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
by (*rule prog.p2s-s2p.insertion[OF spec.interference.closed-ag[where r=UNIV, simplified, OF assms]]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog.ag} \rangle$

lemma $\text{bot}[\text{iff}]$:

shows $\text{prog}.\text{p}2s\ \perp \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
by (*simp add: ag.prog.galois*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path ag} \rangle$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

lemma sup-conv :

shows $\text{prog}.\text{p}2s\ (c_1 \sqcup c_2) \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} \iff \text{prog}.\text{p}2s\ c_1 \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} \wedge \text{prog}.\text{p}2s\ c_2 \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
by (*simp add: prog.p2s.simps*)

lemmas $\text{sup} = \text{iff}D2[OF\ ag.\text{prog}.\text{sup-conv},\ \text{unfolded conj-explode}]$

lemma bind : — Assumptions in weakest-pre order

assumes $\bigwedge v.\ \text{prog}.\text{p}2s\ (g\ v) \leq \{\!|Q'\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $\text{prog}.\text{p}2s\ f \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
shows $\text{prog}.\text{p}2s\ (f \ggg g) \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
by (*simp add: prog.p2s.simps*) (*rule ag.spec.bind; fact*)

lemma action : — Conclusion is insufficiently instantiated for use

fixes $F :: ('v \times 's \times 's)\ \text{set}$
assumes $Q: \bigwedge v\ s\ s'.\ \llbracket P\ s; (v, s, s') \in F \rrbracket \implies Q\ v\ s'$
assumes $G: \bigwedge v\ s\ s'.\ \llbracket P\ s; s \neq s'; (v, s, s') \in F \rrbracket \implies (s, s') \in G$
assumes $sP: \text{stable}\ A\ P$
assumes $sQ: \bigwedge s\ s'\ v.\ \llbracket P\ s; (v, s, s') \in F \rrbracket \implies \text{stable}\ A\ (Q\ v)$
shows $\text{prog}.\text{p}2s\ (\text{prog}.\text{action}\ F) \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
unfolding $\text{prog}.\text{p}2s.\text{action}\ \text{spec}.\text{interference}.\text{cl}.\text{action}$ — sp proof
by (*rule ag.gen-asm*
ag.spec.bind[rotated] ag.spec.stable-interference ag.spec.return
ag.spec.action[where Q= $\lambda v\ s.\ Q\ v\ s \wedge (\exists s s'. P\ s \wedge (v, s, s') \in F)$]
| use assms in auto) $+$

lemma guard :

assumes $\bigwedge s.\ \llbracket P\ s; g\ s \rrbracket \implies Q\ ()\ s$
assumes *stable A P*
assumes *stable A (Q ())*
shows $\text{prog}.\text{p}2s\ (\text{prog}.\text{guard}\ g) \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$

using *assms* by (fastforce simp: prog.guard-def intro: ag.prog.action split: if-splits)

lemma *Parallel*:

assumes $\bigwedge a. a \in as \implies \text{prog.p2s } (Ps \ a) \leq \{P \ a\}, A \cup (\bigcup a' \in as - \{a\}. G \ a') \vdash G \ a, \{\lambda v. Q \ a\}$
shows $\text{prog.p2s } (\text{prog.Parallel } as \ Ps) \leq \{\prod a \in as. P \ a\}, A \vdash \bigcup a \in as. G \ a, \{\lambda v. \prod a \in as. Q \ a\}$
using *assms* by transfer (fast intro: ag.spec.Parallel)

lemma *parallel*:

assumes $\text{prog.p2s } c_1 \leq \{P_1\}, A \cup G_2 \vdash G_1, \{Q_1\}$
assumes $\text{prog.p2s } c_2 \leq \{P_2\}, A \cup G_1 \vdash G_2, \{Q_2\}$
shows $\text{prog.p2s } (\text{prog.parallel } c_1 \ c_2) \leq \{P_1 \wedge P_2\}, A \vdash G_1 \cup G_2, \{\lambda v. Q_1 \ v \wedge Q_2 \ v\}$
unfolding *prog.parallel-alt-def*

by (rule ag.pre[OF ag.prog.Parallel[where $A=A$ and $G=\lambda a. \text{if } a \text{ then } G_1 \text{ else } G_2$ and $P=\langle P_1 \wedge P_2 \rangle$ and $Q=\lambda a. \text{if } a \text{ then } Q_1 \ () \text{ else } Q_2 \ ()$]])

(use *assms* in $\langle \text{auto intro: ag.pre-imp} \rangle$)

lemma *return*:

assumes *sQ*: stable $A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.return } v) \leq \{Q \ v\}, A \vdash G, \{Q\}$
using *assms* by (auto simp: prog.return-def intro: ag.prog.action)

lemma *if*:

assumes $b \implies \text{prog.p2s } c_1 \leq \{P_1\}, A \vdash G, \{Q\}$
assumes $\neg b \implies \text{prog.p2s } c_2 \leq \{P_2\}, A \vdash G, \{Q\}$
shows $\text{prog.p2s } (\text{if } b \text{ then } c_1 \text{ else } c_2) \leq \{\text{if } b \text{ then } P_1 \text{ else } P_2\}, A \vdash G, \{Q\}$
using *assms* by (fastforce intro: ag.pre-ag)

lemma *case-option*:

assumes $x = \text{None} \implies \text{prog.p2s } \text{none} \leq \{P_n\}, A \vdash G, \{Q\}$
assumes $\bigwedge v. x = \text{Some } v \implies \text{prog.p2s } (\text{some } v) \leq \{P_s \ v\}, A \vdash G, \{Q\}$
shows $\text{prog.p2s } (\text{case-option } \text{none } \text{some } x) \leq \{\text{case } x \text{ of } \text{None} \Rightarrow P_n \mid \text{Some } v \Rightarrow P_s \ v\}, A \vdash G, \{Q\}$
using *assms* by (fastforce intro: ag.pre-ag split: option.split)

lemma *case-sum*:

assumes $\bigwedge v. x = \text{Inl } v \implies \text{prog.p2s } (\text{left } v) \leq \{P_l \ v\}, A \vdash G, \{Q\}$
assumes $\bigwedge v. x = \text{Inr } v \implies \text{prog.p2s } (\text{right } v) \leq \{P_r \ v\}, A \vdash G, \{Q\}$
shows $\text{prog.p2s } (\text{case-sum } \text{left } \text{right } x) \leq \{\text{case-sum } P_l \ P_r \ x\}, A \vdash G, \{Q\}$
using *assms* by (fastforce intro: ag.pre-ag split: sum.split)

lemma *case-list*:

assumes $x = [] \implies \text{prog.p2s } \text{nil} \leq \{P_n\}, A \vdash G, \{Q\}$
assumes $\bigwedge v \ vs. x = v \ # \ vs \implies \text{prog.p2s } (\text{cons } v \ vs) \leq \{P_c \ v \ vs\}, A \vdash G, \{Q\}$
shows $\text{prog.p2s } (\text{case-list } \text{nil } \text{cons } x) \leq \{\text{case-list } P_n \ P_c \ x\}, A \vdash G, \{Q\}$
using *assms* by (fastforce intro: ag.pre-ag split: list.split)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

13.5.2 A proof of the parallel rule using Abadi and Plotkin's composition principle

Here we show that the key rule for *Parallel* (*ag.spec.Parallel*) can be established using the *spec.ag-circular* rule (§9.2).

The following proof is complicated by the need to discard a lot of contextual information.

notepad
begin

have *imp-discharge-leL1*:

$x' \leq x \implies x' \sqcap (x \sqcap y \longrightarrow_H z) = x' \sqcap (y \longrightarrow_H z)$ **for** $x \ x' \ y \ z$
by (*simp add: heyting.curry-conv heyting.discharge(1)*)

have *LHS-rel*:

$\{proc \ x\} \times UNIV \cup (-\{proc \ x\}) \times (A \cup (Id \cup \bigcup (G \ ' (as - \{x\}))))$
 $= ((-\{proc \ ' \ as\}) \times (A \cup (Id \cup \bigcup (G \ ' (as - \{x\}))))$
 $\cup (\{proc \ x\} \times UNIV \cup proc \ ' (as - \{x\}) \times (A \cup (Id \cup \bigcup (G \ ' (as - \{x\}))))))$ **for** $A \ G \ as \ x$
by *blast*

have *rel-agents-split*:

$spec.rel \ (as \times r \cup s) = spec.rel \ (as \times r \cup fst \ ' \ s \times UNIV) \sqcap spec.rel \ (as \times UNIV \cup s)$
if $fst \ ' \ s \sqcap as = \{\}$ **for** $as \ r \ s$
using that by (*fastforce simp: image-iff simp flip: spec.rel.inf intro: arg-cong[where f=spec.rel]*)

— *ag.spec.Parallel*

fix $as :: 'a \ set$

fix $A :: 's \ rel$

fix $G :: 'a \Rightarrow 's \ rel$

fix $P :: 'a \Rightarrow 's \ pred$

fix $Q :: 'a \Rightarrow 's \ pred$

fix $P_s :: 'a \Rightarrow (sequential, 's, unit) \ spec$

assume *proc-ag*: $\bigwedge a. a \in as \implies P_s \ a \leq \{\!| P \ a |\!\}, A \cup (\bigcup a' \in as - \{a\}. G \ a') \vdash G \ a, \{\!| \lambda v. Q \ a |\!\}$

have *spec.Parallel* $as \ P_s \leq \{\!| \bigcap a \in as. P \ a |\!\}, A \vdash \bigcup a \in as. G \ a, \{\!| \lambda v. \bigcap a \in as. Q \ a |\!\}$ (**is** $?lhs \leq ?rhs$)

proof(*cases as = \{\}*)

case *True* **then show** $?thesis$

by (*simp add: spec.Parallel.no-agents ag.interference-le*)

next

case *False* **then show** $?thesis$

apply —

supply *inf.bounded-iff[simp del]* — preserve RHS

— replace P_s with a/g specs. guard against empty A, G

apply (*strengthen ord-to-strengthen(1)[OF proc-ag], assumption*)

apply (*subst ag.reflcl-ag*)

apply (*strengthen ord-to-strengthen(2)[OF reflcl-cl.sup-cl-le]*)

— Circular concurrent reasoning

unfolding *ag-def*

— Move a/g hypotheses to LHS, normalize

apply (*simp add: heyting ac-simps*)

— Discharge *spec.pre P*

apply (*subst inf-assoc[symmetric]*)

apply (*subst inf-commute*)

apply (*subst inf-assoc*)

apply (*subst (2) inf-commute*)

apply (*subst spec.Parallel.inf-pre, assumption*)

apply (*simp add: ac-simps*)

— Idiom for rewriting under a quantifier, here *Parallel*

apply (*rule order.trans*)

apply (*rule inf.mono[OF order.refl]*)

apply (*rule spec.Parallel.mono*)

apply (*subst imp-discharge-leL1*)

apply (*simp add: Inf-lower spec.pre.INF; fail*)

— Discard *spec.pre* hypothesis

apply (*rule inf-le2*)

— Move environment assumption A hypothesis under $spec.toSequential$ and the Inf

unfolding $spec.Parallel-def$

apply ($subst\ inf.commute$)

apply ($subst\ spec.map.inf-distr$)

apply ($subst\ spec.invmap.rel$)

apply ($simp\ add: ac-simps\ flip: INF-inf-const1$)

— Eradicate $spec.toSequential$: move to parallel space

apply ($simp\ add: spec.map-invmap.galois\ spec.invmap.inf\ spec.invmap.post\ spec.invmap.rel$
 $flip: spec.term.all.invmap\ spec.term.all.map$)

— Eradicate $spec.toConcurrent$

apply ($simp\ add: ac-simps\ spec.invmap.heyting\ spec.invmap.inf\ spec.invmap.rel$
 $spec.invmap.pre\ spec.invmap.post$)

— Normalize the relations

apply ($simp\ add: inf-sup-distrib1\ Times-Int-Times\ map-prod-vimage-Times\ ac-simps\ spec.rel.reflcl$
 $flip: spec.rel.inf\ image-Int\ inf.assoc$)

— Discharge environment assumption A and that for agents in $-as$

apply ($subst\ LHS-rel$)

— Idiom for rewriting under a quantifier, here Inf

apply ($rule\ order.trans$)

apply ($rule\ INF-mono[where\ B=as]$)

apply ($rule\ rev-bexI, assumption$)

apply ($subst\ (2)\ rel-agents-split, fastforce$)

apply ($subst\ imp-discharge-leL1$)

apply ($rule\ spec.rel.mono, fastforce\ simp: image-Un$)

apply ($rule\ order.refl$)

apply ($simp\ flip: sup.assoc\ Times-Un-distrib1$)

apply ($simp\ add: ac-simps\ INF-inf-const1$)

— Apply Abadi/Plotkin

— Change coordinates

apply ($subst\ INF-rename-bij[where\ X=proc\ 'as\ and\ \pi=the-inv\ proc]$)

apply ($fastforce\ simp: bij-betw-iff-bijections$)

apply ($simp\ add: image-comp\ cong: INF-cong$)

— The circular reasoning principle only applies to the relational part as $spec.post$ is not termination closed. Therefore split the goal

apply ($subst\ heyting.infR$)

apply ($subst\ INF-inf-distrib[symmetric]$)

apply ($rule\ order.trans$)

apply ($rule\ inf-mono[OF\ order.refl]+$)

apply ($rule\ order.trans[rotated]$)

apply ($rule\ spec.ag-circular[where$

$as=proc\ 'as$

and $P_s=\lambda a. spec.rel\ (\{a\} \times (Id \cup G\ (the-inv\ proc\ a))) \cup insert\ env\ (proc\ '(-\ \{the-inv\ proc\ a\})) \times UNIV,$
 $simplified\ spec.idle-le\ spec.term.closed.rel, simplified,$

$OF\ subsetD[OF\ spec.cam.closed.antimono\ spec.cam.closed.rel[OF\ order.refl]]])$

apply ($clarsimp\ simp: image-iff$)

apply ($metis\ ComplI\ agent.exhaust\ singletonD$)

apply ($rule\ INFI$)

apply ($simp\ add: heyting\ ac-simps\ flip: spec.rel.INF\ INF-inf-const1$)

— Idiom for rewriting under a quantifier, here Inf

apply ($rule\ order.trans$)

apply (*rule INF-mono*[**where** $B=as$])
apply (*rule rev-bexI*, *assumption*)
apply (*subst heyting.discharge*)
apply (*rule spec.rel.mono-reflcl*)
apply *fastforce*
apply (*simp flip: spec.rel.inf*)
apply (*rule order.refl*)
apply (*simp flip: spec.rel.INF*)
apply (*rule spec.rel.mono*)
apply (*clarsimp simp: image-iff*)
apply (*metis ComplI agent.exhaust singletonD*)
apply (*simp add: ac-simps flip: spec.rel.INF*)
apply (*subst inf.assoc[symmetric]*)
apply (*simp flip: spec.rel.inf*)

— Conclude guarantee G
apply (*rule le-infI*[*rotated*])
apply (*rule le-infI1*)
apply (*rule spec.rel.mono-reflcl*, *blast*)

— Conclude *spec.post Q*
apply (*subst (2) INF-inf-const1[symmetric]*, *force*)
— Idiom for rewriting under a quantifier, here *Inf*
apply (*rule order.trans*)
apply (*rule INF-mono*[**where** $B=as$])
apply (*rule rev-bexI*, *blast*)
apply (*subst heyting.discharge*)
apply (*force intro: spec.rel.mono*)
apply (*rule order.refl*)
apply (*simp add: spec.post.Ball flip: INF-inf-distrib*)
done
qed

end

13.6 Specification inhabitation

setup $\langle \text{Sign.mandatory-path inhabits.prog} \rangle$

lemma *Sup*:

assumes *prog.p2s* $P -s, xs \rightarrow P'$

assumes $P \in X$

shows *prog.p2s* $(\sqcup X) -s, xs \rightarrow P'$

by (*auto simp: prog.p2s.Sup intro: inhabits.Sup inhabits.supL assms*)

lemma *supL*:

assumes *prog.p2s* $P -s, xs \rightarrow P'$

shows *prog.p2s* $(P \sqcup Q) -s, xs \rightarrow P'$

by (*simp add: prog.p2s.simps assms inhabits.supL*)

lemma *supR*:

assumes *prog.p2s* $Q -s, xs \rightarrow Q'$

shows *prog.p2s* $(P \sqcup Q) -s, xs \rightarrow Q'$

by (*simp add: prog.p2s.simps assms inhabits.supR*)

lemma *bind*:

assumes *prog.p2s* $f -s, xs \rightarrow \text{prog.p2s } f'$

shows *prog.p2s* $(f \gg g) -s, xs \rightarrow \text{prog.p2s } (f' \gg g)$

by (*simp add: prog.p2s.simps inhabits.spec.bind assms*)

lemma *return*:

shows *prog.p2s (prog.return v) -s, [] → spec.return v*

by (*metis prog.p2s.return inhabits.pre inhabits.tau[OF spec.idle.interference.cl-le]
spec.interference.expansive*)

lemma *action-step*:

fixes *F :: ('v × 's × 's) set*

assumes *(v, s, s') ∈ F*

shows *prog.p2s (prog.action F) -s, [(self, s')] → prog.p2s (prog.return v)*

apply (*simp only: prog.p2s.action prog.p2s.return spec.interference.cl.action spec.interference.cl.return*)

apply (*rule inhabits.pre[OF - order.refl]*)

apply (*rule inhabits.spec.bind'[OF inhabits.spec.rel.term]*)

apply (*simp add: spec.bind.returnL spec.idle-le*)

apply (*rule inhabits.spec.bind'[OF inhabits.spec.action.step]*)

using *assms* **apply** *force*

apply (*simp add: spec.bind.returnL spec.idle-le*)

apply (*rule inhabits.tau*)

apply (*simp add: spec.idle-le*)

apply *simp*

done

lemma *action-stutter*:

fixes *F :: ('v × 's × 's) set*

assumes *(v, s, s) ∈ F*

shows *prog.p2s (prog.action F) -s, [] → prog.p2s (prog.return v)*

apply (*simp only: prog.p2s.action prog.p2s.return spec.interference.cl.action spec.interference.cl.return*)

apply (*rule inhabits.pre[OF - order.refl]*)

apply (*rule inhabits.spec.bind'[OF inhabits.spec.rel.term]*)

apply (*simp add: spec.bind.returnL spec.idle-le*)

apply (*rule inhabits.spec.bind'[OF inhabits.spec.action.stutter]*)

using *assms* **apply** *force*

apply (*simp add: spec.bind.returnL spec.idle-le*)

apply (*rule inhabits.tau*)

apply (*simp add: spec.idle-le*)

apply *simp*

done

lemma *parallelL*:

assumes *prog.p2s P -s, xs → prog.p2s P'*

shows *prog.p2s (P || Q) -s, xs → prog.p2s (P' || Q)*

by (*simp add: prog.p2s.simps inhabits.spec.parallelL assms prog.p2s.interference-wind-bind*)

lemma *parallelR*:

assumes *prog.p2s Q -s, xs → prog.p2s Q'*

shows *prog.p2s (P || Q) -s, xs → prog.p2s (P || Q')*

by (*simp add: prog.p2s.simps inhabits.spec.parallelR assms prog.p2s.interference-wind-bind*)

setup *⟨Sign.parent-path⟩*

14 More combinators

Extra combinators:

- *prog.select* shows how we can handle arbitrary choice
- *prog.while* combinator expresses all tail-recursive computations. Its condition is a pure value.

setup $\langle \text{Sign.mandatory-path prog} \rangle$

definition $\text{select} :: 'v \text{ set} \Rightarrow ('s, 'v) \text{ prog}$ **where**
 $\text{select } X = (\bigsqcup x \in X. \text{prog.return } x)$

context

notes $[[\text{function-internals}]]$

begin

partial-function $(\text{lfp}) \text{ while} :: ('k \Rightarrow ('s, 'k + 'v) \text{ prog}) \Rightarrow 'k \Rightarrow ('s, 'v) \text{ prog}$ **where**
 $\text{while } c \ k = c \ k \gg (\lambda rv. \text{case } rv \text{ of } \text{Inl } k' \Rightarrow \text{while } c \ k' \mid \text{Inr } v \Rightarrow \text{prog.return } v)$

end

abbreviation $\text{loop} :: ('s, \text{unit}) \text{ prog} \Rightarrow ('s, 'w) \text{ prog}$ **where**
 $\text{loop } P \equiv \text{prog.while } (\lambda(). P \gg \text{prog.return } (\text{Inl } ())) ()$

abbreviation $\text{guardM} :: \text{bool} \Rightarrow ('s, \text{unit}) \text{ prog}$ **where**
 $\text{guardM } b \equiv \text{if } b \text{ then } \perp \text{ else } \text{prog.return } ()$

abbreviation $\text{unlessM} :: \text{bool} \Rightarrow ('s, \text{unit}) \text{ prog} \Rightarrow ('s, \text{unit}) \text{ prog}$ **where**
 $\text{unlessM } b \ c \equiv \text{if } b \text{ then } \text{prog.return } () \text{ else } c$

abbreviation $\text{whenM} :: \text{bool} \Rightarrow ('s, \text{unit}) \text{ prog} \Rightarrow ('s, \text{unit}) \text{ prog}$ **where**
 $\text{whenM } b \ c \equiv \text{if } b \text{ then } c \text{ else } \text{prog.return } ()$

definition $\text{app} :: ('a \Rightarrow ('s, \text{unit}) \text{ prog}) \Rightarrow 'a \text{ list} \Rightarrow ('s, \text{unit}) \text{ prog}$ **where** — Haskell's mapM -
 $\text{app } f \ xs = \text{foldr } (\lambda x \ m. f \ x \gg m) \ xs \ (\text{prog.return } ())$

definition $\text{set-app} :: ('a \Rightarrow ('s, \text{unit}) \text{ prog}) \Rightarrow 'a \text{ set} \Rightarrow ('s, \text{unit}) \text{ prog}$ **where**
 $\text{set-app } f =$
 $\text{prog.while } (\lambda X. \text{if } X = \{\} \text{ then } \text{prog.return } (\text{Inr } ()))$
 $\text{else } \text{prog.select } X \gg (\lambda x. f \ x \gg \text{prog.return } (\text{Inl } (X - \{x\})))$

primrec $\text{foldM} :: ('b \Rightarrow 'a \Rightarrow ('s, 'b) \text{ prog}) \Rightarrow 'b \Rightarrow 'a \text{ list} \Rightarrow ('s, 'b) \text{ prog}$ **where**
 $\text{foldM } f \ b \ [] = \text{prog.return } b$
 $\mid \text{foldM } f \ b \ (x \# xs) = \text{do } \{$
 $\quad b' \leftarrow f \ b \ x;$
 $\quad \text{foldM } f \ b' \ xs$
 $\}$

primrec $\text{fold-mapM} :: ('a \Rightarrow ('s, 'b) \text{ prog}) \Rightarrow 'a \text{ list} \Rightarrow ('s, 'b \text{ list}) \text{ prog}$ **where**
 $\text{fold-mapM } f \ [] = \text{prog.return } []$
 $\mid \text{fold-mapM } f \ (x \# xs) = \text{do } \{$
 $\quad y \leftarrow f \ x;$
 $\quad ys \leftarrow \text{fold-mapM } f \ xs;$
 $\quad \text{prog.return } (y \# ys)$
 $\}$

setup $\langle \text{Sign.mandatory-path select} \rangle$

lemma empty :

shows $\text{prog.select } \{\} = \perp$

by $(\text{simp add: prog.select-def})$

lemma singleton :

shows $\text{prog.select } \{x\} = \text{prog.return } x$

by $(\text{simp add: prog.select-def})$

lemma *monotone*:

shows *mono prog.select*

by (*simp add: monoI prog.select-def SUP-subset-mono*)

lemmas *strengthen[strg] = st-monotone[OF prog.select.monotone]*

lemmas *mono = monotoneD[OF prog.select.monotone, of P Q for P Q]*

lemmas *mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF prog.select.monotone, simplified, of orda P for orda P]*

lemma *Sup*:

shows *prog.select* $(\bigcup X) = (\bigsqcup_{x \in X} \text{prog.select } x)$

by (*simp add: prog.select-def flip: SUP-UNION*)

lemma *mcont*:

shows *mcont* $\bigcup (\subseteq)$ *Sup* (\leq) *prog.select*

by (*simp add: mcontI contI prog.select.monotone prog.select.Sup*)

lemmas *mcont2mcont[cont-intro] = mcont2mcont[OF prog.select.mcont, of supa orda P for supa orda P]*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path return} \rangle$

lemma *select-le*:

assumes $x \in X$

shows *prog.return* $x \leq \text{prog.select } X$

by (*simp add: assms prog.select-def SUP-upper*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemma *selectL*:

shows *prog.select* $X \ggg g = (\bigsqcup_{x \in X} g \ x)$

by (*simp add: prog.select-def prog.bind.SUPL prog.bind.returnL*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path while} \rangle$

lemma *bot*:

shows *prog.while* $\perp = \perp$

by (*simp add: fun-eq-iff prog.while.simps prog.bind.botL*)

lemma *monotone*: — could hope to prove this with a *strengthen* rule for *lfp.fixp-fun*

shows *mono* $(\lambda P. \text{prog.while } P \ s)$

by (*rule monoI*)

(*induct arbitrary: s rule: prog.while.fixp-induct; simp add: prog.bind.mono le-funD split: sum.split*)

lemmas *strengthen[strg] = st-monotone[OF prog.while.monotone]*

lemmas *mono' = monotoneD[OF prog.while.monotone, of P Q for P Q]* — compare with *prog.while.mono*

lemmas *mono2mono[cont-intro, partial-function-mono] = monotone2monotone[OF prog.while.monotone, simplified, of orda P for orda P]*

lemma *Sup-le*:

shows $(\bigsqcup_{P \in X} \text{prog.while } P \ s) \leq \text{prog.while } (\bigsqcup X) \ s$

by (*simp add: SUP-le-iff SupI prog.while.mono'*)

lemma *Inf-le*:

shows $\text{prog.while } (\sqcap X) s \leq (\sqcap P \in X. \text{prog.while } P s)$
by (*simp add: le-INF-iff Inf-lower prog.while.mono*)

lemma *True-skip-eq-bot*:

shows $\text{prog.while } \langle \text{prog.return } (\text{Inl } x) \rangle s = \perp$
by (*induct arbitrary: s rule: prog.while.fixp-induct (simp-all add: prog.bind.returnL)*)

lemma *Inr-eq-return*:

shows $\text{prog.while } \langle \text{prog.return } (\text{Inr } v) \rangle s = \text{prog.return } v$
by (*subst prog.while.simps (simp add: prog.bind.returnL)*)

lemma *kleene-star*:

shows $\text{prog.kleene.star } P$
 $= \text{prog.while } (\lambda-. (P \gg \text{prog.return } (\text{Inl } ())) \sqcup \text{prog.return } (\text{Inr } ())) () (\text{is } ?lhs = ?rhs)$
proof(*rule antisym*)
show $?lhs \leq ?rhs$
proof(*induct rule: prog.kleene.star.fixp-induct[case-names adm bot step]*)
case (*step P*) **then show** *?case*
by (*subst prog.while.simps (simp add: prog.bind.supL prog.bind.bind prog.bind.mono sup.coboundedI1 prog.bind.returnL)*)
qed *simp-all*
show $?rhs \leq ?lhs$
proof(*induct rule: prog.while.fixp-induct[case-names adm bot step]*)
case (*step k*) **then show** *?case*
by (*subst prog.kleene.star.simps (simp add: prog.bind.supL prog.bind.bind prog.bind.mono prog.bind.returnL le-supI1)*)
qed *simp-all*
qed

lemma *invmap-le*:

fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
shows $\text{prog.while } (\lambda k. \text{prog.invmap } sf (\text{map-sum } id \text{ } vf) (c \ k)) \ k$
 $\leq \text{prog.invmap } sf \text{ } vf (\text{prog.while } c \ k) (\text{is } ?lhs \text{ } \text{prog.while } k \leq ?rhs \ k)$
proof(*rule spec[where x=k]*),
induct rule: prog.while.fixp-induct[where P= $\lambda R. \forall k. ?lhs \ R \ k \leq ?rhs \ k$, case-names adm bot step])
case (*step k*) **show** *?case*
apply (*subst prog.while.simps*)
apply (*strengthen ord-to-strengthen(1)[OF step[rule-format]]*)
apply (*auto intro!: SUPE prog.bind.mono[OF order.refl]*)
split: sum.splits
simp: prog.invmap.bind prog.invmap.return
prog.invmap.split-vinvmap[where sf=sf and vf=map-sum id vf]
prog.bind.bind prog.bind.return prog.bind.SUPL
SUP-upper
order.trans[OF - prog.bind.mono[OF prog.return.rel-le order.refl]]
done
qed (*simp-all add: prog.invmap.bot*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path loop} \rangle$

lemma *bindL*:

fixes $P :: ('s, \text{unit}) \text{ prog}$
fixes $Q :: ('s, 'w) \text{ prog}$

shows $prog.loop P \gg Q = prog.loop P$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*rule prog.while.fixp-induct*[**where** $P=\lambda R. R (\lambda(). P \gg prog.return (Inl ())) () \gg Q \leq ?rhs$]; *simp add: prog.bind.botL*)
(subst prog.while.simps; simp add: prog.bind.bind prog.bind.mono lambda-unit-futzery prog.bind.returnL)
show $?rhs \leq ?lhs$
by (*rule prog.while.fixp-induct*[**where** $P=\lambda R. R (\lambda(). P \gg prog.return (Inl ())) () \leq ?lhs$]; *simp*)
(subst prog.while.simps; simp add: prog.bind.bind prog.bind.mono lambda-unit-futzery prog.bind.returnL)
qed

lemma parallel-le:
shows $prog.loop P \leq lfp (\lambda R. P \parallel R)$
proof(*induct rule: prog.while.fixp-induct*[*case-names adm bot step*])
case (*step k*) **show** $?case$
apply (*subst lfp-unfold, simp*)
apply (*strengthen ord-to-strengthen*[*OF prog.bind.parallel-le*])
apply (*simp add: prog.bind.bind prog.bind.mono prog.bind.returnL step*)
done
qed *simp-all*

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path foldM \rangle$

lemma append:
shows $prog.foldM f b (xs @ ys) = prog.foldM f b xs \gg (\lambda b'. prog.foldM f b' ys)$
by (*induct xs arbitrary: b*) (*simp-all add: prog.bind.returnL prog.bind.bind*)

setup $\langle Sign.parent-path \rangle$

lemma foldM-alt-def:
shows $prog.foldM f b xs = foldr (\lambda x m. prog.bind m (\lambda b. f b x)) (rev xs) (prog.return b)$
by (*induct xs arbitrary: b rule: rev-induct*) (*simp-all add: prog.foldM.append prog.bind.returnR*)

setup $\langle Sign.mandatory-path fold-mapM \rangle$

lemma bot:
shows $prog.fold-mapM \perp = (\lambda xs. case xs of [] \Rightarrow prog.return [] \mid - \Rightarrow \perp)$
by (*simp add: fun-eq-iff prog.bind.botL split: list.split*)

lemma append:
shows $prog.fold-mapM f (xs @ ys)$
 $= prog.fold-mapM f xs \gg (\lambda xs. prog.fold-mapM f ys \gg (\lambda ys. prog.return (xs @ ys)))$
by (*induct xs*) (*simp-all add: prog.bind.bind prog.bind.returnL prog.bind.returnR*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path app \rangle$

lemma bot:
shows $prog.app \perp = (\lambda xs. case xs of [] \Rightarrow prog.return () \mid - \Rightarrow \perp)$
and $prog.app (\lambda -. \perp) = (\lambda xs. case xs of [] \Rightarrow prog.return () \mid - \Rightarrow \perp)$
by (*simp-all add: fun-eq-iff prog.app-def prog.bind.botL split: list.split*)

lemma Nil:
shows $prog.app f [] = prog.return ()$
by (*simp add: prog.app-def*)

lemma *Cons*:

shows $prog.app\ f\ (x\ \# \ xs) = f\ x \gg prog.app\ f\ xs$
by (*simp add: prog.app-def*)

lemmas *simps* =

prog.app.bot
prog.app.Nil
prog.app.Cons

lemma *append*:

shows $prog.app\ f\ (xs\ @\ ys) = prog.app\ f\ xs \gg prog.app\ f\ ys$
by (*induct xs arbitrary: ys*) (*simp-all add: prog.app.simps prog.bind.returnL prog.bind.bind*)

lemma *monotone*:

shows *mono* $(\lambda f. prog.app\ f\ xs)$
by (*induct xs*) (*simp-all add: prog.app.simps le-fun-def monotone-on-def prog.bind.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.app.monotone*]

lemmas *mono* = *monotoneD*[*OF prog.app.monotone*]

lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF prog.app.monotone, simplified, of orda P for orda P*]

lemma *Sup-le*:

shows $(\bigsqcup f \in X. prog.app\ f\ xs) \leq prog.app\ (\bigsqcup X)\ xs$
by (*simp add: SUP-le-iff SupI prog.app.mono*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path invmap \rangle$

lemma *app*:

fixes *sf* :: 's \Rightarrow 't
fixes *vf* :: 'v \Rightarrow unit
shows $prog.invmap\ sf\ vf\ (prog.app\ f\ xs)$
 $= prog.app\ (\lambda x. prog.sinvmap\ sf\ (f\ x))\ xs \gg prog.invmap\ sf\ vf\ (prog.return\ ())$
by (*induct xs*)
(*simp-all add: prog.app.simps prog.bind.return prog.invmap.bind prog.bind.bind id-def*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path sinvmap \rangle$

lemma *app-le*:

fixes *sf* :: 's \Rightarrow 't
fixes *vf* :: 'v \Rightarrow unit
shows $prog.app\ (\lambda x. prog.sinvmap\ sf\ (f\ x))\ xs \leq prog.sinvmap\ sf\ (prog.app\ f\ xs)$
by (*simp add: prog.invmap.app prog.invmap.return prog.bind.return*
order.trans[OF - prog.bind.mono[OF order.refl prog.return.rel-le]])

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path set-app \rangle$

lemma *bot*:

shows $X \neq \{\} \implies prog.set-app\ \perp\ X = \perp$
and $X \neq \{\} \implies prog.set-app\ (\lambda-. \perp)\ X = \perp$
by (*simp-all add: prog.set-app-def prog.while.simps prog.bind.bind prog.bind.botL prog.bind.selectL*)

lemma *empty*:

shows $prog.set\text{-}app\ f\ \{\} = prog.return\ ()$

by (*simp add: prog.set-app-def prog.while.simps prog.bind.returnL*)

lemma *not-empty*:

assumes $X \neq \{\}$

shows $prog.set\text{-}app\ f\ X = prog.select\ X \gg (\lambda x. f\ x \gg prog.set\text{-}app\ f\ (X - \{x\}))$

using *assms by (simp add: prog.set-app-def prog.while.simps prog.bind.returnL prog.bind.bind)*

lemmas *simps =*

prog.set-app.bot

prog.set-app.empty

prog.set-app.not-empty

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle Sign.mandatory\text{-}path\ app \rangle$

lemma *set-app-le*:

assumes $X = set\ xs$

assumes *distinct xs*

shows $prog.app\ f\ xs \leq prog.set\text{-}app\ f\ X$

using *assms*

proof(*induct xs arbitrary: X*)

case (*Cons x xs*) **then show** *?case*

apply (*simp add: prog.set-app.simps prog.app.simps*)

apply (*strengthen ord-to-strengthen(2)[OF prog.return.select-le[of x]], blast*)

apply (*simp add: prog.bind.returnL prog.bind.mono*)

done

qed (*simp add: prog.app.simps prog.set-app.simps*)

setup $\langle Sign.parent\text{-}path \rangle$

lemma *set-app-alt-def*:

assumes *finite X*

shows $prog.set\text{-}app\ f\ X = (\bigsqcup xs \in \{ys. set\ ys = X \wedge distinct\ ys\}. prog.app\ f\ xs)$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

from *assms show ?lhs \leq ?rhs*

proof(*induct rule: finite-remove-induct*)

case (*remove X*)

from $\langle finite\ X \rangle \langle X \neq \{\} \rangle$ **have** $*$: $\{ys. set\ ys = X - \{x\} \wedge distinct\ ys\} \neq \{\}$ **for** *x*

by (*simp add: finite-distinct-list*)

from $\langle X \neq \{\} \rangle$ **show** *?case*

apply (*clarsimp simp: prog.set-app.simps prog.bind.selectL*)

apply (*strengthen ord-to-strengthen[OF remove.hyps(4)], blast*)

apply (*fastforce simp: prog.app.simps prog.bind.SUPR-not-empty[OF *] Sup-le-iff*
intro: rev-SUPI[where $x=x \# xs$ for $x\ xs$])

done

qed (*simp add: prog.set-app.simps prog.app.simps*)

show *?rhs \leq ?lhs*

by (*simp add: Sup-le-iff prog.app.set-app-le*)

qed

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle \text{Sign.mandatory-path ag.prog} \rangle$

lemma *select-sp*:

assumes $\bigwedge s x. \llbracket P s; x \in X \rrbracket \implies Q x s$

assumes $\bigwedge v. \text{stable } A (P \wedge Q v)$

shows $\text{prog.p2s } (\text{prog.select } X) \leq \llbracket P \rrbracket, A \vdash G, \llbracket \lambda v. P \wedge Q v \rrbracket$

by (*clarsimp simp: prog.select-def prog.p2s.Sup spec.interference.cl.bot ag.spec.term.none-inteference*)
(*rule ag.pre[OF ag.prog.return[OF assms(2)]]; blast intro: assms(1)*)

lemma *while*:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$

assumes $c: \bigwedge k. \text{prog.p2s } (c k) \leq \llbracket P k \rrbracket, A \vdash G, \llbracket \text{case-sum } I Q \rrbracket$

assumes $IP: \bigwedge s v. I v s \implies P v s$

assumes $sQ: \bigwedge v. \text{stable } A (Q v)$

shows $\text{prog.p2s } (\text{prog.while } c k) \leq \llbracket I k \rrbracket, A \vdash G, \llbracket Q \rrbracket$

proof –

have $\text{prog.p2s } (\text{prog.while } c k) \leq \llbracket P k \rrbracket, A \vdash G, \llbracket Q \rrbracket$

proof(*induct arbitrary: k rule: prog.while.fixp-induct[case-names adm bot step]*)

case (*step k*) **show** *?case*

apply (*rule ag.prog.bind[OF - c]*)

apply (*rule ag.pre-pre[OF ag.prog.case-sum[OF step ag.prog.return[OF sQ]]]*)

apply (*simp add: IP split: sum.splits*)

done

qed (*simp-all add: ag.prog.galois*)

then show *?thesis*

by (*meson IP ag.pre-pre*)

qed

lemma *app*:

fixes $xs :: 'a \text{ list}$

fixes $f :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$

fixes $P :: 'a \text{ list} \Rightarrow 's \text{ pred}$

assumes $\bigwedge x ys zs. xs = ys @ x \# zs \implies \text{prog.p2s } (f x) \leq \llbracket P ys \rrbracket, A \vdash G, \llbracket \lambda -. P (ys @ [x]) \rrbracket$

assumes $\bigwedge ys. \text{prefix } ys xs \implies \text{stable } A (P ys)$

shows $\text{prog.p2s } (\text{prog.app } f xs) \leq \llbracket P [] \rrbracket, A \vdash G, \llbracket \lambda -. P xs \rrbracket$

using *assms*

by (*induct xs rule: rev-induct;*

fastforce intro: ag.prog.bind ag.prog.return

simp: prog.app.append prog.bind.returnR prog.app.simps)

lemma *app-set*:

fixes $X :: 'a \text{ set}$

fixes $f :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$

fixes $P :: 'a \text{ set} \Rightarrow 's \text{ pred}$

assumes $\bigwedge Y x. \llbracket Y \subseteq X; x \in X - Y \rrbracket \implies \text{prog.p2s } (f x) \leq \llbracket P Y \rrbracket, A \vdash G, \llbracket \lambda -. P (\text{insert } x Y) \rrbracket$

assumes $\bigwedge Y. Y \subseteq X \implies \text{Stability.stable } A (P Y)$

shows $\text{prog.p2s } (\text{prog.set-app } f X) \leq \llbracket P \{\} \rrbracket, A \vdash G, \llbracket \lambda -. P X \rrbracket$

proof –

have $*$: $X - (Y - \{x\}) = \text{insert } x (X - Y)$ **if** $Y \subseteq X$ **and** $x \in Y$ **for** x **and** $X Y :: 'a \text{ set}$

using *that by blast*

show *?thesis*

unfolding *prog.set-app-def*

apply (*rule ag.prog.while[where $I = \lambda Y s. Y \subseteq X \wedge P (X - Y) s$ and $Q = \langle P X \rangle$ and $k = X$, simplified]*)

apply (*rename-tac k*)

apply (*rule ag.prog.if*)

apply (*rule ag.prog.return*)

apply (*simp add: assms; fail*)

apply (*rule-tac $P = \lambda s. k \subseteq X \wedge P (X - k) s$ in ag.prog.bind[rotated]*)

```

apply (rule-tac  $Q = \lambda x s. x \in k$  in ag.prog.select-sp, assumption)
apply (simp add: assms(2) stable.conjI stable.const; fail)
apply (intro ag.gen-asm)
apply (rule ag.prog.bind[rotated])
apply (rule assms(1); force)
apply (rule ag.pre-pre[OF ag.prog.return])
apply (simp add: assms(2) stable.conjI stable.const; fail)
using * apply fastforce
apply force
apply (simp add: assms(2))
done
qed

```

lemma *foldM*:

```

fixes  $xs :: 'a$  list
fixes  $f :: 'b \Rightarrow 'a \Rightarrow ('s, 'b)$  prog
fixes  $I :: 'b \Rightarrow 'a \Rightarrow 's$  pred
fixes  $P :: 'b \Rightarrow 's$  pred
assumes  $f: \bigwedge b x s. x \in \text{set } xs \Longrightarrow \text{prog.p2s } (f b x) \leq \{I b x\}, A \vdash G, \{P\}$ 
assumes  $P: \bigwedge b x s. \llbracket P b s; x \in \text{set } xs \rrbracket \Longrightarrow I b x s$ 
assumes  $sP: \bigwedge b. \text{stable } A (P b)$ 
shows  $\text{prog.p2s } (\text{prog.foldM } f b xs) \leq \{P b\}, A \vdash G, \{P\}$ 
using  $f P$  by (induct xs arbitrary: b) (fastforce intro!: ag.prog.bind intro: ag.pre-pre ag.prog.return[OF sP])+

```

setup $\langle \text{Sign.parent-path} \rangle$

15 Structural local state

15.1 *spec.local*

We develop a few combinators for structural local state. The goal is to encapsulate a local state of type $'ls$ in a process $('a \text{ agent}, 'ls \times 's, 'v)$ *spec*. Applying *spec.smap snd* yields a process of type $('a \text{ agent}, 's, 'v)$ *spec*. We also constrain environment steps to not affect $'ls$, yielding a plausible data refinement rule (see §15.6.1).

abbreviation (*input*) *localize1* $:: ('b \Rightarrow 's \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'ls \times 's \Rightarrow 'a$ **where**
localize1 $f b s \equiv f b (\text{snd } s)$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

setup $\langle \text{Sign.mandatory-path local} \rangle$

definition *qrm* $:: ('a \text{ agent}, 'ls \times 's)$ *steps where* — cf *ag.assm*
 $qrm = \text{range } \text{proc} \times \text{UNIV} \cup \{\text{env}\} \times (\text{Id} \times_R \text{UNIV})$

abbreviation (*input*) *interference* $\equiv \text{spec.rel spec.local.qrm}$

setup $\langle \text{Sign.parent-path} \rangle$

definition *local* $:: ('a \text{ agent}, 'ls \times 's, 'v)$ *spec* $\Rightarrow ('a \text{ agent}, 's, 'v)$ *spec where*
 $\text{local } P = \text{spec.smap snd } (\text{spec.local.interference} \sqcap P)$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma *local-le-conv*:

```

shows  $\langle \sigma \rangle \leq \text{spec.local } P$ 
 $\longleftrightarrow (\exists \sigma'. \langle \sigma' \rangle \leq P$ 
 $\quad \wedge \text{trace.steps } \sigma' \subseteq \text{spec.local.qrm}$ 
 $\quad \wedge \langle \sigma \rangle \leq \langle \text{trace.map id snd id } \sigma' \rangle)$ 

```

by (simp add: spec.local-def spec.singleton.le-conv ac-simps)

setup <Sign.parent-path>

setup <Sign.mandatory-path idle>

lemma local-le[spec.idle-le]: — Converse does not hold

assumes spec.idle \leq P

shows spec.idle \leq spec.local P

by (simp add: spec.local-def assms spec.idle-le)

setup <Sign.parent-path>

setup <Sign.mandatory-path local>

setup <Sign.mandatory-path qrm>

lemma refl:

shows refl (spec.local.qrm “ {a}”)

by (simp add: spec.local.qrm-def refl-onI)

lemma member:

shows (proc a, s, s') \in spec.local.qrm

and (env, s, s') \in spec.local.qrm \longleftrightarrow fst s = fst s'

by (auto simp: spec.local.qrm-def)

lemma inter:

shows UNIV \times Id \cap spec.local.qrm = UNIV \times Id

and spec.local.qrm \cap UNIV \times Id = UNIV \times Id

and spec.local.qrm \cap {self} \times Id = {self} \times Id

and spec.local.qrm \cap {env} \times UNIV = {env} \times (Id \times_R UNIV)

and spec.local.qrm \cap {env} \times (UNIV \times_R Id) = {env} \times Id

and spec.local.qrm \cap A \times (Id \times_R r) = A \times (Id \times_R r)

by (auto simp: spec.local.qrm-def)

lemmas simps[simp] =

spec.local.qrm.refl

spec.local.qrm.member

spec.local.qrm.inter

setup <Sign.parent-path>

setup <Sign.mandatory-path interference>

lemma smap-snd:

shows spec.smap snd spec.local.interference = \top

by (subst spec.map.rel)

(auto simp: spec.local.qrm-def spec.rel.UNIV

image-Un map-prod-image-Times map-prod-image-relprod map-prod-surj

simp flip: Sigma-Un-distrib1)

setup <Sign.parent-path>

lemma inf-interference:

shows spec.local P = spec.local (P \sqcap spec.local.interference)

by (simp add: spec.local-def ac-simps)

lemma bot:

shows $\text{spec.local } \perp = \perp$
by (*simp add: spec.local-def spec.map.bot*)

lemma top:
shows $\text{spec.local } \top = \top$
by (*simp add: spec.local-def spec.local.interference.smap-snd*)

lemma monotone:
shows *mono spec.local*
proof(*rule monotoneI*)
show $\text{spec.local } P \leq \text{spec.local } P'$ **if** $P \leq P'$ **for** $P P' :: ('a \text{ agent}, 's \times 'ls, 'v) \text{ spec}$
unfolding *spec.local-def* **by** (*strengthen ord-to-strengthen(1)[OF ‹P ≤ P'›]*) *simp*
qed

lemmas *strengthen[strg] = st-monotone[OF spec.local.monotone]*
lemmas *mono = monotoneD[OF spec.local.monotone]*
lemmas *mono2mono[cont-intro, partial-function-mono]*
 $= \text{monotone2monotone}[OF \text{spec.local.monotone, simplified, of orda } P \text{ for orda } P]$

lemma Sup:
shows $\text{spec.local } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.local } x)$
by (*simp add: spec.local-def inf-Sup spec.map.Sup image-image*)

lemmas $\text{sup} = \text{spec.local.Sup}[\text{where } X = \{X, Y\} \text{ for } X Y, \text{simplified}]$

lemma mcont2mcont[cont-intro]:
assumes *mcont luba orda Sup (≤) P*
shows *mcont luba orda Sup (≤) (λx. spec.local (P x))*
by (*simp add: spec.local-def assms*)

lemma idle:
shows $\text{spec.local spec.idle} = \text{spec.idle}$
by (*simp add: spec.local-def inf.absorb2[OF spec.idle.rel-le] spec.map.idle*)

lemma action:
fixes $F :: ('v \times 'a \text{ agent} \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
shows $\text{spec.local } (\text{spec.action } F)$
 $= \text{spec.action } (\text{map-prod id } (\text{map-prod id } (\text{map-prod snd snd})) \text{ ‘}$
 $\quad (F \cap \text{UNIV} \times \text{spec.local.qrm}))$
by (*simp add: spec.local-def spec.action.inf-rel spec.map.surj-sf-action*)

lemma return:
shows $\text{spec.local } (\text{spec.return } v) = \text{spec.return } v$
by (*simp add: spec.return-def spec.local.action*
 $\quad \text{Times-Int-Times map-prod-image-Times map-prod-snd-snd-image-Id}$)

lemma bind-le: — Converse does not hold
shows $\text{spec.local } (f \ggg g) \leq \text{spec.local } f \ggg (\lambda v. \text{spec.local } (g v))$
by (*simp add: spec.local-def spec.bind.inf-rel spec.map.bind-le*)

lemma interference:
shows $\text{spec.local } (\text{spec.rel } (\{env\} \times \text{UNIV})) = \text{spec.rel } (\{env\} \times \text{UNIV})$
by (*simp add: spec.local-def spec.map.rel map-prod-image-Times map-prod-image-relprod*
 $\quad \text{flip: spec.rel.inf}$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path map} \rangle$

lemma *local-le*:

shows $\text{spec.map id sf vf (spec.local P)} \leq \text{spec.local (spec.map id (map-prod id sf) vf P)}$
by (*fastforce intro: spec.map.mono inf.mono spec.rel.mono*
simp: spec.local-def spec.map.comp spec.map.inf-rel spec.local.qrm-def)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path vmap} \rangle$

lemma *local*:

shows $\text{spec.vmap vf (spec.local P)} = \text{spec.local (spec.vmap vf P)}$
by (*simp add: spec.local-def spec.map.comp spec.map.inf-rel spec.rel.reflcl(1)[where A=UNIV]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path invmap} \rangle$

lemma *smap-snd*:

fixes $P :: ('a, 'ls \times 't, 'w) \text{ spec}$
fixes $\text{sf} :: 's \Rightarrow 't$
fixes $\text{vf} :: 'v \Rightarrow 'w$
shows $\text{spec.invmap id sf vf (spec.smap snd P)}$
 $= \text{spec.smap snd (spec.invmap id (map-prod id sf) vf P) (is ?lhs = ?rhs)}$

proof(*rule spec.singleton.antisym*)

have *smap-snd-aux*:

$\exists \text{zs. trace.natural}' (ls, sf s) xs = \text{trace.natural}' (ls, sf s) (\text{map (map-prod id (map-prod id sf)) zs})$
 $\wedge \text{trace.natural}' s (\text{map (map-prod id snd) zs}) = \text{trace.natural}' s ys (\text{is } \exists \text{zs. } ?P \text{ } ls \text{ } s \text{ } ys \text{ } zs)$
if $\text{trace.natural}' (sf s) (\text{map (map-prod id sf) ys}) = \text{trace.natural}' (sf s) (\text{map (map-prod id snd) xs})$
for ls **and** $s :: 's$ **and** $\text{sf} :: 's \Rightarrow 't$ **and** $xs :: ('a \times 'ls \times 't) \text{ list}$ **and** $ys :: ('a \times 's) \text{ list}$
using *that*

proof(*induct xs arbitrary: ls s ys*)

case (*Nil ls s ys*) **then show** *?case*

by (*fastforce intro: exI[where x=map (map-prod id (Pair ls)) ys]*
simp: comp-def trace.natural'.eq-Nil-conv)

next

case (*Cons x xs ls s ys*) **show** *?case*

proof(*cases snd (snd x) = sf s*)

case *True with Cons.premis show ?thesis*

by (*cases x*) (*fastforce dest: Cons.hyps[where ls=fst (snd x)]*
intro: exI[where x=(fst x, fst (snd x), s) # zs for zs]
simp flip: id-def)

next

case *False*

with *Cons.premis obtain a s_x us s'* *zs*

where $x = (a, s_x, sf s')$

and $\text{sf } s' \neq \text{sf } s$

and $\text{snd } \text{'map-prod id sf ' set us} \subseteq \{\text{sf } s\}$

and $ys = us @ (a, s') \# zs$

and $\text{trace.natural}' (sf s') (\text{map (map-prod id sf) zs}) = \text{trace.natural}' (sf s') (\text{map (map-prod id snd) xs})$

by (*cases x*) (*clarsimp simp: trace.natural'.eq-Cons-conv map-eq-append-conv simp flip: id-def*)

with *False show ?thesis*

by (*fastforce simp: comp-def trace.natural'.append image-subset-iff trace.natural'.eq-Nil-conv*
intro: exI[where x=map (map-prod id (Pair ls)) us @ (a, (s_x, s')) # zs for zs]
dest: Cons.hyps[where ls=fst (snd x)])

qed

qed

fix σ

```

assume  $\langle \sigma \rangle \leq ?lhs$ 
then obtain  $ls\ xs\ v\ i$ 
  where  $*$ :  $\langle (ls, sf\ (trace.init\ \sigma)), xs, v \rangle \leq P$ 
    and  $**$ :  $trace.natural'\ (sf\ (trace.init\ \sigma))\ (map\ (map\text{-}prod\ id\ sf)\ (trace.rest\ \sigma))$ 
      =  $trace.natural'\ (sf\ (trace.init\ \sigma))\ (map\ (map\text{-}prod\ id\ snd)\ (take\ i\ xs))$ 
    and  $***$ : if  $i \leq length\ xs$  then  $trace.term\ \sigma = None$  else  $map\text{-}option\ vf\ (trace.term\ \sigma) = v$ 
  apply (clarsimp simp: spec.singleton.le-conv spec.singleton-le-conv)
  apply (erule trace.less-eq-takeE)
  apply (erule trace.take.naturalE)
  apply (clarsimp simp: trace.take.map trace.natural-def trace.split-all not-le
    split: if-split-asm)
  apply (metis order.strict-iff-not take-all)
  done
from smap-snd-aux[OF **] obtain  $zs$ 
  where  $trace.natural'\ (ls, sf\ (trace.init\ \sigma))\ (take\ i\ xs)$ 
    =  $trace.natural'\ (ls, sf\ (trace.init\ \sigma))\ (map\ (map\text{-}prod\ id\ (map\text{-}prod\ id\ sf))\ zs)$ 
  and  $trace.natural'\ (trace.init\ \sigma)\ (map\ (map\text{-}prod\ id\ snd)\ zs)$ 
    =  $trace.natural'\ (trace.init\ \sigma)\ (trace.rest\ \sigma)$ 
  by blast
with  $*$   $***$  show  $\langle \sigma \rangle \leq ?rhs$ 
  apply –
  unfolding spec.singleton.le-conv
  apply (rule exI[where  $x=trace.T\ (ls, trace.init\ \sigma)\ zs$  (if  $i \leq length\ xs$  then  $None$  else  $trace.term\ \sigma$ )])
  apply (clarsimp simp: spec.singleton-le-conv trace.natural-def trace.less-eq-None
    split: if-splits
    elim!: order.trans[rotated])
  apply (metis append-take-drop-id prefixI trace.natural'.append)
  done
next
  show  $\langle \sigma \rangle \leq ?lhs$  if  $\langle \sigma \rangle \leq ?rhs$  for  $\sigma$ 
  using that
  by (fastforce dest: spec.singleton.map-le[where  $af=id$  and  $sf=sf$  and  $vf=vf$ ]
    simp: spec.singleton.le-conv)
qed

lemma local:
  fixes  $P :: ('a\ agent, 'ls \times 't, 'v)\ spec$ 
  fixes  $sf :: 's \Rightarrow 't$ 
  shows  $spec.invmap\ id\ sf\ vf\ (spec.local\ P) = spec.local\ (spec.invmap\ id\ (map\text{-}prod\ id\ sf)\ vf\ P)$ 
by (auto simp: spec.local-def spec.local.qrm-def ac-simps
  spec.invmap.smap-snd spec.invmap.inf spec.invmap.rel
  intro!: arg-cong[where  $f=\lambda r. spec.smap\ snd\ (spec.invmap\ id\ (map\text{-}prod\ id\ sf)\ vf\ P \sqcap spec.rel\ r)$ ])

setup  $\langle Sign.parent\text{-}path \rangle$ 

setup  $\langle Sign.mandatory\text{-}path\ term \rangle$ 

setup  $\langle Sign.mandatory\text{-}path\ none \rangle$ 

lemma local:
  shows  $spec.term.none\ (spec.local\ P) = spec.local\ (spec.term.none\ P)$ 
by (simp add: spec.local-def spec.term.none.inf spec.term.none.inf-none-rel spec.term.none.map)

setup  $\langle Sign.parent\text{-}path \rangle$ 

setup  $\langle Sign.mandatory\text{-}path\ all \rangle$ 

lemma local:

```

shows $spec.term.all (spec.local P) = spec.local (spec.term.all P)$
by (*simp add: spec.local-def spec.term.all.map spec.term.all.rel spec.term.all.inf*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path closed \rangle$

lemma *local*:

assumes $P \in spec.term.closed$ -
shows $spec.local P \in spec.term.closed$ -
by (*rule spec.term.closed-clI*)
(simp add: spec.term.all.local spec.term.all.monomorphic
flip: spec.term.closed-conv[OF assms, simplified])

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

15.2 Local state transformations

We want to reorder, introduce and eliminate actions that affect local state while preserving observable behaviour under *spec.local*.

The closure that arises from *spec.local*, i.e.:

lemma

defines $cl \equiv spec.map-invmap.cl - - id snd id$
assumes $spec.local.interference \sqcap P$
 $\leq cl (spec.local.interference \sqcap Q)$
shows $spec.local P \leq spec.local Q$
unfolding *spec.local-def*
by (*strengthen ord-to-strengthen(1)[OF assms(2)]*)
(simp add: spec.map-invmap.galois cl-def spec.map-invmap.cl-def)

expresses all transformations, but does not decompose over (\gg); in other words we do not have $cl f \gg (\lambda v. cl (g v)) \leq cl (f \gg g)$ as the local states that $cl f$ terminates with may not satisfy g . (Observe that we do not expect the converse to hold as then all local states would need to be preserved.)

We therefore define a closure that preserves the observable state and the initial and optionally final (if terminating) local states via a projection:

setup $\langle Sign.mandatory-path seq-ctxt \rangle$

definition $prj :: bool \Rightarrow ('a, 'ls \times 's, 'v) trace.t \Rightarrow ('a, 's, 'v) trace.t \times 'ls \times 'ls option$ **where**
 $prj T \sigma = (\mathfrak{h}(trace.map id snd id \sigma),$
 $fst (trace.init \sigma),$
 $if T then map-option \langle fst (trace.final \sigma) \rangle (trace.term \sigma) else None)$

setup $\langle Sign.mandatory-path prj \rangle$

lemma *natural*:

shows $seq-ctxt.prj T (\mathfrak{h}\sigma) = seq-ctxt.prj T \sigma$
by (*simp add: seq-ctxt.prj-def trace.natural.map-natural*)

lemma *idle*:

shows $seq-ctxt.prj T (trace.T s \sqcap None) = (trace.T (snd s) \sqcap None, fst s, None)$
by (*simp add: seq-ctxt.prj-def trace.natural.simps*)

lemmas $simps[simp] =$

```

seq-ctxt.prj.natural

setup ⟨Sign.parent-path⟩

setup ⟨Sign.parent-path⟩

interpretation seq-ctxt: galois.image-vimage seq-ctxt.prj T for T .

setup ⟨Sign.mandatory-path seq-ctxt.equivalent⟩

lemma partial-sel-equivE:
  assumes seq-ctxt.equivalent T σ1 σ2
  obtains trace.init σ1 = trace.init σ2
    and trace.term σ1 = trace.term σ2
    and [[T; ∃ v. trace.term σ1 = Some v]] ⇒ trace.final σ1 = trace.final σ2
using assms
by (cases σ1)
  (force intro: prod-eqI
    simp: seq-ctxt.prj-def trace.natural.trace-conv
    simp flip: trace.final'.map[where af=id and sf=snd]
    cong: trace.final'.natural'-cong)

lemma downwards-existsE:
  assumes σ1' ≤ σ1
  assumes seq-ctxt.equivalent T σ1 σ2
  obtains σ2'
    where σ2' ≤ σ2
    and seq-ctxt.equivalent T σ1' σ2'
using assms
apply atomize-elim
apply (clarsimp simp: seq-ctxt.prj-def)
apply (rule trace.natural.less-eqE[OF trace.map.mono sym], assumption, assumption)
apply (clarsimp split: if-split-asm)
apply (cases trace.term σ1')
apply (fastforce simp: trace.natural-def elim: trace.less-eqE trace.map.less-eqR)+
done

lemma downwards-existsE2:
  assumes σ1' ≤ σ1
  assumes seq-ctxt.equivalent T σ1' σ2'
  obtains σ2
    where σ2' ≤ σ2
    and seq-ctxt.equivalent T σ1 σ2
proof(atomize-elim, use ⟨σ1' ≤ σ1⟩ in ⟨induct rule: trace.less-eqE⟩)
  case prefix
  from prefix(3) obtain zs
    where *: σ1 = trace.T (trace.init σ1) (trace.rest σ1' @ zs) (trace.term σ1)
    by (cases σ1) (auto elim: prefixE)
  show ?case
  proof(cases trace.term σ1)
  case None with assms(2) prefix(1,2) * show ?thesis
    by (cases σ1, cases σ2')
      (fastforce intro!: exI[where x=trace.T (trace.init σ1) (trace.rest σ2' @ zs) (trace.term σ1)]
        simp: seq-ctxt.prj-def trace.natural-def
          trace.natural'.append trace.less-eq-same-append-conv
          cong: trace.final'.natural'-cong)
  next
  case (Some v)

```

```

from assms(2) prefix(2)
have snd (trace.final  $\sigma_1'$ ) = trace.final (trace.map id snd id  $\sigma_2'$ )
  by (cases  $\sigma_1'$ , cases  $\sigma_2'$ )
    (clarsimp simp: seq-ctxt.prj-def trace.natural-def trace.final'.map
      dest!: arg-cong[where f= $\lambda$ xs. trace.final' (snd (trace.init  $\sigma_1$ )) xs])
with Some assms(2) prefix(1,2) * show ?thesis
  apply (cases  $\sigma_1$ )
  apply (cases  $\sigma_2'$ )
apply (rule exI[where x=trace.T (trace.init  $\sigma_1$ ) (trace.rest  $\sigma_2'$  @ (undefined, trace.final  $\sigma_1'$ ) # zs) (trace.term
 $\sigma_1$ )])
  apply (clarsimp simp: seq-ctxt.prj-def trace.natural-def trace.natural'.append trace.less-eq-same-append-conv)
  apply (clarsimp cong: trace.final'.natural'-cong)
  done
qed
next
  case (maximal v) with assms(2) show ?case
    by blast
qed

```

lemma *map-sf-eq-id*:

```

assumes seq-ctxt.equivalent True  $\sigma_1$   $\sigma_2$ 
shows seq-ctxt.equivalent True (trace.map af id vf  $\sigma_1$ ) (trace.map af id vf  $\sigma_2$ )
using assms
by (auto simp: seq-ctxt.prj-def comp-def trace.final'.map[where sf=id, simplified] trace.natural-def
  simp flip: trace.natural'.map-inj-on-sf
  dest: arg-cong[where f=map (map-prod af id)])

```

lemma *mono*:

```

assumes  $T \implies T'$ 
assumes seq-ctxt.equivalent T'  $\sigma_1$   $\sigma_2$ 
shows seq-ctxt.equivalent T  $\sigma_1$   $\sigma_2$ 
using assms by (clarsimp simp: seq-ctxt.prj-def)

```

lemma *append*:

```

assumes seq-ctxt.equivalent True (trace.T s xs (Some v)) (trace.T s' xs' v')
assumes seq-ctxt.equivalent T (trace.T (trace.final' s xs) ys w) (trace.T t' ys' w')
shows seq-ctxt.equivalent T (trace.T s (xs @ ys) w) (trace.T s' (xs' @ ys') w')
using assms
by (clarsimp simp: seq-ctxt.prj-def trace.natural-def trace.natural'.append
  simp flip: trace.final'.map[where af=id and sf=snd]
  cong: trace.final'.natural'-cong if-cong)
  (simp; metis trace.final'.map[where af=id and sf=fst])

```

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path spec* \rangle

setup \langle *Sign.mandatory-path seq-ctxt* \rangle

definition *cl* :: *bool* \implies (*'a*, *'ls* \times *'s*, *'v*) *spec* \implies (*'a*, *'ls* \times *'s*, *'v*) *spec* **where**
cl T P = \sqcup (*spec.singleton* ' $\{\sigma_1. \exists \sigma_2. \langle \sigma_2 \rangle \leq P \wedge \text{seq-ctxt.equivalent } T \sigma_1 \sigma_2\}$)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path singleton.seq-ctxt* \rangle

lemma *cl-le-conv[spec.singleton.le-conv]*:

shows $\langle \sigma \rangle \leq \text{spec.seq-ctxt.cl } T P \iff (\exists \sigma'. \langle \sigma' \rangle \leq P \wedge \text{seq-ctxt.equivalent } T \sigma \sigma')$ (**is** ?*lhs* \iff ?*rhs*)

proof(*rule iffI*)
show $?lhs \implies ?rhs$
by (*clarsimp simp: spec.seq-ctxt.cl-def spec.singleton-le-conv*)
(*force elim: seq-ctxt.equivalent.downwards-existsE[where $T=T$]*)
(*dest: order.trans[OF spec.singleton.mono]*)
show $?rhs \implies ?lhs$
unfolding *spec.seq-ctxt.cl-def spec.singleton-le-conv* **by** *blast*
qed

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *seq-ctxt: closure-complete-distrib-lattice-distributive-class spec.seq-ctxt.cl T for F*

proof *standard*

show $P \leq \text{spec.seq-ctxt.cl } T \ Q \longleftrightarrow \text{spec.seq-ctxt.cl } T \ P \leq \text{spec.seq-ctxt.cl } T \ Q$ (**is** $?lhs \longleftrightarrow ?rhs$)

for $P \ Q :: ('a, 'ls \times 's, 'v)$ *spec*

proof(*rule iffI*)

show $?lhs \implies ?rhs$

by (*rule spec.singleton-le-extI*)

(*force simp: spec.singleton.seq-ctxt.cl-le-conv dest: order.trans[rotated]*)

show $?rhs \implies ?lhs$

by (*metis spec.singleton.seq-ctxt.cl-le-conv spec.singleton-le-ext-conv*)

qed

show $\text{spec.seq-ctxt.cl } T \ (\bigsqcup X) \leq \bigsqcup (\text{spec.seq-ctxt.cl } T \ ' X) \sqcup \text{spec.seq-ctxt.cl } T \ \perp$

for $X :: ('a, 'ls \times 's, 'v)$ *spec set*

by (*auto simp: spec.seq-ctxt.cl-def*)

qed

setup $\langle \text{Sign.mandatory-path idle.seq-ctxt} \rangle$

lemma *cl-le-conv[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.seq-ctxt.cl } T \ P \longleftrightarrow \text{spec.idle} \leq P$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof(*rule iffI[OF - order.trans[OF - spec.seq-ctxt.expansive]]*)

show $?lhs \implies ?rhs$

by (*clarsimp simp: spec.idle-def spec.singleton.le-conv*)

(*metis trace.take.0 seq-ctxt.equivalent.partial-sel-equivE spec.singleton.takeI trace.t.sel(1)*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path seq-ctxt.cl} \rangle$

lemma *bot[simp]*:

shows $\text{spec.seq-ctxt.cl } T \ \perp = \perp$

by (*simp add: spec.seq-ctxt.cl-def spec.singleton.not-bot*)

lemma *mono*:

assumes $T' \implies T$

assumes $P \leq P'$

shows $\text{spec.seq-ctxt.cl } T \ P \leq \text{spec.seq-ctxt.cl } T' \ P'$

unfolding *spec.seq-ctxt.cl-def*

by (*strengthen ord-to-strengthen(1)[OF $\langle P \leq P' \rangle$]*)

(*blast intro: seq-ctxt.equivalent.mono[OF assms(1)]*)

lemma *strengthen[strg]*:

assumes *st-ord* $(\neg F) \ T \ T'$

assumes *st-ord* $F \ P \ P'$

shows *st-ord* $F \ (\text{spec.seq-ctxt.cl } T \ P) \ (\text{spec.seq-ctxt.cl } T' \ P')$

using *assms* **by** (*cases F; simp add: spec.seq-ctxt.cl.mono le-bool-def*)

lemma *Sup*:

shows $\text{spec.seq-ctxt.cl } T (\bigsqcup X) = \bigsqcup (\text{spec.seq-ctxt.cl } T \text{ ` } X)$
by (*simp add: spec.seq-ctxt.cl-Sup*)

lemmas *sup* = $\text{spec.seq-ctxt.cl.Sup}$ [**where** $X=\{P, Q\}$ **for** $P Q$, *simplified*]

lemma *singleton*:

shows $\text{spec.seq-ctxt.cl } T \langle \sigma \rangle = \bigsqcup (\text{spec.singleton ` } \{\sigma'. \text{seq-ctxt.equivalent } T \sigma \sigma'\})$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*clarsimp simp: spec.seq-ctxt.cl-def spec.singleton-le-conv*)
(*metis seq-ctxt.equivalent.downwards-existsE2 seq-ctxt.prj.natural trace.natural.mono*)
show $?rhs \leq ?lhs$
by (*fastforce simp: spec.seq-ctxt.cl-def*)
qed

lemma *idle*: — not *simp* friendly

shows $\text{spec.seq-ctxt.cl } T (\text{spec.idle} :: ('a, 'ls \times 's, 'v) \text{spec})$
 $= \text{spec.term.none } (\text{spec.rel } (UNIV \times (UNIV \times_R Id)) :: ('a, 'ls \times 's, 'w) \text{spec})$ (**is** $?lhs = ?rhs$)
proof(*rule spec.singleton.antisym*)
have $*$: $s = s'$
if $\text{snd ` set } xs \subseteq \{(ls_0, s_0)\}$
and $\text{trace.natural' } s_0 (\text{map } (\text{map-prod id snd}) ys) = \text{trace.natural' } s_0 (\text{map } (\text{map-prod id snd}) xs)$
and $(a, (ls, s), ls', s') \in \text{trace.steps' } (ls_0, s_0) ys$
for $xs \text{ } ys :: ('a \times ('ls \times 's)) \text{ list}$ **and** $ls_0 \ s_0 \ a \ ls \ s \ ls' \ \text{and } s'$
using *that*
proof(*induct ys rule: rev-induct*)
case *snoc* **from** *snoc.prem*s **show** $?case$
by (*auto simp: trace.natural'.append trace.steps'.append split-pairs*
trace.final'.map[where s=(ls₀, s₀) and sf=snd, simplified]
trace.natural'.map-natural'[where sf=snd and s=(ls₀, s₀), simplified]
simp flip: id-def trace.natural'.eq-Nil-conv
split: if-split-asm
dest: arg-cong[where f=λxs. trace.natural' s₀ (map (map-prod id snd) xs)]
*intro: snoc.hyps[OF snoc.prem*s(1)])

qed *simp*

show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ

using *that*
by (*cases* σ)
(*clarsimp simp: spec.singleton.le-conv * trace.split-all seq-ctxt.prj-def trace.natural-def*)
have $*$: $s' = \text{snd } s$
if $\text{trace.steps' } s \ xs \subseteq UNIV \times (UNIV \times_R Id)$
and $(a, (ls', s')) \in \text{set } xs$
for $a \ s \ ls' \ s'$ **and** $xs :: ('a \times ('ls \times 's)) \text{ list}$
using *that* **by** (*induct xs arbitrary: s*) (*auto simp: trace.steps'.Cons-eq-if split: if-splits*)
show $\langle \sigma \rangle \leq ?lhs$ **if** $\langle \sigma \rangle \leq ?rhs$ **for** σ
using *that*
by (*cases* $\text{trace.init } \sigma$)
(*fastforce simp: spec.singleton.le-conv seq-ctxt.prj-def trace.natural-def map-prod.comp*
*dest: **
intro: exI[where x=trace.map id (map-prod (fst (trace.init σ)) id) id σ])

qed

lemma *invmap-le*:

shows $\text{spec.seq-ctxt.cl } True (\text{spec.invmap af id vf } P) \leq \text{spec.invmap af id vf } (\text{spec.seq-ctxt.cl } True P)$
by (*rule spec.singleton-le-extI*)
(*auto simp: spec.singleton.le-conv dest: seq-ctxt.equivalent.map-sf-eq-id*)

lemma *map-le*:

shows $\text{spec.map af id vf (spec.seq-ctxt.cl True P)} \leq \text{spec.seq-ctxt.cl True (spec.map af id vf P)}$
by (*rule spec.singleton-le-extI*)
(*clarsimp simp: spec.singleton.le-conv*)
 dest!: $\text{seq-ctxt.equivalent.map-sf-eq-id}[\mathbf{where\ af=af\ and\ vf=vf}]$;
meson order.refl order.trans spec.singleton.seq-ctxt.cl-le-conv)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term.none.seq-ctxt} \rangle$

lemma *cl*:

shows $\text{spec.term.none (spec.seq-ctxt.cl T P)} = \text{spec.seq-ctxt.cl T (spec.term.none P)}$
by (*rule spec.singleton.antisym*)
(*auto simp: spec.singleton.le-conv seq-ctxt.prj-def trace.split-Ex trace.natural-def*)

lemma *cl-True-False*:

shows $\text{spec.seq-ctxt.cl True (spec.term.none f)} = \text{spec.seq-ctxt.cl False (spec.term.none f)}$
by (*rule spec.singleton.antisym*)
(*auto simp: spec.singleton.le-conv seq-ctxt.prj-def trace.split-Ex trace.natural-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term.all.seq-ctxt} \rangle$

lemma *cl-le*:

shows $\text{spec.seq-ctxt.cl T (spec.term.all P)} \leq \text{spec.term.all (spec.seq-ctxt.cl T P)}$
by (*metis spec.seq-ctxt.mono-cl spec.term.galois spec.term.lower-upper-contractive spec.term.none.seq-ctxt.cl*)

lemma *cl-False*:

shows $\text{spec.seq-ctxt.cl False (spec.term.all P)} = \text{spec.term.all (spec.seq-ctxt.cl False P)}$
by (*rule spec.singleton.antisym*)
(*auto simp: spec.singleton.le-conv seq-ctxt.prj-def trace.split-Ex trace.natural-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind.seq-ctxt} \rangle$

lemma *cl-le*:

shows $\text{spec.seq-ctxt.cl True } f \ggg (\lambda v. \text{spec.seq-ctxt.cl T (g v)}) \leq \text{spec.seq-ctxt.cl T (f \ggg g)}$
proof(*induct rule: spec.bind-le*)

case *incomplete show ?case*
by (*simp add: spec.seq-ctxt.cl.mono spec.term.none.seq-ctxt.cl*)

next

case (*continue* $\sigma_f \sigma_g v$)
then obtain $\sigma_f' \sigma_g'$
 where $\ast: \langle \sigma_f' \rangle \leq f \text{ seq-ctxt.equivalent True } \sigma_f \sigma_f'$
 $\langle \sigma_g' \rangle \leq g v \text{ seq-ctxt.equivalent T } \sigma_g \sigma_g'$
 by (*clarsimp simp: spec.singleton.seq-ctxt.cl-le-conv*)
let $? \sigma = \text{trace.T (trace.init } \sigma_f') (\text{trace.rest } \sigma_f' @ \text{trace.rest } \sigma_g') (\text{trace.term } \sigma_g')$
from *continue*(2,3) \ast
have $\langle ? \sigma \rangle \leq f \ggg g$
 by (*cases* σ_f' , *cases* σ_g')
 (*fastforce intro: spec.bind.continueI[where v=v] elim: seq-ctxt.equivalent.partial-sel-equivE*)
moreover
from *continue*(2,3) $\ast(2,4)$
have $\text{seq-ctxt.equivalent T (trace.T (trace.init } \sigma_f') (\text{trace.rest } \sigma_f' @ \text{trace.rest } \sigma_g') (\text{trace.term } \sigma_g')) ? \sigma$

by (*cases* σ_f , *cases* σ_g , *cases* $\sigma_{f'}$, *cases* $\sigma_{g'}$) (*auto intro: seq-ctxt.equivalent.append*)
ultimately show *?case*
by (*auto simp: spec.singleton.seq-ctxt.cl-le-conv*)
qed

lemma *clL-le*:

shows *spec.seq-ctxt.cl* $\text{True } f \ggg g \leq \text{spec.seq-ctxt.cl } T (f \ggg g)$
by (*strengthen ord-to-strengthen(2)[OF spec.bind.seq-ctxt.cl-le]*)
(rule spec.bind.mono[OF order.refl spec.seq-ctxt.expansive])

lemma *clR-le*:

shows $f \ggg (\lambda v. \text{spec.seq-ctxt.cl } T (g v)) \leq \text{spec.seq-ctxt.cl } T (f \ggg g)$
by (*strengthen ord-to-strengthen(2)[OF spec.bind.seq-ctxt.cl-le]*)
(rule spec.bind.mono[OF spec.seq-ctxt.expansive order.refl])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path seq-ctxt} \rangle$

lemma *cl-local-le*: — the RHS is the closure that arises from *spec.local*, ignoring the constraint

shows *spec.seq-ctxt.cl* $T P \leq \text{spec.map-invmap.cl} - - - \text{id snd id } P$
by (*fastforce simp: spec.map-invmap.cl-def spec.seq-ctxt.cl-def seq-ctxt.prj-def spec.map.Sup*
spec.map.singleton spec.singleton.map-le-conv spec.singleton-le-conv
simp flip: spec.map-invmap.galois)

lemma *cl-local*:

shows *spec.local* (*spec.seq-ctxt.cl* $T (\text{spec.local.interference} \sqcap P)$)
 $= \text{spec.local } P$ (**is** *?lhs = ?rhs*)
proof(*rule antisym*)
show *?lhs* \leq *?rhs*
by (*simp add: spec.local-def spec.map-invmap.galois le-infI2 spec.seq-ctxt.cl-local-le*
flip: spec.map-invmap.cl-def)
show *?rhs* \leq *?lhs*
unfolding *spec.local-def* **by** (*strengthen ord-to-strengthen(2)[OF spec.seq-ctxt.expansive]*) *simp*
qed

lemma *cl-imp-local-le*:

assumes *spec.local.interference* $\sqcap P$
 $\leq \text{spec.seq-ctxt.cl } \text{False} (\text{spec.local.interference} \sqcap Q)$
shows *spec.local* $P \leq \text{spec.local } Q$
by (*subst (1 2) spec.seq-ctxt.cl-local[where T=False, symmetric]*)
(use assms spec.seq-ctxt.cl[where T=False] in <auto intro: spec.local.mono>)

lemma *cl-inf-pre*:

shows *spec.pre* $P \sqcap \text{spec.seq-ctxt.cl } T c = \text{spec.seq-ctxt.cl } T (\text{spec.pre } P \sqcap c)$
by (*fastforce simp: spec.singleton.le-conv*
intro: spec.singleton.antisym
elim: seq-ctxt.equivalent.partial-sel-equivE)

lemma *cl-pre-le-conv*:

shows *spec.seq-ctxt.cl* $T c \leq \text{spec.pre } P \longleftrightarrow c \leq \text{spec.pre } P$ (**is** *?lhs* \longleftrightarrow *?rhs*)
proof(*rule iffI*)
from *spec.seq-ctxt.cl-inf-pre[where P=P and c=c, symmetric]*
show *?lhs* \implies *?rhs*
by (*auto intro: order.trans[OF spec.seq-ctxt.expansive]*)
from *spec.seq-ctxt.cl-inf-pre[where P=P and c=c, symmetric]*
show *?rhs* \implies *?lhs*
by (*simp add: inf.absorb-iff2*)

qed

lemma *cl-inf-post*:

shows $\text{spec.post } Q \sqcap \text{spec.seq-ctxt.cl True } c = \text{spec.seq-ctxt.cl True } (\text{spec.post } Q \sqcap c)$
by (*fastforce simp: spec.singleton.le-conv*
intro: spec.singleton.antisym
elim: seq-ctxt.equivalent.partial-sel-equivE
split: option.split)

lemma *cl-post-le-conv*:

shows $\text{spec.seq-ctxt.cl True } c \leq \text{spec.post } Q \iff c \leq \text{spec.post } Q$ (**is** *?lhs* \iff *?rhs*)

proof(*rule iffI*)

from *spec.seq-ctxt.cl-inf-post*[**where** $Q=Q$ **and** $c=c$, *symmetric*]

show *?lhs* \implies *?rhs*

by (*auto intro: order.trans[OF spec.seq-ctxt.expansive]*)

from *spec.seq-ctxt.cl-inf-post*[**where** $Q=Q$ **and** $c=c$, *symmetric*]

show *?rhs* \implies *?lhs*

by (*simp add: inf.absorb-iff2*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

15.2.1 Permuting local actions

We can reorder operations on the local state as these are not observable.

Firstly: an initial action F that does not change the observable state can be swapped with an arbitrary action G .

setup $\langle \text{Sign.mandatory-path spec.seq-ctxt} \rangle$

lemma *cl-action-permuteL-le*:

fixes $F :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $G :: 'v \implies ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $G' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $F' :: 'v' \implies ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

— F does not change $'s$, can be partial

assumes $F: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \implies \text{snd } s' = \text{snd } s$

— The final state and return value are independent of the order of actions. F' does not change $'s$, cannot be partial

assumes $FGG'F': \bigwedge v w a a' s s' t. \llbracket P s; (v, a', s, t) \in F; (w, a, t, s') \in G v \rrbracket$

$\implies \exists v' a'' a''' s'' t'. (v', a'', s, t') \in G' \wedge (w, a''', t', s'') \in F' v'$

$\wedge \text{snd } s' = \text{snd } t' \wedge (\text{snd } s \neq \text{snd } t' \longrightarrow a'' = a) \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s') \wedge \text{snd } s'' = \text{snd } t'$

shows $(\text{spec.action } F \gg (\lambda v. \text{spec.action } (G v))) \sqcap \text{spec.pre } P$

$\leq \text{spec.seq-ctxt.cl } T (\text{spec.action } G' \gg (\lambda v. \text{spec.action } (F' v)))$ (**is** \leq *?rhs*)

unfolding *spec.bind.inf-pre*

proof(*rule order.trans[OF spec.bind.mono[OF spec.pre.inf-action-le(2) order.refl]*,

induct rule: spec.bind-le)

case *incomplete*

from F **have** $\text{spec.term.none } (\text{spec.action } (F \sqcap \text{UNIV} \times \text{UNIV} \times \text{Pre } P)) \leq \text{spec.seq-ctxt.cl } T \text{spec.idle}$

by (*fastforce intro: spec.term.none.mono[OF spec.action.rel-le]*

simp: spec.seq-ctxt.cl.idle[**where** $'w='v$])

also have $\dots \leq$ *?rhs*

by (*simp add: spec.idle-le spec.seq-ctxt.cl.mono*)

finally show *?case* .

next

case (*continue* $\sigma_f \sigma_g v$) **then show** *?case*

apply —

```

apply (erule (1) spec.singleton.action-Some-leE)
apply (erule (1) spec.singleton.action-not-idle-leE)
apply clarsimp
apply (frule (1) F)
apply (frule (2) FGG'F')
apply (clarsimp simp: spec.singleton.seq-ctxt.cl-le-conv)
apply (intro exI conjI)
  apply (rule spec.bind.continueI)
    apply (rule spec.action.stepI, assumption, blast)
    apply (rule spec.action.stepI[where w=trace.term  $\sigma_g$ ], simp, force)
apply (clarsimp simp: seq-ctxt.prj-def trace.stuttering.equiv.append-conv
  trace.stuttering.equiv.simps(3)[where xs=[x] and ys=[] for x, simplified])
apply (rule exI[where x=[]])
apply (simp add: image-image trace.natural'.eq-Nil-conv
  trace-steps'-snd-le-const trace-natural'-took-step-shared-changes)
apply (intro conjI impI)
apply (simp add: trace-steps'-snd-le-const)
done

```

qed

Secondly: an initial action G that does change the observable state can be swapped with an arbitrary action F that does not observably change the state.

lemma *cl-action-permuteR-le*:

```

fixes G :: ('v × 'a × ('ls × 's) × ('ls × 's)) set
fixes F :: 'v ⇒ ('w × 'a × ('ls × 's) × ('ls × 's)) set
fixes F' :: ('v' × 'a × ('ls × 's) × ('ls × 's)) set
fixes G' :: 'v' ⇒ ('w × 'a × ('ls × 's) × ('ls × 's)) set
  — F does not stall if G makes an observable state change
assumes G:  $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket$ 
   $\implies \exists v' w a'' t s''. (v', a'', s, t) \in F' \wedge (w, a, t, s'') \in G' \wedge v' \wedge \text{snd } t = \text{snd } s \wedge \text{snd } s'' = \text{snd } s'$ 
  — The final state and return value are independent of the order of actions
assumes GFF'G':  $\bigwedge v w a a' s s' t. \llbracket P s; (v, a, s, t) \in G; (w, a', t, s') \in F' v \rrbracket$ 
   $\implies \text{snd } s' = \text{snd } t \wedge (\exists v' a'' a''' s'' t'. (v', a'', s, t') \in F' \wedge (w, a''', t', s'') \in G' \wedge v' \wedge \text{snd } t' = \text{snd } s \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s') \wedge \text{snd } s'' = \text{snd } s' \wedge (\text{snd } s'' \neq \text{snd } t' \longrightarrow a''' = a))$ 
shows (spec.action G  $\ggg$  ( $\lambda v. \text{spec.action } (F v)$ ))  $\sqcap$  spec.pre P
  < spec.seq-ctxt.cl T (spec.action F'  $\ggg$  ( $\lambda v. \text{spec.action } (G' v)$ ))

```

unfolding spec.bind.inf-pre

proof(rule order.trans[OF spec.bind.mono[OF spec.pre.inf-action-le(2) order.refl]],

induct rule: spec.bind-le)

case incomplete **show** ?case

unfolding spec.term.galois

proof(induct rule: spec.action-le)

case idle **show** ?case

by (simp add: spec.idle-le)

next

case (step v a s s') **then show** ?case

proof(cases snd s' = snd s)

case True **with step show** ?thesis

by (clarsimp simp: spec.singleton.le-conv intro!: exI[**where** x=None] exI[**where** x=trace.T s [] None])
 (simp add: seq-ctxt.prj-def trace.natural.simps order.trans[OF spec.idle.minimal-le] spec.idle-le)

next

case False **with step show** ?thesis

by (fastforce simp: spec.singleton.le-conv seq-ctxt.prj-def trace.natural.simps
 dest: G
 intro: spec.bind.continueI
 elim: spec.action.stepI)

qed

```

qed
next
case (continue  $\sigma_f \sigma_g v$ ) then show ?case
  apply –
  apply (erule (1) spec.singleton.action-Some-leE)
  apply (erule (1) spec.singleton.action-not-idle-leE)
  apply clarsimp
  apply (drule (2) GFF'G')
  apply (clarsimp simp: spec.singleton.seq-ctxt.cl-le-conv)
  apply (intro exI conjI)
  apply (rule spec.bind.continueI)
  apply (rule spec.action.stepI, assumption, blast)
  apply (rule spec.action.stepI[where w=trace.term  $\sigma_g$ ], simp, force)
  apply (clarsimp simp: seq-ctxt.prj-def trace.stuttering.equiv.append-conv
    trace.stuttering.equiv.simps(1)[where xs=[x] for x, simplified])
  apply (rule exI)
  apply (rule exI[where x=[]])
  apply (intro conjI)
  apply simp
  apply (fastforce simp: trace.natural'.eq-Nil-conv
    dest: trace-natural'-took-step-shared-changes trace-natural'-took-step-shared-same)
  apply (auto simp: trace.natural'.eq-Nil-conv
    simp flip: trace.final'.map[where af=id and sf=snd]
    dest!: arg-cong[where f=snd] trace-natural'-took-step-shared-same)
done
qed

```

lemma *cl-action-bind-action-pre-post:*

```

fixes F' :: ('v × 'a × ('ls × 's) × ('ls × 's)) set
fixes G' :: 'v ⇒ ('w × 'a × ('ls × 's) × ('ls × 's)) set
fixes Q :: 'w ⇒ ('ls × 's) pred
assumes  $\bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in F; (w, a', s', s'') \in G v \rrbracket \implies Q w s''$ 
shows spec.pre P  $\sqcap$  spec.seq-ctxt.cl True (spec.action F  $\ggg$  ( $\lambda v.$  spec.action (G v)))  $\leq$  spec.post Q
unfolding spec.seq-ctxt.cl-inf-pre spec.seq-ctxt.cl-post-le-conv spec.bind.inf-pre
proof(induct rule: spec.bind-le)
  case incomplete show ?case
  by (simp add: spec.term.none.post-le)

```

next

```

case (continue  $\sigma_f \sigma_g v$ ) with assms show ?case
  by (cases  $\sigma_f$ )
    (fastforce simp: spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv
      split: option.split-asm)

```

qed

lemma *cl-rev-kleene-star-action-permute-le:*

```

fixes F G :: (unit × 'a × ('ls × 's) × ('ls × 's)) set
— F does not stall if G changes the observable state
assumes G:  $\bigwedge a s s'. \llbracket ((, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket$ 
 $\implies \exists w a'' t s''. ((, a'', s, t) \in F \wedge ((, a, t, s'') \in G \wedge \text{snd } t = \text{snd } s \wedge \text{snd } s'' = \text{snd } s')$ 
— The final state is independent of order of actions, F does not change 's, can be partial
assumes GFFG:  $\bigwedge a a' s s' t. \llbracket ((, a, s, t) \in G; ((, a', t, s') \in F \rrbracket$ 
 $\implies \text{snd } s' = \text{snd } t \wedge (\exists a'' a''' t'. ((, a'', s, t') \in F \wedge ((, a''', t', s') \in G$ 
 $\wedge \text{snd } t' = \text{snd } s \wedge (\text{snd } s' \neq \text{snd } t' \longrightarrow a''' = a))$ 
shows spec.kleene.rev-star (spec.action G)  $\ggg$  ( $\lambda :: \text{unit}.$  spec.action F)
 $\leq$  spec.seq-ctxt.cl True (spec.action F  $\ggg$  spec.kleene.rev-star (spec.action G)) (is ?lhs spec.kleene.rev-star  $\leq$ 
?rhs)
proof(induct rule: spec.kleene.rev-star.fixp-induct[where P= $\lambda R.$  ?lhs R  $\leq$  ?rhs, case-names adm bot step])
  case (step S)

```

from *assms* **have** S (*spec.action* G) \gg *spec.action* G \gg *spec.action* F
 $\leq S$ (*spec.action* G) \gg ($\lambda x. \text{spec.seq-ctxt.cl True (spec.action } F \gg (\lambda v. \text{spec.action } G))$)
by (*simp add: spec.bind.bind*)
(*strengthen ord-to-strengthen(1)[OF spec.seq-ctxt.cl-action-permuteR-le[where P= \top and T=True, simplified]; force*)
also have $\dots \leq ?rhs$
apply (*strengthen ord-to-strengthen(1)[OF spec.bind.seq-ctxt.clR-le]*)
apply (*subst spec.bind.bind[symmetric]*)
apply (*strengthen ord-to-strengthen(1)[OF step]*)
apply (*strengthen ord-to-strengthen(1)[OF spec.bind.seq-ctxt.clL-le[where T=True]*)
apply (*strengthen ord-to-strengthen(2)[OF spec.kleene.fold-rev-starR]*)
apply (*simp add: spec.bind.bind*)
done
moreover have *spec.return* () \gg *spec.action* $F \leq ?rhs$
apply (*simp add: spec.bind.returnL spec.idle-le*)
apply (*rule order.trans[OF - spec.seq-ctxt.expansive]*)
apply (*rule order.trans[OF - spec.bind.mono[OF order.refl spec.kleene.epsilon-rev-star-le]*)
apply *simp*
done
ultimately show *?case*
by (*simp add: spec.bind.supL*)
qed *simp-all*

lemma *cl-local-action-interference-permute-le*: — local actions permute with interference

fixes $F :: (\text{unit} \times 'a \text{ agent} \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $r :: 's \text{ rel}$
— F does not block
assumes $\bigwedge s \text{ ls}. \exists v a \text{ ls}'. (v, a, (ls, s), (ls', s)) \in F$
— F is insensitive to and does not modify the shared state
assumes $\bigwedge v a s s' s'' \text{ ls ls}'. (v, a, (ls, s), (ls', s')) \in F$
 $\implies s' = s \wedge (v, a, (ls, s''), (ls', s'')) \in F$
shows *spec.rel* ($A \times (\text{Id} \times_R r)$) \gg ($\lambda :: \text{unit}. \text{spec.action } F$)
 $\leq \text{spec.seq-ctxt.cl True (spec.action } F \gg \text{spec.rel } (A \times (\text{Id} \times_R r)))$
by (*simp add: spec.rel.monomorphic-conv spec.kleene.star-rev-star spec.rel.act-def*)
(*rule spec.seq-ctxt.cl-rev-kleene-star-action-permute-le; use assms in fastforce*)

lemma *cl-action-mumble-trailing-le*:

assumes $\bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket$
 $\implies \exists a' \text{ ls}'. (v, a', s, (ls', \text{snd } s')) \in F'$
 $\wedge (\text{snd } s' \neq \text{snd } s \longrightarrow a' = a) \wedge (T \longrightarrow \text{fst } s' = \text{fst } s)$
shows *spec.action* $F \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T$ (*spec.action* F')
proof(*rule order.trans[OF spec.pre.inf-action-le(2)], induct rule: spec.action-le*)
case *idle* **show** *?case*
by (*simp add: spec.idle-le*)
next
case (*step* $v a s s'$)
then obtain $a' \text{ ls}'$ **where** $(v, a', s, (ls', \text{snd } s')) \in F'$ **and** $\text{snd } s' \neq \text{snd } s \longrightarrow a' = a$ **and** $T \longrightarrow \text{fst } s' = \text{fst } s'$
by (*blast dest: assms*)
then show *?case*
unfolding *spec.singleton.le-conv*
by (*fastforce intro: exI[where x=trace.T s [(a', (ls', snd s'))] (Some v)] spec.action.stepI*
simp: seq-ctxt.prj-def trace.natural.simps(2)[where xs=[]])
qed

lemma *cl-action-mumbleL-le*:

assumes $\bigwedge w a s s'. \llbracket P s; (w, a, s, s') \in G \rrbracket$
 $\implies \exists v a' a'' t s''. (v, a', s, t) \in F' \wedge (w, a'', t, s') \in G' v$
 $\wedge \text{snd } t = \text{snd } s \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s')$

$$\wedge \text{snd } s'' = \text{snd } s' \wedge (\text{snd } s'' \neq \text{snd } t \longrightarrow a'' = a)$$

shows *spec.action* $G \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T$ (*spec.action* $F' \ggg (\lambda v. \text{spec.action } (G' v))$)
using *assms* **by** (*fastforce intro!*: *spec.seq-ctxt.cl-action-permuteR-le*[**where** $F = \lambda v. (\{v\} \times \text{UNIV} \times \text{Id})$, *simplified*], *spec.return-def*[*symmetric*], *simplified*])

lemma *cl-action-mumbleR-le*:

assumes $\wedge v a s s'. \llbracket P s; (v, a, s, s') \in G \rrbracket$
 $\implies \exists w a' a'' t. (w, a', s, t) \in G' \wedge (v, a'', t, s') \in F' w$
 $\wedge \text{snd } t = \text{snd } s' \wedge (\text{snd } t \neq \text{snd } s \longrightarrow a' = a)$

shows *spec.action* $G \sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T$ (*spec.action* $G' \ggg (\lambda v. \text{spec.action } (F' v))$)
using *assms* **by** (*fastforce intro!*: *spec.seq-ctxt.cl-action-permuteL-le*[**where** $F = (\{\}\} \times \text{UNIV} \times \text{Id})$, *simplified*, *simplified*], *spec.idle-le*, *spec.bind.returnL*, *spec.return-def*[*symmetric*])

lemma *cl-action-mumble-expandL-le*:

assumes $\wedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \implies \text{snd } s' = \text{snd } s$
assumes $\wedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in F; (w, a', s', s'') \in G v \rrbracket$
 $\implies \exists s'''. (w, a', s, s''') \in G' \wedge \text{snd } s''' = \text{snd } s'' \wedge (T \longrightarrow \text{fst } s''' = \text{fst } s'')$

shows (*spec.action* $F \ggg (\lambda v. \text{spec.action } (G v))$) $\sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T$ (*spec.action* G')
using *assms* **by** (*fastforce intro!*: *spec.seq-ctxt.cl-action-permuteL-le*[**where** $F = \lambda v. (\{v\} \times \text{UNIV} \times \text{Id})$, *simplified*], *spec.return-def*[*symmetric*], *simplified*])

lemma *cl-action-mumble-expandR-le*:

assumes $\wedge v a s s'. \llbracket P s; (v, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket \implies \exists v' s''. (v', a, s, s'') \in G' \wedge \text{snd } s'' = \text{snd } s'$
assumes $\wedge v w a a' s s' t. \llbracket P s; (v, a, s, t) \in G; (w, a', t, s') \in F v \rrbracket$
 $\implies \text{snd } s' = \text{snd } t \wedge (\exists a'' s''. (w, a'', s, s'') \in G' \wedge \text{snd } s'' = \text{snd } s' \wedge (T \longrightarrow \text{fst } s'' = \text{fst } s') \wedge$
 $(\text{snd } s'' \neq \text{snd } s \longrightarrow a'' = a))$

shows (*spec.action* $G \ggg (\lambda v. \text{spec.action } (F v))$) $\sqcap \text{spec.pre } P \leq \text{spec.seq-ctxt.cl } T$ (*spec.action* G')
using *assms* **by** (*fastforce intro!*: *spec.seq-ctxt.cl-action-permuteR-le*[**where** $F = (\{\}\} \times \text{UNIV} \times \text{Id})$, *simplified*, *simplified*], *spec.idle-le*, *spec.bind.returnL*, *spec.return-def*[*symmetric*])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.local} \rangle$

lemma *init-write-interference-permute-le*:

fixes $P :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$
shows *spec.local* (*spec.rel* ($\{\text{env}\} \times \text{UNIV}$) $\ggg (\lambda :: \text{unit}. \text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id}) \ggg P)$)
 $\leq \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id}) \ggg (\text{spec.rel } (\{\text{env}\} \times \text{UNIV}) \ggg (\lambda :: \text{unit}. P)))$

apply (*rule* *spec.seq-ctxt.cl-imp-local-le*)

apply (*simp* *add*: *ac-simps spec.bind.inf-rel spec.action.inf-rel*
flip: *spec.rel.inf spec.bind.bind*)

apply (*rule* *order.trans[OF - spec.bind.seq-ctxt.cl-le]*)

apply (*rule* *spec.bind.mono[OF - spec.seq-ctxt.expansive]*)

apply (*rule* *spec.seq-ctxt.cl-local-action-interference-permute-le*)

apply *auto*

done

lemma *init-write-interference2-permute-le*:

fixes $P :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$
shows *spec.local* (*spec.rel* ($A \times (\text{Id} \times_{\mathbb{R}} r)$) $\ggg (\lambda :: \text{unit}. \text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id}) \ggg P)$)
 $\leq \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle \text{ls} \rangle \text{ id}) \ggg (\text{spec.rel } (A \times (\text{Id} \times_{\mathbb{R}} r)) \ggg (\lambda :: \text{unit}. P)))$

apply (*rule* *spec.seq-ctxt.cl-imp-local-le*)

apply (*simp* *add*: *ac-simps spec.bind.inf-rel spec.action.inf-rel*
flip: *spec.rel.inf spec.bind.bind*)

apply (*rule* *order.trans[OF - spec.bind.seq-ctxt.cl-le]*)

apply (*rule* *spec.bind.mono[OF - spec.seq-ctxt.expansive]*)

apply (*rule* *spec.seq-ctxt.cl-local-action-interference-permute-le*)

apply *auto*

done

lemma *trailing-local-act*:

fixes $F :: 'v \Rightarrow ('w \times 'a \text{ agent} \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
shows $\text{spec.local } (P \gg (\lambda v. \text{spec.action } (F v)))$
 $= \text{spec.local } (P \gg (\lambda v. \text{spec.action } \{(w, a, (ls, s), (ls, s')) \mid w a ls s ls' s'. (w, a, (ls, s), (ls', s')) \in F v \wedge (a$
 $= \text{env} \longrightarrow ls' = ls\}))$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \leq ?rhs$

apply (*rule spec.seq-ctxt.cl-imp-local-le*)

apply (*simp add: spec.bind.inf-rel spec.action.inf-rel*)

apply (*rule order.trans[OF - spec.bind.seq-ctxt.clR-le]*)

apply (*rule spec.bind.mono[OF order.refl]*)

apply (*rule spec.seq-ctxt.cl-action-mumble-trailing-le*[**where** $P = \top$, *simplified*])

apply (*force simp: spec.local.qrm-def*)

done

show $?rhs \leq ?lhs$

apply (*rule spec.seq-ctxt.cl-imp-local-le*)

apply (*simp add: spec.bind.inf-rel spec.action.inf-rel*)

apply (*rule order.trans[OF - spec.bind.seq-ctxt.clR-le]*)

apply (*rule spec.bind.mono[OF order.refl]*)

apply (*rule spec.seq-ctxt.cl-action-mumble-trailing-le*[**where** $P = \top$, *simplified*])

apply (*force simp: spec.local.qrm-def*)

done

qed

setup $\langle \text{Sign.parent-path} \rangle$

15.3 *spec.localize*

We can transform a process into one with the same observable behavior that ignores a local state. For compositionality we allow the *env* steps to change the local state but not the *self* steps.

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition *localize* $:: 'ls \text{ rel} \Rightarrow ('a \text{ agent}, 's, 'v) \text{ spec} \Rightarrow ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$ **where**

$\text{localize } r P = \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{UNIV}) \cup \text{range proc} \times (\text{Id} \times_R \text{UNIV})) \sqcap \text{spec.sinvmap snd } P$

setup $\langle \text{Sign.mandatory-path idle} \rangle$

lemma *localize-le*:

assumes $\text{spec.idle} \leq P$

shows $\text{spec.idle} \leq \text{spec.localize } r P$

by (*simp add: spec.localize-def spec.idle.rel-le spec.idle.invmap-le*[*OF assms*] *spec.idle.term.all-le-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path term} \rangle$

setup $\langle \text{Sign.mandatory-path none} \rangle$

lemma *localize*:

shows $\text{spec.term.none } (\text{spec.localize } r P) = \text{spec.localize } r (\text{spec.term.none } P)$

by (*simp add: spec.localize-def spec.term.none.inf spec.term.none.inf-none-rel*)

(*simp add: spec.term.none.invmap*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *localize*:

shows *spec.term.all* (*spec.localize* *r P*) = *spec.localize* *r* (*spec.term.all P*)

by (*simp add: spec.localize-def spec.term.all.rel spec.term.all.inf spec.term.all.invmap*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path closed} \rangle$

lemma *localize*:

assumes $P \in \text{spec.term.closed}$ -

shows *spec.localize* *r P* $\in \text{spec.term.closed}$ -

by (*rule spec.term.closed-clI*)

(*simp add: spec.term.all.localize spec.term.all.monomorphic*
flip: spec.term.closed-conv[OF assms, simplified])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path localize} \rangle$

lemma *singleton*:

fixes $\sigma :: ('a \text{ agent}, 's, 'v) \text{ trace.t}$

shows *spec.localize* *Id* $\langle \sigma \rangle = (\bigsqcup ls::'ls. \langle \text{trace.map id (Pair ls) id } \sigma \rangle)$ (**is** *?lhs* = *?rhs*)

proof(*rule antisym*)

have $*$: *map* (*map-prod id* (*map-prod* $\langle ls \rangle$ *id*)) *xs* = *xs*

if *trace.steps'* (*ls*, *s*) *xs* $\subseteq \text{UNIV} \times (\text{Id} \times_R \text{UNIV})$

for *ls s* **and** *xs* :: $('a \text{ agent} \times ('ls \times 's)) \text{ list}$

using *that* **by** (*induct xs arbitrary: s*) (*auto simp: trace.steps'.Cons-eq-if split: if-split-asm*)

have $\exists x. \langle \sigma'' \rangle \leq \langle \text{trace.map id (Pair } x) \text{ id } \sigma \rangle$

if $\langle \text{trace.map id snd id } \sigma' \rangle \leq \langle \sigma \rangle$

and $\sigma'' \leq \sigma'$

and *trace.steps'* (*trace.init* σ'') (*trace.rest* σ'') $\subseteq \text{UNIV} \times (\text{Id} \times_R \text{UNIV})$

for $\sigma' \sigma'' :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ trace.t}$

using *that*

by - (*cases* σ'' ,

drule spec.singleton.map-le[where af=id and sf=Pair (fst (trace.init σ'')) and vf=id],

*fastforce dest: * trace.map.mono[where af=id and sf= $\lambda x. (fst (trace.init \sigma''), snd x)$ and vf=id]*

spec.singleton.mono

intro: exI[where x=fst (trace.init σ'')]

simp: map-prod-def split-def

simp flip: id-def)

then show *?lhs* \leq *?rhs*

by (*simp add: spec.localize-def spec.invmap.singleton inf-Sup spec.singleton.inf-rel*

flip: Times-Un-distrib1)

show *?rhs* \leq *?lhs*

by (*auto simp: spec.localize-def spec.invmap.singleton spec.singleton.le-conv trace-steps'-map*

intro: exI[where x=trace.map id (Pair a) id σ for a])

qed

lemma *bot*:

shows *spec.localize* *r* $\perp = \perp$

by (*simp add: spec.localize-def spec.invmap.bot*)

lemma *top*:

shows *spec.localize* *r* $\top = \text{spec.rel} (\{\text{env}\} \times (r \times_R \text{UNIV}) \cup \text{range proc} \times (\text{Id} \times_R \text{UNIV}))$

by (*simp add: spec.localize-def spec.invmap.top*)

lemma *Sup*:

shows $\text{spec.localize } r (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.localize } r x)$
by (*simp add: spec.localize-def spec.invmap.Sup inf-Sup image-image*)

lemmas *sup = spec.localize.Sup*[**where** $X = \{X, Y\}$ **for** $X Y$, *simplified*]

lemma *mono*:

assumes $r \subseteq r'$
assumes $P \leq P'$
shows $\text{spec.localize } r P \leq \text{spec.localize } r' P'$
unfolding *spec.localize-def*
apply (*strengthen ord-to-strengthen(1)[OF <r ⊆ r'>]*)
apply (*strengthen ord-to-strengthen(1)[OF <P ≤ P'>]*)
apply *simp*
done

lemma *strengthen*[*strg*]:

assumes *st-ord F r r'*
assumes *st-ord F P P'*
shows *st-ord F (spec.localize r P) (spec.localize r P')*
using *assms by (cases F; simp add: spec.localize.mono)*

lemma *mono2mono*[*cont-intro, partial-function-mono*]:

assumes *monotone orda (≤) r*
assumes *monotone orda (≤) P*
shows *monotone orda (≤) (λx. spec.localize (r x) (P x))*
using *assms by (simp add: monotone-def spec.localize.mono)*

lemma *mcont2mcont*[*cont-intro*]:

assumes *mcont luba orda Sup (≤) P*
shows *mcont luba orda Sup (≤) (λx. spec.localize r (P x))*
using *assms by (simp add: spec.localize-def)*

lemma *bind*:

shows $\text{spec.localize } r (f \ggg g) = \text{spec.localize } r f \ggg (\lambda v. \text{spec.localize } r (g v))$
by (*simp add: spec.localize-def spec.invmap.bind spec.bind.inf-rel*)

lemma *action*:

fixes $F :: ('v \times 'a \text{ agent} \times 's \times 's) \text{ set}$
shows $\text{spec.localize } r (\text{spec.action } F)$
 $= \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{Id}))$
 $\ggg (\lambda :: \text{unit}. \text{spec.action } ((\text{map-prod id } (\text{map-prod id } (\text{map-prod snd snd})) - 'F)$
 $\quad \cap \text{UNIV} \times (\{\text{env}\} \times (r \times_R \text{UNIV}) \cup \text{range proc} \times (\text{Id} \times_R \text{UNIV}) \cup \text{UNIV} \times \text{Id}))$
 $\ggg (\lambda v. \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{Id})) \ggg (\lambda :: \text{unit}. \text{spec.return } v)))$
by (*simp add: spec.localize-def spec.invmap.action spec.bind.inf-rel spec.action.inf-rel-reflcl spec.return.inf-rel*
map-prod-snd-snd-vimage inf-sup-distrib Times-Int-Times relprod-inter
spec.rel.reflcl
flip: spec.rel.inf)

lemma *return*:

shows $(\text{spec.localize } r (\text{spec.return } v) :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec})$
 $= \text{spec.rel } (\{\text{env}\} \times (r \times_R \text{Id})) \ggg (\lambda :: \text{unit}. \text{spec.return } v)$
apply (*simp add: spec.return-def spec.localize.action*)
apply (*simp add: ac-simps map-prod-vimage-Times inf-sup-distrib1 Times-Int-Times*
map-prod-snd-snd-vimage relprod-inter relprod-inter-Id spec.idle-le
flip: spec.return-def)
apply (*simp add: sup.absorb1 Sigma-mono*)

flip: sup.assoc Times-Un-distrib2
apply (*subst spec.action.return-const*[**where** $W=\{()\}$], *simp, simp*)
apply (*simp add: spec.bind.bind spec.bind.return*
flip: spec.rel.act-def[**where** $r=\{\text{env}\} \times (r \times_R \text{Id})$, *simplified ac-simps*])
apply (*simp add: spec.rel.wind-bind*
flip: spec.rel.unfoldL spec.bind.bind)
done

lemma rel:
shows *spec.localize r (spec.rel s)*
 $= \text{spec.rel } (\{\{\text{env}\} \times (r \times_R \text{UNIV}) \cup \text{range proc} \times (\text{Id} \times_R \text{UNIV})\} \cap \text{map-prod id (map-prod snd snd)} - \text{'(s} \cup \text{UNIV} \times \text{Id)})$
by (*simp add: spec.localize-def spec.invmap.rel flip: spec.rel.inf*)

lemma rel-le:
shows *spec.localize Id P \leq spec.rel (UNIV \times (Id \times_R UNIV))*
unfolding *spec.localize-def* **by** (*blast intro: le-infI1 spec.rel.mono*)

lemma parallel:
shows *spec.localize UNIV (P \parallel Q) = spec.localize UNIV P \parallel spec.localize UNIV Q*
by (*simp add: spec.localize-def spec.invmap.parallel spec.parallel.inf-rel*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma localize-le:
assumes $\text{Id} \subseteq r$
shows *spec.action (map-prod id (map-prod id (map-prod snd snd)) - 'F \cap UNIV \times UNIV \times (Id \times_R UNIV))*
 $\leq \text{spec.localize } r \text{ (spec.action } F)$
unfolding *spec.localize.action*
by (*strengthen ord-to-strengthen[OF spec.return.rel-le]*
(use assms in $\langle \text{force intro!}: \text{spec.action.mono}$
simp: spec.bind.return spec.bind.returnL spec.idle-le))

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path interference.closed} \rangle$

lemma localize:
assumes $P \in \text{spec.interference.closed } (\{\text{env}\} \times \text{UNIV})$
shows *spec.localize UNIV P \in spec.interference.closed ($\{\text{env}\} \times \text{UNIV}$)*
by (*force simp: spec.localize-def*
intro: spec.interference.closed.rel subsetD[OF spec.interference.closed.antimono, rotated]
*spec.interference.closed.invmap[**where** $\text{sf}=\text{snd}$ **and** $\text{vf}=\text{id}$, OF assms]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path local} \rangle$

lemma localize:
assumes $\text{Id} \subseteq r$
shows *spec.local (spec.localize r P) = P (is ?lhs = ?rhs)*
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by (*simp add: ac-simps spec.local-def spec.local.qrm-def spec.localize-def spec.map-invmap.galois*)
from *assms* **show** $?rhs \leq ?lhs$
by (*simp add: spec.local-def spec.local.qrm-def spec.localize-def ac-simps*)

(simp add: spec.map.rel spec.rel.UNIV relprod-inter inf.absorb1
 inf-sup-distrib Times-Int-Times map-prod-image-Times map-prod-image-relprod
 flip: Sigma-Un-distrib1 spec.map.inf-distr spec.rel.inf)

qed

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.mandatory-path spec>

setup <Sign.mandatory-path bind>

lemma smap-sndL:

assumes $UNIV \times (Id \times_R UNIV) \subseteq r$

shows $spec.smap\ snd\ f \ggg g = spec.smap\ snd\ (f \ggg (\lambda v. spec.rel\ r \sqcap spec.sinvmap\ snd\ (g\ v)))$ (is ?lhs = ?rhs)

proof(rule antisym)

show ?lhs \leq ?rhs

proof(induct rule: spec.bind-le)

case incomplete show ?case by (simp add: spec.map.mono spec.term.none.map)

next

case (continue σ_f' $\sigma_g\ v$)

from < $\langle \sigma_f' \rangle \leq spec.smap\ snd\ f$ >

obtain σ_f

where *: $\langle \sigma_f \rangle \leq f \langle \sigma_f' \rangle \leq \langle trace.map\ id\ snd\ id\ \sigma_f \rangle$

by (clarsimp simp: spec.singleton.le-conv)

let $?\sigma = trace.T\ (trace.init\ \sigma_f)$

($trace.rest\ \sigma_f\ @\ map\ (map-prod\ id\ (Pair\ (fst\ (trace.final\ \sigma_f))))\ (trace.rest\ \sigma_g)$)

($trace.term\ \sigma_g$)

from continue(3) < $\langle \sigma_f' \rangle \leq \langle trace.map\ id\ snd\ id\ \sigma_f \rangle$ >

have $trace.final\ \sigma_f' = snd\ (trace.final\ \sigma_f)$

by (cases σ_f')

(clarsimp simp flip: trace.final'.map[where af=id and sf=snd]

cong: trace.final'.natural'-cong)

with assms continue(2,3,4) *

have $\langle ?\sigma \rangle \leq f \ggg (\lambda v. spec.rel\ r \sqcap spec.sinvmap\ snd\ (g\ v))$

by (cases σ_f ; cases σ_g)

(force intro!: spec.bind.continueI[where v=v]

simp: spec.singleton.le-conv spec.singleton-le-conv trace-steps'-map

trace.natural-def comp-def)

moreover

from continue(3) < $\langle \sigma_f' \rangle \leq \langle trace.map\ id\ snd\ id\ \sigma_f \rangle$ >

have $\langle trace.init\ \sigma_f',\ trace.rest\ \sigma_f'\ @\ trace.rest\ \sigma_g,\ trace.term\ \sigma_g \rangle \leq \langle trace.map\ id\ snd\ id\ ?\sigma \rangle$

by (clarsimp simp: comp-def spec.singleton-le-conv trace.natural-def trace.natural'.append

cong: trace.final'.natural'-cong)

ultimately show ?case

by (force simp: spec.singleton.le-conv)

qed

show ?rhs \leq ?lhs

by (simp add: spec.bind.mono

spec.invmap.bind spec.map-invmap.galois spec.map-invmap.upper-lower-expansive)

qed

lemma smap-sndR:

assumes $UNIV \times (Id \times_R UNIV) \subseteq r$

shows $f \ggg (\lambda v. spec.smap\ snd\ (g\ v)) = spec.smap\ snd\ (spec.rel\ r \sqcap spec.sinvmap\ snd\ f \ggg g)$ (is ?lhs = ?rhs)

proof(rule antisym)

show ?lhs \leq ?rhs

```

proof(induct rule: spec.bind-le)
  case incomplete show ?case
  proof(rule spec.singleton-le-extI)
    show  $\langle \sigma \rangle \leq ?rhs$  if  $\langle \sigma \rangle \leq spec.term.none f$  for  $\sigma$ 
      using assms that
      by (cases  $\sigma$ )
        (force simp: comp-def spec.singleton-le-conv trace-steps'-map
          intro!: exI[where  $x=trace.T (undefined, trace.init \sigma)$ 
            (map (map-prod id (Pair undefined))
              (trace.rest \sigma)) None])
          spec.bind.incompleteI)

  qed
next
  case (continue  $\sigma_f \sigma_g' v$ )
  from  $\langle \sigma_g' \rangle \leq spec.smap snd (g v)$ 
  obtain  $\sigma_g$ 
    where  $\langle \sigma_g \rangle \leq g v$  and  $\langle \sigma_g' \rangle \leq \langle trace.map id snd id \sigma_g \rangle$ 
    by (clarsimp simp: spec.singleton-le-conv)
  let  $? \sigma = trace.T (fst (trace.init \sigma_g), trace.init \sigma_f)$ 
    (map (map-prod id (Pair (fst (trace.init \sigma_g)))) (trace.rest \sigma_f) @ trace.rest \sigma_g)
    (trace.term \sigma_g)
  from continue(2)  $\langle \sigma_g' \rangle \leq \langle trace.map id snd id \sigma_g \rangle$ 
  have snd (trace.init \sigma_g) = trace.final \sigma_f
    by (metis spec.singleton-le-conv trace.less-eqE trace.natural.sel(1) trace.t.map-sel(1))
  with assms continue(1,3)  $\langle \sigma_g \rangle \leq g v$ 
  have  $\langle ? \sigma \rangle \leq spec.rel r \sqcap spec.invmap snd f \ggg g$ 
    by (cases  $\sigma_f$ , cases  $\sigma_g$ )
    (rule spec.bind.continueI[where  $v=v$ ];
      force simp: spec.singleton-le-conv comp-def trace-steps'-map trace.final'.map)
  moreover
  from continue(2)  $\langle \sigma_g' \rangle \leq \langle trace.map id snd id \sigma_g \rangle$ 
  have  $\langle trace.init \sigma_f, trace.rest \sigma_f @ trace.rest \sigma_g', trace.term \sigma_g' \rangle \leq \langle trace.map id snd id ? \sigma \rangle$ 
    by (cases  $\sigma_f$ )
    (clarsimp simp: comp-def spec.singleton-le-conv trace.natural-def trace.natural'.append;
      metis order-le-less same-prefix-prefix trace.less-eqE trace.less-eq-None(2) trace.t.sel(1) trace.t.sel(2)
      trace.t.sel(3))
  ultimately show ?case
    by (force simp: spec.singleton-le-conv)
  qed
show  $?rhs \leq ?lhs$ 
  by (simp add: spec.bind.mono
    spec.invmap.bind spec.map-invmap.galois spec.map-invmap.upper-lower-expansive)
qed

```

lemma localL:

```

  shows spec.local  $f \ggg g = spec.local (f \ggg (\lambda v. spec.localize Id (g v)))$ 
unfolding spec.local-def spec.localize-def spec.local.qrm-def
by (subst spec.bind.smap-sndL[where  $r=\{env\} \times (Id \times_R UNIV) \cup range proc \times (Id \times_R UNIV)$ ];
  fastforce simp: ac-simps spec.bind.inf-rel inf-sup-distrib1 Times-Int-Times
  simp flip: spec.rel.inf)

```

lemma localR:

```

  shows  $f \ggg (\lambda v. spec.local (g v)) = spec.local (spec.localize Id f \ggg g)$ 
unfolding spec.local-def spec.localize-def spec.local.qrm-def
by (subst spec.bind.smap-sndR[where  $r=\{env\} \times (Id \times_R UNIV) \cup range proc \times (Id \times_R UNIV)$ ];
  fastforce simp: ac-simps spec.bind.inf-rel inf-sup-distrib1 Times-Int-Times
  simp flip: spec.rel.inf)

```

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path local} \rangle$

setup $\langle \text{Sign.mandatory-path cam} \rangle$

lemma *cl-le*:

shows $\text{spec.local} (\text{spec.cam.cl} (\{\text{env}\} \times (s \times_R r)) P) \leq \text{spec.cam.cl} (\{\text{env}\} \times r) (\text{spec.local} P)$

unfolding spec.cam.cl-def spec.local.sup $\text{spec.term.all.local}$ $\text{spec.term.none.local}[\text{symmetric}]$

by (*fastforce* *intro*: *le-supI2* *spec.term.none.mono* *spec.bind.mono* *spec.rel.mono*

simp flip: *spec.map-invmap.galois* *spec.rel.inf*

simp: *spec.local-def* *spec.map-invmap.galois* *spec.bind.inf-rel* *spec.invmap.bind* *spec.invmap.rel*)

lemma *cl*:

assumes $\text{Id} \subseteq r_l$

shows $\text{spec.local} (\text{spec.cam.cl} (\{\text{env}\} \times (r_l \times_R r)) P)$

$= \text{spec.cam.cl} (\{\text{env}\} \times r) (\text{spec.local} P)$ (**is** *?lhs* = *?rhs*)

proof(*rule antisym*[*OF spec.local.cam.cl-le*])

have $\text{spec.local} (\text{spec.term.all} P) \gg \text{spec.rel} (\{\text{env}\} \times r)$

$\leq \text{spec.local} (\text{spec.term.all} P \gg \text{spec.rel} (\{\text{env}\} \times (r_l \times_R r)))$

proof(*induct* *rule*: *spec.bind-le*)

case *incomplete* **show** *?case*

by (*simp add*: *spec.term.none.local* *spec.local.mono* *order.trans*[*OF - spec.term.none.bindL-le*])

next

case (*continue* σ_f σ_g v)

from $\langle \text{trace.term } \sigma_f = \text{Some } v \rangle \langle \sigma_f \rangle \leq \text{spec.local} (\text{spec.term.all} P)$

obtain s xs w

where $P: \langle s, xs, w \rangle \leq P$

$\text{trace.steps}' s xs \subseteq \text{spec.local.qrm}$

$\langle \sigma_f \rangle \leq \langle \text{snd } s, \text{map} (\text{map-prod id snd}) xs, \text{trace.term } \sigma_f \rangle$

by (*clarsimp* *simp*: *spec.singleton.local-le-conv* *spec.singleton.le-conv* *spec.singleton-le-conv*
trace.split-all *trace.natural-def*)

let $?s = \text{trace.T } s$

$(xs @ \text{map} (\text{map-prod id} (\text{Pair} (\text{fst} (\text{trace.final}' s xs)))) (\text{trace.rest } \sigma_g))$

$(\text{trace.term } \sigma_g)$

from *assms* *continue*(2,3,4) $P(1,3)$

have $\langle ?s \rangle \leq \text{spec.term.all} P \gg (\lambda::'g. \text{spec.rel} (\{\text{env}\} \times (r_l \times_R r)))$

by (*cases* σ_f)

(*fastforce* *intro*: *spec.bind.continueI*

simp: *spec.singleton.le-conv* *trace.steps'.map*(1)[**where** *af=id* **and** *sf=Pair* (*fst* (*trace.final'* s

xs)) **and** $s=\text{snd} (\text{trace.final}' s xs)$, *simplified*]

simp flip: *trace.final'.map*[**where** *af=id* **and** *sf=snd*]

cong: *trace.final'.natural'-cong*)

moreover

from *continue*(2,3,4) $P(2,3)$

have $\text{trace.steps}' (\text{trace.init } ?s) (\text{trace.rest } ?s) \subseteq \text{spec.local.qrm}$

by (*fastforce* *simp*: *spec.singleton.le-conv* *spec.singleton-le-conv* *trace.natural-def* *trace.natural'.append*

trace.steps'.append *trace.steps'.map*(1)[**where** *af=id* **and** *sf=Pair* (*fst* (*trace.final'* s xs))

and $s=\text{snd} (\text{trace.final}' s xs)$, *simplified*]

simp flip: *trace.final'.map*[**where** *af=id* **and** *sf=snd*]

cong: *trace.final'.natural'-cong*)

moreover

from $\langle \text{trace.term } \sigma_f = \text{Some } v \rangle \langle \sigma_f \rangle \leq \langle \text{snd } s, \text{map} (\text{map-prod id snd}) xs, \text{trace.term } \sigma_f \rangle$

have $\langle \text{trace.init } \sigma_f, \text{trace.rest } \sigma_f @ \text{trace.rest } \sigma_g, \text{trace.term } \sigma_g \rangle \leq \langle \text{trace.map id snd id } ?s \rangle$

by (*cases* σ_f)

(*simp add*: *spec.singleton-le-conv* *trace.natural'.append* *trace.natural-def* *comp-def*

cong: *trace.final'.natural'-cong*)

ultimately show *?case*

```

    by (force simp: spec.singleton.local-le-conv spec.singleton.le-conv)
qed
then show ?rhs ≤ ?lhs
  by (auto simp: spec.cam.cl-def spec.local.sup spec.term.all.local
      simp flip: spec.term.none.local
      intro: le-supI2 spec.term.none.mono)
qed

setup <Sign.parent-path>

setup <Sign.mandatory-path cam.closed>

lemma local:
  assumes Id ⊆ s
  assumes P ∈ spec.cam.closed ({env} × (s ×R r))
  shows spec.local P ∈ spec.cam.closed ({env} × r)
by (metis spec.cam.closed spec.cam.closed-conv[OF assms(2)] spec.local.cam.cl[OF assms(1)])

setup <Sign.parent-path>

setup <Sign.mandatory-path interference>

lemma cl-le:
  shows spec.local (spec.interference.cl ({env} × (s ×R r)) P)
    ≤ spec.interference.cl ({env} × r) (spec.local P)
by (force intro: spec.interference.cl.mono
    simp: spec.local-def spec.map-invmap.upper-lower-expansive spec.map-invmap.galois
    spec.invmap.interference.cl spec.interference.cl.inf-rel)

lemma cl:
  assumes Id ⊆ s
  shows spec.local (spec.interference.cl ({env} × (s ×R r)) P)
    = spec.interference.cl ({env} × r) (spec.local P)
apply (rule antisym[OF spec.local.interference.cl-le])
apply (simp add: spec.interference.cl-def spec.bind.bind spec.bind.localL spec.bind.localR
    spec.invmap.bind spec.invmap.return spec.invmap.rel
    flip: spec.local.cam.cl[OF order.refl])
apply (simp add: spec.localize.bind spec.localize.rel spec.localize.return)
apply (simp add: spec.rel.wind-bind-trailing flip: spec.bind.bind)
apply (simp add: ac-simps inf-sup-distrib1 map-prod-vimage-Times Times-Int-Times
    map-prod-snd-snd-vimage relprod-inter spec.rel.reflcl spec.rel.Id UNIV-unit)
apply (intro spec.local.mono spec.bind.mono spec.rel.mono spec.cam.cl.mono)
using assms apply force+
done

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.mandatory-path interference.closed>

lemma local:
  assumes P ∈ spec.interference.closed ({env} × (Id ×R r))
  shows spec.local P ∈ spec.interference.closed ({env} × r)
by (rule spec.interference.closed-clI)
  (simp flip: spec.interference.closed-conv[OF assms] spec.local.interference.cl[OF order.refl])

lemma local-UNIV:

```

assumes $P \in \text{spec.interference.closed} (\{env\} \times UNIV)$
shows $\text{spec.local } P \in \text{spec.interference.closed} (\{env\} \times UNIV)$
proof –
have $*$: $\{env\} \times (Id \times_R UNIV) \subseteq \{env\} \times UNIV$ **by** *blast*
show *?thesis*
by (*rule spec.interference.closed-clI*)
*(simp flip: spec.interference.closed-conv[OF subsetD[OF spec.interference.closed.antimono[OF *] assms]]*
spec.local.interference.cl[OF order.refl])
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

15.4 *spec.local_init*

setup $\langle \text{Sign.mandatory-path spec} \rangle$

definition *local-init* :: $'a \Rightarrow 'ls \Rightarrow ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec} \Rightarrow ('a \text{ agent}, 's, 'v) \text{ spec}$ **where**
local-init a ls P = spec.local (spec.write (proc a) (map-prod ⟨ls⟩ id) \gg P)

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma *local-init-le-conv*:

shows $\langle \sigma \rangle \leq \text{spec.local-init } a \text{ } ls \text{ } P$
 $\longleftrightarrow \langle \sigma \rangle \leq \text{spec.idle} \vee (\exists \sigma'. \langle \sigma' \rangle \leq P$
 $\quad \wedge \text{trace.steps } \sigma' \subseteq \text{spec.local.qrm}$
 $\quad \wedge \langle \sigma \rangle \leq \langle \text{trace.map id snd id } \sigma' \rangle$
 $\quad \wedge \text{fst } (\text{trace.init } \sigma') = ls) \text{ (is } ?lhs \longleftrightarrow ?rhs)$

proof(*rule iffI*)

assume *?lhs*

then obtain σ'

where $\langle \sigma' \rangle \leq \text{spec.write } (\text{proc } a) (\text{map-prod } \langle ls \rangle \text{ id}) \gg P$
and $\text{trace.steps}' (\text{trace.init } \sigma') (\text{trace.rest } \sigma') \subseteq \text{spec.local.qrm}$
and $\langle \sigma \rangle \leq \langle \text{trace.map id snd id } \sigma' \rangle$

by (*clarsimp simp: spec.local-init-def spec.singleton.local-le-conv*)

then show *?rhs*

proof(*induct rule: spec.singleton.bind-le*)

case *incomplete* **then show** *?case*

by (*cases* σ')

(*rule disjI1*;

fastforce simp: spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv
elim!: order.trans)

next

case (*continue* $\sigma_f \sigma_g v_f$) **then show** *?case*

by (*cases* σ_g)

(*rule disjI2*; *erule* (1) *spec.singleton.action-Some-leE*;

*force simp: exI[**where** $x = \sigma_g$] spec.singleton.le-conv image-image*
trace.steps'.append trace-steps'-snd-le-const)

qed

next

show *?rhs* \implies *?lhs*

by (*fastforce simp: spec.local-init-def spec.idle-le trace.split-all spec.singleton.local-le-conv*

*intro!: spec.bind.continueI[**where** $xs = []$, *simplified*] spec.action.stutterI*
elim!: order.trans)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path idle} \rangle$

lemma *local-init-le[spec.idle-le]*:

shows $\text{spec.idle} \leq \text{spec.local-init a ls P}$

by (*simp add: spec.local-init-def spec.idle-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path local-init} \rangle$

lemma *Sup*:

shows $\text{spec.local-init a ls } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.local-init a ls } x) \sqcup \text{spec.idle}$

apply (*simp add: spec.local-init-def spec.local.Sup spec.local.sup image-image*

spec.bind.SUPR(1)[where X=X and g=pred-K, simplified]

spec.bind.botR spec.local.action

flip: bot-fun-def spec.term.none.local)

apply (*subst spec.return.cong, force, force*)

apply (*simp add: spec.term.none.return spec.term.none.Sup spec.term.none.sup spec.term.none.idle sup.absorb2*)

done

lemma *Sup-not-empty*:

assumes $X \neq \{\}$

shows $\text{spec.local-init a ls } (\bigsqcup X) = (\bigsqcup x \in X. \text{spec.local-init a ls } x)$

by (*subst spec.local-init.Sup*) (*meson assms spec.local-init.Sup sup.absorb1 SUP1 spec.idle-le ex-in-conv*)

lemmas $\text{sup} = \text{spec.local-init.Sup-not-empty}[\text{where } X=\{X, Y\} \text{ for } X Y, \text{simplified}]$

lemma *bot*:

shows $\text{spec.local-init a ls } \perp = \text{spec.idle}$

using *spec.local-init.Sup[where X={}] by simp*

lemma *top*:

shows $\text{spec.local-init a ls } \top = (\top :: ('a \text{ agent}, 's, 'v) \text{ spec})$

proof –

have $\langle \sigma \rangle \leq \text{spec.local-init a ls } \top$ **for** $\sigma :: ('a \text{ agent}, 's, 'v) \text{ trace.t}$

by (*fastforce simp: spec.local-init-def spec.local.qrm-def spec.singleton.local-le-conv trace.steps'.map comp-def intro: exI[where x=trace.T (ls, trace.init σ) (map (map-prod id (Pair ls)) (trace.rest σ)) (trace.term σ)]*

spec.bind.continueI[where xs=[]], simplified] spec.action.stutterI)

then show *?thesis*

by (*fastforce intro: top-le spec.singleton-le-extI simp: spec.local-init-def*)

qed

lemma *monotone*:

shows $\text{mono} (\text{spec.local-init a ls} :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec} \Rightarrow -)$

proof(*rule monotoneI*)

show $\text{spec.local-init a ls } P \leq \text{spec.local-init a ls } P'$ **if** $P \leq P'$ **for** $P P' :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$

unfolding *spec.local-init-def* **by** (*strengthen ord-to-strengthen(1)[OF $\langle P \leq P' \rangle$] simp*

qed

lemmas $\text{strengthen[strg]} = \text{st-monotone}[OF \text{spec.local-init.monotone}]$

lemmas $\text{mono} = \text{monotoneD}[OF \text{spec.local-init.monotone}]$

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes $\text{monotone orda } (\leq) P$

shows $\text{monotone orda } (\leq) (\lambda x. \text{spec.local-init a ls } (P x))$

by (simp add: monotone2monotone[OF spec.local-init.monotone assms])

lemma mcont2mcont[cont-intro]:

assumes mcont luba orda Sup (\leq) P

shows mcont luba orda Sup (\leq) ($\lambda x. \text{spec.local-init } a \text{ } ls (P \ x)$)

by (simp add: spec.local-init-def assms)

lemma action:

fixes F :: ('v \times 'a agent \times ('ls \times 's) \times ('ls \times 's)) set

shows spec.local-init a ls (spec.action F)

= spec.action {(v, a, s, s') | v a ls' s s'. (v, a, (ls, s), (ls', s')) \in F \wedge (a = env \longrightarrow ls' = ls)} (is ?lhs = ?rhs)

proof(rule antisym)

show ?lhs \leq ?rhs

apply (subst (3) spec.local.localize[**where** r=UNIV, symmetric], simp)

apply (simp add: spec.local-init-def)

apply (rule spec.seq-ctxt.cl-imp-local-le)

apply (simp add: spec.localize.action spec.bind.inf-rel spec.action.inf-rel spec.return.inf-rel)

apply (strengthen ord-to-strengthen(2)[OF spec.return.rel-le])

apply (fastforce intro: spec.seq-ctxt.cl-action-mumble-expandL-le[**where** P= \top , simplified]

simp: spec.local.qrm-def spec.bind.returnL spec.idle-le)

done

show ?rhs \leq ?lhs

apply (subst (1) spec.local.localize[**where** r=UNIV, symmetric], simp)

apply (simp add: spec.local-init-def)

apply (rule spec.seq-ctxt.cl-imp-local-le)

apply (simp add: spec.localize.action spec.bind.inf-rel spec.action.inf-rel spec.return.inf-rel

spec.rel.Id spec.bind.returnL spec.idle-le UNIV-unit

flip: spec.rel.inf)

apply (fastforce intro: spec.seq-ctxt.cl-action-mumbleL-le[**where** P= \top , simplified]

simp: spec.local.qrm-def)

done

qed

lemma return:

shows spec.local-init a ls (spec.return v) = spec.return v

by (auto simp: spec.local-init.action spec.return-def intro: arg-cong[**where** f=spec.action])

lemma localize-le:

assumes spec.idle \leq P

shows spec.local-init a ls (spec.localize r P) \leq P

unfolding spec.local-init-def spec.localize-def

apply (rule order.trans[OF spec.local.bind-le])

apply (simp add: spec.local.action)

apply (subst spec.return.cong, force, force)

apply (simp add: assms spec.bind.SupL spec.bind.supL

spec.bind.returnL spec.idle.local-le spec.idle.invmap-le spec.idle.rel-le)

apply (simp add: le-infI2 spec.local-def spec.map-invmap.galois)

done

lemma localize:

assumes spec.idle \leq P

assumes Id \subseteq r

shows spec.local-init a ls (spec.localize r P) = P (is ?lhs = ?rhs)

proof(rule antisym[OF spec.local-init.localize-le[OF assms(1)] spec.singleton-le-extI])

show $\langle \sigma \rangle \leq$?lhs **if** $\langle \sigma \rangle \leq$?rhs **for** σ

using that

by (cases σ)

(fastforce simp: spec.local-init-def spec.localize-def spec.local.qrm-def comp-def

spec.singleton.le-conv spec.singleton.local-le-conv trace-steps'-map
*intro: exI[**where** $x = \text{trace.map id (Pair ls) id } \sigma]$ subsetD[$OF \langle Id \subseteq r \rangle$]*
*spec.bind.continueI[**where** $xs = []$, simplified] spec.action.stutterI)*

qed

lemma *inf-interference:*

shows *spec.local-init a ls $P = \text{spec.local-init a ls } (P \sqcap \text{spec.local.interference})$*

unfolding *spec.local-init-def*

by (*subst spec.local.inf-interference*)

(*auto simp: ac-simps spec.bind.inf-rel spec.action.inf-rel*

*intro: arg-cong[**where** $f = \lambda x. \text{spec.local (spec.action } x \gg (\lambda x. P \sqcap \text{spec.local.interference}))]$])*)

lemma *eq-local:*

assumes *spec.idle $\leq P$*

shows ($\sqcup ls. \text{spec.local-init a ls } P) = \text{spec.local } P$

proof –

have *spec.local (spec.action {(((), proc a, (ls, s), ls', s) | $ls' ls s. \text{True}$ }) $\gg P$)*
 $= \text{spec.local } P$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

have *?rhs = spec.return () \gg ?rhs*

by (*simp add: spec.bind.returnL assms spec.idle.local-le*)

also have $\dots = \text{spec.smap snd (spec.invmap snd (spec.return ())) $\gg \text{spec.rel spec.local.qrm } \sqcap P$$

by (*simp add: spec.local-def spec.bind.smap-sndR[**where** $r = UNIV$, simplified spec.rel.UNIV]*)

also have *?lhs $\leq \dots$*

by (*force intro: spec.map.mono spec.bind.mono le-infI2 spec.action.rel-le*

simp: spec.local-def spec.invmap.return spec.bind.bind spec.bind.return spec.bind.inf-rel)

finally show *?lhs \leq ?rhs .*

show *?rhs \leq ?lhs*

by (*force simp: assms*

*intro: spec.local.mono order.trans[$OF \text{spec.bind.returnL-le[**where** } g = \langle P \rangle$ and $v = ()$]]*
spec.return.action-le spec.bind.mono)

qed

then show *?thesis*

by (*simp add: spec.local-init-def UNION-eq*

*flip: spec.local.Sup[**where** $X = (\lambda ls. \text{spec.write (proc a) (map-prod } \langle ls \rangle \text{ id}) $\gg P$) ' UNIV, simplified,$*

simplified image-image]
spec.bind.SUPL spec.action.SUP-not-empty)

qed

lemma *ag-le:*

shows *spec.local-init a ls ($\{P\}, Id \times_R A \vdash UNIV \times_R G, \{\lambda v s. Q v (\text{snd } s)\}$)*
 $\leq \{\lambda s. P (ls, s)\}, A \vdash G, \{Q\}$

apply (*subst ag.reflcl-a*)

apply (*simp add: spec.local-init-def spec.local-def spec.local.qrm-def*

spec.map-invmap.galois spec.invmap.ag map-prod-snd-snd-vimage)

apply (*subst inf.commute*)

apply (*subst heyting[symmetric]*)

apply (*subst sup.commute, subst ag.assm-heyting*)

apply (*force intro: ag.spec.bind[rotated] ag.spec.action[**where** $Q = \lambda -. \text{FST } ((=) ls) \wedge P$] ag.mono)*

done

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemma *local-initL:*

shows *spec.local-init a ls $f \gg g = \text{spec.local-init a ls } (f \gg (\lambda v. \text{spec.localize Id } (g v)))$*

by (*simp add: spec.local-init-def spec.bind.localL spec.bind.bind*)

lemma *local-initR*:

shows $f \gg= (\lambda v. \text{spec.local-init } a \text{ } ls (g \ v)) = \text{spec.local-init } a \text{ } ls (\text{spec.localize } Id \ f \gg= g)$
oops

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path } \text{sinvmap} \rangle$

lemma *local-init*:

fixes $P :: ('a \ \text{agent}, 'ls \times 't, 'v) \ \text{spec}$

fixes $sf :: 's \Rightarrow 't$

shows $\text{spec.sinvmap } sf (\text{spec.local-init } a \text{ } ls \ P)$

$= \text{spec.local-init } a \text{ } ls (\text{spec.rel } (UNIV \times (Id \times_R \ \text{map-prod } sf \ sf \ - ' Id)))$
 $\gg= (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod } id \ sf) \ P) \ (\text{is } ?lhs = ?rhs)$

proof(*rule antisym*)

let $?r = UNIV \times (Id \times_R \ \text{map-prod } sf \ sf \ - ' Id)$

have $?lhs = \text{spec.local } (\text{spec.rel } ?r \gg=$

$(\lambda :: \text{unit}. \text{spec.action } (\{((), \ \text{proc } a, (ls', \ s), \ ls, \ s') \mid ls' \ s \ s'. \ sf \ s = sf \ s'\}) \gg=$
 $(\lambda :: \text{unit}. \text{spec.rel } ?r \gg= (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod } id \ sf) \ P))))$

by (*simp add: spec.local-init-def spec.invmap.local spec.invmap.bind spec.invmap.action*
spec.bind.bind map-prod-conv map-prod-map-prod-vimage-Id)

also have $\dots \leq \text{spec.local } (\text{spec.rel } ?r \gg=$

$(\lambda :: \text{unit}. \text{spec.write } (\text{proc } a) (\text{map-prod } \langle ls \rangle \ id) \gg=$

$(\lambda :: \text{unit}. \text{spec.rel.act } ?r \gg=$

$(\lambda :: \text{unit}. \text{spec.rel } ?r \gg= (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod } id \ sf) \ P))))$

apply (*rule spec.seq-ctxt.cl-imp-local-le*)

apply (*simp add: ac-simps spec.bind.inf-rel spec.action.inf-rel spec.rel.act-def flip: spec.rel.inf*)

apply (*subst (4) spec.bind.bind[symmetric]*)

apply (*(rule spec.seq-ctxt.cl-action-mumbleL-le[where P= \top , simplified]; force)*

| rule order.trans[OF spec.bind.mono spec.bind.seq-ctxt.cl-le] spec.seq-ctxt.expansive)+

done

also have $\dots = \text{spec.local } (\text{spec.rel } ?r \gg=$

$(\lambda :: \text{unit}. \text{spec.write } (\text{proc } a) (\text{map-prod } \langle ls \rangle \ id) \gg=$

$(\lambda :: \text{unit}. \text{spec.rel } ?r \gg= (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod } id \ sf) \ P))))$

by (*simp add: spec.rel.wind-bind flip: spec.bind.bind spec.rel.unfoldL*)

also have $\dots \leq \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle ls \rangle \ id) \gg=$

$(\lambda :: \text{unit}. \text{spec.rel } ?r \gg=$

$(\lambda :: \text{unit}. \text{spec.rel } ?r \gg= (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod } id \ sf) \ P))))$

by (*rule spec.local.init-write-interference2-permute-le*)

also have $\dots = \text{spec.local } (\text{spec.write } (\text{proc } a) (\text{map-prod } \langle ls \rangle \ id) \gg=$

$(\lambda :: \text{unit}. \text{spec.rel } ?r \gg= (\lambda :: \text{unit}. \text{spec.sinvmap } (\text{map-prod } id \ sf) \ P))))$

by (*simp add: spec.rel.wind-bind flip: spec.bind.bind*)

also have $\dots = ?rhs$

by (*simp add: spec.local-init-def*)

finally show $?lhs \leq ?rhs$.

show $?rhs \leq ?lhs$

by (*fastforce simp: spec.local-init-def spec.invmap.local spec.invmap.bind spec.invmap.action*

map-prod-conv spec.bind.bind map-prod-map-prod-vimage-Id

intro: spec.local.mono order.trans[OF - spec.bind.relL-le] spec.bind.mono spec.action.mono)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path } \text{vmap} \rangle$

lemma *local-init*:

shows $\text{spec.vmap } vf (\text{spec.local-init } a \text{ } ls \ P) = \text{spec.local-init } a \text{ } ls (\text{spec.vmap } vf \ P)$

by (simp add: spec.local-init-def spec.vmap.local spec.map.bind-inj-sf spec.map.id)

setup <Sign.parent-path>

setup <Sign.mandatory-path vinvmap>

lemma local-init:

shows spec.vinvmap vf (spec.local-init a ls P) = spec.local-init a ls (spec.vinvmap vf P)

by (simp add: spec.local-init-def spec.invmap.local spec.invmap.bind spec.invmap.action spec.rel.Id UNIV-unit spec.bind.returnL spec.idle-le)

setup <Sign.parent-path>

setup <Sign.mandatory-path term.none>

lemma local-init:

shows spec.term.none (spec.local-init a ls P) = spec.local-init a ls (spec.term.none P)

by (simp add: spec.local-init-def spec.term.none.local spec.term.none.bind)

setup <Sign.parent-path>

setup <Sign.mandatory-path term.all>

lemma local-init:

shows spec.term.all (spec.local-init a ls P)

= spec.local-init a ls (spec.term.all P) \sqcup \sqcup range spec.return

apply (simp add: spec.local-init-def spec.term.all.local spec.term.all.bind spec.term.all.action spec.local.Sup spec.local.sup spec.local.action spec.local.return image-image ac-simps)

apply (subst (2) spec.return.cong, force, force intro!: exI[**where** x=proc a])

apply (rule antisym; clarsimp simp: le-supI1 le-supI2 SUP-upper SUP-upper2 spec.idle-le)

done

setup <Sign.parent-path>

setup <Sign.mandatory-path interference.closed>

lemma local-init:

assumes $P \in \text{spec.interference.closed } (\{env\} \times (Id \times_R r))$

shows $\text{spec.local-init } a \text{ } ls \ P \in \text{spec.interference.closed } (\{env\} \times r)$

by (rule spec.interference.closed-clI)

(simp add: spec.local-init-def spec.bind.bind spec.interference.cl.action

spec.interference.cl.bindR[OF assms] spec.interference.closed.bind-relL[OF assms]

order.trans[OF spec.local.init-write-interference2-permute-le[simplified]])

flip: spec.local.interference.cl[**where** s=Id])

setup <Sign.parent-path>

setup <Sign.parent-path>

15.5 Hoist to ('s, 'v) prog

setup <Sign.mandatory-path prog>

lift-definition local :: ('ls \times 's, 'v) prog \Rightarrow ('s, 'v) prog is spec.local

by (blast intro: spec.interference.closed.local subsetD[OF spec.interference.closed.antimono, rotated])

definition local-init :: 'ls \Rightarrow ('ls \times 's, 'v) prog \Rightarrow ('s, 'v) prog **where**

local-init ls P = prog.local (prog.write (map-prod <ls> id) \gg P)

— equivalent to lifting *spec.local-init*; see *prog.p2s.local-init*

lift-definition *localize* :: ('s, 'v) prog ⇒ ('ls × 's, 'v) prog **is** *spec.localize UNIV*
by (*rule spec.interference.closed.localize*)

setup ‹*Sign.mandatory-path p2s*›

lemmas *local[prog.p2s.simps] = prog.local.rep-eq*

lemma *local-init[prog.p2s.simps]*:

shows *prog.p2s (prog.local-init ls P) = spec.local-init () ls (prog.p2s P) (is ?lhs = ?rhs)*

proof(*rule antisym*)

show *?lhs ≤ ?rhs*

by (*simp add: prog.local-init-def spec.local-init-def*
prog.p2s.simps prog.p2s.action prog.p2s.interference-wind-bind
map-prod-image-Collect spec.interference.cl.action spec.bind.bind
order.trans[OF spec.local.init-write-interference-permute-le[simplified]])

show *?rhs ≤ ?lhs*

by (*simp add: prog.local-init-def spec.local-init-def prog.p2s.simps prog.p2s.action*
map-prod-image-Collect
spec.local.mono[OF spec.bind.mono[OF spec.interference.expansive order.refl]])

qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local*›

lemma *Sup*:

shows *prog.local (⊔ X) = (⊔ x∈X. prog.local x)*

by *transfer*

(*simp add: spec.local.Sup spec.local.sup spec.interference.cl.bot spec.local.interference*
flip: spec.term.none.local)

lemmas *sup = prog.local.Sup[where X={X, Y} for X Y, simplified]*

lemma *bot*:

shows *prog.local ⊥ = ⊥*

using *prog.local.Sup[where X={}] by simp*

lemma *top*:

shows *prog.local ⊤ = ⊤*

by *transfer (simp add: spec.local.top)*

lemma *monotone*:

shows *mono prog.local*

by (*rule monotoneI*) (*transfer; erule monotoneD[OF spec.local.monotone]*)

lemmas *strengthen[strg] = st-monotone[OF prog.local.monotone]*

lemmas *mono = monotoneD[OF prog.local.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*

= monotone2monotone[OF prog.local.monotone, simplified, of orda P for orda P]

lemma *mcont2mcont[cont-intro]*:

assumes *mcont luba orda Sup (≤) P*

shows *mcont luba orda Sup (≤) (λx. prog.local (P x))*

proof(*rule mcontI*)

from *assms show monotone orda (≤) (λx. prog.local (P x))*

by (*blast intro: mcont-mono prog.local.mono2mono*)

from *assms* **show** *cont luba orda Sup* (\leq) ($\lambda x. \text{prog.local } (P x)$)
by (*fastforce intro: contI dest: mcont-cont contD simp: prog.local.Sup image-image*)
qed

lemma *bind-botR*:

shows *prog.local* ($P \gg= \perp$) = *prog.local* $P \gg= \perp$
by (*simp add: prog.p2s.simps spec.interference.cl.bot bot-fun-def*
spec.interference.closed.bind-relR spec.interference.closed.local-UNIV
flip: spec.bind.botR prog.p2s-inject)
(*simp add: spec.bind.localL spec.localize.bot*)

lemma *action*:

shows *prog.local* (*prog.action* F) = *prog.action* (*map-prod id* (*map-prod snd snd*) ‘ F)
by *transfer*
(*force simp: spec.local.interference.cl[OF subset-UNIV, where r=UNIV, simplified] spec.local.action*
intro: arg-cong[where f= $\lambda F. \text{spec.interference.cl} (\{env\} \times UNIV) (\text{spec.action } F)$])

lemma *return*:

shows *prog.local* (*prog.return* v) = *prog.return* v
by (*simp add: prog.return-def prog.local.action map-prod-image-Times map-prod-snd-snd-image-Id*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path local-init} \rangle$

lemma *transfer[transfer-rule]*:

shows *rel-fun* (=) (*rel-fun cr-prog cr-prog*) (*spec.local-init* ()) *prog.local-init*
by (*simp add: cr-prog-def prog.p2s.local-init rel-fun-def*)

lemma *Sup*:

shows *prog.local-init* ls ($\bigsqcup X$) = ($\bigsqcup x \in X. \text{prog.local-init } ls x$)
by (*simp add: prog.local-init-def prog.bind.SupR prog.local.Sup prog.local.sup image-image*
prog.local.bind-botR prog.local.action)
(*subst prog.return.cong; force simp: prog.bind.returnL*)

lemmas *sup* = *prog.local-init.Sup*[**where** $X = \{X, Y\}$ **for** $X Y$, *simplified*]

lemma *bot[simp]*:

shows *prog.local-init* $ls \perp = \perp$
using *prog.local-init.Sup*[**where** $ls = ls$ **and** $X = \{\}$] **by** *simp*

lemma *top*:

shows *prog.local-init* $ls \top = \top$
by (*simp add: prog.p2s.simps spec.local-init.top flip: prog.p2s-inject*)

lemma *monotone*:

shows *mono* (*prog.local-init* ls)
unfolding *prog.local-init-def* **by** *simp*

lemmas *strengthen[strg]* = *st-monotone*[*OF prog.local-init.monotone*]

lemmas *mono* = *monotoneD*[*OF prog.local-init.monotone*]

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone orda* (\leq) P
shows *monotone orda* (\leq) ($\lambda x. \text{prog.local-init } ls (P x)$)
by (*simp add: monotone2monotone[OF prog.local-init.monotone assms]*)

lemma *mcont2mcont[cont-intro]*:

assumes $mcont\ luba\ orda\ Sup\ (\leq)\ P$
shows $mcont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. prog.local-init\ ls\ (P\ x))$
proof(*rule mcontI*)
from *assms* **show** $monotone\ orda\ (\leq)\ (\lambda x. prog.local-init\ ls\ (P\ x))$
by (*blast intro: mcont-mono prog.local-init.mono2mono*)
from *assms* **show** $cont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. prog.local-init\ ls\ (P\ x))$
by (*fastforce intro: contI dest: mcont-cont contD simp: prog.local-init.Sup image-image*)
qed

lemma *bind-botR*:
shows $prog.local-init\ ls\ (P\ \gg\ \perp) = prog.local-init\ ls\ P\ \gg\ \perp$
by (*simp add: prog.local-init-def prog.local.bind-botR flip: prog.bind.bind*)

lemma *return*:
shows $prog.local-init\ ls\ (prog.return\ v) = prog.return\ v\ (is\ ?lhs = ?rhs)$
proof –
have $prog.p2s\ ?rhs = spec.local-init\ ()\ ls\ (spec.localize\ UNIV\ (spec.rel\ (\{env\} \times UNIV)\ \gg\ (\lambda :: unit. spec.return\ v)))$
by (*simp add: prog.p2s.simps prog.p2s.return spec.interference.cl.return spec.local-init.localize spec.idle-le*)
also have $\dots = spec.local-init\ ()\ ls\ (spec.rel\ (\{env\} \times UNIV)\ \gg\ (\lambda :: unit. spec.return\ v))$
by (*simp add: spec.localize.bind spec.localize.rel spec.localize.return spec.bind.inf-rel spec.return.inf-rel map-prod-vimage-Times map-prod-snd-snd-vimage ac-simps Int-Un-distrib Int-Un-distrib2 Times-Int-Times relprod-inter spec.rel.reflcl spec.rel.wind-bind-trailing times-subset-iff Id-le-relprod-conv flip: spec.rel.inf spec.bind.bind*)
also have $\dots = prog.p2s\ ?lhs$
by (*simp add: prog.p2s.simps prog.p2s.return spec.interference.cl.return*)
finally show *?thesis*
by (*simp add: p2s-inject*)
qed

lemma *eq-local*:
shows $(\bigsqcup\ ls. prog.local-init\ ls\ P) = prog.local\ P$
by *transfer*
(simp add: spec.local-init.eq-local spec.idle-le sup.absorb1 spec.interference.least spec.interference.closed.local-UNIV)

setup $\langle Sign.parent-path \rangle$

lemma *localize-alt-def*:
shows $prog.localize\ P = prog.rel\ (Id\ \times_R\ UNIV)\ \sqcap\ prog.sinvmap\ snd\ P$
by *transfer* (*simp add: spec.localize-def ac-simps*)

setup $\langle Sign.mandatory-path\ localize \rangle$

lemma *Sup*:
shows $prog.localize\ (\bigsqcup\ X) = (\bigsqcup\ x \in X. prog.localize\ x)$
by (*simp add: prog.localize-alt-def prog.invmap.Sup inf-Sup inf-sup-distrib image-image prog.bind.inf-rel inv-image-alt-def map-prod-snd-snd-vimage relprod-inter prog.rel.Id prog.rel.empty prog.bind.returnL prog.invmap.bot UNIV-unit flip: prog.rel.inf*)

lemmas $sup = prog.localize.Sup$ [where $X = \{X, Y\}$ for $X\ Y$, *simplified*]

lemma *bot*:
shows $prog.localize\ \perp = \perp$

using *prog.localize.Sup*[**where** $X=\{\}$] **by** *simp*

lemma *top*:

shows *prog.localize* $\top = \text{prog.rel } (Id \times_R UNIV)$

by *transfer* (*simp add: spec.localize.top ac-simps*)

lemma *monotone*:

shows *mono prog.localize*

by (*rule monotoneI*) (*transfer; simp add: spec.interference.cl.mono spec.localize.mono*)

lemmas *strengthen[strg] = st-monotone[OF prog.localize.monotone]*

lemmas *mono = monotoneD[OF prog.localize.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]*

= monotone2monotone[OF prog.localize.monotone, simplified, of orda P for orda P]

lemma *mcont2mcont[cont-intro]*:

assumes *mcont luba orda Sup* (\leq) P

shows *mcont luba orda Sup* (\leq) ($\lambda x. \text{prog.localize } (P x)$)

proof(*rule mcontI*)

from *assms show monotone orda* (\leq) ($\lambda x. \text{prog.localize } (P x)$)

by (*blast intro: mcont-mono prog.localize.mono2mono*)

from *assms show cont luba orda Sup* (\leq) ($\lambda x. \text{prog.localize } (P x)$)

by (*fastforce intro: contI dest: mcont-cont contD simp: prog.localize.Sup image-image*)

qed

lemmas *p2s[prog.p2s.simps] = prog.localize.rep-eq*

lemma *bind*:

shows *prog.localize* ($f \ggg g$) = *prog.localize* $f \ggg (\lambda v. \text{prog.localize } (g v))$

by *transfer*

(*simp add: spec.localize.bind spec.interference.least spec.interference.closed.bind spec.bind.mono*)

lemma *parallel*:

shows *prog.localize* ($P \parallel Q$) = *prog.localize* $P \parallel \text{prog.localize } Q$

by *transfer* (*simp add: spec.localize.parallel*)

lemma *rel*:

fixes $r :: 's \text{ rel}$

shows *prog.localize* (*prog.rel* r) = *prog.rel* ($Id \times_R r$)

by (*subst* (2) *prog.rel.reflcl[symmetric]*)

(*transfer; auto simp: spec.localize.rel intro: arg-cong[where f=spec.rel]*)

lemma *action*:

shows *prog.localize* (*prog.action* F)

= prog.action (*map-prod id* (*map-prod snd snd*) $- ' F \cap UNIV \times (Id \times_R UNIV)$)

by (*simp add: prog.localize-alt-def prog.invmap.action prog.bind.inf-rel prog.action.inf-rel*

map-prod-snd-snd-vimage relprod-inter prog.rel.Id UNIV-unit refl-relprod-conv

prog.bind.return prog.return.rel-le

inf.absorb2

flip: prog.rel.inf)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path local} \rangle$

lemma *localize*:

fixes $P :: ('s, 'v) \text{ prog}$

shows *prog.local* (*prog.localize* $P :: ('ls \times 's, 'v) \text{ prog}$) = P

by transfer

(simp add: spec.local.interference.cl[**where** $r=UNIV$ **and** $s=UNIV$, simplified] spec.local.localize
flip: spec.interference.closed-conv)

setup <Sign.parent-path>

setup <Sign.parent-path>

15.6 Refinement rules

setup <Sign.mandatory-path spec>

We use *localizeA* to hoist assumes similarly to *spec.localize*.

definition *localizeA* :: (sequential, 's, 'v) spec \Rightarrow (sequential, 'ls \times 's, 'v) spec **where**
localizeA P = spec.local.interference \sqcap spec.sinvmap snd P

setup <Sign.mandatory-path localizeA>

lemma *bot*:

shows spec.localizeA $\perp = \perp$

by (simp add: spec.localizeA-def spec.invmap.bot)

lemma *top*:

shows spec.localizeA $\top =$ spec.local.interference

by (simp add: spec.localizeA-def spec.invmap.top)

lemma *ag-assm*:

shows spec.localizeA (ag.assm A) = ag.assm (Id \times_R A)

apply (simp add: spec.localizeA-def spec.invmap.rel spec.local.qrm-def flip: spec.rel.inf)

apply (subst (1 2) spec.rel.reflcl[**where** $A=UNIV$, symmetric])

apply (auto intro: arg-cong[**where** $f=spec.rel$])

done

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.mandatory-path refinement>

setup <Sign.mandatory-path spec>

lemma *localI*: — Introduce local state

fixes A :: (sequential, 's, 'v) spec

fixes c :: (sequential, 'ls \times 's, 'v) spec

fixes c' :: (sequential, 's, 'v) spec

fixes P :: 's pred

fixes Q :: 'v \Rightarrow 's pred

assumes $c \leq \{\lambda s. P (snd s)\}$, spec.localizeA A \Vdash spec.sinvmap snd c', $\{\lambda v s. Q v (snd s)\}$

shows spec.local c $\leq \{P\}$, A \Vdash c', $\{Q\}$

apply (simp only: spec.local-def spec.map-invmap.galois spec.invmap.refinement id-apply)

apply (subst inf.commute)

apply (subst heyting[symmetric])

apply (strengthen ord-to-strengthen[OF assms])

apply (strengthen ord-to-strengthen[OF refinement.heyting-le])

apply (simp add: refinement.mono[OF order.refl - - order.refl] heyting spec.localizeA-def)

done

setup <Sign.mandatory-path seq-ctxt>

lemma *local-seq-ctxt-cl*:

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $P :: 's \text{ pred}$
fixes $Q :: 'v \Rightarrow 's \text{ pred}$
fixes $c :: (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$
fixes $c' :: (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$
assumes $\text{spec.local.interference} \sqcap c$
 $\leq \{\!\{ \lambda s. P (\text{snd } s) \}\!\}, \text{spec.localizeA } A \Vdash \text{spec.seq-ctxt.cl False } (\text{spec.local.interference} \sqcap c'), \{\!\{ \lambda v s. Q v (\text{snd } s) \}\!\}$
shows $\text{spec.local } c \leq \{\!\{ P \}\!\}, A \Vdash \text{spec.local } c', \{\!\{ Q \}\!\}$
apply (*simp only: spec.local-def spec.map-invmap.galois spec.invmap.refinement id-apply*)
apply (*subst inf.left-idem[symmetric]*) — non-linear use of env constraint
apply (*strengthen ord-to-strengthen(1)[OF assms]*)
apply (*subst inf commute*)
apply (*subst heyting[symmetric]*)
apply (*strengthen ord-to-strengthen(2)[OF refinement.heyting-le]*)
apply (*subst inf commute, fold spec.localizeA-def*)
apply (*rule refinement.mono[OF order.refl order.refl - order.refl]*)
apply (*strengthen ord-to-strengthen(1)[OF spec.seq-ctxt.cl-local-le]*)
apply (*simp add: heyting flip: spec.map-invmap.cl-def*)
done

lemma *cl-bind*:

fixes $f :: ('a \text{ agent}, 'ls \times 's, 'v) \text{ spec}$
fixes $g :: 'v \Rightarrow ('a \text{ agent}, 'ls \times 's, 'w) \text{ spec}$
assumes $g: \bigwedge v. g v \leq \{\!\{ Q' v \}\!\}, \text{refinement.spec.bind.res } (\text{spec.pre } P \sqcap \text{spec.term.all } A \sqcap \text{spec.seq-ctxt.cl True } f')$
 $A v \Vdash \text{spec.seq-ctxt.cl } T (g' v), \{\!\{ Q \}\!\}$
assumes $f: f \leq \{\!\{ P \}\!\}, \text{spec.term.all } A \Vdash \text{spec.seq-ctxt.cl True } f', \{\!\{ Q' \}\!\}$
shows $f \ggg g \leq \{\!\{ P \}\!\}, A \Vdash \text{spec.seq-ctxt.cl } T (f' \ggg g'), \{\!\{ Q \}\!\}$
by (*strengthen ord-to-strengthen[OF spec.bind.seq-ctxt.cl-le]*) (*rule refinement.spec.bind[OF assms]*)

lemma *cl-action-permuteL*:

fixes $F :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $G :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $G' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $F' :: 'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $Q :: 'w \Rightarrow ('ls \times 's) \text{ pred}$
assumes $F: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in F \rrbracket \Longrightarrow \text{snd } s' = \text{snd } s$
assumes $FGG'F': \bigwedge v w a a' s s' t. \llbracket P s; (v, a', s, t) \in F; (w, a, t, s') \in G v \rrbracket$
 $\Longrightarrow \exists v' a'' a''' t'. (v', a'', s, t') \in G' \wedge (w, a''', t', s') \in F' v'$
 $\wedge \text{snd } s' = \text{snd } t' \wedge (\text{snd } s \neq \text{snd } t' \longrightarrow a'' = a)$
assumes $Q: \bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in G'; (w, a', s', s'') \in F' v \rrbracket \Longrightarrow Q w s''$
shows $\text{spec.action } F \ggg (\lambda v. \text{spec.action } (G v)) \leq \{\!\{ P \}\!\}, A \Vdash \text{spec.seq-ctxt.cl } T (\text{spec.action } G' \ggg (\lambda v. \text{spec.action } (F' v))), \{\!\{ Q \}\!\}$
apply (*strengthen ord-to-strengthen[OF top-greatest[where a=T]]*)
apply (*rule order.trans[OF spec.seq-ctxt.cl-action-permuteL-le[where T=True and F'=F' and G'=G', simplified heyting[symmetric]]]*)
apply (*erule (1) F*)
apply (*drule (2) FGG'F', blast*)
apply (*simp only: refinement-def spec.pre.next-imp-eq-heyting spec.idle-le inf.bounded-iff*)
apply (*rule order.trans[OF - heyting.mono[OF order.refl spec.next-imp.mono[OF top-greatest order.refl]]]*)
apply (*simp add: heyting heyting.detachment spec.seq-ctxt.cl-action-bind-action-pre-post[OF Q]*)
done

lemma *cl-action-permuteR*:

fixes $G :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $F :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$

fixes $F' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
fixes $G' :: 'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's)) \text{ set}$
assumes $G: \bigwedge v a s s'. \llbracket P s; (v, a, s, s') \in G; \text{snd } s' \neq \text{snd } s \rrbracket$
 $\implies \exists v' w a'' t s''. (v', a'', s, t) \in F' \wedge (w, a, t, s'') \in G' \wedge v' \wedge \text{snd } t = \text{snd } s \wedge \text{snd } s'' = \text{snd } s'$
assumes $GFF'G': \bigwedge v w a a' s s' t. \llbracket P s; (v, a, s, t) \in G; (w, a', t, s') \in F' v \rrbracket$
 $\implies \text{snd } s' = \text{snd } t \wedge (\exists v' a'' a''' t'. (v', a'', s, t') \in F' \wedge (w, a''', t', s') \in G' v' \wedge \text{snd } t' = \text{snd } s \wedge (\text{snd } s' \neq \text{snd } t' \longrightarrow a''' = a))$
assumes $Q: \bigwedge v w a a' s s' s''. \llbracket P s; (v, a, s, s') \in F'; (w, a', s', s'') \in G' v \rrbracket \implies Q w s''$
shows $\text{spec.action } G \ggg (\lambda v. \text{spec.action } (F' v)) \leq \llbracket P \rrbracket, A \Vdash \text{spec.seq-ctxt.cl } T (\text{spec.action } F' \ggg (\lambda v. \text{spec.action } (G' v))), \llbracket Q \rrbracket$
apply (*strengthen ord-to-strengthen*[*OF top-greatest*[**where** $a=T$]])
apply (*rule order.trans*[*OF spec.seq-ctxt.cl-action-permuteR-le*[**where** $T=True$, *simplified heyting*[*symmetric*]]])
apply (*erule* (2) G)
apply (*drule* (2) $GFF'G'$, *blast*)
apply (*simp only: refinement-def spec.pre.next-imp-eq-heyting spec.idle-le inf.bounded-iff*)
apply (*rule order.trans*[*OF - heyting.mono*[*OF order.refl spec.next-imp.mono*[*OF top-greatest order.refl*]]])
apply (*simp add: heyting heyting.detachment spec.seq-ctxt.cl-action-bind-action-pre-post*[*OF Q*])
done

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

lemma *localI*: — Introduce local state

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $c :: ('ls \times 's, 'v) \text{ prog}$
fixes $c' :: (\text{sequential}, 's, 'v) \text{ spec}$
fixes $P :: 's \text{ pred}$
fixes $Q :: 'v \Rightarrow 's \text{ pred}$
assumes $\text{prog.p2s } c \leq \llbracket \lambda s. P (\text{snd } s) \rrbracket, \text{spec.localizeA } A \Vdash \text{spec.sinvmap } \text{snd } c', \llbracket \lambda v s. Q v (\text{snd } s) \rrbracket$
shows $\text{prog.p2s } (\text{prog.local } c) \leq \llbracket P \rrbracket, A \Vdash c', \llbracket Q \rrbracket$
using *assms* **by** *transfer* (*erule refinement.spec.localI*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

15.6.1 Data refinement

In this setting a (concrete) specification c is a *data refinement* of (abstract) specification c' if:

- the observable state changes coincide
- concrete local states are mapped to abstract local states by sf which then coincide

Observations:

- pre/post are in terms of the concrete local states
 - sf can be used to lift these to the abstract local states
- we do not require c or c' to disallow the environment from changing the local state
- essentially a Skolemization of Lamport's existentials ([Lamport 1994](#), §8)

References:

- de Roever and Engelhardt (1998, Chapter 14 “Refinement Methods due to Abadi and Lamport and to Lynch”)

– in general c will need to be augmented with auxiliary variables

setup $\langle \text{Sign.mandatory-path refinement} \rangle$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

lemma data:

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$

fixes $c :: (\text{sequential}, 'cls \times 's, 'v) \text{ spec}$

fixes $c' :: (\text{sequential}, 'als \times 's, 'v) \text{ spec}$

fixes $sf :: 'cls \Rightarrow 'als$

assumes $c \leq \{\lambda s. P (\text{snd } s)\}, \text{spec.localizeA } A \Vdash \text{spec.sinvmap } (\text{map-prod } sf \text{ id}) c', \{\lambda v s. Q v (\text{snd } s)\}$

shows $\text{spec.local } c \leq \{P\}, A \Vdash \text{spec.local } c', \{Q\}$

proof –

have $*$: $\text{spec.smap snd } (\text{spec.local.interference} \sqcap \text{spec.sinvmap } (\text{map-prod } sf \text{ id}) c') \leq \text{spec.smap snd } (\text{spec.local.interference} \sqcap c') \text{ (is ?lhs} \leq \text{?rhs)}$

proof(*rule spec.singleton-le-extI*)

show $\langle \sigma \rangle \leq ?rhs$ **if** $\langle \sigma \rangle \leq ?lhs$ **for** σ

using that

by (*clarsimp simp: spec.singleton.le-conv*)

(*fastforce simp: trace.steps'.map spec.local.qrm-def*

simp flip: id-def

intro!: *exI*[**where** $x = \text{trace.map id } (\text{map-prod } sf \text{ id}) \text{ id } \sigma'$ **for** σ'])

qed

show *?thesis*

apply (*simp only: spec.local-def spec.map-invmap.galois spec.invmap.refinement id-apply*)

apply (*subst inf.left-idem[symmetric]*) — non-linear use of env constraint

apply (*strengthen ord-to-strengthen(1)[OF assms]*)

apply (*strengthen ord-to-strengthen(1)[OF refinement.inf-le]*)

apply (*subst inf commute*)

apply (*subst heyting[symmetric]*)

apply (*strengthen ord-to-strengthen(2)[OF refinement.heyting-le]*)

apply (*subst inf commute, fold spec.localizeA-def*)

apply (*rule refinement.mono[OF order.refl - - order.refl]*)

apply (*simp add: spec.localizeA-def; fail*)

apply (*simp add: heyting inf.absorb1 * flip: spec.map-invmap.galois*)

done

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

lemma data:

fixes $A :: (\text{sequential}, 's, 'v) \text{ spec}$

fixes $c :: ('cls \times 's, 'v) \text{ prog}$

fixes $c' :: ('als \times 's, 'v) \text{ prog}$

fixes $sf :: 'cls \Rightarrow 'als$

assumes $\text{prog.p2s } c \leq \{\lambda s. P (\text{snd } s)\}, \text{spec.localizeA } A \Vdash \text{spec.sinvmap } (\text{map-prod } sf \text{ id}) (\text{prog.p2s } c'), \{\lambda v s. Q v (\text{snd } s)\}$

shows $\text{prog.p2s } (\text{prog.local } c) \leq \{P\}, A \Vdash \text{prog.p2s } (\text{prog.local } c'), \{Q\}$

using *assms* **by** *transfer (erule refinement.spec.data)*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

15.7 Assume/guarantee

setup $\langle \text{Sign.mandatory-path ag} \rangle$

setup $\langle \text{Sign.mandatory-path spec} \rangle$

lemma *local*:

fixes $A G :: 's \text{ rel}$

fixes $P :: 's \text{ pred}$

fixes $Q :: 'v \Rightarrow 's \text{ pred}$

fixes $c :: (\text{sequential}, 'ls \times 's, 'v) \text{ spec}$

assumes $c \leq \{\!\{ \lambda s. P (\text{snd } s) \}\!\}, \text{Id} \times_R A \vdash \text{UNIV} \times_R G, \{\!\{ \lambda v s. Q v (\text{snd } s) \}\!\}$

shows $\text{spec.local } c \leq \{\!\{ P \}\!\}, A \vdash G, \{\!\{ Q \}\!\}$

unfolding *spec.local-def*

apply (*subst spec.map-invmap.galois*)

apply (*strengthen ord-to-strengthen(1)[OF assms]*)

apply (*subst (1) ag.reflcl-ag*)

apply (*simp only: spec.invmap.ag inv-image-alt-def map-prod-snd-snd-vimage*)

apply (*subst inf commute*)

apply (*subst heyting[symmetric]*)

apply (*subst spec.local.qrm-def*)

apply (*subst sequential.range-proc-self*)

apply (*subst Un-commute, subst ag.assm-heyting*)

apply (*auto intro: ag.mono*)

done

lemma *localize-lift*:

fixes $A G :: 's \text{ rel}$

fixes $P :: 's \Rightarrow \text{bool}$

fixes $Q :: 'v \Rightarrow 's \Rightarrow \text{bool}$

fixes $c :: (\text{sequential}, 's, 'v) \text{ spec}$

notes *inf.bounded-iff[simp del]*

assumes $c: c \leq \{\!\{ P \}\!\}, A \vdash G, \{\!\{ Q \}\!\}$

shows $\text{spec.localize UNIV } c \leq \{\!\{ \lambda s. P (\text{snd } s) \}\!\}, \text{UNIV} \times_R A \vdash \text{Id} \times_R G, \{\!\{ \lambda v s. Q v (\text{snd } s) \}\!\}$

proof(*rule ag.name-pre-state*)

fix $s :: 'ls \times 's$ **assume** $P (\text{snd } s)$

show $\text{spec.localize UNIV } c \leq \{\!\{ (=) s \}\!\}, \text{UNIV} \times_R A \vdash \text{Id} \times_R G, \{\!\{ \lambda v s. Q v (\text{snd } s) \}\!\}$

apply (*strengthen ord-to-strengthen[OF c]*)

apply (*simp add: spec.localize-def spec.invmap.ag inv-image-snd*)

apply (*simp add: ac-simps ag-def heyting*)

— discharge pre

apply (*subst (2) inf commute*)

apply (*subst (2) inf commute*)

apply (*subst inf.assoc*)

apply (*subst inf.assoc[symmetric]*)

apply (*subst heyting.curry-conv*)

apply (*subst heyting.discharge*)

apply (*simp add: $\langle P (\text{snd } s) \rangle$ predicate1I spec.pre.mono; fail*)

apply (*simp add: ac-simps*)

— discharge assume

apply (*subst inf.assoc[symmetric]*)

apply (*subst inf.assoc[symmetric]*)

apply (*subst heyting.discharge*)

apply (*force intro: le-infI2 spec.rel.mono*)

apply (*simp add: ac-simps*)

— establish post

apply (*subst inf.bounded-iff, rule conjI*)

apply (*simp add: le-infI2; fail*)

— establish guarantee

apply (*force simp: inf.bounded-iff*

simp flip: inf.assoc spec.rel.inf

intro: le-infI2 spec.rel.mono-reflcl)

done

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

lemma *local*:

fixes $A G :: 's \text{ rel}$

fixes $P :: 's \text{ pred}$

fixes $Q :: 'v \Rightarrow 's \text{ pred}$

fixes $c :: ('ls \times 's, 'v) \text{ prog}$

assumes $\text{prog.p2s } c \leq \{\lambda s. P (\text{snd } s)\}, Id \times_R A \vdash UNIV \times_R G, \{\lambda v s. Q v (\text{snd } s)\}$

shows $\text{prog.p2s } (\text{prog.local } c) \leq \{P\}, A \vdash G, \{Q\}$

using *assms* **by** *transfer* (*rule ag.spec.local*)

lemma *localize-lift*:

fixes $A G :: 's \text{ rel}$

fixes $P :: 's \Rightarrow \text{bool}$

fixes $Q :: 'v \Rightarrow 's \Rightarrow \text{bool}$

fixes $c :: ('s, 'v) \text{ prog}$

assumes $\text{prog.p2s } c \leq \{P\}, A \vdash G, \{Q\}$

shows $\text{prog.p2s } (\text{prog.localize } c) \leq \{\lambda s. P (\text{snd } s)\}, UNIV \times_R A \vdash Id \times_R G, \{\lambda v s. Q v (\text{snd } s)\}$

using *assms* **by** *transfer* (*rule ag.spec.localize-lift*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

15.8 Specification inhabitation

setup $\langle \text{Sign.mandatory-path inhabits.spec} \rangle$

lemma *localize*:

assumes $P -s, xs \rightarrow P'$

assumes $Id \subseteq r$

shows $\text{spec.localize } r P -(\text{ls}, s), \text{map } (\text{map-prod } id (\text{Pair } \text{ls})) xs \rightarrow \text{spec.localize } r P'$

by (*auto intro!: inhabits.inf inhabits.spec.rel.rel*

simp: spec.localize-def assms(1) trace-steps'-map subsetD[OF assms(2)] inhabits.spec.invmap comp-def)

lemma *local*:

assumes $P -(\text{ls}, s), xs \rightarrow \text{spec.return } v$

assumes $\text{trace.steps}' (\text{ls}, s) xs \subseteq \text{spec.local.qrm}$

shows $\text{spec.local } P -s, \text{map } (\text{map-prod } id \text{snd}) xs \rightarrow \text{spec.return } v$

unfolding *spec.local-def*

by (*rule inhabits.spec.map[where af=id and sf=snd and vf=id and s=(ls, s), simplified]*)

(*rule inhabits.inf[OF inhabits.spec.rel.rel-term[OF assms(2), where v=v] assms(1), simplified*)

lemma *local-init*:

assumes $P \text{ --}(ls, s), xs \rightarrow P'$

assumes $trace.steps' (ls, s) xs \subseteq spec.local.qrm$

shows $spec.local-init a ls P \text{ --}s, map (map\text{-}prod id snd) xs \rightarrow spec.local-init a (fst (trace.final' (ls, s) xs)) P'$

proof –

have $\langle s, map (map\text{-}prod id snd) xs, Some () \rangle \gg spec.local-init a (fst (trace.final' (ls, s) xs)) P'$

$\leq spec.local-init a ls (\langle (ls, s), xs, Some () \rangle \gg (\lambda\cdot. P'))$

proof(*induct rule: spec.bind-le*)

case *incomplete* **from** *assms(2)* **show** *?case*

by (*fastforce simp: spec.term.none.singleton spec.singleton.local-init-le-conv*
intro: spec.bind.incompleteI)

next

case (*continue* $\sigma_f \sigma_g v$)

consider (*idle*) $\langle \sigma_g \rangle \leq spec.idle$

| (*steps*) σ' **where** $\langle \sigma' \rangle \leq P'$

and $trace.steps' (trace.init \sigma') (trace.rest \sigma') \subseteq spec.local.qrm$

and $\langle \sigma_g \rangle \leq \langle trace.map id snd id \sigma' \rangle$

and $fst (trace.init \sigma') = fst (trace.final' (ls, s) xs)$

using *disjE[OF iffD1[OF spec.singleton.local-init-le-conv continue(4)]]* **by** *metis*

then show *?case*

proof *cases*

case *idle* **with** $\langle \sigma_g \neq trace.T (trace.init \sigma_g) [] None \rangle$ **show** *?thesis*

by (*simp add: spec.singleton.le-conv spec.singleton.local-init-le-conv*
trace.natural-def trace.natural'.eq-Nil-conv)

next

case (*steps* σ')

let $?\sigma' = trace.T (ls, s) (xs @ trace.rest \sigma') (trace.term \sigma')$

from *continue(1,2,3) steps(3)*

have $*$: $snd (trace.final' (ls, s) xs) = snd (trace.init \sigma')$

by (*cases* σ' ; *cases* σ_f)

(*clarsimp; metis snd-conv spec.singleton-le-conv trace.natural.sel(1)*)

trace.final'.map trace.final'.natural' trace.less-eqE trace.t.sel(1))

with *steps(4)* **have** $*$: $trace.final' (ls, s) xs = trace.init \sigma'$

by (*simp add: prod.expand*)

from *steps(1) **

have $\langle ?\sigma' \rangle \leq \langle (ls, s), xs, Some () \rangle \gg P'$

by (*simp add: spec.bind.continueI[OF order.refl]*)

moreover

from *assms(2) steps(2) **

have $trace.steps ?\sigma' \subseteq spec.local.qrm$

by (*simp add: trace.steps'.append*)

moreover

from *continue(1-3) steps **

have $\langle trace.init \sigma_f, trace.rest \sigma_f @ trace.rest \sigma_g, trace.term \sigma_g \rangle \leq \langle trace.map id snd id ?\sigma' \rangle$

by (*auto simp: trace.less-eq-None spec.singleton-le-conv trace.natural-def trace.natural'.append*

cong: trace.final'.natural'-cong

elim: trace.less-eqE)

ultimately show *?thesis*

by (*simp add: spec.singleton.local-init-le-conv exI[where $x = ?\sigma'$]*)

qed

qed

then show *?thesis*

unfolding *inhabits-def*

by (*rule order.trans[OF - spec.local-init.mono[OF assms(1)[unfolds inhabits-def]]]*)

qed

setup $\langle Sign.parent\text{-}path \rangle$

setup $\langle \text{Sign.mandatory-path prog.inhabits} \rangle$

lemma *localize*:

assumes $\text{prog.p2s } P -s, xs \rightarrow \text{prog.p2s } P'$

shows $\text{prog.p2s } (\text{prog.localize } P) - (ls, s), \text{map } (\text{map-prod id } (\text{Pair } ls)) xs \rightarrow \text{prog.p2s } (\text{prog.localize } P')$

using *assms* **by** *transfer* (*rule inhabits.spec.localize; blast*)

lemma *local*:

assumes $\text{prog.p2s } P - (ls, s), xs \rightarrow \text{spec.return } v$

assumes $\text{trace.steps}' (ls, s) xs \subseteq \text{spec.local.qrm}$

shows $\text{prog.p2s } (\text{prog.local } P) -s, \text{map } (\text{map-prod id snd}) xs \rightarrow \text{spec.return } v$

using *assms* **by** *transfer* (*rule inhabits.spec.local*)

lemma *local-init*:

assumes $\text{prog.p2s } P - (ls, s), xs \rightarrow \text{prog.p2s } P'$

assumes $\text{trace.steps}' (ls, s) xs \subseteq \text{spec.local.qrm}$

shows $\text{prog.p2s } (\text{prog.local-init } ls P) -s, \text{map } (\text{map-prod id snd}) xs \rightarrow \text{prog.p2s } (\text{prog.local-init } (\text{fst } (\text{trace.final}' (ls, s) xs)) P')$

using *assms* **by** *transfer* (*rule inhabits.spec.local-init*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

16 A Temporal Logic of Safety (TLS)

We model systems with finite and infinite sequences of states, closed under stuttering following Lamport (1994). This theory relates the safety logic of §8 to the powerset (quotiented by stuttering) representing properties of these sequences (see §16.6). Most of this story is standard but the addition of finite sequences does have some impact.

References:

- historical motivations for future-time linear temporal logic (LTL): Manna and Pnueli (1991); Owicki and Lamport (1982).
- a discussion on the merits of proving liveness: <https://cs.nyu.edu/acsys/beyond-safety/liveness.htm>

Observations:

- Lamport (and Abadi et al) treat infinite stuttering as termination
 - Lamport (2000, p189): “we can represent a terminating execution of any system by an infinite behavior that ends with a sequence of nothing but stuttering steps. We have no need of finite behaviors (finite sequences of states), so we consider only infinite ones.”
 - this conflates divergence with termination
 - we separate those concepts here so we can support sequential composition
- the traditional account of liveness properties breaks down (see §24)

16.1 Stuttering

An infinitary version of *trace.natural'*.

Observations:

- we need to normalize the agent labels for sequences that infinitely stutter

Source materials:

- \$ISABELLE_HOME/src/HOL/Corec_Examples/LFilter.thy.
- \$AFP/Coinductive/Coinductive_List.thy
- \$AFP/Coinductive/TLList.thy
- \$AFP/TLA/Sequence.thy.

definition *trailing* :: 'c \Rightarrow ('a, 'b) tllist \Rightarrow ('c, 'b) tllist **where**
trailing s xs = (if tfinite xs then TNil (terminal xs) else trepeat s)

corecursive *collapse* :: 's \Rightarrow ('a \times 's, 'v) tllist \Rightarrow ('a \times 's, 'v) tllist **where**
collapse s xs = (if snd ' tset xs \subseteq {s} then *trailing* (undefined, s) xs
else if snd (thd xs) = s then *collapse* s (ttl xs)
else TCons (thd xs) (*collapse* (snd (thd xs)) (ttl xs)))

proof –

have (LEAST i. s \neq snd (tnth (ttl xs) i)) < (LEAST i. s \neq snd (tnth xs i))
if *: \neg snd ' tset xs \subseteq {s}
and **: snd (thd xs) = s
for s **and** xs :: ('a \times 's, 'v) tllist

proof –

from * **obtain** a s' **where** (a, s') \in tset xs **and** s \neq s' **by** fastforce
then obtain i **where** snd (tnth xs i) \neq s
by (atomize-elim, induct rule: tset-induct) (auto intro: exI[of - 0] exI[of - Suc i for i])
with * ** **have** (LEAST i. s \neq snd (tnth xs i)) = Suc (LEAST i. s \neq snd (tnth xs (Suc i)))
by (cases xs) (simp-all add: Least-Suc[**where** n=i])
with * **show** (LEAST i. s \neq snd (tnth (ttl xs) i)) < (LEAST i. s \neq snd (tnth xs i))
by (cases xs) simp-all

qed

then show ?thesis

by (relation measure (λ (s, xs). LEAST i. s \neq snd (tnth xs i)); simp)

qed

setup <Sign.mandatory-path tmap>

lemma *trailing*:

shows tmap sf vf (trailing s xs) = trailing (sf s) (tmap sf vf xs)
by (simp add: trailing-def tmap-trepeat)

setup <Sign.parent-path>

setup <Sign.mandatory-path tlength>

lemma *trailing*:

shows tlength (trailing s xs) \leq tlength xs
by (fastforce simp: trailing-def dest: not-lfinite-llength)

setup <Sign.parent-path>

setup <Sign.mandatory-path trailing>

lemma *simps*[simp]:

shows TNil: trailing s (TNil b) = TNil b
and TCons: trailing s (TCons x xs) = trailing s xs
and ttl: ttl (trailing s xs) = trailing s xs
and idempotent: trailing s (trailing s xs) = trailing s xs
and tset-finite: tset (trailing s xs) = (if tfinite xs then {} else {s})

and *trepeat*: $\text{trailing } s (\text{trepeat } s) = \text{trepeat } s$
by (*simp-all add: trailing-def*)

lemma *eq-TNil-conv*:

shows $\text{trailing } s \text{ } xs = \text{TNil } b \longleftrightarrow \text{tfinite } xs \wedge \text{terminal } xs = b$

and $\text{TNil } b = \text{trailing } s \text{ } xs \longleftrightarrow \text{tfinite } xs \wedge \text{terminal } xs = b$

and $\text{is-TNil } (\text{trailing } s \text{ } xs) \longleftrightarrow \text{tfinite } xs$

by (*auto simp: trailing-def dest: is-TNil-tfinite*)

lemma *eq-TCons-conv*:

shows $\text{trailing } s \text{ } xs = \text{TCons } y \text{ } ys \longleftrightarrow \neg \text{tfinite } xs \wedge \text{TCons } y \text{ } ys = \text{trepeat } s$

and $\text{TCons } y \text{ } ys = \text{trailing } s \text{ } xs \longleftrightarrow \neg \text{tfinite } xs \wedge \text{TCons } y \text{ } ys = \text{trepeat } s$

by (*auto simp: trailing-def*)

lemma *tmap*:

shows $\text{trailing } s (\text{tmap } sf \text{ } vf \text{ } xs) = \text{tmap } id \text{ } vf (\text{trailing } s \text{ } xs)$

by (*simp add: trailing-def tmap-trepeat*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path collapse} \rangle$

lemma *unique*:

assumes $\bigwedge s \text{ } xs. f \text{ } s \text{ } xs = (\text{if } \text{snd } ' \text{tset } xs \subseteq \{s\} \text{ then } \text{trailing } (\text{undefined}, s) \text{ } xs$
 $\text{else if } \text{snd } (\text{thd } xs) = s \text{ then } f \text{ } s (\text{ttl } xs)$
 $\text{else } \text{TCons } (\text{thd } xs) (f (\text{snd } (\text{thd } xs)) (\text{ttl } xs)))$

shows $f = \text{collapse}$

proof(*intro ext*)

show $f \text{ } s \text{ } xs = \text{collapse } s \text{ } xs$ **for** $s \text{ } xs$

proof(*coinduction arbitrary: s xs*)

case (*Eq-tllist s xs*) **show** ?*case*

apply (*induct arg \equiv (s, xs) arbitrary: s xs rule: collapse.inner-induct*)

apply (*subst (1 2 3) assms*)

apply (*subst (1 2 3) collapse.code*)

apply *simp*

apply (*subst (1 2 3) assms*)

apply (*subst (1 2 3) collapse.code*)

apply *simp*

apply (*metis assms collapse.code*)

done

qed

qed

lemma *collapse*:

shows $\text{collapse } s (\text{collapse } s \text{ } xs) = \text{collapse } s \text{ } xs$

proof –

have $(\lambda s \text{ } xs. \text{collapse } s (\text{collapse } s \text{ } xs)) = \text{collapse}$

apply (*rule collapse.unique*)

apply (*subst (1 2 3) collapse.code*)

apply *auto*

done

then show ?*thesis*

by (*fastforce simp: fun-eq-iff*)

qed

lemma *simps[simp]*:

shows *TNil*: $\text{collapse } s (\text{TNil } b) = \text{TNil } b$

and *TCons*: $\text{collapse } s (\text{TCons } x \text{ } xs) = (\text{if } \text{snd } x = s \text{ then } \text{collapse } s \text{ } xs \text{ else } \text{TCons } x (\text{collapse } (\text{snd } x) \text{ } xs))$

and *trailing*: $\text{collapse } s \text{ (trailing (undefined, } s) \text{ } xs) = \text{trailing (undefined, } s) \text{ } xs$
by (*simp-all add: collapse.code trailing-def*)

lemma *tshift-stuttering*:

assumes *snd* ' $\text{set } xs \subseteq \{s\}$

shows $\text{collapse } s \text{ (tshift } xs \text{ } ys) = \text{collapse } s \text{ } ys$

using *assms* **by** (*induct xs*) *simp-all*

lemma *infinite-trailing*:

assumes $\neg \text{tfinite } xs$

assumes *snd* ' $\text{tset } xs \subseteq \{s'\}$

shows $\text{collapse } s \text{ } xs = (\text{if } s = s' \text{ then } \text{trepeat (undefined, } s') \text{ else } \text{TCons (thd } xs) \text{ (trepeat (undefined, } s')))$

using *assms* **by** (*cases xs*) (*simp-all add: assms collapse.code trailing-def*)

lemma *eq-TNil-conv*:

shows $\text{collapse } s \text{ } xs = \text{TNil } b \longleftrightarrow \text{tfinite } xs \wedge \text{snd ' tset } xs \subseteq \{s\} \wedge \text{terminal } xs = b$ (**is** *?lhs* \longleftrightarrow *?rhs*)

and $\text{TNil } b = \text{collapse } s \text{ } xs \longleftrightarrow \text{tfinite } xs \wedge \text{snd ' tset } xs \subseteq \{s\} \wedge \text{terminal } xs = b$ (**is** *?thesis1*)

proof –

show *?lhs* \longleftrightarrow *?rhs*

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

proof(*induct arg* \equiv (*s, xs*) *arbitrary: s xs rule: collapse.inner-induct[case-names step]*)

case (*step s xs*) **then show** *?case*

by (*cases xs; clarsimp split: if-splits*)

(*subst (asm) collapse.code; clarsimp simp: trailing.eq-TNil-conv split: if-splits*)

qed

show *?rhs* \implies *?lhs*

by (*simp add: conj-explode*) (*induct arbitrary: s rule: tfinite-induct; simp*)

qed

then show *?thesis1*

by (*rule eq-commute-conv*)

qed

lemma *is-TNil-conv*:

shows $\text{is-TNil (collapse } s \text{ } xs) \longleftrightarrow \text{tfinite } xs \wedge \text{snd ' tset } xs \subseteq \{s\}$ (**is** *?thesis2*)

by (*simp add: is-TNil-def collapse.eq-TNil-conv*)

lemma *eq-TConsE*:

assumes $\text{collapse } s \text{ } xs = \text{TCons } y \text{ } ys$

obtains

(*trailing-stuttering*) $\neg \text{tfinite } xs$

and *snd* ' $\text{tset } xs = \{s\}$

and $\text{TCons } y \text{ } ys = \text{trepeat (undefined, } s)$

| (*step*) *us ys'* **where** $xs = \text{tshift } us \text{ (TCons } y \text{ } ys')$

and *snd* ' $\text{set } us \subseteq \{s\}$

and $\text{snd } y \neq s$

and $\text{collapse (snd } y) \text{ } ys' = ys$

apply *atomize-elim*

using *assms*

proof(*induct arg* \equiv (*s, xs*) *arbitrary: s xs rule: collapse.inner-induct[case-names step]*)

case (*step s xs*) **show** *?case*

proof(*cases xs*)

case (*TNil v*) **with** *step.prem*s **show** *?thesis* **by** *simp*

next

case (*TCons x xs'*) **show** *?thesis*

proof(*cases snd ' tset xs' \subseteq {snd x}*)

case *True* **with** *TCons trans[OF collapse.code[symmetric] step.prem*s] **show** *?thesis*

by (*force simp: trailing.eq-TCons-conv tshift-eq-TCons-conv split: if-split-asm*)

next
case *False* **with** *TCons trans[OF collapse.code[symmetric] step.premis] step.hyps[OF refl]*
show *?thesis*
by (*cases x, cases y*)
(simp add: trailing.eq-TCons-conv tshift-eq-TCons-conv trepeat-eq-TCons-conv
eq-snd-iff exI[where x=[]]
split: if-split-asm; safe; force dest!: spec[where x=(fst x, s) # us for us])
qed
qed
qed

lemma *eq-TCons-conv*:
shows *collapse s xs = TCons y ys*
 $\longleftrightarrow (\neg \text{tfinite } xs \wedge \text{snd } ' \text{tset } xs = \{s\} \wedge \text{TCons } y \text{ } ys = \text{trepeat } (\text{undefined}, s))$
 $\vee (\exists xs' ys'. xs = \text{tshift } xs' (\text{TCons } y \text{ } ys') \wedge \text{snd } ' \text{set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{collapse } (\text{snd } y) \text{ } ys' = ys)$ (**is**
?lhs \longleftrightarrow ?rhs)
and *TCons y ys = collapse s xs*
 $\longleftrightarrow (\neg \text{tfinite } xs \wedge \text{snd } ' \text{tset } xs = \{s\} \wedge \text{TCons } y \text{ } ys = \text{trepeat } (\text{undefined}, s))$
 $\vee (\exists xs' ys'. xs = \text{tshift } xs' (\text{TCons } y \text{ } ys') \wedge \text{snd } ' \text{set } xs' \subseteq \{s\} \wedge \text{snd } y \neq s \wedge \text{collapse } (\text{snd } y) \text{ } ys' = ys)$ (**is**
?thesis1)
proof –
show *?lhs \longleftrightarrow ?rhs*
by (*auto elim: collapse.eq-TConsE simp: collapse.tshift-stuttering collapse.infinite-trailing*)
then show *?thesis1*
by (*rule eq-commute-conv*)
qed

lemma *tfinite*:
shows *tfinite (collapse s xs) \longleftrightarrow tfinite xs* (**is** *?lhs \longleftrightarrow ?rhs*)
proof(*rule iffI*)
show *?lhs if ?rhs*
using that by (*induct arbitrary: s rule: tfinite-induct*) *simp-all*
show *?rhs if ?lhs*
using that by (*induct collapse s xs arbitrary: s xs rule: tfinite-induct*)
(auto simp: collapse.eq-TNil-conv collapse.eq-TCons-conv trepeat-eq-TCons-conv)
qed

lemma *tfinite-conv*:
assumes *collapse s xs = collapse s' xs'*
shows *tfinite xs \longleftrightarrow tfinite xs'*
by (*metis assms collapse.tfinite*)

lemma *terminal*:
shows *terminal (collapse s xs) = terminal xs*
proof(*cases tfinite xs*)
case *True*
then obtain *i where tlength xs \leq enat i*
using *llength-eq-infty-conv-lfinite* **by** *fastforce*
then show *?thesis*
proof(*induct i arbitrary: s xs*)
case (*Suc i s xs*) **then show** *?case*
by (*cases xs*) (*simp-all flip: eSuc-enat*)
qed (*clarsimp simp: enat-0 tlength-0-conv*)
qed (*simp add: collapse.tfinite terminal-tinfinite*)

lemma *tlength*:
shows *tlength (collapse s xs) \leq tlength xs*
proof(*cases tfinite xs*)

case *True* **then show** *?thesis*
by (*induct arbitrary: s rule: tfinite-induct*) (*auto intro: order.trans[OF - ile-eSuc]*)
next
case *False* **then show** *?thesis*
by (*fastforce dest: not-lfinite-llength*)
qed

lemma *tset-memberD*:
assumes $(a, s') \in tset (collapse\ s\ xs)$
shows $s' \in snd\ 'tset\ xs$
using *assms*
by (*induct collapse s xs arbitrary: s xs rule: tset-induct*)
(auto simp: collapse.eq-TCons-conv trepeat-eq-TCons-conv tset-tshift image-Un)

lemma *tset-memberD2*:
assumes $(a, s') \in tset\ xs$
shows $s = s' \vee s' \in snd\ 'tset\ (collapse\ s\ xs)$
using *assms* **by** (*induct xs arbitrary: a s rule: tset-induct; simp; fast*)

lemma *tshift*:
shows $collapse\ s\ (tshift\ xs\ ys) = tshift\ (trace.natural'\ s\ xs)\ (collapse\ (trace.final'\ s\ xs)\ ys)$
by (*induct xs arbitrary: s*) *simp-all*

lemma *trepeat*:
shows $collapse\ s\ (trepeat\ (a, s)) = trepeat\ (undefined, s)$
by (*subst collapse.code*) (*simp add: trailing-def*)

lemma *eq-trepeat-conv*:
shows $trepeat\ (undefined, s) = collapse\ s\ xs \longleftrightarrow \neg tfinite\ xs \wedge snd\ 'tset\ xs = \{s\}$ (**is** *?thesis1*)
and $collapse\ s\ xs = trepeat\ (undefined, s) \longleftrightarrow \neg tfinite\ xs \wedge snd\ 'tset\ xs = \{s\}$ (**is** *?thesis2*)
proof –
show *?thesis1*
by (*rule iffI*,
(subst (asm) trepeat-unfold, simp add: collapse.eq-TCons-conv),
simp add: collapse.infinite-trailing)
then show *?thesis2*
by (*rule eq-commute-conv*)
qed

lemma *trePLICATE*:
shows $collapse\ s\ (trePLICATE\ i\ (a, s)\ v) = TNil\ v$
by (*subst collapse.code*) (*simp add: trailing.eq-TNil-conv split: nat.split*)

lemma *eq-tshift-conv*:
shows $collapse\ s\ xs = tshift\ ys\ zs$
 $\longleftrightarrow (\exists xs'\ xs'' ys'. tshift\ xs'\ xs'' = xs \wedge trace.natural'\ s\ xs' @ ys' = ys$
 $\wedge ((\neg tfinite\ xs'' \wedge snd\ 'tset\ xs'' = \{trace.final'\ s\ xs'\}) \wedge tshift\ ys'\ zs = trepeat\ (undefined, trace.final'\ s$
 $xs''))$
 $\vee (ys' = [] \wedge collapse\ (trace.final'\ s\ xs')\ xs'' = zs))$ (**is** *?lhs* \longleftrightarrow *?rhs*)
and $tshift\ ys\ zs = collapse\ s\ xs$
 $\longleftrightarrow (\exists xs'\ xs'' ys'. tshift\ xs'\ xs'' = xs \wedge trace.natural'\ s\ xs' @ ys' = ys$
 $\wedge ((\neg tfinite\ xs'' \wedge snd\ 'tset\ xs'' = \{trace.final'\ s\ xs'\}) \wedge tshift\ ys'\ zs = trepeat\ (undefined, trace.final'\ s$
 $xs''))$
 $\vee (ys' = [] \wedge collapse\ (trace.final'\ s\ xs')\ xs'' = zs))$ (**is** *?thesis1*)

proof –
show *?lhs* \longleftrightarrow *?rhs*
proof(*rule iffI*)
show *?lhs* \implies *?rhs*

```

proof(induct ys arbitrary: s xs)
  case Nil then show ?case
  by (simp add: exI[where x=[]])
next
  case (Cons y ys s xs)
  from Cons.prem[simplified] show ?case
  proof(cases rule: collapse.eq-TConsE)
    case trailing-stuttering then show ?thesis
    by (simp add: exI[where x=[]])
  next
    case (step xs' ys')
    from step(1-3) Cons.hyps[OF step(4)] show ?thesis
    by (fastforce simp: trace.natural'.append tshift-append
      simp flip: trace.natural'.eq-Nil-conv
      intro: exI[where x=xs' @ y # ys'' for ys''])
  qed
qed
show ?rhs  $\implies$  ?lhs
  by (auto simp: collapse.tshift tshift-append collapse.infinite-trailing)
qed
then show ?thesis1
  by (rule eq-commute-conv)
qed

lemma eq-collapse-ttake-dropn-conv:
  shows collapse s xs = collapse s ys
   $\longleftrightarrow$  ( $\exists j. \text{trace.natural}' s (\text{fst } (\text{ttake } i \text{ xs})) = \text{trace.natural}' s (\text{fst } (\text{ttake } j \text{ ys}))$ 
     $\wedge \text{snd } (\text{ttake } i \text{ xs}) = \text{snd } (\text{ttake } j \text{ ys})$ 
     $\wedge \text{collapse } (\text{trace.final}' s (\text{fst } (\text{ttake } i \text{ xs}))) (\text{tdropn } i \text{ xs})$ 
     $= \text{collapse } (\text{trace.final}' s (\text{fst } (\text{ttake } i \text{ xs}))) (\text{tdropn } j \text{ ys})$ ) (is ?lhs  $\longleftrightarrow$  ( $\exists j. ?rhs \ i \ j \ s \ xs \ ys$ ))

proof(rule iffI)
  show ?lhs  $\implies$  ( $\exists j. ?rhs \ i \ j \ s \ xs \ ys$ )
  proof(induct i arbitrary: s xs ys)
    case (Suc i s xs ys) show ?case
    proof(cases xs)
      case (TNil b) with Suc.prem[s] show ?thesis
      by (fastforce intro: exI[where x=case tlength ys of  $\infty \Rightarrow$  undefined | enat j  $\Rightarrow$  Suc j]
        simp: collapse.eq-TNil-conv trace.natural'.eq-Nil-conv
        ttake-eq-Some-conv tfinite-tlength-conv tdropn-tlength
        dest: in-set-ttakeD)
    next
      case (TCons x xs') show ?thesis
      proof(cases snd x = s)
        case True with Suc TCons show ?thesis by simp
      next
        case False
        note Suc.prem[s] TCons False
        moreover from calculation
        obtain us ys'
          where ys = tshift us (TCons x ys')
          and snd ' set us  $\subseteq$  {s}
          and collapse (snd x) ys' = collapse (snd x) xs'
          by (auto simp: collapse.eq-TCons-conv trepeat-eq-TCons-conv)
        moreover from calculation Suc.hyps[of snd x xs' ys']
        obtain j where ?rhs i j (snd x) xs' ys'
          by presburger
        ultimately show ?thesis
          by (auto simp: ttake-tshift trace.natural'.append tdropn-tshift)

```

```

      simp flip: trace.natural'.eq-Nil-conv
      intro: exI[where x=Suc (length us) + j])
qed
qed
qed (simp add: exI[where x=0])
show  $\exists j. ?rhs\ i\ j\ s\ xs\ ys \implies ?lhs$ 
  by (metis collapse.tshift trace.final'.natural' tshift-fst-ttake-tdropn-id)
qed

lemmas eq-collapse-ttake-dropnE = exE[OF iffD1[OF collapse.eq-collapse-ttake-dropn-conv]]

lemma tshift-tdropn:
  assumes trace.natural' s (fst (ttake i xs)) = trace.natural' s ys
  shows collapse s (tshift ys (tdropn i xs)) = collapse s xs
  by (metis assms collapse.tshift trace.final'.natural' tshift-fst-ttake-tdropn-id)

lemma map-collapse:
  shows collapse (sf s) (tmap (map-prod af sf) vf (collapse s xs))
    = collapse (sf s) (tmap (map-prod af sf) vf xs) (is ?lhs s xs = ?rhs s xs)
proof(coinduction arbitrary: s xs)
  case (Eq-tllist s xs) show ?case
  proof(intro conjI; (intro impI)?)
    have *: sf s' = sf s
      if tfinite xs and sf ' snd ' tset (collapse s xs)  $\subseteq$  {sf s} and (a, s')  $\in$  tset xs
    for a s s'
      using that by (induct arbitrary: s rule: tfinite-induct; clarsimp split: if-split-asm; metis)
  show is-TNil (?lhs s xs)  $\longleftrightarrow$  is-TNil (?rhs s xs)
    by (rule iffI,
        fastforce dest!: * simp: collapse.is-TNil-conv collapse.tfinite tllist.set-map snd-image-map-prod,
        fastforce dest!: collapse.tset-memberD simp: collapse.is-TNil-conv collapse.tfinite tllist.set-map)
  show terminal (?lhs s xs) = terminal (?rhs s xs)
    if is-TNil (?lhs s xs) and is-TNil (?rhs s xs)
    using that by (simp add: collapse.is-TNil-conv collapse.terminal)
  assume  $\neg$ is-TNil (?lhs s xs) and  $\neg$ is-TNil (?rhs s xs)
  then obtain y ys z zs where l: ?lhs s xs = TCons y ys and r: ?rhs s xs = TCons z zs
    by (simp add: tllist.disc-eq-case(2) split: tllist.split-asm)
  from l show thd (?lhs s xs) = thd (?rhs s xs)
     $\wedge$  ( $\exists s' xs'.$  ttl (?lhs s xs) = ?lhs s' xs'  $\wedge$  ttl (?rhs s xs) = ?rhs s' xs')
  proof(cases rule: collapse.eq-TConsE)
    case trailing-stuttering
    note left = this
    from r show ?thesis
    proof(cases rule: collapse.eq-TConsE)
      case trailing-stuttering
      from left(3) trailing-stuttering(3) show ?thesis
        by (fold l r) (simp; metis)
    next
    case (step us zs')
    from left(2) step(1,3) have False
      by (clarsimp simp: tset-tshift tset-tmap tmap-eq-tshift-conv TCons-eq-tmap-conv collapse.tshift
          split: if-split-asm)
      (use step(2) in  $\langle$ fastforce simp flip: trace.final'.map[where af=af] $\rangle$ )
    then show ?thesis ..
  qed
next
case (step us ys')
note left = this
from r show ?thesis

```

```

proof(cases rule: collapse.eq-TConsE)
  case trailing-stuttering
  have False
    if sf s' ≠ sf s
    and (λx. sf (snd x)) ' tset xs = {sf s}
    and (λx. sf (snd x)) ' set us ⊆ {sf s}
    and collapse s xs = tshift us (TCons (a, s') vs)
  for a s' us vs
  using that
  by (force simp: tset-tshift
      dest!: arg-cong[where f=λxs. s' ∈ snd ' tset xs] collapse.tset-memberD
      intro: imageI[where f=λx. sf (snd x)])
  with l left(3) trailing-stuttering(2) have False
    by (fastforce simp: tset-tmap tmap-eq-tshift-conv TCons-eq-tmap-conv collapse.eq-TCons-conv
        trepeat-eq-TCons-conv snd-image-map-prod image-image)
  then show ?thesis ..
next
  case (step vs zs')
  from left step show ?thesis
    unfolding l r
    apply (clarsimp simp: tmap-eq-tshift-conv collapse.tshift TCons-eq-tmap-conv
        tmap-tshift trace.natural'.map-natural'[where af=af and sf=sf and s=s]
        iffD2[OF trace.natural'.eq-Nil-conv(1)]
        dest!: arg-cong[where f=λxs. collapse (sf s) (tmap (map-prod af sf) vf xs)]
        split: if-split-asm)
    apply (use step(2) in ⟨fastforce simp flip: trace.final'.map[where af=af]⟩)
    apply (metis list.set-map trace.final'.idle trace.final'.map trace.final'.natural')
    apply metis
  done
  qed
  qed
  qed
  qed

setup ⟨Sign.parent-path⟩

setup ⟨Sign.mandatory-path behavior⟩

definition natural :: ('a, 's, 'v) behavior.t ⇒ ('a, 's, 'v) behavior.t (⟨hT⟩) where
  hTω = behavior.B (behavior.init ω) (collapse (behavior.init ω) (behavior.rest ω))

setup ⟨Sign.mandatory-path sset⟩

lemma collapse[simp]:
  shows behavior.sset (behavior.B s (collapse s xs)) = behavior.sset (behavior.B s xs)
by (auto simp: behavior.sset.simps collapse.tset-memberD dest: collapse.tset-memberD2[where s=s])

lemma natural[simp]:
  shows behavior.sset (hTω) = behavior.sset ω
by (simp add: behavior.natural-def)

lemma continue:
  shows behavior.sset (σ @-B xs) = trace.sset σ ∪ (case trace.term σ of None ⇒ snd ' tset xs | Some - ⇒ {})
by (cases σ)
  (simp add: behavior.sset.simps behavior.continue-def tshift2-def tset-tshift image-Un trace.sset.simps
  split: option.split)

setup ⟨Sign.parent-path⟩

```

setup $\langle \text{Sign.mandatory-path natural} \rangle$

lemma $\text{sel}[\text{simp}]$:

shows $\text{behavior.init } (\natural_T \omega) = \text{behavior.init } \omega$

and $\text{behavior.rest } (\natural_T \omega) = \text{collapse } (\text{behavior.init } \omega) (\text{behavior.rest } \omega)$

by $(\text{simp-all add: behavior.natural-def})$

lemma $TNil$:

shows $\natural_T(\text{behavior.B } s (\text{TNil } v)) = \text{behavior.B } s (\text{TNil } v)$

by $(\text{simp add: behavior.natural-def})$

lemma $tfinite$:

shows $tfinite (\text{behavior.rest } (\natural_T \omega)) \longleftrightarrow tfinite (\text{behavior.rest } \omega)$

by $(\text{simp add: behavior.natural-def collapse.tfinite})$

lemma $continue$:

shows $\natural_T(\sigma @_{-B} xs) = \natural_T \sigma @_{-B} (\text{collapse } (\text{trace.final } \sigma) xs)$

by $(\text{simp add: behavior.t.expand tshift2-def collapse.tshift split: option.split})$

lemma $tshift$:

shows $\natural_T(\text{behavior.B } s (\text{tshift as } xs)) = \text{behavior.B } s (\text{collapse } s (\text{tshift as } xs))$

by $(\text{simp add: behavior.natural-def})$

lemma $trepeat$:

shows $\natural_T(\text{behavior.B } s (\text{trepeat } (a, s))) = \text{behavior.B } s (\text{trepeat } (\text{undefined}, s))$

by $(\text{simp add: behavior.natural-def collapse.trepeat})$

lemma $trePLICATE$:

shows $\natural_T(\text{behavior.B } s (\text{trePLICATE } i (a, s) v)) = \text{behavior.B } s (\text{TNil } v)$

by $(\text{simp add: behavior.natural-def collapse.trePLICATE})$

lemma map-natural :

shows $\natural_T(\text{behavior.map af sf vf } (\natural_T \omega)) = \natural_T(\text{behavior.map af sf vf } \omega)$

by $(\text{simp add: behavior.natural-def collapse.map-collapse})$

lemma idle :

assumes $\text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$

shows $\natural_T \omega = \text{behavior.B } (\text{behavior.init } \omega) (\text{trailing } (\text{undefined}, \text{behavior.init } \omega) (\text{behavior.rest } \omega))$

using assms **by** $(\text{cases } \omega) (\text{simp add: behavior.natural-def behavior.sset.simps collapse.code})$

setup $\langle \text{Sign.parent-path} \rangle$

interpretation $\text{stuttering: galois.image-vimage-idempotent } \natural_T$

by $\text{standard } (\text{simp add: behavior.natural-def collapse.collapse})$

setup $\langle \text{Sign.mandatory-path stuttering} \rangle$

setup $\langle \text{Sign.mandatory-path equiv} \rangle$

abbreviation $\text{syn} :: ('a, 's, 'v) \text{ behavior.t} \Rightarrow ('a, 's, 'v) \text{ behavior.t} \Rightarrow \text{bool}$ (**infix** $\langle \simeq_T \rangle$ 50) **where**

$\omega_1 \simeq_T \omega_2 \equiv \text{behavior.stuttering.equivalent } \omega_1 \omega_2$

lemma map :

assumes $\omega_1 \simeq_T \omega_2$

shows $\text{behavior.map af sf vf } \omega_1 \simeq_T \text{behavior.map af sf vf } \omega_2$

by $(\text{metis assms behavior.natural.map-natural})$

lemma *takeE*:

assumes $\omega_1 \simeq_T \omega_2$

obtains j **where** $\text{behavior.take } i \ \omega_1 \simeq_S \text{behavior.take } j \ \omega_2$

using *assms*

by (*fastforce simp: behavior.natural-def trace.natural-def*

elim: collapse.eq-collapse-ttake-dropnE[**where** $s = \text{behavior.init } \omega_2$ **and** $i = i$ **and** $xs = \text{behavior.rest } \omega_1$

and $ys = \text{behavior.rest } \omega_2$])

lemma *idle-dropn*:

assumes $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$

assumes $\text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$

shows $\omega \simeq_T \omega'$

proof –

from $\text{behavior.sset.dropn-le}$ [*OF assms(1)*] *assms(2)*

have $\text{behavior.sset } \omega' \subseteq \{\text{behavior.init } \omega'\}$ **and** $\text{behavior.init } \omega' = \text{behavior.init } \omega$

using $\text{behavior.t.set-sel}$ (2) *subset-singletonD* **by** *fastforce+*

from *assms(1)* $\text{behavior.natural.idle}$ [*OF assms(2)*] $\text{behavior.natural.idle}$ [*OF this(1)*] *this(2)*

show *?thesis*

by (*simp add: trailing-def*)

(*metis behavior.dropn.tfiniteD behavior.dropn.eq-Some-tdropnD terminal-tdropn*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path trace.stuttering.equiv.behavior} \rangle$

lemma *takeE*:

fixes $\sigma :: ('a, 's, 'v) \text{trace.t}$

assumes $\text{behavior.take } i \ \omega \simeq_S \sigma$

obtains $\omega' j$ **where** $\omega \simeq_T \omega'$ **and** $\sigma = \text{behavior.take } j \ \omega'$

proof *atomize-elim*

have $\exists ys j. \text{collapse } s \ xs = \text{collapse } s \ ys \wedge \text{trace.T } s \ xs' (\text{snd } (\text{ttake } i \ xs)) = \text{behavior.take } j \ (\text{behavior.B } s \ ys)$

if $\text{trace.natural}' s \ (\text{fst } (\text{ttake } i \ xs)) = \text{trace.natural}' s \ xs'$

for $s \ xs'$ **and** $xs :: ('a \times 's, 'v) \text{tlist}$

using *that*

by (*cases snd (ttake i xs)*)

(*fastforce simp: behavior.take.tshift ttake-eq-Some-conv tdropn-tlength*

trace.take.all trace.take.all-iff

intro: exI[**where** $x = \text{tshift } xs' \ (\text{tdropn } i \ xs)$]

exI[**where** $x = \text{length } xs'$] *exI*[**where** $x = \text{Suc } (\text{length } xs')$]

dest: collapse.tshift-tdropn)+

with *assms* **show** $\exists \omega' j. \omega \simeq_T \omega' \wedge \sigma = \text{behavior.take } j \ \omega'$

by (*cases* σ)

(*clarsimp simp: behavior.natural-def trace.natural-def behavior.split-Ex*)

qed

lemmas $\text{rev-takeE} = \text{trace.stuttering.equiv.behavior.takeE}$ [*OF sym*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path trace.natural.behavior} \rangle$

lemma *takeE*:

fixes $\omega :: ('a, 's, 'v) \text{behavior.t}$

```

obtains j where  $\Downarrow(\text{behavior.take } i \ \omega) = \text{behavior.take } j \ (\Downarrow_T \omega)$ 
proof atomize-elim
  have  $\exists j. \text{trace.natural}' \ s \ (\text{fst} \ (\text{ttake } i \ xs)) = \text{fst} \ (\text{ttake } j \ (\text{collapse } s \ xs))$ 
     $\wedge \text{snd} \ (\text{ttake } i \ xs) = \text{snd} \ (\text{ttake } j \ (\text{collapse } s \ xs))$ 
  for s and xs :: ('a × 's, 'v) tllist
proof(induct i arbitrary: s xs)
  case 0 show ?case by (fastforce simp: ttake-eq-Nil-conv)
next
  case (Suc i s xs) show ?case
  proof(cases xs)
    case (TCons x' xs') with Suc[where s=snd x' and xs=xs'] show ?thesis
    by (fastforce intro: exI[where x=Suc j for j])
  qed (simp add: exI[where x=1])
qed
then show  $\exists j. \Downarrow(\text{behavior.take } i \ \omega) = \text{behavior.take } j \ (\Downarrow_T \omega)$ 
  by (simp add: behavior.take-def trace.natural-def split-def)
qed

```

```

setup  $\langle \text{Sign.parent-path} \rangle$ 

```

16.2 The ('a, 's, 'v) tls lattice

This is our version of Lamport's TLA lattice which we treat in a "semantic" way similarly to [Abadi and Merz \(1996\)](#).

Observations:

- there is a somewhat natural partial ordering on the *tls* lattice induced by the connection with the *spec* lattice (see §16.6 and §24) which we do not use

```

typedef ('a, 's, 'v) tls = behavior.stuttering.closed :: ('a, 's, 'v) behavior.t set set
morphisms unTLS TLS
by blast

```

```

setup-lifting type-definition-tls

```

```

instantiation tls :: (type, type, type) complete-boolean-algebra
begin

```

```

lift-definition bot-tls :: ('a, 's, 'v) tls is empty ..
lift-definition top-tls :: ('a, 's, 'v) tls is UNIV ..
lift-definition sup-tls :: ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls is sup ..
lift-definition inf-tls :: ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls is inf ..
lift-definition less-eq-tls :: ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls  $\Rightarrow$  bool is less-eq .
lift-definition less-tls :: ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls  $\Rightarrow$  bool is less .
lift-definition Inf-tls :: ('a, 's, 'v) tls set  $\Rightarrow$  ('a, 's, 'v) tls is Inf ..
lift-definition Sup-tls :: ('a, 's, 'v) tls set  $\Rightarrow$  ('a, 's, 'v) tls is  $\lambda X. \text{Sup } X \sqcup \text{behavior.stuttering.cl } \{ \}$  ..
lift-definition minus-tls :: ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls is minus ..
lift-definition uminus-tls :: ('a, 's, 'v) tls  $\Rightarrow$  ('a, 's, 'v) tls is uminus ..

```

```

instance

```

```

by (standard; transfer;
  auto simp: behavior.stuttering.cl-bot
  behavior.stuttering.closed-strict-complete-distrib-lattice-axiomI[OF behavior.stuttering.cl-bot])

```

```

end

```

```

declare

```

```

  SUPE[where 'a=( 'a, 's, 'v) tls, intro!]

```

SupE[**where** 'a=(*'a, 's, 'v*) tls, *intro!*]
Sup-le-iff[**where** 'a=(*'a, 's, 'v*) tls, *simp*]
SupI[**where** 'a=(*'a, 's, 'v*) tls, *intro*]
SUPI[**where** 'a=(*'a, 's, 'v*) tls, *intro*]
rev-SUPI[**where** 'a=(*'a, 's, 'v*) tls, *intro?*]
INFE[**where** 'a=(*'a, 's, 'v*) tls, *intro*]

setup $\langle \text{Sign.mandatory-path tls} \rangle$

lemma *boolean-implication-transfer*[*transfer-rule*]:

shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=))) (\longrightarrow_B) (\longrightarrow_B)
unfolding *boolean-implication-def* **by** *transfer-prover*

lemma *bot-not-top*:

shows $\perp \neq (\top :: ('a, 's, 'v) \text{ tls})$
by *transfer simp*

setup $\langle \text{Sign.parent-path} \rangle$

16.3 Irreducible elements

setup $\langle \text{Sign.mandatory-path raw} \rangle$

definition *singleton* :: (*'a, 's, 'v*) *behavior.t* \Rightarrow (*'a, 's, 'v*) *behavior.t* **set** **where**
singleton $\omega = \text{behavior.stuttering.cl } \{\omega\}$

lemma *singleton-le-conv*:

shows *raw.singleton* $\sigma_1 \leq \text{raw.singleton } \sigma_2 \iff \Downarrow_T \sigma_1 = \Downarrow_T \sigma_2$
by (*rule iffI*; *fastforce simp: raw.singleton-def simp flip: behavior.stuttering.cl-axiom*
dest: behavior.stuttering.clE behavior.stuttering-equiv-cl-singleton)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path tls} \rangle$

lift-definition *singleton* :: (*'a, 's, 'v*) *behavior.t* \Rightarrow (*'a, 's, 'v*) *tls* ($\langle \Downarrow \rangle_T [0]$) **is** *raw.singleton*
by (*simp add: raw.singleton-def*)

abbreviation *singleton-behavior-syn* :: 's \Rightarrow (*'a* \times 's, 'v) *tllist* \Rightarrow (*'a, 's, 'v*) *tls* ($\langle \Downarrow \rangle_T [0, 0]$) **where**
 $\langle s, xs \rangle_T \equiv \langle \text{behavior.B } s \text{ } xs \rangle_T$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma *Sup-prime*:

shows *Sup-prime* $\langle \omega \rangle_T$
by (*clarsimp simp: Sup-prime-on-def*)
(transfer; auto simp: raw.singleton-def behavior.stuttering.cl-bot
elim!: Sup-prime-onE[OF behavior.stuttering.Sup-prime-on-singleton])

lemma *nchotomy*:

shows $\exists X \in \text{behavior.stuttering.closed. } x = \bigsqcup (\text{tls.singleton } 'X)$
by *transfer*
(use behavior.stuttering.closed-conv in <auto simp add: raw.singleton-def
simp flip: behavior.stuttering.distributive>)

lemmas *exhaust* = *bexE[OF tls.singleton.nchotomy]*

lemma *collapse*[*simp*]:

shows $\sqcup (tls.singleton \text{ ' } \{\omega. \langle \omega \rangle_T \leq P\}) = P$
by (rule *tls.singleton.exhaust*[of *P*]) (simp add: *antisym SUP-le-iff SUP-upper*)

lemmas *not-bot* = *Sup-prime-not-bot*[OF *tls.singleton.Sup-prime*] — Non-triviality

setup $\langle \text{Sign.parent-path} \rangle$

lemma *singleton-le-ext-conv*:
shows $P \leq Q \iff (\forall \omega. \langle \omega \rangle_T \leq P \implies \langle \omega \rangle_T \leq Q)$ (is ?lhs \iff ?rhs)
proof(rule *iffI*)
show ?rhs \implies ?lhs
by (rule *tls.singleton.exhaust*[where $x=P$]; rule *tls.singleton.exhaust*[where $x=Q$]; blast)
qed *fastforce*

lemmas *singleton-le-conv* = *raw.singleton-le-conv*[*transferred*]
lemmas *singleton-le-extI* = *iffD2*[OF *tls.singleton-le-ext-conv*, *rule-format*]

lemma *singleton-eq-conv*[*simp*]:
shows $\langle \omega \rangle_T = \langle \omega' \rangle_T \iff \omega \simeq_T \omega'$
using *tls.singleton-le-conv* **by** (*force intro: antisym*)

lemma *singleton-cong*:
assumes $\omega \simeq_T \omega'$
shows $\langle \omega \rangle_T = \langle \omega' \rangle_T$
using *assms* **by** *simp*

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

named-theorems *le-conv* $\langle \text{simplification rules for } \langle \langle \sigma \rangle_T \leq \text{const } \dots \rangle \rangle$

lemma *boolean-implication-le-conv*[*tls.singleton.le-conv*]:
shows $\langle \sigma \rangle_T \leq P \implies_B Q \iff (\langle \sigma \rangle_T \leq P \implies \langle \sigma \rangle_T \leq Q)$
by *transfer*
(auto simp: *raw.singleton-def boolean-implication.set-alt-def elim!*: *behavior.stuttering.clE behavior.stuttering.closed-in*[OF - *sym*])

lemmas *antisym* = *antisym*[OF *tls.singleton-le-extI tls.singleton-le-extI*]

lemmas *top* = *tls.singleton.collapse*[of \top , *simplified*, *symmetric*]

lemma *simps*[*simp*]:
shows $\langle \natural_T \omega \rangle_T = \langle \omega \rangle_T$
and $\langle s, xs \rangle_T \leq \langle s, \text{collapse } s \text{ } xs \rangle_T$
and *snd* ' *set* $ys \subseteq \{s\} \implies \langle s, \text{tshift } ys \text{ } xs \rangle_T = \langle s, xs \rangle_T$
and $\langle s, \text{TCons } (a, s) \text{ } xs \rangle_T = \langle s, xs \rangle_T$
by (*simp-all* add: *antisym tls.singleton-le-conv behavior.natural-def behavior.stuttering.f-idempotent collapse.collapse collapse.tshift-stuttering*)

lemmas *Sup-irreducible* = *iffD1*[OF *heyting.Sup-prime-Sup-irreducible-iff tls.singleton.Sup-prime*]
lemmas *sup-irreducible* = *Sup-irreducible-on-imp-sup-irreducible-on*[OF *tls.singleton.Sup-irreducible*, *simplified*]
lemmas *Sup-leE*[*elim*] = *Sup-prime-onE*[OF *tls.singleton.Sup-prime*, *simplified*]
lemmas *sup-le-conv*[*simp*] = *sup-irreducible-le-conv*[OF *tls.singleton.sup-irreducible*]
lemmas *Sup-le-conv*[*simp*] = *Sup-prime-on-conv*[OF *tls.singleton.Sup-prime*, *simplified*]
lemmas *compact-point* = *Sup-prime-is-compact*[OF *tls.singleton.Sup-prime*]
lemmas *compact*[*cont-intro*] = *compact-points-are-ccpo-compact*[OF *tls.singleton.compact-point*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

16.4 The idle process

The idle process contains no transitions and does not terminate.

setup $\langle \text{Sign.mandatory-path raw} \rangle$

definition $\text{idle} :: ('a, 's, 'v) \text{ behavior.t set where}$
 $\text{idle} = (\bigcup s. \text{raw.singleton} (\text{behavior.B } s (\text{trepeat} (\text{undefined}, s))))$

lemma idle-alt-def :

shows $\text{raw.idle} = \{\omega. \neg \text{tfinite} (\text{behavior.rest } \omega) \wedge \text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}\}$ (**is** $?lhs = ?rhs$)

proof(*rule antisym[OF - subsetI]*)

show $?lhs \subseteq ?rhs$

by (*force simp: raw.idle-def raw.singleton-def behavior.split-all behavior.natural-def*
behavior.sset.simps collapse.trepeat collapse.eq-trepeat-conv
elim: behavior.stuttering.clE
dest: collapse.tfinite-conv)

show $\omega \in ?lhs$ **if** $\omega \in ?rhs$ **for** ω

using *that*

by (*cases* ω)

(*auto simp: raw.idle-def raw.singleton-def behavior.natural-def behavior.sset.simps*
behavior.stuttering.idemI collapse.infinite-trailing
elim: behavior.stuttering.clE
intro: exI[where $x = \text{behavior.init } \omega$])

qed

setup $\langle \text{Sign.mandatory-path idle} \rangle$

lemma not-tfinite :

assumes $\omega \in \text{raw.idle}$

shows $\neg \text{tfinite} (\text{behavior.rest } \omega)$

using *assms* **by** (*simp add: raw.idle-alt-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path behavior.stuttering.closed} \rangle$

lemma $\text{idle}[iff]$:

shows $\text{raw.idle} \in \text{behavior.stuttering.closed}$

by (*simp add: raw.idle-def raw.singleton-def*
behavior.stuttering.closed-UNION[simplified behavior.stuttering.cl-bot, simplified])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path tls} \rangle$

lift-definition $\text{idle} :: ('a, 's, 'v) \text{ tls is raw.idle ..}$

lemma idle-alt-def :

shows $\text{tls.idle} = (\bigcup s. \downarrow s, \text{trepeat} (\text{undefined}, s))_T$

by *transfer (simp add: raw.idle-def behavior.stuttering.cl-bot)*

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma $\text{idle-le-conv}[tls.singleton.le-conv]$:

shows $\langle \omega \rangle_T \leq tls.idle \iff \neg tfinite (behavior.rest \omega) \wedge behavior.sset \omega \subseteq \{behavior.init \omega\}$
by transfer (*simp add: raw.singleton-def behavior.stuttering.least-conv; simp add: raw.idle-alt-def*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path idle \rangle$

lemma *minimal-le:*

shows $\langle s, trepeat (undefined, s) \rangle_T \leq tls.idle$
by (*simp add: tls.singleton.idle-le-conv behavior.sset.simps*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

16.5 Temporal Logic for ('a, 's, 'v) tls

The following is a straightforward shallow embedding of the now-traditional anchored semantics of LTL [Manna and Pnueli \(1988\)](#).

References:

- [\\$AFP/TLA/Liveness.thy](#)
- [\\$ISABELLE_HOME/src/HOL/TLA/TLA.thy](#)
- https://en.wikipedia.org/wiki/Linear_temporal_logic
- [Kröger and Merz \(2008\)](#)
- [Warford, Vega, and Staley \(2020\)](#)

Observations:

- as we lack `next/X/⊙` (due to stuttering closure) we do not have induction for \mathcal{U} (until)
- [Lamport \(1994\)](#) omitted the LTL “until” operator from TLA as he considered it too hard to use
- As [De Giacomo and Vardi \(2013\)](#) observe, things get non-standard on finite traces
 - see §24 for an example
 - [Maier \(2004\)](#) provides an alternative account

setup $\langle Sign.mandatory-path raw \rangle$

definition *state-prop* :: 's pred \Rightarrow ('a, 's, 'v) behavior.t set **where**
state-prop P = $\{\omega. P (behavior.init \omega)\}$

definition

until :: ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set
where
until P Q = $\{\omega. \exists i. \exists \omega' \in Q. behavior.dropn i \omega = Some \omega' \wedge (\forall j < i. the (behavior.dropn j \omega) \in P)\}$

definition

eventually :: ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set
where
eventually P = *raw.until UNIV P*

definition

always :: ('a, 's, 'v) behavior.t set \Rightarrow ('a, 's, 'v) behavior.t set
where

$always P = \neg raw.eventually (\neg P)$

abbreviation (*input*) $unless P Q \equiv raw.until P Q \cup raw.always P$

definition *terminated* :: (*'a*, *'s*, *'v*) *behavior.t set where*

$terminated = \{\omega. tfinite (behavior.rest \omega) \wedge behavior.sset \omega \subseteq \{behavior.init \omega\}\}$

lemma *untilI*:

assumes $behavior.dropn i \omega = Some \omega'$

assumes $\omega' \in Q$

assumes $\bigwedge j. j < i \implies the (behavior.dropn j \omega) \in P$

shows $\omega \in raw.until P Q$

using *assms unfolding raw.until-def by blast*

lemma *eventually-alt-def*:

shows $raw.eventually P = \{\omega . \exists \omega' \in P. \exists i. behavior.dropn i \omega = Some \omega'\}$

by (*auto simp: raw.eventually-def raw.until-def*)

lemma *always-alt-def*:

shows $raw.always P = \{\omega . \forall i \omega'. behavior.dropn i \omega = Some \omega' \longrightarrow \omega' \in P\}$

by (*auto simp: raw.always-def raw.eventually-alt-def*)

lemma *alwaysI*:

assumes $\bigwedge i \omega'. behavior.dropn i \omega = Some \omega' \implies \omega' \in P$

shows $\omega \in raw.always P$

by (*simp add: raw.always-alt-def assms*)

lemma *alwaysD*:

assumes $\omega \in raw.always P$

assumes $behavior.dropn i \omega = Some \omega'$

shows $\omega' \in P$

using *assms by (simp add: raw.always-alt-def)*

setup $\langle Sign.mandatory-path state-prop \rangle$

lemma *monotone*:

shows *mono raw.state-prop*

by (*fastforce intro: monoI simp: raw.state-prop-def*)

lemma *simps*:

shows

$raw.state-prop \langle False \rangle = \{\}$

$raw.state-prop \perp = \{\}$

$raw.state-prop \langle True \rangle = UNIV$

$raw.state-prop \top = UNIV$

$\neg raw.state-prop P = raw.state-prop (\neg P)$

$raw.state-prop P \cup raw.state-prop Q = raw.state-prop (P \sqcup Q)$

$raw.state-prop P \cap raw.state-prop Q = raw.state-prop (P \sqcap Q)$

$(raw.state-prop P \longrightarrow_B raw.state-prop Q) = raw.state-prop (P \longrightarrow_B Q)$

by (*auto simp: raw.state-prop-def boolean-implication.set-alt-def*)

lemma *Inf*:

shows $raw.state-prop (\bigcap X) = \bigcap (raw.state-prop \text{' } X)$

by (*fastforce simp: raw.state-prop-def*)

lemma *Sup*:

shows $raw.state-prop (\bigcup X) = \bigcup (raw.state-prop \text{' } X)$

by (*fastforce simp: raw.state-prop-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path terminated} \rangle$

lemma *inf-always-le*:

fixes $P :: ('a, 's, 'v) \text{ behavior.t set}$

assumes $P \in \text{behavior.stuttering.closed}$

shows $\text{raw.terminated} \cap P \subseteq \text{raw.always } P$

by (*rule subsetI*[*OF raw.alwaysI*])

(*auto simp: raw.terminated-def*

elim: behavior.stuttering.closed-in[*OF - - assms*] *behavior.stuttering.equiv.idle-dropn*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path until} \rangle$

lemma *base*:

shows $\omega \in Q \implies \omega \in \text{raw.until } P Q$

and $Q \subseteq \text{raw.until } P Q$

by (*force simp: raw.until-def*)⁺

lemma *step*:

assumes $\omega \in P$

assumes $\text{behavior.tl } \omega = \text{Some } \omega'$

assumes $\omega' \in \text{raw.until } P Q$

shows $\omega \in \text{raw.until } P Q$

proof –

from $\langle \omega' \in \text{raw.until } P Q \rangle$

obtain $i \omega''$

where $\omega'' \in Q$ **and** $\forall j < i. \text{the } (\text{behavior.dropn } j \omega') \in P$ **and** $\text{behavior.dropn } i \omega' = \text{Some } \omega''$

by (*clarsimp simp: raw.until-def*)

with *assms*(1,2) **show** *?thesis*

by (*clarsimp simp: raw.until-def behavior.dropn.Suc less-Suc-eq-0-disj*

intro!:: exI[**where** $x = \text{Suc } i$])

qed

lemmas *intro*[*intro*] =

raw.until.base

raw.until.step

lemma *induct*[*case-names base step, consumes 1, induct set: raw.until*]:

assumes $\omega \in \text{raw.until } P Q$

assumes *base*: $\bigwedge \omega. \omega \in Q \implies R \omega$

assumes *step*: $\bigwedge \omega \omega'. [\omega \in P; \text{behavior.tl } \omega = \text{Some } \omega'; \omega' \in \text{raw.until } P Q; R \omega'] \implies R \omega$

shows $R \omega$

proof –

from $\langle \omega \in \text{raw.until } P Q \rangle$ **obtain** $\omega' i$

where $\text{behavior.dropn } i \omega = \text{Some } \omega'$ **and** $\omega' \in Q$ **and** $\forall j < i. \text{the } (\text{behavior.dropn } j \omega) \in P$

unfolding *raw.until-def* **by** *blast*

then show *?thesis*

proof(*induct i arbitrary: \omega*)

case 0 **then show** *?case*

by (*force intro: base*)

next

case *Suc* **from** *Suc.prem*s **show** *?case*

by (*fastforce intro: step Suc.hyps dest: spec*[**where** $x = \text{Suc } j$ **for** j]

simp: behavior.dropn.Suc raw.until-def)

split: Option.bind-split-asm)

qed
qed

lemma *mono*:

assumes $P \subseteq P'$

assumes $Q \subseteq Q'$

shows $\text{raw.until } P \ Q \subseteq \text{raw.until } P' \ Q'$

unfolding *raw.until-def* using *assms* by *blast*

lemma *botL*:

shows $\text{raw.until } \{\} \ Q = Q$

by (*force simp: raw.until-def*)

lemma *botR*:

shows $\text{raw.until } P \ \{\} = \{\}$

by (*force simp: raw.until-def*)

lemma *untilR*:

shows $\text{raw.until } P \ (\text{raw.until } P \ Q) = \text{raw.until } P \ Q$ (*is ?lhs = ?rhs*)

proof(*rule antisym[OF subsetI]*)

show $\omega \in ?rhs$ if $\omega \in ?lhs$ for ω using *that* by *induct blast+*

show $?rhs \subseteq ?lhs$ by *blast*

qed

lemma *InfL-not-empty*:

assumes $X \neq \{\}$

shows $\text{raw.until } (\bigcap X) \ Q = (\bigcap_{x \in X} \text{raw.until } x \ Q)$ (*is ?lhs = ?rhs*)

proof(*rule antisym[OF - subsetI]*)

show $?lhs \subseteq ?rhs$

by (*simp add: INT-greatest Inter-lower raw.until.mono*)

show $\omega \in ?lhs$ if $\omega \in ?rhs$ for ω

proof –

from $\langle X \neq \{\} \rangle$ obtain P where $P \in X$ by *blast*

with *that* obtain $i \ \omega'$

where $*$: $\text{behavior.dropn } i \ \omega = \text{Some } \omega' \ \omega' \in Q \ \forall j < i$. *the* ($\text{behavior.dropn } j \ \omega \in P$)

unfolding *raw.until-def* by *blast*

from *this*(1,2) obtain $k \ \omega''$

where $**$: $k \leq i$ $\text{behavior.dropn } k \ \omega = \text{Some } \omega'' \ \omega'' \in Q \ \forall j < k$. *the* ($\text{behavior.dropn } j \ \omega \notin Q$)

using *ex-has-least-nat*[**where** $k=i$ **and** $P=\lambda k. k \leq i \wedge (\forall \omega''. \text{behavior.dropn } k \ \omega = \text{Some } \omega'' \longrightarrow \omega'' \in Q)$

and $m=id$]

by *clarsimp* (*metis* (*no-types*, *lifting*) *behavior.dropn.shorterD leD nle-le option.sel order.trans*)

from *that* $**$ show *?thesis*

by (*clarsimp simp: raw.until-def intro!: exI*[**where** $x=k$])

(*metis* *order.strict-trans1 linorder-not-le option.sel*)

qed

qed

lemma *SupR*:

shows $\text{raw.until } P \ (\bigcup X) = \bigcup (\text{raw.until } P \ ` X)$

unfolding *raw.until-def* by *blast*

lemma *weakenL*:

shows $\text{raw.until } UNIV \ P = \text{raw.until } (- \ P) \ P$ (*is ?lhs = ?rhs*)

proof(*rule antisym[OF subsetI]*)

show $\omega \in ?rhs$ if $\omega \in ?lhs$ for ω using *that* by *induct blast+*

show $?rhs \subseteq ?lhs$ by (*simp add: raw.until.mono*)

qed

lemma *implication-ordering-le*: — Warford et al. (2020, (16))
shows $\text{raw.until } P \ Q \cap \text{raw.until } (-Q) \ R \subseteq \text{raw.until } P \ R$
by (*clarsimp simp: raw.until-def*) (*metis order.trans linorder-not-le option.sel*)

lemma *infR-ordering-le*: — Warford et al. (2020, (18))
shows $\text{raw.until } P \ (Q \cap R) \subseteq \text{raw.until } (\text{raw.until } P \ Q) \ R$ (**is** $?lhs \subseteq ?rhs$)
proof(*rule subsetI*)
show $\omega \in ?rhs$ **if** $\omega \in ?lhs$ **for** ω
using *that*
proof *induct*
case (*step* $\omega \ \omega'$) **then show** $?case$
by — (*rule raw.until.step, rule raw.until.step;*
blast intro: subsetD[OF raw.until.mono, rotated -1])
qed *blast*
qed

lemma *untilL*:
shows $\text{raw.until } (\text{raw.until } P \ Q) \ Q \subseteq \text{raw.until } P \ Q$ (**is** $?lhs \subseteq ?rhs$)
proof(*rule subsetI*)
show $\omega \in ?rhs$ **if** $\omega \in ?lhs$ **for** ω
using *that* **by** *induct auto*
qed

lemma *alwaysR-le*:
shows $\text{raw.until } P \ (\text{raw.always } Q) \subseteq \text{raw.always } (\text{raw.until } P \ Q)$ (**is** $?lhs \subseteq ?rhs$)
proof(*rule subsetI*)
show $\omega \in ?rhs$ **if** $\omega \in ?lhs$ **for** ω
using *that*
proof *induct*
case (*base* ω) **then show** $?case$ **by** (*auto simp: raw.always-alt-def*)
next
case (*step* $\omega \ \omega'$) **show** $?case$
proof(*rule raw.alwaysI*)
fix $i \ \omega''$ **assume** $\text{behavior.dropn } i \ \omega = \text{Some } \omega''$
with *step behavior.dropn.0* **show** $\omega'' \in \text{raw.until } P \ Q$
by (*cases i; clarsimp simp: raw.always-alt-def behavior.dropn.Suc; blast*)
qed
qed
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path unless} \rangle$

lemma *neg*:
shows — $(\text{raw.until } P \ Q \cup \text{raw.always } P) = \text{raw.until } (-Q) \ (-P \cap -Q)$ (**is** $?lhs = ?rhs$)
proof(*rule antisym[OF subsetI], (unfold Compl-Un Int-iff conj-explode Compl-iff)[1]*)
fix ω
assume $*$: $\omega \notin \text{raw.until } P \ Q$
assume $\omega \notin \text{raw.always } P$
then obtain $k \ \omega'$
where $\text{behavior.dropn } k \ \omega = \text{Some } \omega'$
and $\omega' \notin P$
by (*clarsimp simp: raw.always-alt-def*)
with *ex-has-least-nat*[**where** $k=k$ **and** $P=\lambda i. \exists \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \wedge \omega' \notin P$ **and** $m=id$]
obtain $k \ \omega'$
where $\text{behavior.dropn } k \ \omega = \text{Some } \omega'$

and $\omega' \notin P$
and $\forall j < k. \text{the } (\text{behavior.dropn } j \ \omega) \in P$
by *clarsimp* (*metis* *behavior.dropn.shorterD less-le-not-le option.distinct(1) option.exhaust-sel*)
with $* \text{behavior.dropn.shorterD}$ **show** $\omega \in ?rhs$
by (*fastforce simp: raw.until-def intro: exI[where x=k]*)
next
show $?rhs \subseteq ?lhs$
by (*clarsimp simp: raw.always-alt-def raw.until-def subset-iff; metis nat-neq-iff option.sel*)
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path eventually} \rangle$

lemma *terminated*:

shows $\text{raw.eventually } \text{raw.terminated} = \{\omega. \text{tfinite } (\text{behavior.rest } \omega)\}$ (**is** $?lhs = ?rhs$)
proof(*rule antisym[OF - subsetI]*)
show $?lhs \subseteq ?rhs$
by (*clarsimp simp: raw.eventually-alt-def raw.terminated-def behavior.dropn.tfiniteD*)
show $\omega \in ?lhs$ **if** $\omega \in ?rhs$ **for** ω
proof –
note $\langle \omega \in ?rhs \rangle$
moreover from *calculation*
obtain i **where** $\text{tlength } (\text{behavior.rest } \omega) = \text{enat } i$
by (*clarsimp simp: tfinite-tlength-conv*)
moreover from *calculation*
obtain ω' **where** $\text{behavior.dropn } i \ \omega = \text{Some } \omega'$
using *behavior.dropn.eq-Some-tlength-conv* **by** *fastforce*
moreover from *calculation*
have $\text{behavior.sset } \omega' \subseteq \{\text{behavior.init } \omega'\}$
by (*cases* ω')
(*clarsimp dest!: behavior.dropn.eq-Some-tdropnD simp: tdropn-tlength behavior.sset.simps*)
ultimately show $\omega \in ?lhs$
by (*auto simp: raw.eventually-alt-def raw.terminated-def dest: behavior.dropn.tfiniteD*)
qed
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path behavior.stuttering.closed.raw} \rangle$

lemma *state-prop[intro]*:

shows $\text{raw.state-prop } P \in \text{behavior.stuttering.closed}$
by (*fastforce simp: raw.state-prop-def behavior.natural-def elim: behavior.stuttering.clE*)

lemma *terminated[intro]*:

shows $\text{raw.terminated} \in \text{behavior.stuttering.closed}$
by (*rule behavior.stuttering.closedI*)
(*clarsimp simp: raw.terminated-def elim!: behavior.stuttering.clE;*
metis behavior.natural.sel(1) behavior.natural.tfinite behavior.sset.natural)

lemma *until[intro]*:

assumes $P \in \text{behavior.stuttering.closed}$
assumes $Q \in \text{behavior.stuttering.closed}$
shows $\text{raw.until } P \ Q \in \text{behavior.stuttering.closed}$
proof –

```

have  $\omega_2 \in \text{raw.until } P \ Q$  if  $\omega_1 \in \text{raw.until } P \ Q$  and  $\omega_1 \simeq_T \omega_2$  for  $\omega_1 \ \omega_2$ 
using that
proof(induct arbitrary:  $\omega_2$  rule: raw.until.induct)
  case (base  $\omega_1 \ \omega_2$ ) with assms(2) show ?case
    by (blast intro: behavior.stuttering.closed-in)
next
case (step  $\omega_1 \ \omega' \ \omega_2$ )
show ?case
proof(cases  $\omega' \simeq_T \omega_1$ )
  case True with  $\langle \omega_1 \simeq_T \omega_2 \rangle$  step.hyps(4) show ?thesis
    by simp
next
case False
from assms(1)  $\langle \omega_1 \in P \rangle \langle \omega_1 \simeq_T \omega_2 \rangle$  have  $\omega_2 \in P$ 
  by (blast intro: behavior.stuttering.closed-in)
from False  $\langle \omega_1 \simeq_T \omega_2 \rangle \langle \text{behavior.tl } \omega_1 = \text{Some } \omega' \rangle$ 
obtain  $a \ s_0 \ s_1 \ xs_1 \ xs' \ ys'$ 
  where  $\omega_1: \omega_1 = \text{behavior.B } s_0 \ (\text{TCons } (a, s_1) \ xs_1)$ 
    and  $\omega_2: \omega_2 = \text{behavior.B } s_0 \ (\text{tshift } xs' \ (\text{TCons } (a, s_1) \ ys'))$ 
    and  $*$ :  $\text{collapse } s_0 \ (\text{TCons } (a, s_1) \ xs_1) = \text{collapse } s_0 \ (\text{tshift } xs' \ (\text{TCons } (a, s_1) \ ys'))$ 
       $s_0 \neq s_1$ 
    and  $**$ :  $\text{collapse } s_1 \ ys' = \text{collapse } s_1 \ xs_1$ 
    and  $xs'$ :  $\text{snd } ' \ \text{set } xs' \subseteq \{s_0\}$ 
  by (cases  $\omega_1$ ; cases  $\omega_2$ ; cases behavior.rest  $\omega_1$ ; simp)
    (fastforce simp: behavior.natural-def collapse.eq-TCons-conv trepeat-eq-TCons-conv
      split: if-splits)
from  $\omega_2 \ \langle \omega_2 \in P \rangle \ xs'$  show ?thesis
proof(induct  $xs'$  arbitrary:  $\omega_2$ )
  case Nil with  $\omega_1 \ ** \ \text{step.hyps}(2,4)$  show ?case
    by (auto simp: behavior.natural-def)
next
case (Cons  $x' \ xs'$ )
  with behavior.stuttering.closed-in[OF - -  $\langle P \in \text{behavior.stuttering.closed} \rangle$ ]  $\omega_1 \ ** \ \text{step}(3)$ 
  show ?case
    by (auto simp: behavior.natural-def behavior.split-all)
qed
qed
qed
then show ?thesis
  by (fastforce elim: behavior.stuttering.clE)
qed

```

```

lemma eventually[intro]:
  assumes  $P \in \text{behavior.stuttering.closed}$ 
  shows raw.eventually  $P \in \text{behavior.stuttering.closed}$ 
using assms by (auto simp: raw.eventually-def)

```

```

lemma always[intro]:
  assumes  $P \in \text{behavior.stuttering.closed}$ 
  shows raw.always  $P \in \text{behavior.stuttering.closed}$ 
using assms by (auto simp: raw.always-def)

```

```

setup  $\langle \text{Sign.parent-path} \rangle$ 

```

```

setup  $\langle \text{Sign.mandatory-path } tls \rangle$ 

```

```

definition valid ::  $(\prime a, \prime s, \prime v) \ tls \Rightarrow \text{bool}$  where
  valid  $P \longleftrightarrow P = \top$ 

```

lift-definition *state-prop* :: $'s \text{ pred} \Rightarrow ('a, 's, 'v) \text{ tls}$ **is** *raw.state-prop* ..

lift-definition *terminated* :: $('a, 's, 'v) \text{ tls}$ **is** *raw.terminated* ..

lift-definition *until* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **is** *raw.until* ..

definition *eventually* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

eventually $P = \text{tls.until } \top P$

definition *always* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

always $P = \neg \text{tls.eventually } (\neg P)$

definition *release* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

release $P Q = \neg(\text{tls.until } (\neg P) (\neg Q))$

definition *unless* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

unless $P Q = \text{tls.until } P Q \sqcup \text{tls.always } P$

abbreviation (*input*) *always-imp-syn* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

always-imp-syn $P Q \equiv \text{tls.always } (P \longrightarrow_B Q)$

abbreviation (*input*) *leads-to* :: $('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls} \Rightarrow ('a, 's, 'v) \text{ tls}$ **where**

leads-to $P Q \equiv \text{tls.always-imp-syn } P (\text{tls.eventually } Q)$

open-bundle *syntax*

begin

notation *tls.valid* ($\langle \models \rightarrow [30] 30 \rangle$)

notation *tls.state-prop* ($\langle \langle - \rangle \rangle [0] \rangle$)

notation *tls.until* (**infix** $\langle \mathcal{U} \rangle 85$)

notation *tls.eventually* ($\langle \diamond \rightarrow [87] 87 \rangle$)

notation *tls.always* ($\langle \square \rightarrow [87] 87 \rangle$)

notation *tls.release* (**infixr** $\langle \mathcal{R} \rangle 85$)

notation *tls.unless* (**infixr** $\langle \mathcal{W} \rangle 85$)

notation *tls.always-imp-syn* (**infixr** $\langle \longrightarrow_{\square} \rangle 75$)

notation *tls.leads-to* (**infixr** $\langle \rightsquigarrow \rangle 75$)

end

bundle *no-syntax*

begin

no-notation *tls.valid* ($\langle \models \rightarrow [30] 30 \rangle$)

no-notation *tls.state-prop* ($\langle \langle - \rangle \rangle [0] \rangle$)

no-notation *tls.until* (**infixr** $\langle \mathcal{U} \rangle 85$)

no-notation *tls.eventually* ($\langle \diamond \rightarrow [87] 87 \rangle$)

no-notation *tls.always* ($\langle \square \rightarrow [87] 87 \rangle$)

no-notation *tls.release* (**infixr** $\langle \mathcal{R} \rangle 85$)

no-notation *tls.unless* (**infixr** $\langle \mathcal{W} \rangle 85$)

no-notation *tls.always-imp-syn* (**infixr** $\langle \longrightarrow_{\square} \rangle 75$)

no-notation *tls.leads-to* (**infixr** $\langle \rightsquigarrow \rangle 75$)

end

lemma *validI*:

assumes $\top \leq P$

shows $\models P$

by (*simp add: assms tls.valid-def top.extremum-uniqueI*)

setup $\langle \text{Sign.mandatory-path valid} \rangle$

lemma *trans[trans]*:

assumes $\models P$

assumes $P \leq Q$
shows $\models Q$
using *assms* **by** (*simp add: tls.valid-def top.extremum-unique*)

lemma *mp*:
assumes $\models P \longrightarrow_B Q$
assumes $\models P$
shows $\models Q$
using *assms* **by** (*simp add: tls.valid-def*)

lemmas *rev-mp* = *tls.valid.mp[rotated]*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma *uminus-le-conv*[*tls.singleton.le-conv*]:
shows $\langle \omega \rangle_T \leq -P \longleftrightarrow \neg \langle \omega \rangle_T \leq P$
by *transfer*
(simp add: raw.singleton-def behavior.stuttering.closed-uminus behavior.stuttering.least-conv)

lemma *state-prop-le-conv*[*tls.singleton.le-conv*]:
shows $\langle \omega \rangle_T \leq \text{tls.state-prop } P \longleftrightarrow P$ (*behavior.init* ω)
by *transfer*
(simp add: raw.singleton-def behavior.stuttering.least-conv[OF behavior.stuttering.closed.raw.state-prop]; simp add: raw.state-prop-def)

lemma *terminated-le-conv*[*tls.singleton.le-conv*]:
shows $\langle \omega \rangle_T \leq \text{tls.terminated} \longleftrightarrow \text{tfinite}(\text{behavior.rest } \omega) \wedge \text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$
by *transfer*
(simp add: raw.singleton-def behavior.stuttering.least-conv[OF behavior.stuttering.closed.raw.terminated]; simp add: raw.terminated-def)

lemma *until-le-conv*[*tls.singleton.le-conv*]:
fixes $P :: ('a, 's, 'v) \text{tls}$
shows $\langle \omega \rangle_T \leq P \mathcal{U} Q \longleftrightarrow (\exists i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega' \wedge \langle \omega' \rangle_T \leq Q \wedge (\forall j < i. \langle \text{the } (\text{behavior.dropn } j \omega) \rangle_T \leq P))$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

proof *transfer*

fix ω **and** $P Q :: ('a, 's, 'v) \text{behavior.t set}$

assume $*$: $P \in \text{behavior.stuttering.closed}$ $Q \in \text{behavior.stuttering.closed}$

and $\text{raw.singleton } \omega \subseteq \text{raw.until } P Q$

then have $\exists i. \exists \omega' \in Q. \text{behavior.dropn } i \omega = \text{Some } \omega' \wedge (\forall j < i. \text{the } (\text{behavior.dropn } j \omega) \in P)$

by (*auto simp: raw.singleton-def raw.until-def*)

with $*$ **show** $\exists i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega'$

$\wedge \text{raw.singleton } \omega' \subseteq Q \wedge (\forall j < i. \text{raw.singleton } (\text{the } (\text{behavior.dropn } j \omega)) \subseteq P)$

by (*auto simp: raw.singleton-def behavior.stuttering.least-conv*)

qed

show *?rhs* \implies *?lhs*

by *transfer*

(*unfold raw.singleton-def;*

rule behavior.stuttering.least[OF - behavior.stuttering.closed.raw.until];

auto 10 0 intro: iffD2[OF eqset-imp-iff[OF raw.until-def]])

qed

lemma *eventually-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \diamond P \iff (\exists i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega' \wedge \langle \omega' \rangle_T \leq P)$
by (*simp add: tls.eventually-def tls.singleton.le-conv*)

lemma *always-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.always } P \iff (\forall i \omega'. \text{behavior.dropn } i \omega = \text{Some } \omega' \implies \langle \omega' \rangle_T \leq P)$
by (*simp add: tls.always-def tls.singleton.le-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *until: closure-complete-lattice-distributive-class* *tls.until* *P* **for** *P*

proof *standard*

show $(x \leq \text{tls.until } P y) = (\text{tls.until } P x \leq \text{tls.until } P y)$ **for** $x y$

by *transfer*

(*intro iffD2[OF order-class.order.closure-axioms-alt-def[unfolded closure-axioms-def], rule-format]*
conjI allI raw.until.base monoI raw.until.mono order.refl raw.until.untilR, assumption)

show $\text{tls.until } P (\bigsqcup X) \leq \bigsqcup (\text{tls.until } P \text{ ` } X) \sqcup \text{tls.until } P \perp$ **for** X

by *transfer (simp add: raw.until.SupR behavior.stuttering.cl-bot)*

qed

setup $\langle \text{Sign.mandatory-path until} \rangle$

lemmas *botL* = *raw.until.botL*[*transferred*]

lemmas *botR* = *raw.until.botR*[*transferred*]

lemmas *topR* = *tls.until.cl-top*

lemmas *expansiveR* = *tls.until.expansive*[*of P Q for P Q*]

lemmas *weakenL* = *raw.until.weakenL*[*transferred*]

lemmas *mono* = *raw.until.mono*[*transferred*]

lemma *strengthen*[*strg*]:

assumes *st-ord F P P'*

assumes *st-ord F Q Q'*

shows *st-ord F (P U Q) (P' U Q')*

using *assms* **by** (*cases F*) (*auto simp: tls.until.mono*)

lemma *SupL-le*:

shows $(\bigsqcup x \in X. x \mathcal{U} R) \leq (\bigsqcup X) \mathcal{U} R$

by (*simp add: SupI tls.until.mono*)

lemma *supL-le*:

shows $P \mathcal{U} R \sqcup Q \mathcal{U} R \leq (P \sqcup Q) \mathcal{U} R$

by (*simp add: tls.until.mono*)

lemma *SupR*:

shows $P \mathcal{U} (\bigsqcup X) = \bigsqcup ((\mathcal{U}) P \text{ ` } X)$

by (*simp add: tls.until.cl-Sup tls.until.botR*)

lemmas *supR* = *tls.until.cl-sup*

lemmas *InfL-not-empty* = *raw.until.InfL-not-empty*[*transferred*]

lemmas *infL* = *tls.until.InfL-not-empty*[**where** $X = \{P, Q\}$ **for** $P Q$, *simplified, of P Q R for P Q R*]

lemmas *InfR-le* = *tls.until.cl-Inf-le*

lemmas *infR-le* = *tls.until.cl-inf-le*[*of P Q R for P Q R*]

lemma *implication-ordering-le*: — Warford et al. (2020, (16))

shows $P \mathcal{U} Q \sqcap (\neg Q) \mathcal{U} R \leq P \mathcal{U} R$

by *transfer (rule raw.until.implication-ordering-le)*

lemma *supL-ordering-le*: — Warford et al. (2020, (17))
shows $P \mathcal{U} (Q \mathcal{U} R) \leq (P \sqcup Q) \mathcal{U} R$ (**is** $?lhs \leq ?rhs$)
proof –
have $?rhs = (P \sqcup Q) \mathcal{U} ((P \sqcup Q) \mathcal{U} R)$ **by** (*rule* $tls.until.idempotent(1)[symmetric]$)
also have $?lhs \leq \dots$ **by** (*blast intro*: $tls.until.mono$ $le-supI1$ $le-supI2$)
finally show $?thesis$.
qed

lemma *infR-ordering-le*: — Warford et al. (2020, (18))
shows $P \mathcal{U} (Q \sqcap R) \leq (P \mathcal{U} Q) \mathcal{U} R$
by *transfer* (*rule* $raw.until.infR-ordering-le$)

lemma *boolean-implication-distrib-le*: — Warford et al. (2020, (19))
shows $(P \longrightarrow_B Q) \mathcal{U} R \leq (P \mathcal{U} R) \longrightarrow_B (Q \mathcal{U} R)$
by (*metis* $galois.conj-imp.galois$ $order.refl$ $tls.until.infL$ $tls.until.mono$)

lemma *excluded-middleR*: — Warford et al. (2020, (23))
shows $\models P \mathcal{U} Q \sqcup P \mathcal{U} (\neg Q)$
by (*simp add*: $tls.validI$ $tls.until.cl-top$ *flip*: $tls.until.cl-sup$)

lemmas $untilR = tls.until.idempotent(1)[of\ P\ Q\ for\ P\ Q]$

lemma *untilL*:
shows $(P \mathcal{U} Q) \mathcal{U} Q = P \mathcal{U} Q$ (**is** $?lhs = ?rhs$)
proof(*rule antisym*)
show $?lhs \leq ?rhs$
by *transfer* (*rule* $raw.until.untilL$)
show $?rhs \leq ?lhs$
using $tls.until.infR-ordering-le$ [**where** $P=P$ **and** $Q=Q$ **and** $R=Q$] **by** *simp*
qed

lemma *absorb*:
shows $P \mathcal{U} P = P$
by (*metis* $tls.until.botL$ $tls.until.untilL$)

lemma *absorb-supL*: — Warford et al. (2020, (23))
shows $P \sqcup P \mathcal{U} Q = P \sqcup Q$
by (*metis* $inf-commute$ $inf-sup-absorb$ $le-iff-sup$
 $tls.until.absorb$ $tls.until.cl-sup$ $tls.until.expansive$ $tls.until.infL$)

lemma *absorb-supR*: — Warford et al. (2020, (23))
shows $Q \sqcup P \mathcal{U} Q = P \mathcal{U} Q$
by (*simp add*: $sup.absorb2$ $tls.until.expansive$)

lemma *eventually-le*:
shows $P \mathcal{U} Q \leq \diamond Q$
by (*simp add*: $tls.eventually-def$ $tls.until.mono$)

lemma *absorb-eventually*:
shows *inf-eventually-absorbR*: $P \mathcal{U} Q \sqcap \diamond Q = P \mathcal{U} Q$ — Warford et al. (2020, (39))
and *sup-eventually-absorbR*: $P \mathcal{U} Q \sqcup \diamond Q = \diamond Q$ — Warford et al. (2020, (40))
and *eventually-absorbR*: $P \mathcal{U} \diamond Q = \diamond Q$ — Warford et al. (2020, (41))
by (*simp-all add*: $tls.eventually-def$ $sup.absorb2$ $tls.until.mono$
 $order.eq-iff$ $order.trans[OF\ tls.until.supL-ordering-le]$ $tls.until.expansiveR$
 $flip: tls.until.infL$)

lemma *sup-le*: — Warford et al. (2020, (28))

shows $P \mathcal{U} Q \leq P \sqcup Q$
by (*simp add: ac-simps sup.absorb-iff1 tls.until.absorb-supL tls.until.absorb-supR*)

lemma *ordering*: — Warford et al. (2020, (251))
shows $(-P) \mathcal{U} Q \sqcup (-Q) \mathcal{U} P = \diamond(P \sqcup Q)$ (**is** *?lhs = ?rhs*)

proof —

have $?lhs = \top \mathcal{U} P \sqcap (-Q) \mathcal{U} P \sqcup \top \mathcal{U} Q \sqcap (-P) \mathcal{U} Q$
by (*simp add: ac-simps inf.absorb2 tls.until.mono*)
also have $\dots = (-P) \mathcal{U} P \sqcap (-Q) \mathcal{U} P \sqcup (-Q) \mathcal{U} Q \sqcap (-P) \mathcal{U} Q$
by (*simp add: tls.until.weakenL*)
also have $\dots = (- (P \sqcup Q)) \mathcal{U} (P \sqcup Q)$
by (*simp add: ac-simps tls.until.cl-sup flip: tls.until.infL*)
also have $\dots = ?rhs$
by (*simp add: tls.eventually-def tls.until.weakenL*)
finally show *?thesis* .

qed

lemmas *simps* =
tls.until.expansiveR
tls.until.botL
tls.until.botR
tls.until.absorb
tls.until.absorb-supL
tls.until.absorb-supR
tls.until.untilL
tls.until.untilR

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *eventually*: *closure-complete-lattice-distributive-class tls.eventually*
unfolding *tls.eventually-def*
by (*simp add: tls.until.closure-complete-lattice-distributive-class-axioms*)

lemma *eventually-alt-def*:
shows $\diamond P = (-P) \mathcal{U} P$
by (*simp add: tls.eventually-def tls.until.weakenL*)

setup $\langle \text{Sign.mandatory-path eventually} \rangle$

lemma *transfer[transfer-rule]*:
shows *rel-fun (pcr-tls (=) (=) (=)) (pcr-tls (=) (=) (=)) raw.eventually tls.eventually*
unfolding *tls.eventually-def raw.eventually-def* **by** *transfer-prover*

lemma *bot*:
shows $\diamond \perp = \perp$
by (*simp add: tls.eventually-def tls.until.simps*)

lemma *bot-conv*:
shows $\diamond P = \perp \longleftrightarrow P = \perp$
by (*auto simp: tls.eventually.bot bot.extremum-uniqueI[OF order.trans[OF tls.eventually.expansive]]*)

lemmas *top = tls.eventually.cl-top*

lemmas *monotone = tls.eventually.monotone-cl*

lemmas *mono = tls.eventually.mono-cl*

lemmas *Sup = tls.eventually.cl-Sup[simplified tls.eventually.bot, simplified]*

lemmas *sup = tls.eventually.Sup[where X={P, Q} for P Q, simplified]*

lemmas $Inf\text{-}le = tls.eventually.cl\text{-}Inf\text{-}le$

lemmas $inf\text{-}le = tls.eventually.cl\text{-}inf\text{-}le$

lemma *neg*:

shows $-\diamond P = \square(-P)$

by (*simp add: tls.always-def*)

lemma *boolean-implication-le*:

shows $\diamond P \longrightarrow_B \diamond Q \leq \diamond(P \longrightarrow_B Q)$

by (*simp add: boolean-implication.conv-sup tls.eventually.sup*)

(*meson le-supI1 compl-mono order.trans le-supI1 tls.eventually.expansive*)

lemmas *simps* =

tls.eventually.bot

tls.eventually.top

tls.eventually.expansive

tls.eventually-def[symmetric]

lemma *terminated*:

shows $\diamond tls.terminated = \bigsqcup (tls.singleton \text{ ‘ } \{\omega. tfinite (behavior.rest \omega)\})$

by *transfer*

(*auto elim!: behavior.stuttering.clE*

dest: iffD2[OF behavior.natural.tfinite]

simp: raw.eventually.terminated behavior.stuttering.cl-bot raw.singleton-def collapse.tfinite)

lemma *always-imp-le*:

shows $P \longrightarrow_{\square} Q \leq \diamond P \longrightarrow_B \diamond Q$

by (*simp add: tls.always-def boolean-implication.conv-sup flip: shunt2*)

(*metis inf-commute order.refl shunt2 sup commute tls.eventually.mono tls.eventually.sup*)

lemma *until*:

shows $\diamond(P \mathcal{U} Q) = \diamond Q$

by (*meson antisym tls.eventually.cl tls.eventually.mono tls.until.eventually-le tls.until.expansiveR*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *always-alt-def*:

shows $\square P = P \mathcal{W} \perp$

by (*simp add: tls.unless-def tls.until.simps*)

setup $\langle \text{Sign.mandatory-path always} \rangle$

lemma *transfer[transfer-rule]*:

shows $rel\text{-}fun (pcr\text{-}tls (=) (=) (=)) (pcr\text{-}tls (=) (=) (=)) raw.always tls.always$

unfolding *tls.always-def raw.always-def* **by** *transfer-prover*

tls.always is an interior operator

lemma *idempotent[simp]*:

shows $\square\square P = \square P$

by (*simp add: tls.always-def*)

lemma *contractive*:

shows $\square P \leq P$

by (*simp add: tls.always-def compl-le-swap2 tls.eventually.expansive*)

lemma *monotone[iff]*:

shows *mono tls.always*

by (simp add: tls.always-def monoI tls.eventually.mono)

lemmas strengthen[strg] = st-monotone[OF tls.always.monotone]

lemmas mono[trans] = monoD[OF tls.always.monotone]

lemma bot:

shows $\Box \perp = \perp$

by (simp add: tls.always-def tls.eventually.simps)

lemma top:

shows $\Box \top = \top$

by (simp add: tls.always-def tls.eventually.simps)

lemma top-conv:

shows $\Box P = \top \iff P = \top$

by (auto simp: tls.always.top intro: top.extremum-uniqueI[OF order.trans[OF - tls.always.contractive]])

lemma Sup-le:

shows $\Box (\bigsqcup (tls.always \text{ ` } X) \leq \Box (\bigsqcup X)$

by (simp add: SupI tls.always.mono)

lemma sup-le:

shows $\Box P \sqcup \Box Q \leq \Box (P \sqcup Q)$

by (simp add: tls.always.mono)

lemma Inf:

shows $\Box (\bigsqcap X) = \bigsqcap (tls.always \text{ ` } X)$

by (rule iffD1[OF compl-eq-compl-iff])

(simp add: tls.always-def image-image tls.eventually.Sup uminus-Inf)

lemma inf:

shows $\Box (P \sqcap Q) = \Box P \sqcap \Box Q$

by (simp add: tls.always-def tls.eventually.sup)

lemma neg:

shows $-\Box P = \Diamond(-P)$

by (simp add: tls.always-def)

lemma always-necessitation:

assumes $\models P$

shows $\models \Box P$

using assms by (simp add: tls.valid-def tls.always.top)

lemma valid-conv:

shows $\models \Box P \iff \models P$

by (simp add: tls.valid-def tls.always.top-conv)

lemma always-imp-le:

shows $P \longrightarrow_{\Box} Q \leq \Box P \longrightarrow_B \Box Q$

by (simp add: galois.conj-imp.lower-upper-contractive tls.always.mono
flip: galois.conj-imp.galois tls.always.inf)

lemma eventually-le:

shows $\Box P \leq \Diamond P$

using tls.always.contractive tls.eventually.cl tls.eventually.mono by blast

lemma not-until-le: — Warford et al. (2020, (81))

shows $\Box P \leq -(Q \text{ U } (-P))$

by (simp add: compl-le-swap1 tls.always.neg tls.until.eventually-le)

lemmas_simps =
 tls.always.bot
 tls.always.top
 tls.always.contractive
 tls.always-alt-def[symmetric]

setup <Sign.parent-path>

lemma until-alwaysR-le: — Warford et al. (2020, (140))
 shows $P \mathcal{U} \Box Q \leq \Box(P \mathcal{U} Q)$
by transfer (rule raw.until.alwaysR-le)

lemma until-alwaysR: — Warford et al. (2020, (141))
 shows $P \mathcal{U} \Box P = \Box P$
by (simp add: order.eq-iff order.trans[OF tls.until-alwaysR-le] tls.until_simps)

lemma eventually-always-always-eventually-le: — Warford et al. (2020, (145))
 shows $\Diamond \Box P \leq \Box \Diamond P$
by (simp add: tls.eventually-def tls.until-alwaysR-le)

lemma always-inf-eventually-eventually-le:
 shows $\Box P \sqcap \Diamond Q \leq \Diamond(P \sqcap Q)$
by (simp add: shunt1 order.trans[OF - tls.eventually.always-imp-le] boolean-implication.mp
 tls.always.mono
 flip: boolean-implication-def)

lemma always-always-imp: — Kröger and Merz (2008, §2.2: T33 frame)
 shows $\models \Box P \longrightarrow_B \Box Q \longrightarrow_B \Box(P \sqcap Q)$
by (simp add: tls.validI tls.always.inf flip: boolean-implication.infl)

lemma always-eventually-imp: — Kröger and Merz (2008, §2.2: T34 frame)
 shows $\models \Box P \longrightarrow_B \Diamond Q \longrightarrow_B \Diamond(P \sqcap Q)$
by (simp add: tls.validI boolean-implication.mp tls.always-inf-eventually-eventually-le)

lemma always-imp-always-generalization: — Kröger and Merz (2008, §2.2: T35)
 shows $\Box P \longrightarrow_{\Box} Q \leq \Box P \longrightarrow_B \Box Q$
by (simp add: order.trans[OF tls.always.always-imp-le])

lemma always-imp-eventually-generalization: — Kröger and Merz (2008, §2.2: T36)
 shows $P \longrightarrow_{\Box} \Diamond Q \leq \Diamond P \longrightarrow_B \Diamond Q$
by (metis tls.eventually.always-imp-le tls.eventually.idempotent(1))

The following show that there is no point nesting *tls.always* and *tls.eventually* more than two deep.

lemma always-eventually-always-absorption: — Kröger and Merz (2008, §2.2: T37)
 shows $\Diamond \Box \Diamond P = \Box \Diamond P$
by (metis order.eq-iff tls.eventually.expansive tls.eventually.idempotent(1)
 tls.eventually-always-always-eventually-le)

lemma eventually-always-eventually-absorption: — Kröger and Merz (2008, §2.2: T38)
 shows $\Box \Diamond \Box P = \Diamond \Box P$
by (metis tls.always.neg tls.always-def tls.always-eventually-always-absorption)

lemma always-imp-always-eventually-le: — Warford et al. (2020, (157))
 shows $P \longrightarrow_{\Box} Q \leq \Box \Diamond P \longrightarrow_B \Box \Diamond Q$
by (simp add: order.trans[OF - tls.always.always-imp-le]
 order.trans[OF - tls.always.mono[OF tls.eventually.always-imp-le]])

lemma *always-imp-eventually-always-le*: — Warford et al. (2020, (158))

shows $P \longrightarrow_{\square} Q \leq \diamond \square P \longrightarrow_B \diamond \square Q$

by (*simp add: order.trans[OF - tls.eventually.always-imp-le]*
order.trans[OF - tls.always.mono[OF tls.always.always-imp-le]])

lemma *always-eventually-inf-le*:

shows $\square \diamond (P \sqcap Q) \leq \square \diamond P \sqcap \square \diamond Q$ — Warford et al. (2020, (159))

by (*simp add: tls.always.mono tls.eventually.mono*)

lemma *eventually-always-sup-le*:

shows $\diamond \square P \sqcap \diamond \square Q \leq \diamond \square (P \sqcup Q)$ — Warford et al. (2020, (160))

by (*simp add: le-infI2 tls.always.mono tls.eventually.mono*)

lemma *always-eventually-sup*: — Warford et al. (2020, (161))

fixes $P :: ('a, 's, 'v) \text{tls}$

shows $\square \diamond (P \sqcup Q) = \square \diamond P \sqcup \square \diamond Q$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

show *?lhs ≤ ?rhs*

proof *transfer*

fix $P Q :: ('a, 's, 'v) \text{behavior.t set}$

have $\exists \omega' \in P. \exists i. \text{behavior.dropn } i \ \omega_j = \text{Some } \omega'$

if $\forall i \ \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \longrightarrow (\exists \omega'' \in P \cup Q. \exists i. \text{behavior.dropn } i \ \omega' = \text{Some } \omega'')$

and $\text{behavior.dropn } i \ \omega = \text{Some } \omega_i$

and $\forall \omega' \in Q. \forall i. \text{behavior.dropn } i \ \omega_i \neq \text{Some } \omega'$

and $\text{behavior.dropn } j \ \omega = \text{Some } \omega_j$

for $\omega \ i \ j \ \omega_i \ \omega_j$

using *spec[where x=max i j, OF that(1)] that(2,3,4)*

by (*clarsimp simp: nat-le-iff-add split: split-asm-max;*

metis add-diff-inverse-nat behavior.dropn.dropn bind.bind-lunit order.asym)

then show $\text{raw.always } (\text{raw.eventually } (P \cup Q))$

$\subseteq \text{raw.always } (\text{raw.eventually } P) \cup \text{raw.always } (\text{raw.eventually } Q)$

by (*clarsimp simp: raw.eventually-alt-def raw.always-alt-def*)

qed

show *?rhs ≤ ?lhs*

by (*simp add: tls.eventually.sup order.trans[OF - tls.always.sup-le]*)

qed

lemma *eventually-always-inf*: — Warford et al. (2020, (162))

shows $\diamond \square (P \sqcap Q) = \diamond \square P \sqcap \diamond \square Q$

by (*subst compl-eq-compl-iff[symmetric]*)

(*simp add: tls.always.neg tls.always-eventually-sup tls.eventually.neg*)

lemma *eventual-latching*: — Warford et al. (2020, (163))

shows $\diamond \square (P \longrightarrow_B \square Q) = \diamond \square (-P) \sqcup \diamond \square Q$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*)

show *?lhs ≤ ?rhs*

by (*rule order.trans[OF tls.eventually.mono[OF tls.always-imp-always-eventually-le]]*)

(*simp add: boolean-implication.conv-sup tls.always.neg*

tls.eventually.neg tls.eventually.sup tls.eventually-always-eventually-absorption)

have $\diamond \square Q \leq \diamond \square (-P \sqcup \square Q)$

apply (*rule order.trans[OF tls.eventually.mono[OF eq-refl[OF tls.always.idempotent[symmetric]]]]*)

apply (*rule tls.eventually.mono[OF tls.always.mono]*)

apply *simp*

done

then show *?rhs ≤ ?lhs*

by (*simp add: le-sup-iff boolean-implication.conv-sup monoD*)

qed

setup $\langle \text{Sign.mandatory-path unless} \rangle$

lemma *transfer*[*transfer-rule*]:

shows $\text{rel-fun } (\text{pcr-tls } (=) (=) (=)) (\text{rel-fun } (\text{pcr-tls } (=) (=) (=)) (\text{pcr-tls } (=) (=) (=)))$
 $(\lambda P Q. \text{raw.until } P Q \cup \text{raw.always } P)$
tls.unless

unfolding *tls.unless-def* **by** *transfer-prover*

lemma *neg*: — Warford et al. (2020, (170))

shows $\neg(P \mathcal{W} Q) = (\neg Q) \mathcal{U} (\neg P \sqcap \neg Q)$

by *transfer* (*rule raw.unless.neg*)

lemma *alwaysR-le*: — Warford et al. (2020, (177))

shows $P \mathcal{W} \square Q \leq \square(P \mathcal{W} Q)$

by (*simp add: tls.unless-def order.trans[OF tls.until-alwaysR-le] tls.always.mono*
order.trans[OF - tls.always.sup-le])

lemma *sup-le*: — Warford et al. (2020, (206))

shows $P \mathcal{W} Q \leq P \sqcup Q$

by (*rule iffD1[OF compl-le-compl-iff]*) (*simp add: tls.unless.neg tls.until.expansive*)

lemma *ordering*: — Warford et al. (2020, (252))

shows $\models (\neg P) \mathcal{W} Q \sqcup (\neg Q) \mathcal{W} P$

by (*simp add: ac-simps tls.validI tls.unless-def tls.until.ordering tls.eventually.sup flip: tls.eventually.neg*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path until} \rangle$

lemma *eq-unless-inf-eventually*:

shows $P \mathcal{U} Q = (P \mathcal{W} Q) \sqcap \diamond Q$

by *transfer* (*force simp: raw.until-def raw.eventually-def raw.always-alt-def behavior.dropn.shorterD*)

lemma *always-strengthen-le*: — Warford et al. (2020, (83))

shows $\square P \sqcap (Q \mathcal{U} R) \leq (P \sqcap Q) \mathcal{U} (P \sqcap R)$

by *transfer*

(*clarsimp simp: raw.until-def raw.always-alt-def;*
fastforce simp: behavior.dropn.shorterD del: exI intro!: exI)

lemma *until-weakI*:

shows $\square P \sqcap \diamond Q \leq P \mathcal{U} Q$ (**is** *?lhs* \leq *?rhs*) — Warford et al. (2020, (84))

by (*simp add: tls.eventually-def order.trans[OF tls.until.always-strengthen-le] tls.until.mono*)

lemma *always-impL*: — Warford et al. (2020, (86))

shows $P \longrightarrow_{\square} P' \sqcap P \mathcal{U} Q \leq P' \mathcal{U} Q$ (**is** *?thesis1*)

and $P \mathcal{U} Q \sqcap P \longrightarrow_{\square} P' \leq P' \mathcal{U} Q$ (**is** *?thesis2*)

proof —

show *?thesis1*

by (*rule order.trans[OF tls.until.always-strengthen-le]*)
(*simp add: tls.until.mono boolean-implication.shunt1*)

then show *?thesis2*

by (*simp add: inf-commute*)

qed

lemma *always-impR*: — Warford et al. (2020, (85))

shows $Q \longrightarrow_{\square} Q' \sqcap P \mathcal{U} Q \leq P \mathcal{U} Q'$ (**is** *?thesis1*)

and $P \mathcal{U} Q \sqcap Q \longrightarrow_{\square} Q' \leq P \mathcal{U} Q'$ (**is** *?thesis2*)

proof –
show *?thesis1*
 by (rule order.trans[*OF tls.until.always-strengthen-le*])
 (simp add: *tls.until.mono boolean-implication.shunt1*)
then show *?thesis2*
 by (simp add: *inf-commute*)
qed

lemma *neg*: — Warford et al. (2020, (173))
shows $-(P \mathcal{U} Q) = (-Q) \mathcal{W} (-P \sqcap -Q)$
unfolding *tls.unless-def*
by (simp flip: *tls.until.eq-unless-inf-eventually tls.unless.neg tls.eventually.neg*
boolean-algebra.de-Morgan-conj)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path state-prop} \rangle$

lemmas *monotone* = *raw.state-prop.monotone*[*transferred*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF tls.state-prop.monotone*]
lemmas *mono* = *monoD*[*OF tls.state-prop.monotone*]

lemma *Sup*:
shows *tls.state-prop* ($\bigsqcup X$) = \bigsqcup (*tls.state-prop* ‘ *X*)
by *transfer* (simp add: *raw.state-prop.Sup behavior.stuttering.cl-bot*)

lemma *Inf*:
shows *tls.state-prop* ($\bigsqcap X$) = \bigsqcap (*tls.state-prop* ‘ *X*)
by *transfer* (simp add: *raw.state-prop.Inf*)

lemmas *simps* = *raw.state-prop.simps*[*transferred*]

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path terminated} \rangle$

lemma *not-bot*:
shows *tls.terminated* $\neq \perp$
by *transfer*
 (simp add: *raw.terminated-def exI*[**where** *x=behavior.B undefined (TNil undefined)*] *behavior.sset.simps*)

lemma *not-top*:
shows *tls.terminated* $\neq \top$
by *transfer*
 (*fastforce simp: raw.terminated-def*
dest: subsetD[*OF Set.equalityD2*, **where** *c=behavior.B undefined (trepeat (undefined, undefined))*]))

lemma *inf-always*:
shows *tls.terminated* $\sqcap \square P = \text{tls.terminated} \sqcap P$
by (rule *antisym*[*OF inf.mono*[*OF order.refl tls.always.contractive*]])
 (*transfer*; *simp add: raw.terminated.inf-always-le*)

lemma *always-le-conv*:
shows *tls.terminated* $\leq \square P \longleftrightarrow \text{tls.terminated} \leq P$
by (simp add: *inf.order-iff tls.terminated.inf-always*)

lemma *inf-eventually*:
shows *tls.terminated* $\sqcap \diamond P = \text{tls.terminated} \sqcap P$ (**is** *?lhs = ?rhs*)

proof(*rule antisym*[*OF - inf.mono*[*OF order.refl tls.eventually.expansive*]])
have $tls.terminated \sqcap - P \leq tls.terminated \sqcap - \diamond P$
by (*simp add: tls.terminated.inf-always tls.eventually.neg*)
then show $?lhs \leq ?rhs$
by (*simp add: boolean-implication.shunt1 boolean-implication.imp-trivialI*)
qed

lemma *eventually-le-conv*:
shows $tls.terminated \leq tls.eventually P \longleftrightarrow tls.terminated \leq P$
by (*simp add: inf.order-iff tls.terminated.inf-eventually*)

lemma *eq-always-terminated*:
shows $tls.terminated = \sqcap tls.terminated$
by (*rule order.antisym*[*OF - tls.always.contractive*])
(*simp add: tls.terminated.always-le-conv*)

setup $\langle Sign.parent-path \rangle$

16.5.1 Leads-to and leads-to-via

So-called *response* properties are of the form $P \longrightarrow_{\square} \diamond Q$ (pronounced “ P leads to Q ”, written $P \rightsquigarrow Q$) (Manna and Pnueli 1991). This connective is similar to the “ensures” modality of Chandy and Misra (1989, §3.4.4). Jackson (1998) used the more general “ P leads to Q via I ” form $P \longrightarrow_{\square} I \mathcal{U} Q$ to establish liveness properties in a sequential setting.

lemma *leads-to-refl*:
shows $\models P \rightsquigarrow P$
by (*simp add: tls.validI boolean-implication.shunt-top tls.always.top-conv tls.eventually.expansive top.extremum-unique*)

lemma *leads-to-mono*:
assumes $P' \leq P$
assumes $Q \leq Q'$
shows $P \rightsquigarrow Q \leq P' \rightsquigarrow Q'$
by (*simp add: assms boolean-implication.mono tls.always.mono tls.eventually.mono*)

lemma *leads-to-supL*:
shows $(P \rightsquigarrow R) \sqcap (Q \rightsquigarrow R) \leq (P \sqcup Q) \rightsquigarrow R$
by (*simp add: boolean-implication.conv-sup sup-inf-distrib2 tls.always.inf*)

lemma *always-imp-leads-to*:
shows $P \longrightarrow_{\square} Q \leq P \rightsquigarrow Q$
by (*simp add: boolean-implication.mono tls.always.mono tls.eventually.expansive*)

lemma *leads-to-eventually*:
shows $\diamond P \sqcap (P \rightsquigarrow Q) \leq \diamond Q$
by (*simp add: galois.conj-imp.galois tls.always-imp-eventually-generalization*)

lemma *leads-to-leads-to-via*:
shows $P \longrightarrow_{\square} Q \mathcal{U} R \leq P \rightsquigarrow R$
by (*simp add: boolean-implication.mono tls.always.mono tls.until.eventually-le*)

lemma *leads-to-trans*:
shows $P \rightsquigarrow Q \sqcap Q \rightsquigarrow R \leq P \rightsquigarrow R$ (**is** $?lhs \leq ?rhs$)
proof –
have $?lhs \leq P \rightsquigarrow Q \sqcap \sqcap(Q \rightsquigarrow R)$
by (*simp add: tls.always.simps*)
also have $\dots \leq P \rightsquigarrow Q \sqcap \diamond Q \rightsquigarrow R$

by (*meson order.refl inf-mono tls.always.mono tls.always-imp-eventually-generalization*)
 also have ... \leq ?*rhs*
 by (*simp add: boolean-implication.trans tls.always.mono flip: tls.always.inf*)
 finally show ?*thesis* .
 qed

lemma *leads-to-via-weakenR*:

shows $Q \longrightarrow_{\square} Q' \sqcap P \longrightarrow_{\square} I \mathcal{U} Q \leq P \longrightarrow_{\square} I \mathcal{U} Q'$
 by *transfer*
 (*clarsimp simp: raw.always-alt-def raw.until-def boolean-implication.set-alt-def;*
metis behavior.dropn.dropn Option.bind.bind-lunit)

lemma *leads-to-via-supL*: — useful for case distinctions

shows $P \longrightarrow_{\square} I \mathcal{U} Q \sqcap P' \longrightarrow_{\square} I' \mathcal{U} Q \leq P \sqcup P' \longrightarrow_{\square} (I \sqcup I') \mathcal{U} Q$
 by (*simp add: boolean-implication.conv-sup ac-simps le-infI2 le-supI2*
monoD[OF tls.always.monotone] tls.until.mono)

lemma *leads-to-via-trans*:

shows $(P \longrightarrow_{\square} I \mathcal{U} Q) \sqcap (Q \longrightarrow_{\square} I' \mathcal{U} R) \leq P \longrightarrow_{\square} (I \sqcup I') \mathcal{U} R$ (is ?*lhs* \leq ?*rhs*)
proof –
 have ?*lhs* \leq $\square(P \longrightarrow_B I \mathcal{U} (I' \mathcal{U} R))$
 by (*subst inf.commute*) (*rule tls.leads-to-via-weakenR*)
 also have ... \leq ?*rhs*
 by (*strengthen ord-to-strengthen(1)[OF tls.until.supL-ordering-le]*) (*rule order.refl*)
 finally show ?*thesis* .
 qed

lemma *leads-to-via-disj*: — more like a chaining rule

shows $(P \longrightarrow_{\square} I \mathcal{U} Q) \sqcap (Q \longrightarrow_{\square} I' \mathcal{U} R) \leq (P \sqcup Q \longrightarrow_{\square} (I \sqcup I') \mathcal{U} R)$
 by (*simp add: boolean-implication-def inf.coboundedI2 le-supI2 tls.always.mono tls.until.mono*)

16.5.2 Fairness

A few renderings of weak fairness. [van Glabbeek and Höfner \(2019\)](#) call this “response to insistence” as a generalisation of weak fairness.

definition *weakly-fair* :: $(\prime a, \prime s, \prime v) \text{tls} \Rightarrow (\prime a, \prime s, \prime v) \text{tls} \Rightarrow (\prime a, \prime s, \prime v) \text{tls}$ **where**
weakly-fair enabled taken = $\square \text{enabled} \longrightarrow_{\square} \diamond \text{taken}$

lemma *weakly-fair-def2*:

shows *tls.weakly-fair enabled taken* = $\square(\neg(\square(\text{enabled} \sqcap \neg \text{taken})))$
 by (*simp add: tls.weakly-fair-def tls.always-def tls.eventually.sup*)

lemma *weakly-fair-def3*:

shows *tls.weakly-fair enabled taken* = $\diamond \square \text{enabled} \longrightarrow_B \square \diamond \text{taken}$
 by (*simp add: tls.weakly-fair-def boolean-implication.conv-sup*
tls.always.neg tls.always-eventually-sup tls.eventually.neg
flip: tls.eventually.sup)

lemma *weakly-fair-def4*:

shows *tls.weakly-fair enabled taken* = $\square \diamond (\text{enabled} \longrightarrow_B \text{taken})$
 by (*simp add: tls.weakly-fair-def boolean-implication.conv-sup tls.always.neg tls.eventually.sup*)

setup $\langle \text{Sign.mandatory-path weakly-fair} \rangle$

lemma *mono*:

assumes $P' \leq P$
 assumes $Q \leq Q'$
 shows *tls.weakly-fair* $P Q \leq \text{tls.weakly-fair} P' Q'$

unfolding *tls.weakly-fair-def*
apply (*strengthen ord-to-strengthen(1)[OF assms(1)]*)
apply (*strengthen ord-to-strengthen(1)[OF assms(2)]*)
apply (*rule order.refl*)
done

lemma *strengthen[strg]:*
assumes *st-ord* $(\neg F) P P'$
assumes *st-ord* $F Q Q'$
shows *st-ord* $F (tls.weakly-fair P Q) (tls.weakly-fair P' Q')$
using *assms* **by** (*cases F*) (*auto simp: tls.weakly-fair.mono*)

lemma *weakly-fair-triv:*
shows $\Box \Diamond (\neg enabled) \leq tls.weakly-fair\ enabled\ taken$
by (*simp add: tls.weakly-fair-def3 boolean-implication.conv-sup tls.always.neg tls.eventually.neg*)

lemma *mp:*
shows $tls.weakly-fair\ enabled\ taken \sqcap \Box enabled \leq \Diamond taken$
by (*simp add: tls.weakly-fair-def boolean-implication.shunt1 tls.always.contractive*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ always \rangle$

lemma *weakly-fair:*
shows $\Box (tls.weakly-fair\ enabled\ taken) = tls.weakly-fair\ enabled\ taken$
by (*simp add: tls.weakly-fair-def tls.always.simps*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ eventually \rangle$

lemma *weakly-fair:*
shows $\Diamond (tls.weakly-fair\ enabled\ taken) = tls.weakly-fair\ enabled\ taken$
by (*simp add: tls.weakly-fair-def4 tls.always-eventually-always-absorption*)

setup $\langle Sign.parent-path \rangle$

Similarly for strong fairness. [van Glabbeek and Höfner \(2019\)](#) call this "response to persistence" as a generalisation of strong fairness.

definition *strongly-fair* $:: ('a, 's, 'v)\ tls \Rightarrow ('a, 's, 'v)\ tls \Rightarrow ('a, 's, 'v)\ tls$ **where**
strongly-fair enabled taken $= \Box \Diamond enabled \longrightarrow_{\Box} \Diamond taken$

lemma *strongly-fair-def2:*
shows $tls.strongly-fair\ enabled\ taken = \Box (\neg \Box (\Diamond enabled \sqcap \neg taken))$
by (*simp add: tls.strongly-fair-def boolean-implication.conv-sup tls.always.neg tls.eventually.sup*)

lemma *strongly-fair-def3:*
shows $tls.strongly-fair\ enabled\ taken = \Box \Diamond enabled \longrightarrow_B \Box \Diamond taken$
by (*simp add: tls.strongly-fair-def boolean-implication.conv-sup tls.always.neg tls.eventually.neg*
tls.always-eventually-sup tls.eventually-always-eventually-absorption
flip: tls.eventually.sup)

setup $\langle Sign.mandatory-path\ strongly-fair \rangle$

lemma *mono:*
assumes $P' \leq P$
assumes $Q \leq Q'$

shows $tls.\text{strongly-fair } P \ Q \leq \text{tls.strongly-fair } P' \ Q'$
unfolding $tls.\text{strongly-fair-def}$
apply ($\text{strengthen ord-to-strengthen}(1)[OF \text{ assms}(1)]$)
apply ($\text{strengthen ord-to-strengthen}(1)[OF \text{ assms}(2)]$)
apply (rule order.refl)
done

lemma $\text{strengthen}[strg]$:
assumes $st\text{-ord } (\neg F) \ P \ P'$
assumes $st\text{-ord } F \ Q \ Q'$
shows $st\text{-ord } F \ (\text{tls.strongly-fair } P \ Q) \ (\text{tls.strongly-fair } P' \ Q')$
using assms **by** ($\text{cases } F$) ($\text{auto simp: tls.strongly-fair.mono}$)

lemma supL : — does not hold for $tls.\text{weakly-fair}$
shows $tls.\text{strongly-fair } (\text{enabled1} \sqcup \text{enabled2}) \ \text{taken}$
 $= (\text{tls.strongly-fair } \text{enabled1} \ \text{taken} \sqcap \text{tls.strongly-fair } \text{enabled2} \ \text{taken})$
by ($\text{simp add: boolean-implication.conv-sup sup-inf-distrib2 tls.always.inf tls.always-eventually-sup}$
 $tls.\text{strongly-fair-def}$)

lemma weakly-fair-le :
shows $tls.\text{strongly-fair } \text{enabled} \ \text{taken} \leq \text{tls.weakly-fair } \text{enabled} \ \text{taken}$
by ($\text{simp add: tls.strongly-fair-def3 tls.weakly-fair-def3 boolean-implication.mono}$
 $tls.\text{eventually-always-always-eventually-le}$)

lemma $\text{always-enabled-weakly-fair-strongly-fair}$:
shows $\square \text{enabled} \leq \text{tls.weakly-fair } \text{enabled} \ \text{taken} \longleftrightarrow_B \text{tls.strongly-fair } \text{enabled} \ \text{taken}$
by ($\text{simp add: boolean-eq-def boolean-implication-def}$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path always} \rangle$

lemma strongly-fair :
shows $\square(\text{tls.strongly-fair } \text{enabled} \ \text{taken}) = \text{tls.strongly-fair } \text{enabled} \ \text{taken}$
by ($\text{simp add: tls.strongly-fair-def}$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path eventually} \rangle$

lemma strongly-fair :
shows $\diamond(\text{tls.strongly-fair } \text{enabled} \ \text{taken}) = \text{tls.strongly-fair } \text{enabled} \ \text{taken}$
by ($\text{simp add: tls.strongly-fair-def2 tls.always.neg tls.always-eventually-always-absorption}$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

16.6 Safety Properties

We now carve the safety properties out of the ($'a$, $'s$, $'v$) tls lattice.

References:

- [Alpern and Schneider \(1985\)](#); [Alpern, Demers, and Schneider \(1986\)](#); [Schneider \(1987, §2\)](#)
 - observes that Lamport’s earlier definitions do not work without stuttering
 - provides the now standard definition that works with and without stuttering

- [Abadi and Lamport \(1991, §2.2\)](#): topological definitions and intuitions
- [Sistla \(1994, §2.2\)](#)

We go a different way: we establish a Galois connection with $(\prime a, \prime s, \prime v)$ *spec*.

Observations:

- our safety closure for $(\prime a, \prime s, \prime v)$ *tls* introduces infinite sequences to stand for the prefixes in $(\prime a, \prime s, \prime v)$ *spec*
 - i.e., the non-termination of trace σ ($\text{trace.term } \sigma = \text{None}$) is represented by a behavior ending with $\text{trace.final } \sigma$ infinitely stuttered
 - [Abadi and Lamport \(1991, §2.1\)](#) consider these behaviors to represent terminating processes

setup $\langle \text{Sign.mandatory-path raw} \rangle$

definition $\text{to-spec} :: (\prime a, \prime s, \prime v)$ *behavior.t set* \Rightarrow $(\prime a, \prime s, \prime v)$ *trace.t set* **where**
 $\text{to-spec } T = \{\text{behavior.take } i \ \omega \mid \omega \ i. \ \omega \in T\}$

definition $\text{from-spec} :: (\prime a, \prime s, \prime v)$ *trace.t set* \Rightarrow $(\prime a, \prime s, \prime v)$ *behavior.t set* **where**
 $\text{from-spec } S = \{\omega . \forall i. \text{behavior.take } i \ \omega \in S\}$

interpretation *safety*: $\text{galois.powerset raw.to-spec raw.from-spec}$
by *standard* ($\text{fastforce simp: raw.to-spec-def raw.from-spec-def}$)

setup $\langle \text{Sign.mandatory-path from-spec} \rangle$

lemma *empty*:

shows $\text{raw.from-spec } \{\} = \{\}$

by ($\text{simp add: raw.from-spec-def}$)

lemma *singleton*:

shows $\text{raw.from-spec } (\text{Safety-Logic.raw.singleton } \sigma)$

$= \bigcup (\text{raw.singleton } \prime \{\omega . \forall i. \text{behavior.take } i \ \omega \in \text{Safety-Logic.raw.singleton } \sigma\})$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \subseteq ?rhs$ **by** ($\text{force simp: raw.from-spec-def TLS.raw.singleton-def}$)

show $?rhs \subseteq ?lhs$

by ($\text{clarsimp simp: raw.from-spec-def TLS.raw.singleton-def Safety-Logic.raw.singleton-def}$
 $\text{elim!: behavior.stuttering.clE}$)

($\text{metis behavior.stuttering.equiv.takeE raw.spec.closed raw.spec.closed.stuttering-closed}$
 $\text{trace.stuttering.clI trace.stuttering.closed-conv}$)

qed

lemma *sup*:

assumes $P \in \text{raw.spec.closed}$

assumes $Q \in \text{raw.spec.closed}$

shows $\text{raw.from-spec } (P \cup Q) = \text{raw.from-spec } P \cup \text{raw.from-spec } Q$

by ($\text{rule antisym}[OF - \text{raw.safety.sup-upper-le}]$)

($\text{clarsimp simp: raw.from-spec-def}$;

$\text{meson behavior.take.mono downwards.closed-in linorder-le-cases}$

$\text{raw.spec.closed.downwards-closed}[OF \text{assms}(1)] \text{raw.spec.closed.downwards-closed}[OF \text{assms}(2)])$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path to-spec} \rangle$

lemma *singleton*:

shows $\text{raw.to-spec } (\text{TLS.raw.singleton } \omega)$

$= (\bigcup i. \text{Safety-Logic.raw.singleton } (\text{behavior.take } i \ \omega))$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)
show $?lhs \subseteq ?rhs$
 by (*fastforce simp: TLS.raw.singleton-def raw.to-spec-def*
Safety-Logic.raw.singleton-def raw.spec.cl-def
elim: behavior.stuttering.clE behavior.stuttering.equiv.takeE[OF sym]
trace.stuttering.cII[OF - sym, rotated])
show $?rhs \subseteq ?lhs$
 by (*fastforce simp: Safety-Logic.raw.singleton-def raw.spec.cl-def TLS.raw.singleton-def*
raw.to-spec-def trace.less-eq-take-def trace.take.behavior.take
elim: downwards.clE trace.stuttering.clE trace.stuttering.equiv.behavior.takeE)
qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path safety} \rangle$

lemma *cl-altI*:

assumes $\bigwedge i. \exists \omega' \in P. \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$
shows $\omega \in \text{raw.safety.cl } P$
using *assms* **by** (*fastforce simp: raw.safety.cl-def raw.from-spec-def raw.to-spec-def*)

lemma *cl-altE*:

assumes $\omega \in \text{raw.safety.cl } P$
obtains ω' **where** $\omega' \in P$ **and** $\text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$
proof(*atomize-elim, cases enat i ≤ tlength (behavior.rest ω)*)
case *True* **with** *assms* **show** $\exists \omega'. \omega' \in P \wedge \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$
 by (*clarsimp simp: raw.safety.cl-def raw.from-spec-def raw.to-spec-def*)
 (*metis behavior.take.length behavior.take.sel(3) ttake-eq-None-conv(1)*
min.absorb2 min-enat2-conv-enat the-enat.simps)

next

case *False* **with** *assms* **show** $\exists \omega'. \omega' \in P \wedge \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$
 by (*clarsimp simp: raw.safety.cl-def raw.from-spec-def raw.to-spec-def*)
 (*metis behavior.continue.take-drop-id behavior.take.continue-id leI*)

qed

lemma *cl-alt-def*: — [Alpern et al. \(1986\)](#): the classical definition: ω belongs to the safety closure of P if every prefix of ω can be extended to a behavior in P

shows $\text{raw.safety.cl } P = \{\omega. \forall i. \exists \beta. \text{behavior.take } i \ \omega @_{-B} \beta \in P\}$ (**is** $?lhs = ?rhs$)

proof(*rule antisym*)

show $?lhs \subseteq ?rhs$
 by *clarsimp* (*metis behavior.continue.take-drop-id raw.safety.cl-altE*)
show $?rhs \subseteq ?lhs$
proof(*clarify intro!: raw.safety.cl-altI*)
fix $\omega \ i$
assume $\forall j. \exists \beta. \text{behavior.take } j \ \omega @_{-B} \beta \in P$
then show $\exists \omega' \in P. \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega'$
 by (*force dest: spec[where x=i]*
intro: exI[where x=i] rev-bexI
simp: behavior.take.continue trace.take.behavior.take trace.continue.self-conv
ttake-eq-None-conv length-ttake
split: option.split enat.split)

qed

qed

lemma *closed-alt-def*: — If ω is not in P then some prefix of ω has irretrievably gone wrong

shows $\text{raw.safety.closed} = \{P. \forall \omega. \omega \notin P \longrightarrow (\exists i. \forall \beta. \text{behavior.take } i \ \omega @_{-B} \beta \notin P)\}$

unfolding *raw.safety.closed-def raw.safety.cl-alt-def* **by** *fast*

lemma *closed-alt-def2*: — Contraposition gives the customary prefix-closure definition
shows $\text{raw.safety.closed} = \{P. \forall \omega. (\forall i. \exists \beta. \text{behavior.take } i \ \omega \ @_{-B} \ \beta \in P) \longrightarrow \omega \in P\}$
unfolding *raw.safety.closed-alt-def* **by** *fast*

lemma *closedI2*:
assumes $\bigwedge \omega. (\bigwedge i. \exists \beta. \text{behavior.take } i \ \omega \ @_{-B} \ \beta \in P) \implies \omega \in P$
shows $P \in \text{raw.safety.closed}$
using *assms* **unfolding** *raw.safety.closed-alt-def2* **by** *fast*

lemma *closedE2*:
assumes $P \in \text{raw.safety.closed}$
assumes $\bigwedge i. \omega \notin P \implies \exists \beta. \text{behavior.take } i \ \omega \ @_{-B} \ \beta \in P$
shows $\omega \in P$
using *assms* **unfolding** *raw.safety.closed-alt-def2* **by** *blast*

setup $\langle \text{Sign.mandatory-path } cl \rangle$

lemma *state-prop*:
shows $\text{raw.safety.cl } (\text{raw.state-prop } P) = \text{raw.state-prop } P$
by (*simp* *add: raw.safety.cl-alt-def raw.state-prop-def*)

lemma *terminated-iff*:
assumes $\omega \in \text{raw.terminated}$
shows $\omega \in \text{raw.safety.cl } P \longleftrightarrow \omega \in P$ (**is** *?lhs* \longleftrightarrow *?rhs*)
proof(*rule iffI*)
from *assms* **obtain** *i* **where** $\text{tlength } (\text{behavior.rest } \omega) = \text{enat } i$
by (*clarsimp simp: raw.terminated-def tfinite-tlength-conv*)
then show *?lhs* \implies *?rhs*
by (*metis raw.safety.cl-altE[where i=Suc i]*
behavior.continue.take-drop-id behavior.take.continue-id enat-ord-simps(2) lessI)
qed (*simp add: raw.safety.expansive'*)

lemma *terminated*:
shows $\text{raw.safety.cl } \text{raw.terminated} = \text{raw.idle} \cup \text{raw.terminated}$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym[OF subsetI subsetI]*)
fix ω
assume $\omega \in ?lhs$
then have $\text{snd } (\text{tnth } (\text{behavior.rest } \omega) \ i) = \text{behavior.init } \omega$
if $\text{enat } i < \text{tlength } (\text{behavior.rest } \omega)$
for *i*
using *that*
by (*clarsimp simp: raw.terminated-def behavior.take-def behavior.split-all behavior.sset.simps*
split-def
simp del: ttake.simps
elim!: raw.safety.cl-altE[where i=Suc i]
(metis (no-types, lifting) Suc-ile-eq in-tset-conv-tnth nth-ttake
doubleton-eq-iff insert-image insert-absorb2 lessI subset-singletonD ttake-eq-None-conv(1)))
then have $\text{behavior.sset } \omega \subseteq \{\text{behavior.init } \omega\}$
by (*cases* ω) (*clarsimp simp: behavior.sset.simps tset-conv-tnth*)
then show $\omega \in ?rhs$
by (*simp add: raw.idle-alt-def raw.terminated-def*)
next
show $\omega \in ?lhs$ **if** $\omega \in ?rhs$ **for** ω
using *that*
proof(*cases rule: UnE[consumes 1, case-names idle terminated]*)
case *idle* **show** *?thesis*
proof(*rule raw.safety.cl-altI*)
fix *i*

```

let ? $\omega'$  = behavior.take  $i$   $\omega$  @-B TNil undefined
from idle have ? $\omega' \in$  raw.terminated
  by (auto simp: raw.idle-alt-def raw.terminated-def behavior.sset.continue
    dest: subsetD[OF behavior.sset.take-le]
    split: option.split)
moreover
from idle have behavior.take  $i$   $\omega$  = behavior.take  $i$  ? $\omega'$ 
  by (simp add: raw.idle-alt-def behavior.take.continue trace.take.behavior.take
    length-ttake tfinite-tlength-conv)
ultimately show  $\exists \omega' \in$  raw.terminated. behavior.take  $i$   $\omega$  = behavior.take  $i$   $\omega'$ 
  by blast
qed
qed (auto intro: raw.safety.expansive')
qed

```

lemma *le-terminated-bot*:

```

assumes  $P \in$  behavior.stuttering.closed
assumes raw.safety.cl  $P \subseteq$  raw.terminated
shows  $P = \{\}$ 
proof(rule ccontr)
assume  $\langle P \neq \{\} \rangle$  then obtain  $\omega$  where  $\omega \in P$  by blast
let ? $\omega'$  = behavior.B (behavior.init  $\omega$ ) (trepeat (undefined, behavior.init  $\omega$ ))
from  $\langle \omega \in P \rangle$  have ? $\omega' \in$  raw.safety.cl  $P$ 
  by (fastforce intro: exI[where x=behavior.rest  $\omega]$ 
    behavior.stuttering.f-closedI[OF  $\langle P \in$  behavior.stuttering.closed $\rangle$ 
    simp: raw.safety.cl-alt-def behavior.take.trepeat behavior.continue.simps
    behavior.natural.tshift collapse.tshift trace.natural'.replicate
    trace.final'.replicate
    behavior.stuttering.f-closed[OF  $\langle P \in$  behavior.stuttering.closed $\rangle$ 
    simp flip: behavior.natural-def)
moreover have ? $\omega' \notin$  raw.terminated
  by (simp add: raw.terminated-def)
moreover note  $\langle$ raw.safety.cl  $P \subseteq$  raw.terminated $\rangle$ 
ultimately show False by blast
qed

```

lemma *always-le*:

```

shows raw.safety.cl (raw.always  $P$ )  $\subseteq$  raw.always (raw.safety.cl  $P$ )
unfolding raw.always-alt-def raw.safety.cl-alt-def subset-iff mem-Collect-eq
proof(intro allI impI)
  fix  $\omega$   $i$   $\omega' j$ 
  assume *:  $\forall i. \exists \beta. \forall k \omega'. \text{behavior.dropn } k \text{ (behavior.take } i \omega \text{ @}_{-B} \beta) = \text{Some } \omega' \longrightarrow \omega' \in P$ 
  and **: behavior.dropn  $i$   $\omega$  = Some  $\omega'$ 
  from spec[where x=i + j, OF *] ** behavior.take.dropn[OF **, where j=j]
  show  $\exists \beta. \text{behavior.take } j \omega' \text{ @}_{-B} \beta \in P$ 
  by (clarsimp dest!: spec[where x=i])
    (subst (asm) behavior.dropn.continue-shorter;
    force simp: length-ttake trace.dropn.behavior.take
    dest: behavior.dropn.eq-Some-tlengthD
    split: enat.split)
qed

```

lemma *eventually*:

```

assumes  $P \neq \perp$ 
shows raw.safety.cl (raw.eventually  $P$ )
  =  $\neg$ raw.eventually raw.terminated  $\cup$  raw.eventually  $P$  (is ?lhs = ?rhs)
proof(rule antisym[OF subsetI iffD2[OF Un-subset-iff, simplified conj-explode, rule-format, OF subsetI]])
  show  $\omega \in$  ?rhs if  $\omega \in$  ?lhs for  $\omega$ 

```

```

proof(cases tlength (behavior.rest  $\omega$ ))
  case (enat  $i$ ) with that show ?thesis
    by (fastforce dest: spec[where  $x = \text{Suc } i$ ]
      simp: raw.safety.cl-alt-def raw.terminated-def behavior.take.continue-id)
qed (simp add: raw.eventually.terminated tfinite-tlength-conv)
from assms obtain  $\omega_P \in P$  where  $\omega_P \in P$  by blast
show  $\omega \in ?lhs$  if  $\omega \in -\text{raw.eventually raw.terminated}$  for  $\omega$ 
proof(intro raw.safety.cl-altI exI beXI)
  fix  $i$ 
  let  $?w' = \text{behavior.take } i \ \omega \ @_{-B} \ \text{TCons } (\text{undefined}, \text{behavior.init } \omega_P) (\text{behavior.rest } \omega_P)$ 
  from  $\langle \omega_P \in P \rangle \langle \omega \in -\text{raw.eventually raw.terminated} \rangle$  show  $?w' \in \text{raw.eventually } P$ 
    unfolding raw.eventually.terminated
    by (auto intro!: exI[where  $x = \text{Suc } i$ ]
      simp: raw.eventually-alt-def tfinite-tlength-conv behavior.dropn.continue
        length-ttake ttake-eq-None-conv)
  from  $\langle \omega \in -\text{raw.eventually raw.terminated} \rangle$  show  $\text{behavior.take } i \ \omega = \text{behavior.take } i \ ?w'$ 
    by (simp add: raw.eventually.terminated behavior.take.continue trace.take.behavior.take
      length-ttake tfinite-tlength-conv
      split: enat.split)
qed
show  $\text{raw.eventually } P \subseteq ?lhs$ 
  by (fast intro!: order.trans[OF - raw.safety.expansive])
qed

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.mandatory-path closed} \rangle$ 

lemma always-eventually:
  assumes  $P \in \text{raw.safety.closed}$ 
  assumes  $\forall i. \exists j \geq i. \exists \beta. \text{behavior.take } j \ \omega \ @_{-B} \ \beta \in P$ 
  shows  $\omega \in P$ 
using assms(1)
proof(rule raw.safety.closedE2)
  fix  $i$ 
  from spec[OF assms(2), where  $x = i$ ] obtain  $j \ \beta$  where  $i \leq j$  and  $\text{behavior.take } j \ \omega \ @_{-B} \ \beta \in P$ 
  by blast
  then show  $\exists \beta. \text{behavior.take } i \ \omega \ @_{-B} \ \beta \in P$  if  $\omega \notin P$ 
    using that
    by (clarsimp simp: tdropn-tshift2 behavior.continue.tshift2 behavior.continue.take-drop-shorter length-ttake
      behavior.continue.term-Some behavior.take.term-Some-conv ttake-eq-Some-conv
      split: enat.split split-min
      intro!: exI[where  $x = \text{tdropn } i (\text{behavior.rest } (\text{behavior.take } j \ \omega \ @_{-B} \ \beta))$ ])
qed

lemma sup:
  assumes  $P \in \text{raw.safety.closed}$ 
  assumes  $Q \in \text{raw.safety.closed}$ 
  shows  $P \cup Q \in \text{raw.safety.closed}$ 
by (clarsimp simp: raw.safety.closed-alt-def2)
  (meson assms raw.safety.closed.always-eventually sup.cobounded1 sup.cobounded2)

lemma unless: — Sistla (1994, §3.1) – minimality is irrelevant
  assumes  $P \in \text{raw.safety.closed}$ 
  assumes  $Q \in \text{raw.safety.closed}$ 
  shows  $\text{raw.unless } P \ Q \in \text{raw.safety.closed}$ 
proof(rule raw.safety.closedI2)
  fix  $\omega$  assume *:  $\exists \beta. \text{behavior.take } i \ \omega \ @_{-B} \ \beta \in \text{raw.unless } P \ Q$  for  $i$ 

```

```

show  $\omega \in \text{raw.unless } P \ Q$ 
proof(cases  $\forall i \ j \ \omega'. \exists \beta. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } j \ \omega' @_{-B} \beta \in P$ )
  case True
    with  $\langle P \in \text{raw.safety.closed} \rangle$  have  $\text{behavior.dropn } i \ \omega = \text{Some } \omega' \longrightarrow \omega' \in P$  for  $i \ \omega'$ 
      by (blast intro: raw.safety.closedE2)
    then show ?thesis
      by (simp add: raw.always-alt-def)
  next
    case False
    then obtain  $\omega' \ k \ l$ 
      where  $** : \text{behavior.dropn } k \ \omega = \text{Some } \omega' \forall \beta. \text{behavior.take } l \ \omega' @_{-B} \beta \notin P$ 
      by clarsimp
    {
      fix  $i \ \beta$ 
      assume  $kli : k + l \leq i$ 
      moreover
      note  $**$ 
      moreover
      from  $kli$  have  $\exists j. i - k = l + j$  by presburger
      moreover
      from  $\langle \text{behavior.dropn } k \ \omega = \text{Some } \omega' \rangle$   $kli$ 
      have  $*** : k \leq \text{length } (\text{trace.rest } (\text{behavior.take } i \ \omega))$ 
        by (fastforce simp: length-ttake split: enat.splits
          dest: behavior.dropn.eq-Some-tlengthD)
      ultimately have  $**** : \forall \omega''. \text{behavior.dropn } k \ (\text{behavior.take } i \ \omega @_{-B} \beta) = \text{Some } \omega'' \longrightarrow \omega'' \notin P$ 
        by (force simp: behavior.dropn.continue-shorter trace.dropn.behavior.take behavior.take.add
          simp flip: behavior.continue.tshift2)
    }
    {
      assume  $PQ : \text{behavior.take } i \ \omega @_{-B} \beta \in \text{raw.unless } P \ Q$ 
      from  $**** \ PQ$  obtain  $m$ 
        where  $m \leq k$ 
        and  $\forall \omega'. \text{behavior.dropn } m \ (\text{behavior.take } i \ \omega @_{-B} \beta) = \text{Some } \omega' \longrightarrow \omega' \in Q$ 
        and  $\forall p < m. (\forall \omega'. \text{behavior.dropn } p \ (\text{behavior.take } i \ \omega @_{-B} \beta) = \text{Some } \omega' \longrightarrow \omega' \in P)$ 
        by (auto 6 0 simp: raw.until-def raw.always-alt-def)
        (metis behavior.dropn.shorterD leI nle-le option.sel)
      with  $kli \ ***$ 
      have  $(\exists m \leq k. (\forall \omega'. \text{behavior.dropn } m \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } (i - m) \ \omega' @_{-B} \beta \in Q)$ 
         $\wedge (\forall p < m. (\forall \omega'. \text{behavior.dropn } p \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } (i - p) \ \omega' @_{-B} \beta \in P))$ )
        by (clarsimp simp: exI[where x=m] behavior.dropn.continue-shorter trace.dropn.behavior.take)
    }
  }
  }
  then have  $\forall i. \exists n \geq i. \exists m \leq k. \exists \beta. (\forall \omega'. \text{behavior.dropn } m \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } (n - m) \ \omega' @_{-B} \beta \in Q)$ 
     $\wedge (\forall p < m. \forall \omega'. \text{behavior.dropn } p \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } (n - p) \ \omega' @_{-B} \beta \in P)$ 
    using  $*$  by (metis nle-le)
  then obtain  $m$ 
    where  $m \leq k \forall i. \exists n \geq i. \exists \beta. (\forall \omega'. \text{behavior.dropn } m \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } (n - m) \ \omega' @_{-B} \beta \in Q)$ 
     $\wedge (\forall p < m. \forall \omega'. \text{behavior.dropn } p \ \omega = \text{Some } \omega' \longrightarrow \text{behavior.take } (n - p) \ \omega' @_{-B} \beta \in P)$ 
    by (clarsimp simp: always-eventually-pigeonhole)
  with  $\text{behavior.dropn.shorterD}[OF \langle \text{behavior.dropn } k \ \omega = \text{Some } \omega' \rangle \langle m \leq k \rangle]$ 
     $\text{raw.safety.closed.always-eventually}[OF \langle P \in \text{raw.safety.closed} \rangle]$ 
     $\text{raw.safety.closed.always-eventually}[OF \langle Q \in \text{raw.safety.closed} \rangle]$ 
  show  $\omega \in \text{raw.unless } P \ Q$ 
  apply  $-$ 
  apply clarsimp
  apply (rule raw.untilI, assumption)

```

```

    apply (meson add-le-imp-le-diff)
    apply (metis add-le-imp-le-diff option.sel behavior.dropn.shorterD[OF - less-imp-le])
  done
qed
qed

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path downwards.closed›

lemma to-spec:
  shows range raw.to-spec  $\subseteq$  downwards.closed
by (fastforce elim: downwards.clE simp: raw.to-spec-def trace.less-eq-take-def trace.take.behavior.take)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path trace.stuttering.closed›

lemma to-spec:
  shows raw.to-spec ‘ behavior.stuttering.closed  $\subseteq$  trace.stuttering.closed
by (fastforce simp: raw.to-spec-def
    elim: trace.stuttering.clE trace.stuttering.equiv.E trace.stuttering.equiv.behavior.takeE
    dest: behavior.stuttering.closed-in)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path raw.spec.closed›

lemma to-spec:
  shows raw.to-spec ‘ behavior.stuttering.closed  $\subseteq$  raw.spec.closed
using downwards.closed.to-spec trace.stuttering.closed.to-spec by (blast intro: raw.spec.closed.I)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path behavior.stuttering.closed›

lemma from-spec:
  shows raw.from-spec ‘ trace.stuttering.closed
     $\subseteq$  (behavior.stuttering.closed :: ('a, 's, 'v) behavior.t set set)
proof –
  have *: behavior.take i  $\omega_2 \in P$ 
  if  $\omega_1 \simeq_T \omega_2$  and  $\forall i. \text{behavior.take } i \omega_1 \in P$  and  $P \in \text{trace.stuttering.closed}$ 
  for  $\omega_1 \omega_2 i$  and  $P :: ('a, 's, 'v) \text{trace.t set}$ 
  using that(2-)
  by – (rule behavior.stuttering.equiv.takeE[OF sym[OF ‹ $\omega_1 \simeq_T \omega_2$ ›]], where  $i=i$ );
    fastforce intro: trace.stuttering.closed-in)
  show ?thesis
  by (fastforce simp: raw.from-spec-def elim: behavior.stuttering.clE *)
qed

lemma safety-cl:
  assumes  $P \in \text{behavior.stuttering.closed}$ 
  shows raw.safety.cl  $P \in \text{behavior.stuttering.closed}$ 
unfolding raw.safety.cl-def using assms

```

by (*blast intro: subsetD[OF behavior.stuttering.closed.from-spec]*
subsetD[OF trace.stuttering.closed.to-spec])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path tls} \rangle$

lift-definition *to-spec* :: $(\text{'a}, \text{'s}, \text{'v}) \text{ tls} \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ spec}$ **is** *raw.to-spec*
using *raw.spec.closed.to-spec* **by** *blast*

lift-definition *from-spec* :: $(\text{'a}, \text{'s}, \text{'v}) \text{ spec} \Rightarrow (\text{'a}, \text{'s}, \text{'v}) \text{ tls}$ **is** *raw.from-spec*
by (*meson image-subset-iff behavior.stuttering.closed.from-spec raw.spec.closed.stuttering-closed*)

interpretation *safety: galois.complete-lattice-class tls.to-spec tls.from-spec*
by *standard (transfer; simp add: raw.safety.galois)*

setup $\langle \text{Sign.mandatory-path from-spec} \rangle$

lemma *singleton:*

notes *spec.singleton.transfer[transfer-rule]*

shows *tls.from-spec (spec.singleton σ)*

$= \bigsqcup (\text{tls.singleton } \{ \omega . \forall i. \text{behavior.take } i \omega \in \text{Safety-Logic.raw.singleton } \sigma \})$

by *transfer (simp add: behavior.stuttering.cl-bot raw.from-spec.singleton)*

lemmas *bot = raw.from-spec.empty[transferred]*

lemma *sup:*

shows *tls.from-spec (P \sqcup Q) = tls.from-spec P \sqcup tls.from-spec Q*

by *transfer (rule raw.from-spec.sup)*

lemmas *Inf = tls.safety.upper-Inf*

lemmas *inf = tls.safety.upper-inf*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path to-spec} \rangle$

lemma *singleton:*

notes *spec.singleton.transfer[transfer-rule]*

shows *tls.to-spec (tls.singleton ω) = ($\bigsqcup i. \text{spec.singleton (behavior.take } i \omega)$)*

by *transfer (simp add: raw.to-spec.singleton)*

lemmas *bot = tls.safety.lower-bot*

lemmas *Sup = tls.safety.lower-Sup*

lemmas *sup = tls.safety.lower-sup*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path safety} \rangle$

setup $\langle \text{Sign.mandatory-path cl} \rangle$

lemma *transfer[transfer-rule]:*

shows *rel-fun (pcr-tls (=) (=) (=)) (pcr-tls (=) (=) (=)) raw.safety.cl tls.safety.cl*

unfolding *raw.safety.cl-def tls.safety.cl-def* **by** *transfer-prover*

lemma *bot[iff]:*

shows $tls.safety.cl \perp = \perp$
by (*simp add: tls.safety.cl-def tls.from-spec.bot tls.safety.lower-bot*)

lemma *sup*:

shows $tls.safety.cl (P \sqcup Q) = tls.safety.cl P \sqcup tls.safety.cl Q$
by (*simp add: tls.safety.cl-def tls.from-spec.sup tls.to-spec.sup*)

lemmas *state-prop* = *raw.safety.cl.state-prop*[*transferred*]

lemmas *always-le* = *raw.safety.cl.always-le*[*transferred*]

lemma *eventually*: — all the infinite traces and any finite ones that satisfy $\diamond P$

assumes $P \neq \perp$

shows $tls.safety.cl (\diamond P) = -\diamond tls.terminated \sqcup \diamond P$

using *assms* **by** *transfer* (*rule raw.safety.cl.eventually*)

lemma *terminated-iff*:

assumes $\langle \omega \rangle_T \leq tls.terminated$

shows $\langle \omega \rangle_T \leq tls.safety.cl P \longleftrightarrow \langle \omega \rangle_T \leq P$ (**is** *?lhs* \longleftrightarrow *?rhs*)

using *assms*

by *transfer*

(*simp add: raw.singleton-def behavior.stuttering.least-conv raw.safety.cl.terminated-iff
behavior.stuttering.closed.safety-cl behavior.stuttering.closed.raw.terminated*)

lemma *terminated*:

shows $tls.safety.cl tls.terminated = tls.idle \sqcup tls.terminated$

by *transfer* (*simp add: raw.safety.cl.terminated*)

lemma *not-terminated*:

shows $tls.safety.cl (- tls.terminated) = - tls.terminated$ (**is** *?lhs* = *?rhs*)

proof —

have *?lhs* = $tls.safety.cl (\diamond (- tls.terminated))$

by (*simp flip: tls.always.neg tls.terminated.eq-always-terminated*)

also have $\dots = -\diamond tls.terminated \sqcup \diamond (- tls.terminated)$

by (*metis tls.safety.cl.eventually tls.terminated.not-top*

boolean-algebra.compl-zero boolean-algebra-class.boolean-algebra.double-compl)

also have $\dots = ?rhs$

by (*simp add: sup.absorb2 tls.eventually.expansive*

flip: tls.always.neg tls.terminated.eq-always-terminated)

finally show *?thesis* .

qed

lemma *le-terminated-conv*:

shows $tls.safety.cl P \leq tls.terminated \longleftrightarrow P = \perp$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof(*rule iffI*)

show *?lhs* \implies *?rhs*

by *transfer* (*rule raw.safety.cl.le-terminated-bot*)

show *?rhs* \implies *?lhs*

by *simp*

qed

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path closed \rangle$

lemma *transfer*[*transfer-rule*]:

shows *rel-set* (*pcr-tls* (=) (=) (=))

(*behavior.stuttering.closed* \cap *raw.safety.closed*)

tls.safety.closed (**is** *rel-set* - *?lhs* *?rhs*)

proof(*rule rel-setI*)

fix X **assume** $X \in ?lhs$ **then show** $\exists Y \in ?rhs. pcr-tls (=) (=) (=) X Y$

by (*metis (no-types, opaque-lifting) raw.safety.cl-def raw.safety.closed-conv tls.safety.closed-upper
tls.from-spec.rep-eq TLS-inverse cr-tls-def tls.pcr-cr-eq tls.to-spec.rep-eq Int-iff*)

next

fix Y **assume** $Y \in ?rhs$ **then show** $\exists X \in ?lhs. pcr-tls (=) (=) (=) X Y$

by (*metis tls.safety.cl-def tls.safety.closed-conv tls.from-spec.rep-eq
tls.pcr-cr-eq cr-tls-def unTLS raw.safety.closed-upper Int-iff*)

qed

lemma *bot*:

shows $\perp \in tls.safety.closed$

by (*simp add: tls.safety.closed-clI*)

lemma *sup*:

assumes $P \in tls.safety.closed$

assumes $Q \in tls.safety.closed$

shows $P \sqcup Q \in tls.safety.closed$

by (*simp add: assms tls.safety.closed-clI tls.safety.cl.sup flip: tls.safety.closed-conv*)

lemmas *inf = tls.safety.closed-inf*

lemma *boolean-implication*:

assumes $\neg P \in tls.safety.closed$

assumes $Q \in tls.safety.closed$

shows $P \longrightarrow_B Q \in tls.safety.closed$

by (*simp add: assms boolean-implication.conv-sup tls.safety.closed.sup*)

lemma *state-prop*:

shows *tls.state-prop* $P \in tls.safety.closed$

by (*simp add: tls.safety.closed-clI tls.safety.cl.state-prop*)

lemma *not-terminated*:

shows $\neg tls.terminated \in tls.safety.closed$

by (*simp add: tls.safety.closed-clI tls.safety.cl.not-terminated*)

lemma *unless*:

assumes $P \in tls.safety.closed$

assumes $Q \in tls.safety.closed$

shows *tls.unless* $P Q \in tls.safety.closed$

using *assms* **by** *transfer (blast intro: raw.safety.closed.unless)*

lemma *always*:

assumes $P \in tls.safety.closed$

shows *tls.always* $P \in tls.safety.closed$

by (*simp add: assms tls.always-alt-def tls.safety.closed.bot tls.safety.closed.unless*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path cl \rangle$

lemma *until-unless-le*:

assumes $P \in tls.safety.closed$

assumes $Q \in tls.safety.closed$

shows *tls.safety.cl* (*tls.until* $P Q$) \leq *tls.unless* $P Q$

by (*simp add: order.trans[OF tls.safety.cl-inf-le] tls.until.eq-unless-inf-eventually
flip: tls.safety.closed-conv[OF tls.safety.closed.unless[OF assms]]*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path singleton} \rangle$

lemma *to-spec-le-conv*[*tls.singleton.le-conv*]:

notes *spec.singleton.transfer*[*transfer-rule*]

shows $\langle \sigma \rangle \leq \text{tls.to-spec } P \iff (\exists \omega \ i. \langle \omega \rangle_T \leq P \wedge \sigma = \text{behavior.take } i \ \omega)$

by *transfer*

(*simp add: TLS.raw.singleton-def behavior.stuttering.least-conv Safety-Logic.raw.singleton-def raw.spec.least-conv*[*OF subsetD*[*OF raw.spec.closed.to-spec*]];
fastforce simp: raw.to-spec-def)

lemma *from-spec-le-conv*[*tls.singleton.le-conv*]:

notes *spec.singleton.transfer*[*transfer-rule*]

shows $\langle \omega \rangle_T \leq \text{tls.from-spec } P \iff (\forall i. \langle \text{behavior.take } i \ \omega \rangle \leq P)$

by *transfer*

(*simp add: TLS.raw.singleton-def Safety-Logic.raw.singleton-def raw.spec.least-conv behavior.stuttering.least-conv subsetD*[*OF behavior.stuttering.closed.from-spec imageI*[*OF raw.spec.closed.stuttering-closed*]];
simp add: raw.from-spec-def)

lemma *safety-cl-le-conv*[*tls.singleton.le-conv*]:

shows $\langle \omega \rangle_T \leq \text{tls.safety.cl } P \iff (\forall i. \exists \omega'. \langle \omega' \rangle_T \leq P \wedge \text{behavior.take } i \ \omega = \text{behavior.take } i \ \omega')$

by *transfer*

(*simp add: TLS.raw.singleton-def behavior.stuttering.least-conv behavior.stuttering.closed.safety-cl; fastforce intro: raw.safety.cl-altI elim: raw.safety.cl-altE*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

16.7 Maps

setup $\langle \text{Sign.mandatory-path } \text{tls} \rangle$

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('b, 't, 'w) \text{tls}$ **where**

map af sf vf P = $\sqcup (\text{tls.singleton } \langle \text{behavior.map } af \ sf \ vf \ \{ \sigma. \langle \sigma \rangle_T \leq P \})$

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

invmap af sf vf P = $\sqcup (\text{tls.singleton } \langle \text{behavior.map } af \ sf \ vf \ - \ \{ \sigma. \langle \sigma \rangle_T \leq P \})$

abbreviation *amap* :: $('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('b, 's, 'v) \text{tls}$ **where**

amap af $\equiv \text{tls.map } af \ id \ id$

abbreviation *ainvmap* :: $('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

ainvmap af $\equiv \text{tls.invmap } af \ id \ id$

abbreviation *smap* :: $('s \Rightarrow 't) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('a, 't, 'v) \text{tls}$ **where**

smap sf $\equiv \text{tls.map } id \ sf \ id$

abbreviation *sinvmap* :: $('s \Rightarrow 't) \Rightarrow ('a, 't, 'v) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

sinvmap sf $\equiv \text{tls.invmap } id \ sf \ id$

abbreviation *vmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v) \text{tls} \Rightarrow ('a, 's, 'w) \text{tls}$ **where** — aka *liftM*

vmap vf $\equiv \text{tls.map } id \ id \ vf$

abbreviation *vinvmap* :: $('v \Rightarrow 'w) \Rightarrow ('a, 's, 'w) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

vinvmap vf $\equiv \text{tls.invmap } id \ id \ vf$

interpretation *map-invmap: galois.complete-lattice-distributive-class*

tls.map af sf vf

tls.invmap af sf vf for af sf vf

proof *standard*

show *tls.map af sf vf P ≤ Q ↔ P ≤ tls.invmap af sf vf Q (is ?lhs ↔ ?rhs) for P Q*

proof(*rule iffI*)

show *?lhs ⇒ ?rhs*

by (*fastforce simp: tls.map-def tls.invmap-def intro: tls.singleton-le-extI*)

show *?rhs ⇒ ?lhs*

by (*fastforce simp: tls.map-def tls.invmap-def tls.singleton-le-conv*

dest: order.trans[of - P] behavior.stuttering.equiv.map[where af=af and sf=sf and vf=vf]

cong: tls.singleton-cong)

qed

show *tls.invmap af sf vf (⊔ X) ≤ ⊔ (tls.invmap af sf vf ‘ X) for X*

by (*fastforce simp: tls.invmap-def*)

qed

setup *⟨Sign.mandatory-path singleton⟩*

lemma *map-le-conv[$tls.singleton.le-conv$]:*

shows $\langle \omega \rangle_T \leq tls.map\ af\ sf\ vf\ P \longleftrightarrow (\exists \omega'. \langle \omega' \rangle_T \leq P \wedge \langle \omega \rangle_T \leq \langle behavior.map\ af\ sf\ vf\ \omega' \rangle_T)$

by (*simp add: tls.map-def*)

lemma *invmap-le-conv[$tls.singleton.le-conv$]:*

shows $\langle \omega \rangle_T \leq tls.invmap\ af\ sf\ vf\ P \longleftrightarrow \langle behavior.map\ af\ sf\ vf\ \omega \rangle_T \leq P$

by (*simp add: tls.invmap-def tls.singleton-le-conv*)

(*metis behavior.natural.map-natural tls.singleton-eq-conv*)

setup *⟨Sign.parent-path⟩*

setup *⟨Sign.mandatory-path map⟩*

lemmas *bot = tls.map-invmap.lower-bot*

lemmas *monotone = tls.map-invmap.monotone-lower*

lemmas *mono = monotoneD[OF tls.map.monotone]*

lemmas *Sup = tls.map-invmap.lower-Sup*

lemmas *sup = tls.map-invmap.lower-sup*

lemmas *Inf-le = tls.map-invmap.lower-Inf-le* — Converse does not hold

lemmas *inf-le = tls.map-invmap.lower-inf-le* — Converse does not hold

lemmas *invmap-le = tls.map-invmap.lower-upper-contractive*

lemma *singleton:*

shows $tls.map\ af\ sf\ vf\ \langle \omega \rangle_T = \langle behavior.map\ af\ sf\ vf\ \omega \rangle_T$

by (*auto simp: tls.map-def order.eq-iff tls.singleton-le-conv intro: behavior.stuttering.equiv.map*)

lemma *top:*

assumes *surj af*

assumes *surj sf*

assumes *surj vf*

shows $tls.map\ af\ sf\ vf\ \top = \top$

by (*rule antisym*)

(*auto simp: assms tls.singleton.top tls.map.Sup tls.map.singleton surj-f-inv-f*

intro: exI[where x=behavior.map (inv af) (inv sf) (inv vf) σ for σ])

lemma *id*:

shows $tls.map\ id\ id\ id\ P = P$

and $tls.map\ (\lambda x. x)\ (\lambda x. x)\ (\lambda x. x)\ P = P$

by (*simp-all add: tls.map-def flip: id-def*)

lemma *comp*:

shows $tls.map\ af\ sf\ vf\ \circ\ tls.map\ ag\ sg\ vg = tls.map\ (af\ \circ\ ag)\ (sf\ \circ\ sg)\ (vf\ \circ\ vg)$ (**is** *?lhs = ?rhs*)

and $tls.map\ af\ sf\ vf\ (tls.map\ ag\ sg\ vg\ P) = tls.map\ (\lambda a. af\ (ag\ a))\ (\lambda s. sf\ (sg\ s))\ (\lambda v. vf\ (vg\ v))\ P$ (**is** *?thesis1*)

proof –

have $?lhs\ P = ?rhs\ P$ **for** P

by (*rule tls.singleton.exhaust[where x=P]*)

(*simp add: tls.map.Sup tls.map.singleton map-prod.comp image-image comp-def*)

then show $?lhs = ?rhs$ **and** *?thesis1* **by** (*simp-all add: comp-def*)

qed

lemmas $map = tls.map.comp$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ invmap \rangle$

lemmas $bot = tls.map-invmap.upper-bot$

lemmas $top = tls.map-invmap.upper-top$

lemmas $monotone = tls.map-invmap.monotone-upper$

lemmas $mono = monotoneD[OF\ tls.invmap.monotone]$

lemmas $Sup = tls.map-invmap.upper-Sup$

lemmas $sup = tls.map-invmap.upper-sup$

lemmas $Inf = tls.map-invmap.upper-Inf$

lemmas $inf = tls.map-invmap.upper-inf$

lemma *singleton*:

shows $tls.invmap\ af\ sf\ vf\ \langle \omega \rangle_T = \bigsqcup (tls.singleton\ \langle \omega' \rangle_{behavior.map\ af\ sf\ vf\ \omega'})_T \leq \langle \omega \rangle_T$

by (*simp add: tls.invmap-def*)

lemma *id*:

shows $tls.invmap\ id\ id\ id\ P = P$

and $tls.invmap\ (\lambda x. x)\ (\lambda x. x)\ (\lambda x. x)\ P = P$

unfolding *id-def[symmetric]* **by** (*metis tls.map.id(1) tls.map-invmap.lower-upper-lower(2)+*)

lemma *comp*:

shows $tls.invmap\ af\ sf\ vf\ (tls.invmap\ ag\ sg\ vg\ P) = tls.invmap\ (\lambda x. ag\ (af\ x))\ (\lambda s. sg\ (sf\ s))\ (\lambda v. vg\ (vf\ v))\ P$
(**is** *?lhs P = ?rhs P*)

and $tls.invmap\ af\ sf\ vf\ \circ\ tls.invmap\ ag\ sg\ vg = tls.invmap\ (ag\ \circ\ af)\ (sg\ \circ\ sf)\ (vg\ \circ\ vf)$ (**is** *?thesis1*)

proof –

show $?lhs\ P = ?rhs\ P$ **for** P

by (*auto intro: tls.singleton.antisym tls.singleton-le-extI simp: tls.singleton.le-conv*)

then show *?thesis1*

by (*simp add: fun-eq-iff comp-def*)

qed

lemmas $invmap = tls.invmap.comp$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ to-spec \rangle$

lemma *map*:

shows $tls.to-spec (tls.map\ af\ sf\ vf\ P) = spec.map\ af\ sf\ vf (tls.to-spec\ P)$

by (*rule* $tls.singleton.exhaust[of\ P]$)

(*simp* *add*: $tls.map.Sup\ tls.map.singleton\ spec.map.Sup\ spec.map.singleton\ image-image\ tls.to-spec.singleton\ tls.to-spec.Sup\ behavior.take.map$)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

16.8 Abadi's axioms for TLA

The axioms for “propositional” TLA due to [Abadi \(1990\)](#) hold in this model. These are complete for *tls.always* and *tls.eventually*.

Observations:

- Abadi says that the temporal system is D aka S4.3Dum; see [Goldblatt \(1992, §8\)](#)
 - the only interesting axiom here is 5: the discrete-time Dummett axiom
- “propositional” means that actions are treated separately; we omit this part as we don't have actions ala TLA

setup $\langle Sign.mandatory-path\ tls.Abadi \rangle$

lemma *Ax1*:

shows $\models \Box(P \longrightarrow_B Q) \longrightarrow_B \Box P \longrightarrow_B \Box Q$

by (*simp* *add*: $tls.valid-def\ boolean-implication.shunt-top\ tls.always.always-imp-le$)

lemma *Ax2*:

shows $\models \Box P \longrightarrow_B P$

by (*simp* *add*: $tls.valid-def\ boolean-implication.shunt-top\ tls.always.contractive$)

lemma *Ax3*:

shows $\models \Box P \longrightarrow_B \Box \Box P$

by (*simp* *add*: $tls.validI$)

lemma *Ax4*:

— “a classical way to express that time is linear – that any two instants in the future are ordered” [Warford et al. \(2020, \(254\) Lemmon formula\)](#)

shows $\models \Box(\Box P \longrightarrow_B Q) \sqcup \Box(\Box Q \longrightarrow_B P)$

proof –

have $\models (-\Box P) \mathcal{W} \Box Q \sqcup (-\Box Q) \mathcal{W} \Box P$ **by** (*rule* $tls.unless.ordering$)

also have $\dots \leq \Box((-\Box P) \mathcal{W} \Box Q) \sqcup \Box((-\Box Q) \mathcal{W} \Box P)$

by (*metis* *sup-mono* $tls.always.idempotent\ tls.unless.alwaysR-le$)

also have $\dots \leq \Box(-\Box P \sqcup Q) \sqcup \Box(-\Box Q \sqcup P)$

by (*strengthen* *ord-to-strengthen*(1)[*OF* $tls.unless.sup-le$])

(*meson* *order.refl* *sup-mono* $tls.always.contractive\ tls.always.mono$)

also have $\dots = \Box(\Box P \longrightarrow_B Q) \sqcup \Box(\Box Q \longrightarrow_B P)$

by (*simp* *add*: $boolean-implication.conv-sup$)

finally show *?thesis* .

qed

lemma *Ax5*:

— “expresses the discreteness of time” See also [Warford et al. \(2020, §4.1 “the Dummett formula”\)](#): for them “next” encodes discreteness

fixes $P :: ('a, 's, 'v)\ tls$

shows $\models \Box(\Box(P \longrightarrow_B \Box P) \longrightarrow_B P) \longrightarrow_B \Diamond\Box P \longrightarrow_B P$ (is \models ?goal)

proof –

have *raw-Ax5*: *raw.always* (*raw.eventually* ($P \cap$ *raw.eventually* ($-P$)) \cup P)
 \cap *raw.eventually* (*raw.always* P)
 $\subseteq P$ (is ?lhs \subseteq ?rhs)

for $P :: ('a, 's, 'v)$ *behavior.t set*

proof(*rule subsetI*)

fix ω **assume** $\omega \in ?lhs$

from *IntD2*[*OF* $\langle \omega \in ?lhs \rangle$]

obtain i

where $\exists \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \wedge \omega' \in \text{raw.always } P$

by (*force simp*: *raw.always-alt-def* *raw.eventually-alt-def*)

then obtain i

where $i: \exists \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \wedge \omega' \in \text{raw.always } P$

and $\forall j < i. \forall \omega'. \text{behavior.dropn } j \ \omega = \text{Some } \omega' \longrightarrow \omega' \notin \text{raw.always } P$

using *ex-has-least-nat*[**where** $k=i$ **and** $P=\lambda i. \exists \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \wedge \omega' \in \text{raw.always } P$ **and** $m=id$]

by (*auto dest*: *leD*)

have $\exists \omega'. \text{behavior.dropn } (i - j) \ \omega = \text{Some } \omega' \wedge \omega' \in \text{raw.always } P$ **for** j

proof(*induct j*)

case (*Suc j*) **show** ?case

proof(*cases j < i*)

case *True* **show** ?thesis

proof(*rule ccontr*)

assume $\nexists \omega'. \text{behavior.dropn } (i - \text{Suc } j) \ \omega = \text{Some } \omega' \wedge \omega' \in \text{raw.always } P$

with $\langle \exists \omega'. \text{behavior.dropn } i \ \omega = \text{Some } \omega' \wedge \omega' \in \text{raw.always } P \rangle$

have $\exists \omega'. \text{behavior.dropn } (i - \text{Suc } j) \ \omega = \text{Some } \omega' \wedge \omega' \notin \text{raw.always } P$

using *behavior.dropn.shorterD*[*OF* - *diff-le-self*] **by** *blast*

then obtain k **where** $\exists \omega'. \text{behavior.dropn } (i - \text{Suc } j + k) \ \omega = \text{Some } \omega' \wedge \omega' \notin P$

by (*clarsimp simp*: *raw.always-alt-def* *behavior.dropn.add* *behavior.dropn.Suc*) *blast*

with *Suc.hyps* $\langle j < i \rangle$

have $\exists \omega'. \text{behavior.dropn } (i - \text{Suc } j) \ \omega = \text{Some } \omega' \wedge \omega' \notin P$

by (*fastforce simp*: *raw.always-alt-def* *behavior.dropn.add*
split: *nat-diff-split-asm*
dest: *spec*[**where** $x=k - 1$])

with $\langle j < i \rangle$ *IntD1*[*OF* $\langle \omega \in ?lhs \rangle$]

obtain $m \ n$ **where** $\exists \omega' \ \omega'' \ \omega'''. \text{behavior.dropn } (i - \text{Suc } j) \ \omega = \text{Some } \omega' \wedge \omega' \notin P$
 $\wedge \text{behavior.dropn } m \ \omega'' = \text{Some } \omega'' \wedge \omega'' \in P$
 $\wedge \text{behavior.dropn } n \ \omega''' = \text{Some } \omega''' \wedge \omega''' \notin P$

by (*simp add*: *raw.always-alt-def* *raw.eventually-alt-def*)
(*blast dest*: *spec*[**where** $x=i - \text{Suc } j$])

with $\langle j < i \rangle$ *Suc.hyps*

show *False*

by (*clarsimp simp*: *raw.always-alt-def* *dest!*: *spec*[**where** $x=m + n - 1$] *split*: *nat-diff-split-asm*)
(*metis* *behavior.dropn.Suc* *behavior.dropn.bind-tl-commute* *behavior.dropn.dropn* *bind*.*bind-lunit*)

qed

qed (*use* *Suc.hyps* **in** *simp*)

qed (*use* i **in** *simp*)

from *this*[*of* i] **show** $\omega \in P$

by (*fastforce simp*: *raw.always-alt-def* *dest*: *spec*[**where** $x=0$])

qed

show ?thesis

proof(*rule tIs.validI*)

have $\Box(\Diamond(P \sqcap \Diamond(- P)) \sqcup P) \sqcap \Diamond\Box P \leq P$

by (*rule raw-Ax5*[*transferred*])

then have $\Box(\Diamond(P \sqcap \Diamond(- P)) \sqcup P) \sqcap \Diamond\Box P \leq P$

by (*simp add*: *boolean-implication.conv-sup* *tIs.always.neg*)

then show $\top \leq ?goal$

by – (*intro iffD1*[*OF boolean-implication.shunt1*];
simp add: boolean-implication.conv-sup tls.always.neg)

qed
qed

lemma *Ax6*:

assumes $\models P$

shows $\models \Box P$

by (*rule tls.always.always-necessitation*[*OF assms*])

— Ax7: propositional tautologies: given by the *boolean-algebra* instance

lemma *Ax8*:

assumes $\models P$

assumes $\models P \longrightarrow_B Q$

shows $\models Q$

by (*rule tls.valid.rev-mp*[*OF assms*])

setup $\langle \text{Sign.parent-path} \rangle$

16.9 Tweak syntax

unbundle *tls.no-syntax*

no-notation *tls.singleton* ($\langle \dashv \rangle_T$)

setup $\langle \text{Sign.mandatory-path } \text{tls} \rangle$

bundle *extra-syntax*

begin

notation *tls.singleton* ($\langle \dashv \rangle_T$ [*0*])

notation *tls.from-spec* ($\langle \dashv \rangle$ [*0*])

end

setup $\langle \text{Sign.parent-path} \rangle$

17 Atomic sections

By restricting the environment to stuttering steps we can consider arbitrary processes to be atomic, i.e., free of interference.

setup $\langle \text{Sign.mandatory-path } \text{spec} \rangle$

definition *atomic* :: $'a \Rightarrow ('a, 's, 'v) \text{ spec} \Rightarrow ('a, 's, 'v) \text{ spec}$ **where**

atomic a P = P \sqcap spec.rel ($\{a\} \times \text{UNIV}$)

setup $\langle \text{Sign.mandatory-path } \text{idle} \rangle$

lemma *atomic-le-conv*[*spec.idle-le*]:

shows *spec.idle* \leq *spec.atomic a P* \longleftrightarrow *spec.idle* \leq *P*

by (*simp add: spec.atomic-def spec.idle.rel-le*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path } \text{term} \rangle$

setup $\langle \text{Sign.mandatory-path } \text{none} \rangle$

lemma *atomic*:

shows $\text{spec.term.none} (\text{spec.atomic } a \ P) = \text{spec.atomic } a (\text{spec.term.none } P)$
by (*simp add: spec.atomic-def spec.term.none.inf spec.term.none.inf-rel*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path all} \rangle$

lemma *atomic*:

shows $\text{spec.term.all} (\text{spec.atomic } a \ P) = \text{spec.atomic } a (\text{spec.term.all } P)$
by (*simp add: spec.atomic-def spec.term.all.inf spec.term.all.rel*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path atomic} \rangle$

lemma *bot[simp]*:

shows $\text{spec.atomic } a \ \perp = \perp$
by (*simp add: spec.atomic-def*)

lemma *top[simp]*:

shows $\text{spec.atomic } a \ \top = \text{spec.rel} (\{a\} \times \text{UNIV})$
by (*simp add: spec.atomic-def*)

lemma *contractive*:

shows $\text{spec.atomic } a \ P \leq P$
by (*simp add: spec.atomic-def*)

lemma *idempotent[simp]*:

shows $\text{spec.atomic } a (\text{spec.atomic } a \ P) = \text{spec.atomic } a \ P$
by (*simp add: spec.atomic-def*)

lemma *monotone*:

shows *mono* ($\text{spec.atomic } a$)
by (*simp add: spec.atomic-def monoI le-infI1*)

lemmas *strengthen[strg]* = *st-monotone*[*OF spec.atomic.monotone*]

lemmas *mono* = *monotoneD*[*OF spec.atomic.monotone*]

lemmas *mono2mono*[*cont-intro, partial-function-mono*]
= *monotone2monotone*[*OF spec.atomic.monotone, simplified, of orda P for orda P*]

lemma *Sup*:

shows $\text{spec.atomic } a (\bigsqcup X) = \bigsqcup (\text{spec.atomic } a \ ' X)$
by (*simp add: spec.atomic-def ac-simps heyting.inf-Sup-distrib*)

lemmas *sup* = *spec.atomic.Sup*[**where** $X = \{P, Q\}$ **for** $P \ Q$, *simplified*]

lemma *mcont2mcont*[*cont-intro*]:

assumes *mcont luba orda Sup* (\leq) P
shows *mcont luba orda Sup* (\leq) ($\lambda x. \text{spec.atomic } a (P \ x)$)
by (*simp add: spec.atomic-def assms*)

lemma *Inf-not-empty*:

assumes $X \neq \{\}$
shows $\text{spec.atomic } a (\bigsqcap X) = \bigsqcap (\text{spec.atomic } a \ ' X)$
using *assms* **by** (*simp add: spec.atomic-def INF-inf-const2*)

lemmas *inf = spec.atomic.Inf-not-empty*[**where** $X=\{P, Q\}$ **for** $P Q$, *simplified*]

lemma *idle*:

shows *spec.atomic a spec.idle = spec.idle*

by (*simp add: spec.atomic-def inf.absorb1 spec.idle.rel-le*)

lemma *action*:

shows *spec.atomic a (spec.action F) = spec.action (F \cap UNIV \times ($\{a\} \times$ UNIV \cup UNIV \times Id))*

by (*simp add: spec.atomic-def spec.action.inf-rel-reflcl*)

lemma *return*:

shows *spec.atomic a (spec.return v) = spec.return v*

by (*simp add: spec.return-def spec.atomic.action Times-Int-Times*)

lemma *bind*:

shows *spec.atomic a (f \gg g) = spec.atomic a f \gg (λv . spec.atomic a (g v))*

by (*simp add: spec.atomic-def spec.bind.inf-rel ac-simps*)

lemma *map-le*:

fixes *af :: 'a \Rightarrow 'b*

shows *spec.map af sf vf (spec.atomic a P) \leq spec.atomic (af a) (spec.map af sf vf P)*

by (*auto simp: spec.atomic-def spec.map.inf-rel*

intro!: spec.map.mono inf.mono order.refl spec.rel.mono)

lemma *invmap*:

fixes *af :: 'a \Rightarrow 'b*

shows *spec.atomic a (spec.invmap af sf vf P) \leq spec.invmap af sf vf (spec.atomic (af a) P)*

by (*auto simp: spec.atomic-def spec.invmap.inf spec.invmap.rel*

intro!: le-infI2 spec.rel.mono)

lemma *rel*:

shows *spec.atomic a (spec.rel r) = spec.rel (r \cap $\{a\} \times$ UNIV)*

by (*simp add: spec.atomic-def flip: spec.rel.inf*)

lemma *interference*:

shows *spec.atomic (proc a) (spec.rel ($\{env\} \times$ UNIV)) = spec.rel $\{\}$*

by (*simp add: spec.atomic.rel flip: Times-Int-distrib1*)

setup \langle *Sign.mandatory-path cam* \rangle

lemma *cl*:

shows *spec.atomic (proc a) (spec.cam.cl ($\{env\} \times$ UNIV) P) = spec.atomic (proc a) P*

by (*simp add: spec.cam.cl-def spec.atomic.sup spec.atomic.bind spec.atomic.interference*

spec.rel.empty spec.term.none.bind spec.term.none.Sup spec.term.none.return

image-image spec.bind.botR spec.bind.idleR sup-iff-le

flip: spec.term.none.atomic spec.term.all.atomic)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path interference* \rangle

lemma *cl*:

shows *spec.atomic (proc a) (spec.interference.cl ($\{env\} \times$ UNIV) P) = spec.return () \gg spec.atomic (proc a)*

P

by (*simp add: spec.interference.cl-def UNIV-unit spec.atomic.bind spec.atomic.interference*

spec.rel.empty spec.atomic.cam.cl spec.bind.return spec.atomic.return)

setup \langle *Sign.parent-path* \rangle

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

lift-definition $\text{atomic} :: ('s, 'v) \text{ prog} \Rightarrow ('s, 'v) \text{ prog is}$
 $\lambda P. \text{spec.interference.cl} (\{\text{env}\} \times \text{UNIV}) (\text{spec.atomic self } P) ..$

setup $\langle \text{Sign.mandatory-path atomic} \rangle$

lemma bot[simp] :
shows $\text{prog.atomic} \perp = \perp$

by transfer
($\text{simp add: spec.interference.cl.bot spec.atomic.interference spec.interference.cl.rel}$
 $\text{flip: spec.term.none.atomic spec.term.none.interference.cl}$)

lemma contractive :
shows $\text{prog.atomic} P \leq P$
by transfer ($\text{simp add: spec.atomic.contractive spec.interference.least}$)

lemma idempotent[simp] :
shows $\text{prog.atomic} (\text{prog.atomic } P) = \text{prog.atomic } P$
by transfer ($\text{metis spec.atomic.idempotent spec.atomic.interference.cl spec.interference.closed-conv}$)

lemma monotone :
shows mono prog.atomic
by (rule monoI) ($\text{transfer; simp add: spec.atomic.mono spec.interference.mono-cl}$)

lemmas $\text{strengthen[strg]} = \text{st-monotone[OF prog.atomic.monotone]}$
lemmas $\text{mono} = \text{monotoneD[OF prog.atomic.monotone]}$
lemmas $\text{mono2mono[cont-intro, partial-function-mono]} = \text{monotone2monotone[OF prog.atomic.monotone, simplified, of orda } P \text{ for orda } P]$

lemma Sup :
shows $\text{prog.atomic} (\bigsqcup X) = \bigsqcup (\text{prog.atomic } ' X)$
by transfer
($\text{simp add: spec.atomic.Sup spec.atomic.sup spec.interference.cl-Sup spec.interference.cl-sup}$
 $\text{image-image spec.interference.cl.bot spec.atomic.interference spec.interference.cl.rel}$
 $\text{flip: spec.term.none.atomic spec.term.none.interference.cl}$)

lemmas $\text{sup} = \text{prog.atomic.Sup}[\text{where } X=\{P, Q\} \text{ for } P Q, \text{ simplified}]$

lemma mcont :
shows $\text{mcont Sup} (\leq) \text{Sup} (\leq) \text{prog.atomic}$
by ($\text{simp add: mcontI contI prog.atomic.Sup}$)

lemmas $\text{mcont2mcont[cont-intro]} = \text{mcont2mcont[OF prog.atomic.mcont, of luba orda } P \text{ for luba orda } P]$

lemma Inf-le :
shows $\text{prog.atomic} (\bigsqcap X) \leq \bigsqcap (\text{prog.atomic } ' X)$
by transfer ($\text{simp add: Inf-lower le-INF-iff spec.atomic.mono spec.interference.mono-cl}$)

lemmas $\text{inf-le} = \text{prog.atomic.Inf-le}[\text{where } X=\{P, Q\} \text{ for } P Q, \text{ simplified}]$

lemma action :
shows $\text{prog.atomic} (\text{prog.action } F) = \text{prog.action } F$

by *transfer*

(*simp add: spec.atomic.interference.cl spec.atomic.action spec.bind.returnL spec.idle.action-le;*
rule arg-cong; blast)

lemma *return:*

shows *prog.atomic (prog.return v) = prog.return v*

by (*simp add: prog.return-def prog.atomic.action*)

lemma *bind-le:*

shows *prog.atomic (f \ggg g) \leq prog.atomic f \ggg ($\lambda v.$ prog.atomic (g v))*

by *transfer*

(*simp add: spec.atomic.bind spec.bind.mono*
spec.interference.closed.bind spec.interference.expansive spec.interference.least)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path p2s* \rangle

lemmas *atomic = prog.atomic.rep-eq*

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.parent-path* \rangle

17.1 Inhabitation

setup \langle *Sign.mandatory-path inhabits.spec* \rangle

lemma *atomic:*

assumes $P -s, xs \rightarrow P'$

assumes $trace.steps' s xs \subseteq \{a\} \times UNIV$

shows $spec.atomic a P -s, xs \rightarrow spec.atomic a P'$

unfolding *spec.atomic-def* **by** (*rule inhabits.inf[OF assms(1) inhabits.spec.rel.rel[OF assms(2)]]*)

lemma *atomic-term:*

assumes $P -s, xs \rightarrow spec.return v$

assumes $trace.steps' s xs \subseteq \{a\} \times UNIV$

shows $spec.atomic a P -s, xs \rightarrow spec.return v$

by (*rule inhabits.spec.atomic[where P'=spec.return v, simplified spec.atomic.return, OF assms]*)

lemma *atomic-diverge:*

assumes $P -s, xs \rightarrow \perp$

assumes $trace.steps' s xs \subseteq \{a\} \times UNIV$

shows $spec.atomic a P -s, xs \rightarrow \perp$

by (*rule inhabits.spec.atomic[where P'= \perp , simplified spec.atomic.bot, OF assms]*)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path inhabits.prog* \rangle

lemma *atomic-term:*

assumes $prog.p2s P -s, xs \rightarrow spec.return v$

assumes $trace.steps' s xs \subseteq \{self\} \times UNIV$

shows $prog.p2s (prog.atomic P) -s, xs \rightarrow spec.return v$

unfolding *prog.p2s.atomic*

by (*rule inhabits.mono[OF spec.interference.expansive order.refl*
inhabits.spec.atomic-term[OF assms]])

lemma *atomic-diverge*:
assumes *prog.p2s* $P -s, xs \rightarrow \perp$
assumes *trace.steps'* $s \ xs \subseteq \{\text{self}\} \times UNIV$
shows *prog.p2s* (*prog.atomic* P) $-s, xs \rightarrow \perp$
unfolding *prog.p2s.atomic*
by (*rule inhabits.mono*[*OF spec.interference.expansive order.refl*
inhabits.spec.atomic-diverge[*OF assms*]])

setup $\langle \text{Sign.parent-path} \rangle$

17.2 Assume/guarantee

setup $\langle \text{Sign.mandatory-path ag.prog} \rangle$

lemma *atomic*:

assumes *prog.p2s* $c \leq \{\!\{P\}\!\}, Id \vdash G, \{\!\{Q\}\!\}$
assumes *P*: *stable* $A \ P$
assumes *Q*: $\bigwedge v. \text{stable } A \ (Q \ v)$
shows *prog.p2s* (*prog.atomic* c) $\leq \{\!\{P\}\!\}, A \vdash G, \{\!\{Q\}\!\}$
apply (*subst ag.assm-heyting*[**where** $A=A$ **and** $r=A$, *simplified, symmetric*])
apply (*simp add: prog.p2s.atomic*)
apply (*strengthen ord-to-strengthen*[*OF assms(1)*])
apply (*simp add: spec.atomic-def heyting ac-simps spec.interference.cl.inf-rel inf-sup-distrib Times-Int-Times*
flip: spec.rel.inf)
using *assms*
apply (*force intro: order.trans*[*OF - spec.interference.cl-ag-le*[**where** $A=A$ **and** $r=A$, *simplified*]]
spec.interference.cl.mono[*OF order.refl*] *ag.pre-a*
simp add: heyting[*symmetric*] *ag.assm-heyting*[**where** $r=\{\}$, *simplified*])
done

setup $\langle \text{Sign.parent-path} \rangle$

18 Exceptions

A sketch of how we might handle exceptions in this framework.

setup $\langle \text{Sign.mandatory-path raw} \rangle$

type-synonym $(\ 's, \ 'x, \ 'v) \text{ exn} = (\ 's, \ 'x + \ 'v) \text{ prog}$

definition *action* $:: (\ 'v \times \ 's \times \ 's) \text{ set} \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}$ **where**
action = *prog.action* \circ *image* (*map-prod* *Inr id*)

definition *return* $:: \ 'v \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}$ **where**
return = *prog.return* \circ *Inr*

definition *throw* $:: \ 'x \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}$ **where**
throw = *prog.return* \circ *Inl*

definition *catch* $:: (\ 's, \ 'x, \ 'v) \text{ raw.exn} \Rightarrow (\ 'x \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}) \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}$ **where**
catch *f handler* = *f* $\gg=$ *case-sum handler raw.return*

definition *bind* $:: (\ 's, \ 'x, \ 'v) \text{ raw.exn} \Rightarrow (\ 'v \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}) \Rightarrow (\ 's, \ 'x, \ 'v) \text{ raw.exn}$ **where**
bind *f g* = *f* $\gg=$ *case-sum raw.throw g*

definition *parallel* $:: (\ 's, \ 'x, \ \text{unit}) \text{ raw.exn} \Rightarrow (\ 's, \ 'x, \ \text{unit}) \text{ raw.exn} \Rightarrow (\ 's, \ 'x, \ \text{unit}) \text{ raw.exn}$ **where**
parallel $P \ Q = (P \gg= \text{case-sum } \perp \ \text{prog.return} \parallel Q \gg= \text{case-sum } \perp \ \text{prog.return}) \gg= \text{raw.return}$

setup $\langle \text{Sign.mandatory-path bind} \rangle$

lemma *bind*:

shows $\text{raw.bind } (\text{raw.bind } f g) h = \text{raw.bind } f (\lambda x. \text{raw.bind } (g x) h)$

by (*simp add: raw.bind-def prog.bind.bind sum.case-distrib*[**where** $h = \lambda f. f \gg= \text{case-sum raw.throw } h$])
(*simp add: raw.throw-def comp-def prog.bind.return cong: sum.case-cong*)

lemma *return*:

shows *returnL*: $\text{raw.bind } (\text{raw.return } v) = (\lambda g. g v)$

and *returnR*: $\text{raw.bind } f \text{ raw.return} = f$

by (*simp-all add: fun-eq-iff raw.bind-def raw.return-def raw.throw-def prog.bind.return case-sum-Inl-Inr-L*)

lemma *throwL*:

shows $\text{raw.bind } (\text{raw.throw } x) = (\lambda g. \text{raw.throw } x)$

by (*simp add: fun-eq-iff raw.bind-def raw.throw-def prog.bind.return*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path catch} \rangle$

lemma *catch*:

shows $\text{raw.catch } (\text{raw.catch } f \text{ handler}_1) \text{ handler}_2 = \text{raw.catch } f (\lambda x. \text{raw.catch } (\text{handler}_1 x) \text{ handler}_2)$

by (*simp add: raw.catch-def prog.bind.bind sum.case-distrib*[**where** $h = \lambda f. f \gg= \text{case-sum handler}_2 \text{ raw.return}$])
(*simp add: raw.return-def comp-def prog.bind.return cong: sum.case-cong*)

lemma *returnL*:

shows $\text{raw.catch } (\text{raw.return } v) = (\lambda \text{handler. raw.return } v)$

by (*simp add: fun-eq-iff raw.catch-def raw.return-def prog.bind.return*)

lemma *throw*:

shows *throwL*: $\text{raw.catch } (\text{raw.throw } x) = (\lambda g. g x)$

and *throwR*: $\text{raw.catch } f \text{ raw.throw} = f$

by (*simp-all add: fun-eq-iff raw.catch-def raw.return-def raw.throw-def prog.bind.return case-sum-Inl-Inr-L*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path parallel} \rangle$

lemma *commute*:

shows $\text{raw.parallel } P Q = \text{raw.parallel } Q P$

by (*simp add: raw.parallel-def prog.parallel.commute*)

lemma *assoc*:

shows $\text{raw.parallel } P (\text{raw.parallel } Q R) = \text{raw.parallel } (\text{raw.parallel } P Q) R$

by (*simp add: raw.parallel-def raw.return-def prog.bind.bind prog.bind.return prog.parallel.assoc*)

lemma *return*:

shows $\text{raw.parallel } (\text{raw.return } ()) P = \text{raw.catch } P (\lambda x. \perp)$ (**is** *?thesis1*)

and $\text{raw.parallel } P (\text{raw.return } ()) = \text{raw.catch } P (\lambda x. \perp)$ (**is** *?thesis2*)

proof –

show *?thesis1*

by (*simp add: raw.parallel-def raw.return-def*

prog.bind.bind prog.bind.return prog.parallel.return prog.bind.botL

sum.case-distrib[**where** $h = \lambda f. f \gg= \text{prog.return} \circ \text{Inr}$]

flip: raw.catch-def[*unfolded raw.return-def o-def*]

cong: sum.case-cong)

then show *?thesis2*

by (*simp add: raw.parallel.commute*)

qed

lemma *throw*:

shows $\text{raw.parallel } (\text{raw.throw } x) P = \text{raw.bind } (\text{raw.catch } P (\lambda x. \perp)) (\lambda x. \perp)$ **(is ?thesis1)**
and $\text{raw.parallel } P (\text{raw.throw } x) = \text{raw.bind } (\text{raw.catch } P (\lambda x. \perp)) (\lambda x. \perp)$ **(is ?thesis2)**

proof –

show *?thesis1*

by (*simp add: raw.parallel-def raw.throw-def raw.bind-def raw.return-def raw.catch-def*
prog.bind.bind prog.bind.return prog.bind.botL prog.parallel.bot
sum.case-distrib[where h= $\lambda f. \text{prog.bind } f (\lambda x. \perp)$]
sum.case-distrib[where h= $\lambda f. f \gg= \text{case-sum } (\text{prog.return } \circ \text{Inl}) (\lambda x. \perp)$]
cong: sum.case-cong)

then show *?thesis2*

by (*simp add: raw.parallel.commute*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

typedef $(\prime s, \prime x, \prime v) \text{ exn} = \text{UNIV} :: (\prime s, \prime x, \prime v) \text{ raw.exn set}$

by *blast*

setup-lifting *type-definition-exn*

instantiation *exn* :: $(\text{type}, \text{type}, \text{type})$ *complete-distrib-lattice*

begin

lift-definition *bot-exn* :: $(\prime s, \prime x, \prime v) \text{ exn}$ **is** \perp .

lift-definition *top-exn* :: $(\prime s, \prime x, \prime v) \text{ exn}$ **is** \top .

lift-definition *sup-exn* :: $(\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *sup* .

lift-definition *inf-exn* :: $(\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *inf* .

lift-definition *less-eq-exn* :: $(\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn} \Rightarrow \text{bool}$ **is** *less-eq* .

lift-definition *less-exn* :: $(\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn} \Rightarrow \text{bool}$ **is** *less* .

lift-definition *Inf-exn* :: $(\prime s, \prime x, \prime v) \text{ exn set} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *Inf* .

lift-definition *Sup-exn* :: $(\prime s, \prime x, \prime v) \text{ exn set} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *Sup* .

instance **by** *standard* (*transfer; auto intro: Inf-lower InfI le-supI1 SupI SupE Inf-Sup*)**+**

end

setup $\langle \text{Sign.mandatory-path } \text{exn} \rangle$

lift-definition *action* :: $(\prime v \times \prime s \times \prime s) \text{ set} \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *raw.action* .

lift-definition *return* :: $\prime v \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *raw.return* .

lift-definition *throw* :: $\prime x \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *raw.throw* .

lift-definition *catch* :: $(\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime x \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}) \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *raw.catch* .

lift-definition *bind* :: $(\prime s, \prime x, \prime v) \text{ exn} \Rightarrow (\prime v \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}) \Rightarrow (\prime s, \prime x, \prime v) \text{ exn}$ **is** *raw.bind* .

lift-definition *parallel* :: $(\prime s, \prime x, \text{unit}) \text{ exn} \Rightarrow (\prime s, \prime x, \text{unit}) \text{ exn} \Rightarrow (\prime s, \prime x, \text{unit}) \text{ exn}$ **is** *raw.parallel* .

adhoc-overloading

Monad-Syntax.bind $\equiv \text{exn.bind}$

adhoc-overloading

parallel $\equiv \text{exn.parallel}$

setup $\langle \text{Sign.mandatory-path } \text{bind} \rangle$

lemma *bind*:

shows $f \ggg g \ggg h = \text{exn.bind } f (\lambda x. g x \ggg h)$
by *transfer* (rule *raw.bind.bind*)

lemma *return*:

shows *returnL*: $(\ggg) (\text{exn.return } v) = (\lambda g. g v)$ (**is** *?thesis1*)
and *returnR*: $f \ggg \text{exn.return} = f$ (**is** *?thesis2*)
by (*transfer*; rule *raw.bind.return*)+

lemma *throwL*:

shows $(\ggg) (\text{exn.throw } x) = (\lambda g. \text{exn.throw } x)$
by *transfer* (rule *raw.bind.throwL*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path catch} \rangle$

lemma *catch*:

shows $\text{exn.catch } (\text{exn.catch } f \text{ handler}_1) \text{ handler}_2 = \text{exn.catch } f (\lambda x. \text{exn.catch } (\text{handler}_1 x) \text{ handler}_2)$
by *transfer* (rule *raw.catch.catch*)

lemma *returnL*:

shows $\text{exn.catch } (\text{exn.return } v) = (\lambda \text{handler. exn.return } v)$
by *transfer* (rule *raw.catch.returnL*)

lemma *throw*:

shows *throwL*: $\text{exn.catch } (\text{exn.throw } x) = (\lambda g. g x)$
and *throwR*: $\text{exn.catch } f \text{ exn.throw} = f$
by (*transfer*; rule *raw.catch.throw*)+

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path parallel} \rangle$

lemma *commute*:

shows $\text{exn.parallel } P Q = \text{exn.parallel } Q P$
by *transfer* (rule *raw.parallel.commute*)

lemma *assoc*:

shows $\text{exn.parallel } P (\text{exn.parallel } Q R) = \text{exn.parallel } (\text{exn.parallel } P Q) R$
by *transfer* (rule *raw.parallel.assoc*)

lemma *return*:

shows *returnL*: $\text{exn.return } () \parallel P = \text{exn.catch } P \perp$
and *returnR*: $P \parallel \text{exn.return } () = \text{exn.catch } P \perp$
unfolding *bot-fun-def* **by** (*transfer*; rule *raw.parallel.return*)+

lemma *throw*:

shows *throwL*: $\text{exn.throw } x \parallel P = \text{exn.catch } P \perp \ggg \perp$
and *throwR*: $P \parallel \text{exn.throw } x = \text{exn.catch } P \perp \ggg \perp$
unfolding *bot-fun-def* **by** (*transfer*; rule *raw.parallel.throw*)+

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

19 Assume/Guarantee rule sets

The rules in *ConcurrentHOL.Refinement* are deficient in various ways:

- redundant stability requirements
- interleaving of program decomposition with stability goals
- insufficiently instantiated

The following are some experimental rules aimed at practical assume/guarantee reasoning.

19.1 Implicit stabilisation

We can define a relation *ceilr* P to be the largest (weakest assumption) for which P is stable. This always yields a preorder (i.e., it is reflexive and transitive). Later we use this to inline stability side conditions into assume/guarantee rules (§19.1.1).

This relation is not very pleasant to work with: it is not monotonic and does not have many useful algebraic properties. However it suffices to defer the checking of assumes (see §19.1.1).

This is a cognate of the *strongest guarantee* used by de Roever et al. (2001, Definition 8.31) in their completeness proof for the rely-guarantee method.

definition *ceilr* :: 'a pred \Rightarrow 'a rel **where**

$$\textit{ceilr } P = \bigsqcup \{r. \textit{stable } r P\}$$

lemma *ceilr-alt-def*:

$$\textbf{shows } \textit{ceilr } P = \{(s, s'). P s \longrightarrow P s'\}$$

by (*auto simp: ceilr-def stable-def monotone-def le-bool-def order-eq-iff Sup-upper*)

lemma *ceilrE[elim]*:

$$\textbf{assumes } (x, y) \in \textit{ceilr } P$$

$$\textbf{assumes } P x$$

$$\textbf{shows } P y$$

using *assms* **by** (*simp add: ceilr-alt-def*)

setup $\langle \textit{Sign.mandatory-path } \textit{ceilr} \rangle$

named-theorems *simps* $\langle \textit{simp rules for } \textbf{const} \langle \textit{ceilr} \rangle \rangle$

lemma *bot[ceilr.simps]*:

$$\textbf{shows } \textit{ceilr } \perp = \textit{UNIV}$$

by (*simp-all add: ceilr-alt-def*)

lemma *top[ceilr.simps]*:

$$\textbf{shows } \textit{ceilr } \top = \textit{UNIV}$$

by (*simp-all add: ceilr-alt-def*)

lemma *const[ceilr.simps]*:

$$\textbf{shows } \textit{ceilr } \langle c \rangle = \textit{UNIV}$$

$$\textbf{and } \textit{ceilr } (P \wedge \langle c \rangle) = (\textit{if } c \textit{ then } \textit{ceilr } P \textit{ else } \textit{UNIV})$$

$$\textbf{and } \textit{ceilr } (\langle c \rangle \wedge P) = (\textit{if } c \textit{ then } \textit{ceilr } P \textit{ else } \textit{UNIV})$$

$$\textbf{and } \textit{ceilr } (P \wedge \langle c \rangle \wedge P') = (\textit{if } c \textit{ then } \textit{ceilr } (P \wedge P') \textit{ else } \textit{UNIV})$$

by (*simp-all add: ceilr-alt-def*)

lemma *Id-le*:

$$\textbf{shows } \textit{Id} \subseteq \textit{ceilr } P$$

by (*auto simp: ceilr-alt-def*)

lemmas *refl[iff]* = *ceilr.Id-le*[*folded refl-alt-def*]

lemma *trans*[*iff*]:

shows *trans* (*ceilr* *P*)

by (*simp* *add*: *ceilr-alt-def trans-def*)

lemma *stable*[*stable.intro*]:

shows *stable* (*ceilr* *P*) *P*

by (*simp* *add*: *ceilr-def stable.Union-conv*)

lemma *largest*[*stable.intro*]:

assumes *stable* *r P*

shows $r \subseteq \text{ceilr } P$

by (*simp* *add*: *ceilr-def assms Sup-upper*)

lemma *disj-subseteq*: — Converse does not hold

shows $\text{ceilr } (P \vee Q) \subseteq \text{ceilr } P \cup \text{ceilr } Q$

by (*fastforce simp*: *ceilr-alt-def*)

lemma *Ex-subseteq*: — Converse does not hold

shows $\text{ceilr } (\exists x. P x) \subseteq (\bigcup x. \text{ceilr } (P x))$

by (*fastforce simp*: *ceilr-alt-def*)

lemma *conj-subseteq*: — Converse does not hold

shows $\text{ceilr } P \cap \text{ceilr } Q \subseteq \text{ceilr } (P \wedge Q)$

by (*fastforce simp*: *ceilr-alt-def*)

lemma *All-subseteq*: — Converse does not hold

shows $(\bigcap x. \text{ceilr } (P x)) \subseteq \text{ceilr } (\forall x. P x)$

by (*fastforce simp*: *ceilr-alt-def*)

lemma *const-implies*[*ceilr.simps*]:

shows $\text{ceilr } (\langle P \rangle \longrightarrow Q) = (\text{if } P \text{ then } \text{ceilr } Q \text{ else } UNIV)$

by (*simp* *add*: *ceilr.simps*)

lemma *Id-proj-on*:

shows $(\bigcap c. \text{ceilr } (\langle c \rangle = f)) = Id_f$

and $(\bigcap c. \text{ceilr } (f = \langle c \rangle)) = Id_f$

by (*fastforce simp*: *ceilr-alt-def*)⁺

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable} \rangle$

lemma *Inter-ceilr*:

shows *stable* $(\bigcap v. \text{ceilr } (Q v)) (Q v)$

by (*rule antimonoD*[**where** $y = \text{ceilr } (Q v)$, *OF stable.antimono-rel*, *unfolded le-bool-def*, *rule-format*, *rotated*])
(*auto simp*: *ceilr.stable*)

setup $\langle \text{Sign.parent-path} \rangle$

We can internalize the stability conditions; see §19.1.1 for further discussion.

setup $\langle \text{Sign.mandatory-path ag.prog} \rangle$

lemma *p2s-s2p-ag-ceilr*:

shows *prog.p2s* (*prog.s2p* ($\{\!\{P\}\!\}$, $\text{ceilr } P \cap (\bigcap v. \text{ceilr } (Q v)) \vdash G$, $\{\!\{Q\}\!\}$)

$= \{\!\{P\}\!\}$, $\text{ceilr } P \cap (\bigcap v. \text{ceilr } (Q v)) \vdash G$, $\{\!\{Q\}\!\}$

by (*simp* *add*: *ag.prog.p2s-s2p-ag-stable ceilr.stable stable.Inter-ceilr stable.infI*)

setup $\langle \text{Sign.parent-path} \rangle$

19.1.1 Assume/guarantee rules using implicit stability

We use *ceilr* to incorporate stability side conditions directly into the assume/guarantee rules. In other words, instead of working with arbitrary relations, we work with the largest (most general) *assume* that makes the relevant predicates *stable*.

In practice this allows us to defer all stability obligations to the end of a proof, which may be in any convenient context (typically a function). This approach could be considered a semantic version of how [Zakowski, Cachera, Demange, Petri, Pichardie, Jagannathan, and Vitek \(2019\)](#) split sequential and assume/guarantee reasoning. See [Vafeiadis \(2008, §4\)](#) for a discussion on when to check stability.

We defer the *guarantee* proofs by incorporating them into preconditions. This also allows control flow context to be accumulated.

These are backchaining (“weakest precondition”) rules: the guarantee and post condition need to be instantiated and the rules instantiate assume and pre condition schematics.

Note that the rule for (\gg) duplicates stability goals.

See §22 for an example of using these rules.

setup $\langle \text{Sign.mandatory-path iag} \rangle$

named-theorems *intro* $\langle \text{safe backchaining intro rules} \rangle$

lemma *init*:

assumes $c \leq \{P\}, A \vdash G, \{Q\}$

assumes $\bigwedge s. P' s \implies P s$

assumes $A' \subseteq A$ — these rules use *ceilr* which always yields a reflexive relation (*ceilr.refl*)

shows $c \leq \{P'\}, A' \vdash G, \{Q\}$

using *assms(2,3)* **by** (*auto intro: ag.mono order.trans[OF assms(1)]*)

lemmas *mono = ag.mono*

lemmas *gen-asm = ag.gen-asm*

lemmas *pre = ag.pre*

lemmas *pre-pre = ag.pre-pre*

lemmas *pre-post = ag.pre-post*

lemmas *pre-ag = ag.pre-ag*

lemmas *pre-a = ag.pre-a*

lemmas *pre-g = ag.pre-g*

lemmas *post-imp = ag.post-imp*

lemmas *conj-lift = ag.conj-lift*

lemmas *disj-lift = ag.disj-lift*

lemmas *all-lift = ag.all-lift*

lemmas *augment-a = ag.augment-a*

lemmas *augment-post = ag.augment-post*

lemmas *augment-post-imp = ag.augment-post-imp*

lemmas *stable-augment-base = ag.stable-augment-base*

lemmas *stable-augment = ag.stable-augment*

lemmas *stable-augment-post = ag.stable-augment-post*

lemmas *stable-augment-frame = ag.stable-augment-frame*

lemma *bind[iag.intro]*:

assumes $\bigwedge v. \text{prog.p2s } (g v) \leq \{Q' v\}, A_2 v \vdash G, \{Q\}$

assumes $\text{prog.p2s } f \leq \{P\}, A_1 \vdash G, \{Q'\}$

shows $prog.p2s (f \ggg g) \leq \{\!\{P\}\!\}, A_1 \cap (\bigcap v. A_2 v) \vdash G, \{\!\{Q\}\!\}$
by (*auto simp: prog.p2s.simps intro: assms ag.spec.bind ag.pre*)

lemmas *rev-bind = iag.bind[rotated]*

lemma *read[iag.intro]:*

shows $prog.p2s (prog.read F) \leq \{\!\{\lambda s. Q (F s) s\}\!\}, ceilr (\lambda s. Q (F s) s) \cap (\bigcap s. ceilr (Q (F s))) \vdash G, \{\!\{Q\}\!\}$
by (*rule ag.pre[OF ag.prog.action[where P= $\lambda s. Q (F s) s$ and Q=Q and A= $ceilr (\lambda s. Q (F s) s) \cap (\bigcap s. ceilr (Q (F s))$ and G=G]];*
fastforce intro: stable.intro stable.Inter-ceilr stable.infI))

lemma *return[iag.intro]:*

shows $prog.p2s (prog.return v) \leq \{\!\{Q v\}\!\}, ceilr (Q v) \vdash G, \{\!\{Q\}\!\}$
unfolding *prog.return-alt-def* **by** (*rule iag.init[OF iag.read]; fastforce*)

lemma *write[iag.intro]:* — this is where *guarantee* obligations arise

shows $prog.p2s (prog.write F) \leq \{\!\{\lambda s. Q () (F s) \wedge (s, F s) \in G\}\!\}, ceilr (\lambda s. Q () (F s) \wedge (s, F s) \in G) \cap ceilr (Q ()) \vdash G, \{\!\{Q\}\!\}$
by (*rule ag.prog.action; fastforce intro: stable.intro stable.Inter-ceilr stable.infI1 stable.infI2*)

lemma *parallel:* — not in the *iag* format; instantiate the first two assumptions

assumes $prog.p2s c_1 \leq \{\!\{P_1\}\!\}, A_1 \vdash G_1, \{\!\{Q_1\}\!\}$
assumes $prog.p2s c_2 \leq \{\!\{P_2\}\!\}, A_2 \vdash G_2, \{\!\{Q_2\}\!\}$
assumes $\bigwedge s. [Q_1 () s; Q_2 () s] \implies Q () s$
assumes $G_2 \subseteq A_1$
assumes $G_1 \subseteq A_2$
assumes $G_1 \cup G_2 \subseteq G$
shows $prog.p2s (prog.parallel c_1 c_2) \leq \{\!\{P_1 \wedge P_2\}\!\}, A_1 \cap A_2 \vdash G, \{\!\{Q\}\!\}$
by (*rule order.trans[OF ag.prog.parallel[OF iag.pre-a[OF assms(1)] iag.pre-a[OF assms(2)],*
*where A= $A_1 \cap A_2$, *simplified*, OF $\langle G_2 \subseteq A_1 \rangle \langle G_1 \subseteq A_2 \rangle$]];*
use assms(3,6) in $\langle force intro: iag.mono \rangle$)

lemmas *local = ag.prog.local* — not in the *iag* format

lemma *if[iag.intro]:*

assumes $b \implies prog.p2s c_1 \leq \{\!\{P_1\}\!\}, A_1 \vdash G, \{\!\{Q\}\!\}$
assumes $\neg b \implies prog.p2s c_2 \leq \{\!\{P_2\}\!\}, A_2 \vdash G, \{\!\{Q\}\!\}$
shows $prog.p2s (if b then c_1 else c_2) \leq \{\!\{if b then P_1 else P_2\}\!\}, A_1 \cap A_2 \vdash G, \{\!\{Q\}\!\}$
using *assms* **by** (*fastforce intro: ag.pre-ag*)

lemma *case-option[iag.intro]:*

assumes $x = None \implies prog.p2s none \leq \{\!\{P_n\}\!\}, A_n \vdash G, \{\!\{Q\}\!\}$
assumes $\bigwedge v. x = Some v \implies prog.p2s (some v) \leq \{\!\{P_s v\}\!\}, A_s v \vdash G, \{\!\{Q\}\!\}$
shows $prog.p2s (case-option none some x) \leq \{\!\{case x of None \Rightarrow P_n \mid Some v \Rightarrow P_s v\}\!\}, case-option A_n A_s x \vdash G, \{\!\{Q\}\!\}$
using *assms* **by** (*fastforce intro: ag.pre-ag split: option.split*)

lemma *case-sum[iag.intro]:*

assumes $\bigwedge v. x = Inl v \implies prog.p2s (left v) \leq \{\!\{P_l v\}\!\}, A_l v \vdash G, \{\!\{Q\}\!\}$
assumes $\bigwedge v. x = Inr v \implies prog.p2s (right v) \leq \{\!\{P_r v\}\!\}, A_r v \vdash G, \{\!\{Q\}\!\}$
shows $prog.p2s (case-sum left right x) \leq \{\!\{case-sum P_l P_r x\}\!\}, case-sum A_l A_r x \vdash G, \{\!\{Q\}\!\}$
using *assms* **by** (*fastforce intro: ag.pre-ag split: sum.split*)

lemma *case-list[iag.intro]:*

assumes $x = [] \implies prog.p2s nil \leq \{\!\{P_n\}\!\}, A_n \vdash G, \{\!\{Q\}\!\}$
assumes $\bigwedge v vs. x = v \# vs \implies prog.p2s (cons v vs) \leq \{\!\{P_c v vs\}\!\}, A_c v vs \vdash G, \{\!\{Q\}\!\}$

shows $\text{prog.p2s } (\text{case-list nil cons } x) \leq \{\!\{ \text{case-list } P_n P_c x \}\!\}$, $\text{case-list } A_n A_c x \vdash G$, $\{\!\{ Q \}\!\}$
using *assms* **by** (*fastforce intro: ag.pre-ag split: list.split*)

lemma *while*:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$
assumes $c: \bigwedge k. \text{prog.p2s } (c k) \leq \{\!\{ P k \}\!\}$, $A \vdash G$, $\{\!\{ \text{case-sum } I Q \}\!\}$
shows $\text{prog.p2s } (\text{prog.while } c k) \leq \{\!\{ (\forall v s. I v s \longrightarrow P v s) \wedge I k \}\!\}$, $A \cap (\bigcap v. \text{ceilr } (Q v)) \vdash G$, $\{\!\{ Q \}\!\}$
by (*rule iag.gen-asm*)
(rule ag.prog.while[OF ag.pre-a[OF c]]; blast intro: stable.Inter-ceilr stable.infI2)

lemmas $\text{whenM} = \text{iag.if}[\mathbf{where } c_1=c \mathbf{ and } A_1=A \mathbf{ and } P_1=P, \text{OF} - \text{iag.return}[\mathbf{where } v=()]]$ **for** $A c P$

setup $\langle \text{Sign.parent-path} \rangle$

19.2 Refinement with relational assumes

Two sets of refinement rules:

- relational assumes
- relational assumes and *prog.sinvmap* (inverse state abstraction)

setup $\langle \text{Sign.mandatory-path rar.prog} \rangle$

lemma *bind*:

assumes $\bigwedge v. \text{prog.p2s } (g v) \leq \{\!\{ Q' v \}\!\}$, $\text{ag.assm } A \Vdash \text{prog.p2s } (g' v)$, $\{\!\{ Q \}\!\}$
assumes $\text{prog.p2s } f \leq \{\!\{ P \}\!\}$, $\text{ag.assm } A \Vdash \text{prog.p2s } f'$, $\{\!\{ Q' \}\!\}$
shows $\text{prog.p2s } (f \ggg g) \leq \{\!\{ P \}\!\}$, $\text{ag.assm } A \Vdash \text{prog.p2s } (f' \ggg g')$, $\{\!\{ Q \}\!\}$
by (*rule refinement.prog.bind[OF refinement.pre-a[OF assms(1) refinement.spec.bind.res.rel-le[OF order.refl]]]*)
(simp add: spec.term.all.rel assms(2))

lemmas $\text{rev-bind} = \text{rar.prog.bind}[\text{rotated}]$

lemma *action*:

fixes $F :: ('v \times 's \times 's) \text{ set}$
fixes $F' :: ('v \times 's \times 's) \text{ set}$
assumes $Q: \bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \Longrightarrow Q v s'$
assumes $F': \bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \Longrightarrow (v, s, s') \in F'$
assumes $sP: \text{stable } A P$
assumes $sQ: \bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \Longrightarrow \text{stable } A (Q v)$
shows $\text{prog.p2s } (\text{prog.action } F) \leq \{\!\{ P \}\!\}$, $\text{ag.assm } A \Vdash \text{prog.p2s } (\text{prog.action } F')$, $\{\!\{ Q \}\!\}$
by (*rule refinement.prog.action*)
(use assms in $\langle \text{simp-all add: spec.steps.rel stable-def monotone-def} \rangle$)

lemma *return*:

assumes $sQ: \text{stable } A (Q v)$
shows $\text{prog.p2s } (\text{prog.return } v) \leq \{\!\{ Q v \}\!\}$, $\text{ag.assm } A \Vdash \text{prog.p2s } (\text{prog.return } v)$, $\{\!\{ Q \}\!\}$
using *assms* **by** (*simp add: refinement.prog.return spec.steps.rel stable-def monotone-def*)

lemma *parallel-refinement*:

assumes $c_1: \text{prog.p2s } c_1 \leq \{\!\{ P_1 \}\!\}$, $\text{ag.assm } (A \cup G_2) \Vdash \text{prog.p2s } (c_1' \sqcap \text{prog.rel } G_1)$, $\{\!\{ Q_1 \}\!\}$
assumes $c_2: \text{prog.p2s } c_2 \leq \{\!\{ P_2 \}\!\}$, $\text{ag.assm } (A \cup G_1) \Vdash \text{prog.p2s } (c_2' \sqcap \text{prog.rel } G_2)$, $\{\!\{ Q_2 \}\!\}$
shows $\text{prog.p2s } (c_1 \parallel c_2) \leq \{\!\{ P_1 \wedge P_2 \}\!\}$, $\text{ag.assm } A \Vdash \text{prog.p2s } (c_1' \sqcap \text{prog.rel } G_1 \parallel c_2' \sqcap \text{prog.rel } G_2)$, $\{\!\{ \lambda v. Q_1 v \wedge Q_2 v \}\!\}$
apply (*rule refinement.prog.parallel[OF refinement.pre-a[OF c₁] refinement.pre-a[OF c₂]]*)
apply (*rule order.trans[OF refinement.spec.env-hyp.mono[OF order.refl] ag.spec.Parallel-assm[$\mathbf{where } a=\text{True}$ and $as=\text{UNIV}$ and $G=\lambda a. \text{if } a \text{ then } G_1 \text{ else } G_2, \text{simplified}]]$*)
(simp add: ac-simps prog.p2s.simps; fail)

apply (*rule order.trans*[*OF refinement.spec.env-hyp.mono*[*OF order.refl*] *ag.spec.Parallel-assm*[**where** *a=False*
and *as=UNIV* **and** *G=λa. if a then G₁ else G₂, simplified*]]];
simp add: ac-simps prog.p2s.simps; fail)
done

lemma parallel:

assumes *prog.p2s c₁ ≤ {P₁}*, *ag.assm (A ∪ G₂) ⊢ prog.p2s c₁'*, *{Q₁}*
assumes *prog.p2s c₁ ≤ {P₁}*, *A ∪ G₂ ⊢ G₁*, *{⊤}*
assumes *prog.p2s c₂ ≤ {P₂}*, *ag.assm (A ∪ G₁) ⊢ prog.p2s c₂'*, *{Q₂}*
assumes *prog.p2s c₂ ≤ {P₂}*, *A ∪ G₁ ⊢ G₂*, *{⊤}*
shows *prog.p2s (c₁ || c₂) ≤ {P₁ ∧ P₂}*, *ag.assm A ⊢ prog.p2s (c₁' || c₂')*, *{λv. Q₁ v ∧ Q₂ v}*
by (*rule order.trans*[*OF rar.prog.parallel-refinement*
refinement.mono[*OF order.refl order.refl prog.p2s.mono*[*OF prog.parallel.mono*[*OF inf.cobounded1*
inf.cobounded1]]] *order.refl*]])
(force simp: prog.p2s.simps refinement.infR-conv[**where** *Q₂=⊤*, *simplified*]
simp flip: ag.refinement
intro: assms)+

lemma while:

fixes *c :: 'k ⇒ ('s, 'k + 'v) prog*
fixes *c' :: 'k ⇒ ('s, 'k + 'v) prog*
assumes *c: λk. prog.p2s (c k) ≤ {P k}*, *ag.assm A ⊢ prog.p2s (c' k)*, *{case-sum I Q}*
assumes *IP: λs v. I v s ⇒ P v s*
assumes *sQ: λv. stable A (Q v)*
shows *prog.p2s (prog.while c k) ≤ {I k}*, *ag.assm A ⊢ prog.p2s (prog.while c' k)*, *{Q}*
proof –
have *∀k. prog.p2s (prog.while c k) ≤ {P k}*, *ag.assm A ⊢ prog.p2s (prog.while c' k)*, *{Q}*
proof(*induct rule: prog.while.fixp-induct*[**where** *P=λR. ∀k. prog.p2s (R c k) ≤ {P k}*, *ag.assm A ⊢ prog.p2s*
(prog.while c' k), *{Q}*], *case-names adm bot step*])
case (*step R*) **from** *sQ* **show** *?case*
apply (*subst prog.while.simps*)
apply (*intro allI rar.prog.rev-bind*[*OF c*
refinement.pre-pre[*OF refinement.prog.case-sum*[*OF step*[*rule-format*
rar.prog.return[*OF sQ*]]]])
apply (*simp add: IP split: sum.splits*)
done
qed *simp-all*
then show *?thesis*
by (*meson IP refinement.pre-pre*)
qed

lemma app:

fixes *xs :: 'a list*
fixes *f :: 'a ⇒ ('s, unit) prog*
fixes *P :: 'a list ⇒ 's pred*
assumes *λx ys zs. xs = ys @ x # zs ⇒ prog.p2s (f x) ≤ {P ys}*, *ag.assm A ⊢ prog.p2s (f' x)*, *{λ-. P (ys @*
[x])}
assumes *λys. prefix ys xs ⇒ stable A (P ys)*
shows *prog.p2s (prog.app f xs) ≤ {P []}*, *ag.assm A ⊢ prog.p2s (prog.app f' xs)*, *{λ-. P xs}*
using *assms*
by (*induct xs rule: rev-induct*;
force intro: rar.prog.return
simp: prog.app.append prog.app.simps spec.steps.rel prog.bind.return rar.prog.rev-bind)

lemmas *if = refinement.prog.if*[**where** *A=ag.assm A* **for** *A*]

lemmas *case-option = refinement.prog.case-option*[**where** *A=ag.assm A* **for** *A*]

setup *⟨Sign.parent-path⟩*

setup $\langle \text{Sign.mandatory-path rair.prog} \rangle$

abbreviation $(\text{input}) \text{ absfn } sf \ c \equiv \text{prog.p2s } (\text{prog.invmap } sf \ c)$

lemma *bind*:

assumes $\bigwedge v. \text{prog.p2s } (g \ v) \leq \{\!\{Q' \ v\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (g' \ v), \{\!\{Q\}\!\}$
assumes $\text{prog.p2s } f \leq \{\!\{P\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ f', \{\!\{Q'\}\!\}$
shows $\text{prog.p2s } (f \ggg g) \leq \{\!\{P\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (f' \ggg g'), \{\!\{Q\}\!\}$
by (*simp add: prog.invmap.bind rar.prog.bind[OF assms]*)

lemmas *rev-bind = rair.prog.bind[rotated]*

lemma *action*:

fixes $F :: ('v \times 's \times 's) \text{ set}$
fixes $F' :: ('v \times 't \times 't) \text{ set}$
fixes $sf :: 's \Rightarrow 't$
assumes $Q: \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \Longrightarrow Q \ v \ s'$
assumes $F': \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \Longrightarrow (v, sf \ s, sf \ s') \in F'$
assumes $sP: \text{stable } A \ P$
assumes $sQ: \bigwedge v \ s \ s'. \llbracket P \ s; (v, s, s') \in F \rrbracket \Longrightarrow \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.action } F) \leq \{\!\{P\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (\text{prog.action } F'), \{\!\{Q\}\!\}$
by (*strengthen ord-to-strengthen(2)[OF prog.action.invmap-le]*)
(*simp add: rar.prog.action assms*)

lemma *return*:

assumes $sQ: \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.return } v) \leq \{\!\{Q \ v\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (\text{prog.return } v), \{\!\{Q\}\!\}$
using *assms* **by** (*simp add: refinement.prog.invmap-return spec.steps.rel stable-def monotone-def*)

lemma *parallel*:

fixes $sf :: 's \Rightarrow 't$
assumes $\text{prog.p2s } c_1 \leq \{\!\{P_1\}\!\}$, $ag.assm \ (A \cup G_2) \Vdash \text{rair.prog.absfn } sf \ c_1', \{\!\{Q_1\}\!\}$
assumes $\text{prog.p2s } c_1 \leq \{\!\{P_1\}\!\}$, $A \cup G_2 \vdash G_1, \{\!\{\top\}\!\}$
assumes $\text{prog.p2s } c_2 \leq \{\!\{P_2\}\!\}$, $ag.assm \ (A \cup G_1) \Vdash \text{rair.prog.absfn } sf \ c_2', \{\!\{Q_2\}\!\}$
assumes $\text{prog.p2s } c_2 \leq \{\!\{P_2\}\!\}$, $A \cup G_1 \vdash G_2, \{\!\{\top\}\!\}$
shows $\text{prog.p2s } (c_1 \parallel c_2) \leq \{\!\{P_1 \wedge P_2\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (c_1' \parallel c_2'), \{\!\{\lambda v. Q_1 \ v \wedge Q_2 \ v\}\!\}$
unfolding *prog.invmap.parallel* **by** (*rule rar.prog.parallel[OF assms]*)

lemma *while*:

fixes $c :: 'k \Rightarrow ('s, 'k + 'v) \text{ prog}$
fixes $c' :: 'k \Rightarrow ('t, 'k + 'v) \text{ prog}$
fixes $sf :: 's \Rightarrow 't$
assumes $c: \bigwedge k. \text{prog.p2s } (c \ k) \leq \{\!\{P \ k\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (c' \ k), \{\!\{\text{case-sum } I \ Q\}\!\}$
assumes $IP: \bigwedge s \ v. I \ v \ s \Longrightarrow P \ v \ s$
assumes $sQ: \bigwedge v. \text{stable } A \ (Q \ v)$
shows $\text{prog.p2s } (\text{prog.while } c \ k) \leq \{\!\{I \ k\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (\text{prog.while } c' \ k), \{\!\{Q\}\!\}$
by (*strengthen ord-to-strengthen(2)[OF prog.while.invmap-le]*)
(*simp add: assms map-sum.id rar.prog.while[OF c]*)

lemma *app*:

fixes $xs :: 'a \text{ list}$
fixes $f :: 'a \Rightarrow ('s, \text{unit}) \text{ prog}$
fixes $P :: 'a \text{ list} \Rightarrow 's \text{ pred}$
assumes $\bigwedge x \ ys \ zs. xs = ys @ x \ \# \ zs \Longrightarrow \text{prog.p2s } (f \ x) \leq \{\!\{P \ ys\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (f' \ x), \{\!\{\lambda-. P \ xs\}\!\}$
 $P \ (ys @ [x])\}$
assumes $\bigwedge ys. \text{prefix } ys \ xs \Longrightarrow \text{stable } A \ (P \ ys)$
shows $\text{prog.p2s } (\text{prog.app } f \ xs) \leq \{\!\{P \ []\}\!\}$, $ag.assm \ A \Vdash \text{rair.prog.absfn } sf \ (\text{prog.app } f' \ xs), \{\!\{\lambda-. P \ xs\}\!\}$

by (*strengthen ord-to-strengthen*(2)[*OF prog.sinvmap.app-le*])
(*simp add: rar.prog.app assms*)

lemma if:

assumes $i \implies \text{prog.p2s } t \leq \{\!\{P\}\!\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } t'$, $\{\!\{Q\}\!\}$

assumes $\neg i \implies \text{prog.p2s } e \leq \{\!\{P'\}\!\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } e'$, $\{\!\{Q\}\!\}$

shows $\text{prog.p2s } (\text{if } i \text{ then } t \text{ else } e) \leq \{\!\{\text{if } i \text{ then } P \text{ else } P'\}\!\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } (\text{if } i \text{ then } t' \text{ else } e')$,
 $\{\!\{Q\}\!\}$

using *assms* **by** *fastforce*

lemma case-option:

assumes $\text{opt} = \text{None} \implies \text{prog.p2s } \text{none} \leq \{\!\{P_n\}\!\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } \text{none}'$, $\{\!\{Q\}\!\}$

assumes $\bigwedge v. \text{opt} = \text{Some } v \implies \text{prog.p2s } (\text{some } v) \leq \{\!\{P_s v\}\!\}$, $\text{ag.assm } A \Vdash \text{rair.prog.absfn sf } (\text{some}' v)$, $\{\!\{Q\}\!\}$

shows $\text{prog.p2s } (\text{case-option none some opt}) \leq \{\!\{\text{case opt of None} \Rightarrow P_n \mid \text{Some } v \Rightarrow P_s v\}\!\}$, $\text{ag.assm } A \Vdash$
 $\text{rair.prog.absfn sf } (\text{case-option none}' \text{ some}' \text{ opt})$, $\{\!\{Q\}\!\}$

using *assms* **by** (*simp add: option.case-eq-if*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

20 Wickerson, Dodds and Parkinson: explicit stabilisation

Notes on [Wickerson, Dodds, and Parkinson \(2010\)](#) (all references here are to the technical report):

- motivation: techniques for eliding redundant stability conditions
 - the standard rules check the interstitial assertion in $c ; d$ twice
- they claim in §7 to supersede the “mid stability” of [Vafeiadis \(2008, §4.1\)](#) (wssa, sswa)
- Appendix D:
 - not a complete set of rules
 - ATOMR-S does not self-compose: consider $c ; d$ – the interstitial assertion is either a floor or ceiling
 - * every step therefore requires a use of weakening/monotonicity

The basis of their approach is to make assertions a function of a relation (a *rely*). By considering a set of relations, a single rely-guarantee specification can satisfy several call sites. Separately they tweak the RGSep rules of [Vafeiadis \(2008\)](#).

The definitions are formally motivated as follows (§3):

Our operators can also be defined using Dijkstra’s predicate transformer semantics: $\lfloor p \rfloor R$ is the weakest precondition of R^* given postcondition p , while $\lceil p \rceil R$ is the strongest postcondition of R^* given precondition p .

The following adapts their definitions and proofs to our setting.

setup $\langle \text{Sign.mandatory-path wdp} \rangle$

definition *floor* $:: 'a \text{ rel} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred}$ **where** — An interior operator, or a closure in the dual lattice
floor $r P s \iff (\forall s'. (s, s') \in r^* \longrightarrow P s')$

definition *ceiling* $:: 'a \text{ rel} \Rightarrow 'a \text{ pred} \Rightarrow 'a \text{ pred}$ **where** — A closure operator
ceiling $r P s \iff (\exists s'. (s', s) \in r^* \wedge P s')$

setup $\langle \text{Sign.mandatory-path floor} \rangle$

lemma *empty-rel*[*simp*]:

shows $\text{wdp.floor } \{\} P = P$

by (*simp add: wdp.floor-def fun-eq-iff*)

lemma *reflcl*:

shows $\text{wdp.floor } (r^\complement) = \text{wdp.floor } r$

by (*simp add: wdp.floor-def fun-eq-iff*)

lemma *const*:

shows $\text{wdp.floor } r \langle c \rangle = \langle c \rangle$

by (*auto simp: wdp.floor-def fun-eq-iff*)

lemma *contractive*:

shows $\text{wdp.floor } r P \leq P$

by (*simp add: wdp.floor-def le-fun-def le-bool-def*)

lemma *idempotent*:

shows $\text{wdp.floor } r (\text{wdp.floor } r P) = \text{wdp.floor } r P$

by (*auto simp: fun-eq-iff wdp.floor-def dest: rtrancl-trans*)

lemma *mono*:

assumes $r' \subseteq r$

assumes $P \leq P'$

shows $\text{wdp.floor } r P \leq \text{wdp.floor } r' P'$

using *assms* **by** (*auto 6 0 simp add: wdp.floor-def le-bool-def le-fun-def dest: rtrancl-mono*)

lemma *strengthen*[*strg*]:

assumes $\text{st-ord } (\neg F) r r'$

assumes $\text{st-ord } F P P'$

shows $\text{st-ord } F (\text{wdp.floor } r P) (\text{wdp.floor } r' P')$

using *assms* **by** (*cases F; simp add: wdp.floor.mono*)

lemma *weakest*:

assumes $Q \leq P$

assumes $\text{stable } r Q$

shows $Q \leq \text{wdp.floor } r P$

using *assms* **by** (*simp add: wdp.floor-def stable-def monotone-def le-fun-def le-bool-def*) (*metis rtrancl-induct*)

lemma *Chernoff*:

assumes $P \leq Q$

shows $(\text{wdp.floor } r P \wedge Q) \leq \text{wdp.floor } r Q$

using *assms* **by** (*simp add: wdp.floor-def le-fun-def le-bool-def*)

lemma *floor1*:

assumes $r \subseteq r'$

shows $\text{wdp.floor } r' (\text{wdp.floor } r P) = \text{wdp.floor } r' P$

unfolding *wdp.floor-def* **by** (*meson assms rtrancl-cl.cl-mono rtrancl-eq-or-trancl rtrancl-trans*)

lemma *floor2*:

assumes $r \subseteq r'$

shows $\text{wdp.floor } r (\text{wdp.floor } r' P) = \text{wdp.floor } r' P$

by (*metis assms antisym wdp.floor.contractive wdp.floor.idempotent wdp.floor.mono*)

setup $\langle \text{Sign.parent-path} \rangle$

interpretation *ceiling*: *closure-complete-lattice-distributive-class* $\text{wdp.ceiling } r$ **for** r

by *standard* (*auto 5 0 simp: wdp.ceiling-def le-fun-def le-bool-def dest: rtrancl-trans*)

setup $\langle \text{Sign.mandatory-path ceiling} \rangle$

lemma *empty-rel*[simp]:

shows $\text{wdp.ceiling } \{\} P = P$

by (*simp add: wdp.ceiling-def fun-eq-iff*)

lemma *reflcl*:

shows $\text{wdp.ceiling } (r^=) = \text{wdp.ceiling } r$

by (*simp add: wdp.ceiling-def fun-eq-iff*)

lemma *const*:

shows $\text{wdp.ceiling } r \langle c \rangle = \langle c \rangle$

by (*auto simp: wdp.ceiling-def fun-eq-iff*)

lemma *mono*:

assumes $r \subseteq r'$

assumes $P \leq P'$

shows $\text{wdp.ceiling } r P \leq \text{wdp.ceiling } r' P'$

using *assms* **by** (*auto 7 0 simp: wdp.ceiling-def le-bool-def le-fun-def dest: rtrancl-mono*)

lemma *strengthen*[strg]:

assumes *st-ord* $F r r'$

assumes *st-ord* $F P P'$

shows *st-ord* $F (\text{wdp.ceiling } r P) (\text{wdp.ceiling } r' P')$

using *assms* **by** (*cases F; simp add: wdp.ceiling.mono*)

lemma *strongest*:

assumes $P \leq Q$

assumes *stable* $r Q$

shows $\text{wdp.ceiling } r P \leq Q$

using *assms* **by** (*simp add: wdp.ceiling-def stable-def monotone-def le-fun-def le-bool-def*) (*metis rtrancl-induct*)

lemma *ceiling1*:

assumes $r \subseteq r'$

shows $\text{wdp.ceiling } r' (\text{wdp.ceiling } r P) = \text{wdp.ceiling } r' P$

unfolding *wdp.ceiling-def* **by** (*meson assms rtrancl-cl.cl-mono rtrancl-eq-or-trancl rtrancl-trans*)

lemma *ceiling2*:

assumes $r \subseteq r'$

shows $\text{wdp.ceiling } r (\text{wdp.ceiling } r' P) = \text{wdp.ceiling } r' P$

by (*metis assms antisym wdp.ceiling.ceiling1 wdp.ceiling.expansive wdp.ceiling.idempotent(1)*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable} \rangle$

lemma *floor*:

shows *stable* $r (\text{wdp.floor } r P)$

unfolding *wdp.floor-def stable-def monotone-def* **by** (*simp add: converse-rtrancl-into-rtrancl le-boolI*)

lemma *ceiling*:

shows *stable* $r (\text{wdp.ceiling } r P)$

by (*fastforce simp: wdp.ceiling-def stable-def monotone-def le-bool-def*
dest: rtrancl-into-rtrancl)

lemma *floor-conv*:

assumes *stable* $r P$

shows $P = \text{wdp.floor } r P$

using *assms unfolding wdp.floor-def stable-def monotone-def le-bool-def fun-eq-iff*
by (*metis rtrancl-refl rtrancl-induct*)

lemma *ceiling-conv*:

assumes *stable r P*

shows $P = \text{wdp.ceiling } r P$

using *assms unfolding wdp.ceiling-def stable-def monotone-def le-bool-def fun-eq-iff*
by (*metis rtrancl-refl rtrancl-induct*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *floor-alt-def*: — Wickerson et al. (2010, §3)

shows $\text{wdp.floor } r P = \bigsqcup \{Q. Q \leq P \wedge \text{stable } r Q\}$

by (*rule antisym*)

(*auto simp: Sup-upper wdp.floor.contractive wdp.stable.floor*
intro: wdp.floor.weakest[unfolded le-bool-def le-fun-def, rule-format])

lemma *ceiling-alt-def*: — Wickerson et al. (2010, §3)

shows $\text{wdp.ceiling } r P = \bigsqcap \{Q. P \leq Q \wedge \text{stable } r Q\}$

by (*rule antisym*)

(*auto simp: Inf-lower wdp.ceiling.expansive wdp.stable.ceiling*
dest: wdp.ceiling.strongest[unfolded le-bool-def le-fun-def, rule-format, rotated])

lemma *duality-floor-ceiling*:

shows $\text{wdp.ceiling } r (\neg P) = (\neg \text{wdp.floor } (r^{-1}) P)$

by (*simp add: wdp.ceiling-def wdp.floor-def fun-eq-iff rtrancl-converse*)

lemma *ceiling-floor*:

assumes $r \subseteq r'$

shows $\text{wdp.ceiling } r (\text{wdp.floor } r' P) = \text{wdp.floor } r' P$

by (*metis assms wdp.ceiling.ceiling2 wdp.stable.ceiling-conv wdp.stable.floor*)

lemma *floor-ceiling*:

assumes $r \subseteq r'$

shows $\text{wdp.floor } r (\text{wdp.ceiling } r' P) = \text{wdp.ceiling } r' P$

by (*metis assms wdp.floor.floor2 wdp.stable.ceiling wdp.stable.floor-conv*)

lemma *floor-ceilr*:

shows $\text{wdp.floor } (\text{ceilr } P) P = P$

by (*metis ceilr.stable wdp.stable.floor-conv*)

lemma *ceiling-ceilr*:

shows $\text{wdp.ceiling } (\text{ceilr } P) P = P$

by (*metis ceilr.stable wdp.stable.ceiling-conv*)

setup $\langle \text{Sign.parent-path} \rangle$

20.1 Assume/Guarantee rules

§3.2 traditional assume/guarantee rules **setup** $\langle \text{Sign.mandatory-path wdp} \rangle$

lemma *action*: — arbitrary A

fixes $F :: ('v \times 's \times 's) \text{ set}$

assumes $Q: \bigwedge v s s'. \llbracket P s; (v, s, s') \in F \rrbracket \implies Q v s'$

assumes $G: \bigwedge v s s'. \llbracket P s; s \neq s'; (v, s, s') \in F \rrbracket \implies (s, s') \in G$

shows $\text{prog.p2s } (\text{prog.action } F) \leq \{\text{wdp.floor } A P\}, A \vdash G, \{\lambda v. \text{wdp.ceiling } A (Q v)\}$

by (*rule ag.prog.action*)

(*auto simp: wdp.stable.floor wdp.stable.ceiling*)

intro: G
dest: $Q[\text{rotated}]$
elim: $\text{wdp.floor.contractive}[\text{unfolded le-fun-def le-bool-def, rule-format}]$
 $\text{wdp.ceiling.expansive}[\text{unfolded le-fun-def le-bool-def, rule-format}]$

lemmas $\text{mono} = \text{ag.mono}$
lemmas $\text{bind} = \text{ag.prog.bind}$

etc. – the other rules are stock

setup $\langle \text{Sign.parent-path} \rangle$

§4, Appendix C parametric specifications definition $\text{pag} :: ('s \text{ rel} \Rightarrow 's \text{ pred}) \Rightarrow 's \text{ rel set} \Rightarrow 's \text{ rel} \Rightarrow ('s \text{ rel} \Rightarrow 'v \Rightarrow 's \text{ pred}) \Rightarrow (\text{sequential}, 's, 'v) \text{ spec} (\langle \{-\}, - / \vdash_P -, \{-\} \rangle [0,0,0,0] 100)$ **where**
 $\{\!|P|\!\}, \text{As} \vdash_P G, \{\!|Q|\!\} = (\prod A \in \text{As}. \{\!|P A|\!\}, A \vdash G, \{\!|Q A|\!\})$

setup $\langle \text{Sign.mandatory-path pag} \rangle$

lemma *empty*:

shows $\{\!|P|\!\}, \{\} \vdash_P G, \{\!|Q|\!\} = \top$
by (*simp add: pag-def*)

lemma *singleton*:

shows $\{\!|P|\!\}, \{A\} \vdash_P G, \{\!|Q|\!\} = \{\!|P A|\!\}, A \vdash G, \{\!|Q A|\!\}$
by (*simp add: pag-def*)

lemma *mono*: — strengthening of the WEAKEN rule in Figure 4, needed for the example

assumes $\bigwedge A. A \in \text{As}' \Longrightarrow P' A \leq P A$
assumes $\text{As}' \leq \text{As}$
assumes $G \leq G'$
assumes $\bigwedge A. A \in \text{As}' \Longrightarrow Q A \leq Q' A$
shows $\{\!|P|\!\}, \text{As} \vdash_P G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, \text{As}' \vdash_P G', \{\!|Q'|\!\}$
by (*simp add: asms pag-def INF-superset-mono[OF $\langle \text{As}' \leq \text{As} \rangle$ ag.mono] le-funD*)

lemma *action*: — allow assertions to depend on assume A , needed for the example

fixes $F :: ('v \times 's \times 's) \text{ set}$
assumes $Q: \bigwedge A v s s'. \llbracket A \in \text{As}; P A s; (v, s, s') \in F \rrbracket \Longrightarrow Q A v s'$
assumes $G: \bigwedge A v s s'. \llbracket A \in \text{As}; P A s; s \neq s'; (v, s, s') \in F \rrbracket \Longrightarrow (s, s') \in G$
shows $\text{prog.p2s} (\text{prog.action } F) \leq \{\!|\lambda A. \text{wdp.floor } A (P A)|\!\}, \text{As} \vdash_P G, \{\!|\lambda A v. \text{wdp.ceiling } A (Q A v)|\!\}$
by (*simp add: asms pag-def wdp.action INFI*)

lemmas $\text{sup} = \text{ag.prog.sup}$

lemma *bind*:

assumes $\bigwedge v. \text{prog.p2s} (g v) \leq \{\!|\lambda A. Q' A v|\!\}, \text{As} \vdash_P G, \{\!|Q|\!\}$
assumes $\text{prog.p2s} f \leq \{\!|P|\!\}, \text{As} \vdash_P G, \{\!|Q|\!\}$
shows $\text{prog.p2s} (f \ggg g) \leq \{\!|P|\!\}, \text{As} \vdash_P G, \{\!|Q|\!\}$
unfolding *pag-def*
by (*fastforce intro: INFI ag.prog.bind[rotated]*
 $\text{order.trans}[\text{OF asms}(2) \text{ pag.mono}[\text{OF } - - - \text{order.refl}]]$
 $\text{order.trans}[\text{OF asms}(1) \text{ pag.mono}[\text{where } \text{As}' = \{A\} \text{ for } A], \text{simplified pag.singleton}]$
simp flip: pag.singleton) $+$

lemma *parallel*:

assumes $\text{prog.p2s} c_1 \leq \{\!|P_1|\!\}, (\cup) G_2 \text{ ' } A \vdash_P G_1, \{\!|Q_1|\!\}$
assumes $\text{prog.p2s} c_2 \leq \{\!|P_2|\!\}, (\cup) G_1 \text{ ' } A \vdash_P G_2, \{\!|Q_2|\!\}$
shows $\text{prog.p2s} (\text{prog.parallel } c_1 c_2)$
 $\leq \{\!|\lambda R. P_1 (R \cup G_2) \wedge P_2 (R \cup G_1)|\!\}, A \vdash_P G_1 \cup G_2, \{\!|\lambda R v. Q_1 (R \cup G_2) v \wedge Q_2 (R \cup G_1) v|\!\}$
unfolding *pag-def*

by (force intro: INFI ag.prog.parallel order.trans[OF assms(1) pag.mono]
order.trans[OF assms(2) pag.mono]
simp flip: pag.singleton)

etc. – the other rules follow similarly

setup <Sign.parent-path>

20.2 Examples

There is not always a single (traditional) most general assume/guarantee specification (§2.1).

type-synonym $state = int$ — just x

abbreviation (input) $incr \equiv prog.write ((+) 1)$ — atomic increment

abbreviation (input) $increases :: int\ rel$ **where** $increases \equiv \{(x, x'). x \leq x'\}$

lemma *ag-incr1*: — the precondition is stable as the rely is very strong

shows $prog.p2s\ incr \leq \{(=)\ c\}, \{\} \vdash increases, \{((=)\ (c + 1))\}$

by (rule ag.prog.action; simp add: stable.empty)

lemma *ag-incr2*: — note the weaker precondition due to the larger assume

shows $prog.p2s\ incr \leq \{(\leq)\ c\}, increases \vdash increases, \{(\leq)\ (c + 1)\}$

by (rule ag.prog.action; auto simp: stable-def monotone-def le-bool-def)

lemma *ag-incr1-par-incr1*:

shows $prog.p2s\ (incr \parallel incr) \leq \{\lambda x. c \leq x\}, increases \vdash increases, \{\lambda x. c + 1 \leq x\}$

apply (rule ag.pre-pre)

apply (rule ag.post-post)

apply (rule ag.prog.parallel[**where** $P_1 = \lambda x. c \leq x$ **and** $P_2 = \lambda x. c \leq x$
and $Q_1 = \lambda x. c + 1 \leq x$ **and** $Q_2 = \lambda x. c + 1 \leq x$
and $G_1 = increases$ **and** $G_2 = increases$, simplified])

apply (rule ag.prog.action; simp add: stable-def monotone-def le-boolI; fail)

apply (rule ag.prog.action; simp add: stable-def monotone-def le-boolI; fail)

apply simp-all

done

Using explicit stabilisation we can squash the two specifications for *incr* into a single one (§4).

lemma — postcondition cannot be simplified for arbitrary A

shows $prog.p2s\ incr \leq \{wdp.ceiling\ A\ ((=)\ c)\}, A \vdash increases, \{\langle wdp.ceiling\ A\ (\lambda s. wdp.ceiling\ A\ ((=)\ c)\ (s - 1)) \rangle\}$

by (rule ag.pre-pre[OF wdp.action]) (simp add: wdp.floor-ceiling)+

— The set of assumes that commute with the increment

abbreviation (input) $comm-xpp :: int\ rel\ set$ **where**

$comm-xpp \equiv \{A. \forall p\ s. wdp.ceiling\ A\ p\ (s - 1) = wdp.ceiling\ A\ (\lambda s. p\ (s - 1))\ s\}$

lemma *pag-incr*: — postcondition can be simplified wrt *comm-xpp*

shows $prog.p2s\ incr \leq \{\lambda A. wdp.ceiling\ A\ ((=)\ c)\}, comm-xpp \vdash_P increases, \{\lambda A. \langle wdp.ceiling\ A\ ((=)\ (c + 1)) \rangle\}$

apply (rule order.trans[OF - pag.mono[OF - order.refl order.refl]])

apply (rule pag.action[**where** $P = \lambda A. wdp.ceiling\ A\ ((=)\ c)$

and $Q = \lambda A\ v\ s. wdp.ceiling\ A\ ((=)\ c)\ (s - 1)$])

apply (simp-all add: wdp.floor-ceiling eq-diff-eq)

done

— the two earlier specifications can be recovered

lemma

shows $prog.p2s\ incr \leq \{(=)\ c\}, \{\} \vdash increases, \{((=)\ (c + 1))\}$

apply (rule order.trans[OF pag-incr])

apply (subst pag.singleton[symmetric])

```

apply (rule pag.mono; force)
done

```

```

lemma

```

```

  shows prog.p2s incr ≤ {((≤) c}, increases ⊢ increases, {((≤) (c + 1))}
apply (rule order.trans[OF pag-incr[where c=c]])
apply (subst pag.singleton[symmetric])
apply (rule pag.mono; force simp: wdp.ceiling-def order-rtrancl dest: zless-imp-add1-zle)
done

```

21 Example: inhabitation

The following is a simple example of showing that a specification is inhabited.

```

lemma

```

```

  shows ⟨0::nat, [(self, 1), (self, 2)], Some ()⟩
    ≤ prog.p2s (prog.while ⟨prog.write ((+) 1) ≫ (prog.return (Inl ()) ⊔ prog.return (Inr ()))⟩ ())
apply (rule inhabits.I)
apply (rule inhabits.pre)
  apply (subst prog.while.simps)
  apply (simp add: prog.bind.bind)
  apply (rule inhabits.trans)
  apply (rule inhabits.prog.bind)
  apply (rule inhabits.prog.action-step)
  apply force
  apply (simp add: prog.bind.returnL)
  apply (rule inhabits.trans)
  apply (rule inhabits.prog.bind)
  apply (rule inhabits.prog.supL)
  apply (rule inhabits.tau)
  apply (simp add: spec.idle.p2s-le; fail)
  apply (simp add: prog.bind.returnL)
  apply (subst prog.while.simps)
  apply (simp add: prog.bind.bind)
  apply (rule inhabits.trans)
  apply (rule inhabits.prog.bind)
  apply (rule inhabits.prog.action-step)
  apply force
  apply (simp add: prog.bind.returnL)
  apply (rule inhabits.trans)
  apply (rule inhabits.prog.bind)
  apply (rule inhabits.prog.supR)
  apply (rule inhabits.tau)
  apply (simp add: spec.idle.p2s-le; fail)
  apply (simp add: prog.bind.returnL)
  apply (rule inhabits.tau)
  apply (simp add: spec.idle.p2s-le; fail)
apply (simp add: prog.p2s.return spec.interference.cl.return spec.return.rel-le; fail)
apply simp
done

```

22 Example: findP

We demonstrate assume/guarantee reasoning by showing the safety of *findP*, a classic exercise in concurrency verification. It has been treated by at least:

- Karp and Miller (1969, Example 5.1)
- Rosen (1976, §3)
- Owicki and Gries (1976, §4 Example 2)
- Jones (1983, §2.4)
- Xu et al. (1994, §3.1)
- Brookes (1996, p161) (no proof)
- de Roever et al. (2001, Examples 3.57 and 8.26) (atomic guarded commands)
- Dingel (2002, §6.2) (refinement)
- Prensa Nieto (2003, §10) (mechanized, arbitrary number of threads)
- Apt, de Boer, and Olderog (2009, §7.4, §8.6)
- Hayes and Jones (2017, §4) (refinement)

We take the task to be of finding the first element of a given array A that satisfies a given predicate $pred$, if it exists, or yielding $length\ A$ if it does not. This search is performed with two threads: one searching the even indices and the other the odd. There is the possibility of a thread terminating early if it notices that the other thread has found a better candidate than it could.

We generalise previous treatments by allowing the predicate to be specified modularly and to be a function of the state. It is required to be pure, i.e., it cannot change the observable/shared state, though it could have its own local state.

Our search loops are defined recursively; one could just as easily use *prog.while*. We use a list and not an array for simplicity – at this level of abstraction there is no difference – and a mix of variables, where the monadic ones are purely local and the state-based are shared between the threads. The lens allows the array to be a value or reside in the (observable/shared) state.

type-synonym $'s\ state = (nat \times nat) \times 's$

abbreviation $foundE :: nat \implies 's\ state$ **where** $foundE \equiv fst_L ;_L fst_L$

abbreviation $foundO :: nat \implies 's\ state$ **where** $foundO \equiv snd_L ;_L fst_L$

context

fixes $pred :: 'a \Rightarrow ('s, bool)\ prog$

fixes $predPre :: 's\ pred$

fixes $predP :: 'a \Rightarrow 's\ pred$

fixes $A :: 's\ rel$

fixes $array :: 'a\ list \implies 's$

— A guarantee of Id indicates that $pred\ a$ is observationally pure.

assumes $iag\text{-}pred: \bigwedge a. prog.p2s\ (pred\ a) \leq \{\{predPre \wedge \langle a \rangle \in SET\ get_array\}\}, A^= \cap Id_{get_array} \cap ceilr\ predPre$
 $\cap Id_{predP\ a} \vdash Id, \{\{\lambda rv. \langle rv \rangle = predP\ a\}\}$

begin

abbreviation $array' :: 'a\ list \implies 's\ state$ **where** $array' \equiv array ;_L snd_L$

partial-function $(lfp)\ findP\text{-}loop\text{-}evens :: nat \Rightarrow ('s\ state, unit)\ prog$ **where**

$findP\text{-}loop\text{-}evens\ i =$

$do\ \{ fO \leftarrow prog.read\ get_{foundO}$

$;\ prog.whenM\ (i < fO)$

$(do\ \{ v \leftarrow prog.read\ (\lambda s. get_{array'}\ s\ !\ i)$

$;\ b \leftarrow prog.localize\ (pred\ v)$

$;\ if\ b\ then\ prog.write\ (\lambda s. put_{foundE}\ s\ i)\ else\ findP\text{-}loop\text{-}evens\ (i + 2)$

$\})$

$\}$

partial-function (*lfp*) *findP-loop-odds* :: *nat* \Rightarrow (*'s state, unit*) *prog where*

```

findP-loop-odds i =
  do { fE  $\leftarrow$  prog.read getfoundE
      ; prog.whenM (i < fE)
        (do { v  $\leftarrow$  prog.read ( $\lambda s.$  getarray' s ! i)
            ; b  $\leftarrow$  prog.localize (pred v)
            ; if b then prog.write ( $\lambda s.$  putfoundO s i) else findP-loop-odds (i + 2)
            })
      }

```

definition *findP* :: (*'s, nat*) *prog where*

```

findP = prog.local (
  do { N  $\leftarrow$  prog.read (SIZE getarray')
      ; prog.write ( $\lambda s.$  putfoundE s N)
      ; prog.write ( $\lambda s.$  putfoundO s N)
      ; (findP-loop-evens 0 || findP-loop-odds 1)
      ; fE  $\leftarrow$  prog.read (getfoundE)
      ; fO  $\leftarrow$  prog.read (getfoundO)
      ; prog.return (min fE fO)
    })

```

Relies and guarantees **abbreviation** (*input*) *A'* :: *'s rel where* $A' \equiv A^= \cap \text{ceilr } \text{predPre} \cap (\bigcap a. \text{Id}_{\text{predP } a})$

definition *AE* :: *'s state rel where*

$$AE = UNIV \times_R A' \cap \text{Id}_{\text{get}_{array'}} \cap \text{Id}_{\text{get}_{foundE}} \cap \leq_{\text{get}_{foundO}}$$

definition *GE* :: *'s state rel where*

$$GE = \text{Id}_{\text{snd}} \cap \text{Id}_{\text{get}_{foundO}} \cap \leq_{\text{get}_{foundE}}$$

definition *AO* :: *'s state rel where*

$$AO = UNIV \times_R A' \cap \text{Id}_{\text{get}_{array'}} \cap \text{Id}_{\text{get}_{foundO}} \cap \leq_{\text{get}_{foundE}}$$

definition *GO* :: *'s state rel where*

$$GO = \text{Id}_{\text{snd}} \cap \text{Id}_{\text{get}_{foundE}} \cap \leq_{\text{get}_{foundO}}$$

lemma *AG-refl-trans*:

shows

```

refl AE
refl AO
trans A  $\implies$  trans AE
trans A  $\implies$  trans AO
refl GE
refl GO
trans GE
trans GO

```

unfolding *AE-def* *AO-def* *GE-def* *GO-def*

by (*auto simp: refl-inter-conv refl-relprod-conv*
intro!: trans-Int refl-UnionI refl-INTER trans-INTER)

lemma *AG-containment*:

shows $GO \subseteq AE$
and $GE \subseteq AO$

by (*auto simp: AE-def AO-def GO-def GE-def refl-onD[OF ceilr.refl]*)

lemma *G-containment*:

shows $GE \cup GO \subseteq UNIV \times_R \text{Id}$

by (*fastforce simp: GE-def GO-def*)

Safety proofs lemma *ag-findP-loop-evens*:

shows *prog.p2s* (*findP-loop-evens i*)
 $\leq \{\langle \text{even } i \rangle \wedge (\lambda s. \text{predPre } (\text{snd } s)) \wedge \text{get}_{\text{foundE}} = \text{SIZE } \text{get}_{\text{array}'} \wedge \text{get}_{\text{foundO}} \leq \text{SIZE } \text{get}_{\text{array}'}\}, \text{AE} \vdash \text{GE},$

$$\{\lambda -. (\text{get}_{\text{foundE}} < \text{SIZE } \text{get}_{\text{array}'}) \longrightarrow \text{localize1 } \text{predP } \mathbf{\$ \$} \text{get}_{\text{array}' } ! \text{get}_{\text{foundE}}) \\ \wedge (\forall j. \langle i \leq j \wedge \text{even } j \rangle \wedge \langle j \rangle < \text{pred-min } \text{get}_{\text{foundE}} \text{get}_{\text{foundO}} \longrightarrow \neg \text{localize1 } \text{predP } \mathbf{\$ \$} \text{get}_{\text{array}'}$$

! $\langle j \rangle\}$

proof(*intro ag.gen-asm*,

induct arbitrary: i rule: findP-loop-evens.fixp-induct[case-names bot adm step])

case (*step R i*) **show** ?*case*

apply (*rule iag.init*)

apply (*rule iag.intro*)+

— else branch, recursive call

apply (*rename-tac v va vb*)

apply (*rule-tac P= $\langle va \rangle = \text{get}_{\text{array}' } ! \langle i \rangle \wedge \langle vb \rangle = \text{localize1 } \text{predP } va$*
in *iag.stable-augment[OF step.hyps]*)

apply (*simp add: $\langle \text{even } i \rangle$; fail*)

apply *clarsimp*

apply (*metis $\langle \text{even } i \rangle$ even-Suc less-Suc not-less*)

apply (*force simp: GE-def AE-def stable-def monotone-def*)

— *prog.localize (pred ...)*

apply (*rename-tac v va*)

apply (*rule-tac Q= $\lambda vb. (\lambda s. \text{predPre } (\text{snd } s)) \wedge \text{get}_{\text{foundE}} = \text{SIZE } \text{get}_{\text{array}' } \wedge \text{get}_{\text{foundO}} \leq \text{SIZE } \text{get}_{\text{array}' } \wedge \langle v \rangle \leq \text{SIZE } \text{get}_{\text{array}' } \wedge \langle va \rangle = \text{get}_{\text{array}' } ! \langle i \rangle \wedge \langle vb \rangle = \text{localize1 } \text{predP } va$*

in *ag.post-imp*)

apply (*clarsimp simp: GE-def exI[where $x = \langle i \rangle \leq \text{get}_{\text{foundE}}$]; fail*)

apply (*rule iag.pre-g[where $G' = GE$, OF iag.stable-augment-post[OF iag.augment-a[where $A' = AE$, OF ag.prog.localize-lift[OF iag-pred, simplified]]]]*)

apply (*fastforce simp: AE-def stable-def monotone-def*)

apply (*metis AG-refl-trans(5) refl-alt-def*)

apply (*rule iag.intro*)+

— precondition

apply *force*

— assume

apply (*intro conjI Int-greatest INT-greatest ceilr.largest*)

apply ((*fastforce simp: stable-def monotone-def AE-def*)+)[6]

apply (*clarsimp simp: stable-def monotone-def AE-def GE-def; rule exI[where $x = \langle i \rangle \leq \text{get}_{\text{foundE}}$]; clarsimp; metis*)

apply (*fastforce simp: stable-def monotone-def AE-def*)+

done

qed *simp-all*

lemma *ag-findP-loop-odds*:

shows *prog.p2s* (*findP-loop-odds i*)

$\leq \{\langle \text{odd } i \rangle \wedge (\lambda s. \text{predPre } (\text{snd } s)) \wedge \text{get}_{\text{foundO}} = \text{SIZE } \text{get}_{\text{array}' } \wedge \text{get}_{\text{foundE}} \leq \text{SIZE } \text{get}_{\text{array}'}\}, \text{AO} \vdash \text{GO},$

$$\{\lambda -. (\text{get}_{\text{foundO}} < \text{SIZE } \text{get}_{\text{array}'}) \longrightarrow \text{localize1 } \text{predP } \mathbf{\$ \$} \text{get}_{\text{array}' } ! \text{get}_{\text{foundO}})$$

$$\wedge (\forall j. \langle i \leq j \wedge \text{odd } j \rangle \wedge \langle j \rangle < \text{pred-min } \text{get}_{\text{foundE}} \text{get}_{\text{foundO}} \longrightarrow \neg \text{localize1 } \text{predP } \mathbf{\$ \$} \text{get}_{\text{array}'}$$

! $\langle j \rangle\}$

proof(*intro ag.gen-asm*,

induct arbitrary: i rule: findP-loop-odds.fixp-induct[case-names bot adm step])

case (*step R i*) **show** ?*case*

apply (*rule iag.init*)

apply (*rule iag.intro*)+

— else branch, recursive call

apply (*rename-tac v va vb*)

apply (*rule-tac P= $\langle va \rangle = \text{get}_{\text{array}' } ! \langle i \rangle \wedge \langle vb \rangle = \text{localize1 } \text{predP } va$*
in *iag.stable-augment[OF step.hyps]*)

apply (*simp add: $\langle \text{odd } i \rangle$; fail*)

```

apply clarsimp
apply (metis  $\langle \text{odd } i \rangle$  even-Suc less-Suc-eq not-less)
apply (force simp: GO-def AO-def stable-def monotone-def)
— prog.localize (pred ...)
apply (rename-tac v va)
apply (rule-tac  $Q = \lambda v b. (\lambda s. \text{predPre } (s \text{nd } s)) \wedge \text{get}_{\text{found}O} = \text{SIZE } \text{get}_{\text{array}' } \wedge \text{get}_{\text{found}E} \leq \text{SIZE } \text{get}_{\text{array}' } \wedge \langle v \rangle \leq \text{SIZE } \text{get}_{\text{array}' } \wedge \langle va \rangle = \text{get}_{\text{array}' } ! \langle i \rangle \wedge \langle vb \rangle = \text{localize1 } \text{predP } va$ )
in ag.post-imp)
apply (clarsimp simp: GO-def exI[where  $x = \langle i \rangle \leq \text{get}_{\text{found}O}$ ]; fail)
apply (rule iag.pre-g[where  $G' = GO, OF \text{ iag.stable-augment-post}[OF \text{ iag.augment-a}[\text{where } A' = AO, OF \text{ ag.prog.localize-lift}[OF \text{ iag-pred, simplified}]]]]$ )]])
apply (fastforce simp: AO-def stable-def monotone-def)
apply (metis AG-refl-trans(6) refl-alt-def)
apply (rule iag.intro)+
— precondition
apply force
— assume
apply (intro conjI Int-greatest INT-greatest ceilr.largest)
apply ((fastforce simp: AO-def stable-def monotone-def)+)[6]
apply (clarsimp simp: AO-def GO-def stable-def monotone-def; rule exI[where  $x = \langle i \rangle \leq \text{get}_{\text{found}O}$ ]; clarsimp; metis)
apply (fastforce simp: AO-def stable-def monotone-def)+
done
qed simp-all

```

theorem *ag-findP*:

shows *prog.p2s findP*

$\leq \{\text{predPre}\}, A' \cap \text{Id}_{\text{get}_{\text{array}}}$

$\vdash \text{Id}, \{\lambda v s. v = (\text{LEAST } i. i < \text{SIZE } \text{get}_{\text{array}} s \longrightarrow \text{predP } (\text{get}_{\text{array}} s ! i) s)\}$

unfolding *findP-def*

apply (*rule ag.prog.local*)

apply (*rule iag.init*)

apply (*rule iag.intro*)+

apply (*rule iag.augment-post-imp*[**where** $Q = \lambda v. \text{get}_{\text{found}E} \leq \text{SIZE } \text{get}_{\text{array}' } \wedge \text{get}_{\text{found}O} \leq \text{SIZE } \text{get}_{\text{array}' }$])

apply (*rule iag.pre-g*[*OF - G-containment*])

apply (*rule iag.stable-augment-frame*)

apply (*rule iag.parallel*[*OF ag-findP-loop-evens ag-findP-loop-odds - AG-containment order.refl*])

— postcondition from *iag.parallel*

apply *clarsimp*

apply (*rule Least-equality, linarith*)

subgoal for $x y s z$ **by** (*clarsimp simp: min-le-iff-disj not-less not-le dest!*: *spec*[**where** $x = z$])

— stability for *iag.stable-augment-frame*

apply (*force simp: stable-def monotone-def AE-def AO-def GE-def GO-def*)

apply (*rule iag.intro*)+

— precondition

apply *fastforce*

— assume

apply (*simp*;

intro conjI Int-greatest INT-greatest ceilr.largest;

fastforce simp: AE-def AO-def stable-def monotone-def)

done

end

We conclude by showing how we can instantiate the above with a *coprime* predicate.

setup $\langle \text{Sign.mandatory-path gcd} \rangle$

type-synonym $'s \text{ state} = (\text{nat} \times \text{nat}) \times 's$

abbreviation $x :: nat \Rightarrow 's \text{ gcd.state}$ **where** $x \equiv fst_L ;_L fst_L$
abbreviation $y :: nat \Rightarrow 's \text{ gcd.state}$ **where** $y \equiv snd_L ;_L fst_L$

definition $seq :: nat \Rightarrow nat \Rightarrow ('s, nat) \text{ prog}$ **where**

```
seq a b =
  prog.local (
    do { prog.write ( $\lambda s. put_{gcd.x} s a$ )
      ; prog.write ( $\lambda s. put_{gcd.y} s b$ )
      ; prog.while ( $\lambda -.$ 
        do {  $xv \leftarrow prog.read\ get_{gcd.x}$ 
          ;  $yv \leftarrow prog.read\ get_{gcd.y}$ 
          ; if  $xv = yv$ 
            then prog.return (Inr ())
            else (do { (if  $xv < yv$ 
                      then prog.write ( $\lambda s. put_{gcd.y} s (yv - xv)$ )
                      else prog.write ( $\lambda s. put_{gcd.x} s (xv - yv)$ ))
                ; prog.return (Inl ()) } )
        }) ()
    ; prog.read  $get_{gcd.x}$ 
  })
```

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ ag.gcd \rangle$

lemma seq :

shows $prog.p2s (gcd.seq a b) \leq \{\langle True \rangle\}$, $UNIV \vdash Id$, $\{\lambda v. \langle v = gcd a b \rangle\}$

unfolding $gcd.seq-def$

apply ($rule\ ag.prog.local$)

apply ($rule\ iag.init$)

apply ($rule\ iag.intro\ iag.while$ [**where** $I=\lambda - s. gcd (get_{gcd.x} s) (get_{gcd.y} s) = gcd a b$])+
 — precondition

apply ($auto\ simp: gcd-diff1-nat$)[1]

apply ($metis\ gcd.commute\ gcd-diff1-nat\ less-or-eq-imp-le$)

— assume

apply ($intro\ stable.intro\ stable.local-only\ INFI\ infI$)

apply $auto$

done

setup $\langle Sign.parent-path \rangle$

definition $findP-coprime :: (nat \times nat\ list, nat) \text{ prog}$ **where**

$findP-coprime = findP (\lambda a. prog.read\ get_{fst_L} \ggg gcd.seq\ a \ggg (\lambda c. prog.return (c = 1)))\ snd_L$

lemma $ag-findP-coprime'$:

shows $prog.p2s\ findP-coprime$

$\leq \{\langle True \rangle\}$, Id

$\vdash Id$, $\{\lambda rv\ s. rv = (LEAST\ i. i < length (get_{snd_L} s) \longrightarrow coprime (get_{fst_L} s) (get_{snd_L} s ! i))\}$

unfolding $findP-coprime-def$

apply ($rule\ iag.init$)

apply ($rule\ ag-findP$ [**where** $A=Id$ **and** $array=snd_L$ **and** $predP=\lambda b\ s. coprime (get_{fst_L} s) b$ **and** $pred-Pre=\langle True \rangle$])

apply ($rule\ iag.init$)

apply ($rule\ iag.intro$)+

apply ($rule-tac\ Q=\langle v \rangle = get_{fst_L}$) **in** $iag.augment-post-imp$)

apply ($rule\ iag.stable-augment-frame$)

apply ($rule\ iag.pre$ [$OF\ ag.gcd.seq$, **where** $A'=Id$ **and** $P'=\langle True \rangle$, $simplified$, $OF\ order.refl$])

```

apply (clarsimp simp: ac-simps coprime-iff-gcd-eq-1 simp flip: One-nat-def; fail)
apply (force simp: stable-def monotone-def)
apply (rule iag.intro)+
apply (simp; intro conjI INT-greatest ceilr.largest; fastforce simp: stable-def monotone-def)+
done

```

lemma *ag-findP-coprime*: — Shuffle the parameter to the precondition, exploiting purity.

```

shows prog.p2s findP-coprime
  ≤ {⟦⟨a⟩ = getfstL⟧, Id
    ⊢ Id, {⟦λrv s. rv = (LEAST i. i < length (getsndL s) → coprime a (getsndL s ! i))⟧}}
apply (rule ag.pre-pre)
apply (rule ag.stable-augment-post[OF ag-findP-coprime])
apply (fastforce simp: stable-def)+
done

```

23 Example: data refinement (search)

We show a very simple example of data refinement: implementing sets with functional queues for breadth-first search (BFS). The objective here is to transfer a simple correctness property proven on the abstract level to the concrete level.

Observations:

- there is no concurrency in the BFS: this is just about data refinement
 - however arbitrary interference is allowed
- the abstract level does not require the implementation of the pending set to make progress
- the concrete level does not require a representation invariant
- depth optimality is not shown

References:

- queue ADT: \$ISABELLE_HOME/src/Doc/Codegen/Introduction.thy
- BFS verification:
 - J. C. Filliâtre http://toccata.lri.fr/gallery/vstte12_bfs.en.html
 - \$AFP/Refine_Monadic/examples/Breadth_First_Search.thy
 - our model is quite different

```

setup <Sign.mandatory-path pending>

```

```

record ('a, 's) interface =
  init :: ('s, unit) prog
  enq :: 'a ⇒ ('s, unit) prog
  deq :: ('s, 'a option) prog

```

```

type-synonym 'a abstract = 'a set

```

definition *abstract* :: ('a, 'a pending.abstract × 's) pending.interface **where**

```

abstract =
  (⟦ pending.interface.init = prog.write (map-prod {⟦⟧} id)
    , pending.interface.enq = λx. prog.write (map-prod (insert x) id)
    , pending.interface.deq = prog.action (⟦(None, s, s) | s. fst s = {⟦⟧}
      ∪ ⟨⟦(Some x, (insert x X, s), (X, s)) | X s x. True⟧⟩
    ⟦

```

type-synonym $'a$ concrete = $'a$ list \times $'a$ list — a queue

fun cdeq-update :: $'a$ pending.concrete \times $'s \Rightarrow 'a$ option \times $'a$ pending.concrete \times $'s$ **where**
 cdeq-update (([], []), s) = (None, (([], []), s))
 | cdeq-update ((xs, []), s) = cdeq-update (([], rev xs), s)
 | cdeq-update ((xs, y # ys), s) = (Some y, ((xs, ys), s))

definition concrete :: ($'a$, $'a$ pending.concrete \times $'s$) pending.interface **where**
 concrete =
 (| pending.interface.init = prog.write (map-prod <{[], []}> id)
 , pending.interface.eng = λx . prog.write (map-prod (map-prod ((#) x) id) id)
 , pending.interface.deq = prog.det-action pending.cdeq-update
 |)

abbreviation absfn' :: $'a$ pending.concrete $\Rightarrow 'a$ list **where** — queue as a list
 absfn' s \equiv snd s @ rev (fst s)

definition absfn :: $'a$ pending.concrete $\Rightarrow 'a$ pending.abstract **where**
 absfn s = set (pending.absfn' s)

setup <Sign.mandatory-path ag>

lemma init:

fixes Q :: unit $\Rightarrow 'a$ pending.abstract \times $'s \Rightarrow$ bool
fixes A :: $'s$ rel
assumes stable (Id \times_R A) (Q ())
shows prog.p2s (pending.init pending.abstract) \leq $\{\lambda s$. Q () ({} , snd s) $\}$, Id \times_R A \vdash UNIV \times_R Id, $\{Q\}$
using assms **by** (auto intro: ag.prog.action simp: pending.abstract-def stable-def monotone-def)

lemma enq:

fixes x :: $'a$
fixes Q :: unit $\Rightarrow 'a$ pending.abstract \times $'s \Rightarrow$ bool
fixes A :: $'s$ rel
assumes stable (Id \times_R A) (Q ())
shows prog.p2s (pending.enq pending.abstract x) \leq $\{\lambda s$. Q () (insert x (fst s), snd s) $\}$, Id \times_R A \vdash UNIV \times_R Id, $\{Q\}$
using assms **by** (auto intro: ag.prog.action simp: pending.abstract-def stable-def monotone-def)

lemma deq:

fixes Q :: $'a$ option $\Rightarrow 'a$ pending.abstract \times $'s \Rightarrow$ bool
fixes A :: $'s$ rel
assumes $\bigwedge v$. stable (Id \times_R A) (Q v)
shows prog.p2s (pending.deq pending.abstract) \leq $\{\lambda s$. if fst s = {} then Q None s else ($\forall x X$. fst s = insert x X \longrightarrow Q (Some x) (X, snd s)) $\}$, Id \times_R A \vdash UNIV \times_R Id, $\{Q\}$
unfolding pending.abstract-def pending.interface.simps
by (rule ag.prog.action)
 (use assms **in** <auto simp: stable-def monotone-def le-bool-def split: if-split-asm>)

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.mandatory-path set>

record ($'a$, $'s$) interface =
 init :: ($'s$, unit) prog
 ins :: $'a \Rightarrow ('s$, unit) prog

$mem :: 'a \Rightarrow ('s, bool) prog$

type-synonym $'a abstract = 'a list$ — model finite sets

definition $abstract :: ('a, 's \times 'a set.abstract \times 't) set.interface$ **where**

$abstract =$
 \langle $set.interface.init = prog.write (map-prod id (map-prod \langle [] \rangle id))$
 $, set.interface.ins = \lambda x. prog.write (map-prod id (map-prod ((\#) x) id))$
 $, set.interface.mem = \lambda x. prog.read (\lambda s. x \in set (fst (snd s)))$
 \rangle

setup $\langle Sign.mandatory-path ag \rangle$

lemma $init:$

fixes $Q :: unit \Rightarrow 's \times 'a set.abstract \times 't \Rightarrow bool$

fixes $A :: 's rel$

assumes $stable (A \times_R Id \times_R B) (Q ())$

shows $prog.p2s (set.init set.abstract) \leq \{\lambda s. Q () (fst s, [], snd (snd s))\}, A \times_R Id \times_R B \vdash Id \times_R UNIV \times_R Id, \{Q\}$

using $assms$ **by** $(auto intro: ag.prog.action simp: set.abstract-def stable-def monotone-def)$

lemma $ins:$

fixes $x :: 'a$

fixes $Q :: unit \Rightarrow 's \times 'a set.abstract \times 't \Rightarrow bool$

fixes $A :: 's rel$

assumes $stable (A \times_R Id \times_R B) (Q ())$

shows $prog.p2s (set.ins set.abstract x) \leq \{\lambda s. Q () (fst s, x \# fst (snd s), snd (snd s))\}, A \times_R Id \times_R B \vdash Id \times_R UNIV \times_R Id, \{Q\}$

using $assms$ **by** $(auto intro: ag.prog.action simp: set.abstract-def stable-def monotone-def)$

lemma $mem:$

fixes $Q :: bool \Rightarrow 's \times 'a set.abstract \times 't \Rightarrow bool$

assumes $\bigwedge v. stable (A \times_R Id \times_R B) (Q v)$

shows $prog.p2s (set.mem set.abstract x) \leq \{\lambda s. Q (x \in set (fst (snd s))) s\}, A \times_R Id \times_R B \vdash Id \times_R UNIV \times_R Id, \{Q\}$

using $assms$ **by** $(auto intro: ag.prog.action simp: set.abstract-def stable-def monotone-def)$

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

context

fixes $pending :: ('a, 'p \times 'a set.abstract \times 's) pending.interface$

fixes $f :: 'a \Rightarrow 'a list$

begin

definition $loop :: 'a pred \Rightarrow ('p \times 'a set.abstract \times 's, 'a option) prog$ **where**

$loop p =$

$prog.while (\lambda -.$

$do \{ aopt \leftarrow pending.deq pending$

$; case aopt of$

$None \Rightarrow prog.return (Inr None)$

$| Some x \Rightarrow$

$if p x$

$then prog.return (Inr (Some x))$

$else do \{ prog.app (\lambda y. do \{ b \leftarrow set.mem set.abstract y;$

$prog.unlessM b (do \{ set.ins set.abstract y$

$; pending.enq pending y \}) \}$

```

                (f x)
            ; prog.return (Inl ())
        }
    }) ()

```

definition *main* :: 'a pred ⇒ 'a ⇒ ('p × 'a set.abstract × 's, 'a option) prog **where**

```

main p x =
  do {
    set.init set.abstract
  ; pending.init pending
  ; set.ins set.abstract x
  ; pending.enq pending x
  ; loop p
  }

```

definition *search* :: 'a pred ⇒ 'a ⇒ ('s, 'a option) prog **where**

```

search p x = prog.local (prog.local (main p x))

```

end

abbreviation (input) *aloop* ≡ loop pending.abstract

abbreviation (input) *amain* ≡ main pending.abstract

abbreviation (input) *asearch* ≡ search pending.abstract

abbreviation (input) *bfs* ≡ search pending.concrete

lemma

shows *pending-g*: $UNIV \times_R Id \subseteq UNIV \times_R UNIV \times_R Id$

and *set-g*: $Id \times_R UNIV \times_R Id \subseteq UNIV \times_R UNIV \times_R Id$

by *fastforce+*

context

fixes *f* :: 'a ⇒ 'a list

fixes *P* :: 'a pred

fixes *x*₀ :: 'a

begin

abbreviation (input) *step* :: 'a rel **where**

```

step ≡ {(x, y). y ∈ set (f x)}

```

abbreviation (input) *path* :: 'a rel **where**

```

path ≡ step*

```

definition *aloop-invP* :: 'a pending.abstract ⇒ 'a set.abstract ⇒ bool **where**

```

aloop-invP q v ⟷

```

```

  q ⊆ set v

```

```

  ∧ set v ⊆ path “ {x0}

```

```

  ∧ set v ∩ Collect P ⊆ q

```

```

  ∧ x0 ∈ set v

```

definition *vclosureP* :: 'a ⇒ 'a pending.abstract ⇒ 'a set.abstract ⇒ bool **where**

```

vclosureP x q v ⟷ (x ∈ set v - q ⟶ step “ {x} ⊆ set v)

```

definition *search-postP* :: 'a option ⇒ bool **where**

```

search-postP rv = (case rv of

```

```

  None ⇒ finite (path “ {x0}) ∧ (path “ {x0} ∩ Collect P = {})

```

```

  | Some y ⇒ (x0, y) ∈ path ∧ P y)

```

abbreviation *aloop-inv s* ≡ *aloop-invP* (fst s) (fst (snd s))

abbreviation $vclosure\ x\ s \equiv vclosureP\ x\ (fst\ s)\ (fst\ (snd\ s))$

abbreviation $search\text{-}post\ rv \equiv \langle search\text{-}postP\ rv \rangle$

lemma $vclosureP\text{-}closed$:

assumes $set\ v \subseteq path\ \{\{x_0\}\}$

assumes $\forall y. vclosureP\ y\ \{\} v$

assumes $x_0 \in set\ v$

shows $path\ \{\{x_0\}\} = set\ v$

proof –

have $y \in set\ v$ **if** $(x_0, y) \in path$ **for** y

using $that\ assms(2,3)$ **by** $induct\ (auto\ simp: vclosureP\text{-}def)$

with $assms(1)$ **show** $?thesis$

by $fast$

qed

lemma $vclosureP\text{-}app$:

assumes $\forall y. x \neq y \longrightarrow local.vclosureP\ y\ q\ v$

assumes $set\ (f\ x) \subseteq set\ v$

shows $vclosureP\ y\ q\ v$

using $assms$ **by** $(fastforce\ simp: vclosureP\text{-}def)$

lemma $vclosureP\text{-}init$:

shows $vclosureP\ x\ \{x_0\}\ [x_0]$

by $(simp\ add: vclosureP\text{-}def)$

lemma $vclosureP\text{-}step$:

assumes $\forall z. x \neq z \longrightarrow vclosureP\ z\ q\ v$

assumes $x \neq z$

shows $vclosureP\ z\ (insert\ y\ q)\ (y \# v)$

using $assms$ **by** $(fastforce\ simp: vclosureP\text{-}def)$

lemma $vclosureP\text{-}dequeue$:

assumes $\forall z. vclosureP\ z\ (insert\ x\ q)\ v$

assumes $x \neq z$

shows $vclosureP\ z\ q\ v$

using $assms$ **by** $(fastforce\ simp: vclosureP\text{-}def)$

lemma $aloop\text{-}invP$:

assumes $aloop\text{-}invP\ q\ v$

assumes $x \in q$

shows $(x_0, x) \in path$

using $assms$ **by** $(auto\ simp: aloop\text{-}invP\text{-}def)$

lemma $aloop\text{-}invP\text{-}init$:

shows $aloop\text{-}invP\ \{x_0\}\ [x_0]$

by $(simp\ add: aloop\text{-}invP\text{-}def)$

lemma $aloop\text{-}invP\text{-}step$:

assumes $aloop\text{-}invP\ q\ v$

assumes $(x_0, x) \in path$

assumes $y \in set\ (f\ x) - set\ v$

shows $aloop\text{-}invP\ (insert\ y\ q)\ (y \# v)$

using $assms$ **by** $(auto\ simp: aloop\text{-}invP\text{-}def\ elim: rtrancl\text{-}into\text{-}rtrancl)$

lemma $aloop\text{-}invP\text{-}dequeue$:

assumes $aloop\text{-}invP\ (insert\ x\ q)\ v$

assumes $\neg P\ x$

shows $aloop\text{-}invP\ q\ v$

using *assms* by (*auto simp: aloop-invP-def*)

lemma *search-postcond-None*:

assumes *aloop-invP* $\{ \} v$

assumes $\forall y. \text{vclosureP } y \{ \} v$

shows *search-postP* *None*

using *assms* by (*auto simp: search-postP-def aloop-invP-def dest: vclosureP-closed*)

lemma *search-postcond-Some*:

assumes *aloop-invP* $q v$

assumes $x \in q$

assumes $P x$

shows *search-postP* (*Some* x)

using *assms* by (*auto simp: search-postP-def aloop-invP-def*)

lemmas *stable-simps* =

prod.sel

split-def

sum.simps

lemma *ag-aloop*:

shows $\text{prog.p2s } (aloop f P) \leq \{ \{ aloop\text{-inv} \wedge (\forall x. \text{vclosure } x) \} \}, Id \times_R Id \times_R UNIV \vdash UNIV \times_R UNIV \times_R Id,$
 $\{ \{ search\text{-post} \} \}$

unfolding *loop-def*

apply (*rule ag.prog.while*[*OF - - stable.Id-fst-fst-snd*])

apply (*rule ag.prog.bind*)

apply (*rule ag.prog.case-option*)

apply (*rule ag.prog.return*)

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.prog.if*)

apply (*rule ag.prog.return*)

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.prog.bind*)

apply (*rule ag.prog.return*)

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rename-tac* x)

apply (*rule-tac* $Q = \lambda-. aloop\text{-inv} \wedge (\forall y. \langle x \neq y \rangle \longrightarrow \text{vclosure } y) \wedge (\lambda s. \text{set } (f x) \subseteq \text{set } (fst (snd s)) \wedge (x_0, x)$
 $\in \text{path})$ **in** *ag.post-imp*)

apply (*force elim: vclosureP-app*)

apply (*rule ag.prog.app*)

apply (*rule ag.prog.bind*)

apply (*rule ag.prog.if*)

apply (*rule ag.prog.return*)

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.prog.bind*)

apply (*rule ag.pre-g*[*OF pending.ag.enq pending-g*])

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.pre-g*[*OF set.ag.ins set-g*])

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.pre-pre*[*OF ag.pre-g*[*OF set.ag.mem set-g*]])

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*force simp: aloop-invP-step vclosureP-step*)

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.pre-g*[*OF pending.ag.deq pending-g*])

apply ((*simp only: stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*auto elim: search-postcond-Some search-postcond-None aloop-invP-dequeue*
aloop-invPD vclosureP-dequeue)

done

lemma *ag-amain*:

shows $\text{prog.p2s } (\text{amain } f P x_0) \leq \{\langle \text{True} \rangle\}, \text{Id} \times_R \text{Id} \times_R \text{UNIV} \vdash \text{UNIV} \times_R \text{UNIV} \times_R \text{Id}, \{\text{search-post}\}$
unfolding *main-def*
apply (*rule ag.pre-pre*)
apply (*rule ag.prog.bind*)
apply (*rule ag.aloop*)
apply (*rule ag.post-imp*[**where** $Q = \langle \lambda(q, v, -). q = \{x_0\} \wedge v = [x_0] \rangle$])
apply (*clarsimp simp: aloop-invP-init vclosureP-init; fail*)
apply (*rule ag.pre-g*[*OF pending.ag.enq pending-g*])
apply (*(simp only: stable-simps)?; (rule stable.intro)+; fail*)
apply (*rule ag.pre-g*[*OF set.ag.ins set-g*])
apply (*(simp only: stable-simps)?; (rule stable.intro)+; fail*)
apply (*rule ag.pre-g*[*OF pending.ag.init pending-g*])
apply (*(simp only: stable-simps)?; (rule stable.intro)+; fail*)
apply (*rule ag.pre-g*[*OF set.ag.init set-g*])
apply (*(simp only: stable-simps)?; (rule stable.intro)+; fail*)
apply *simp*
done

lemma *ag-asearch*:

shows $\text{prog.p2s } (\text{asearch } f P x_0 :: ('s, 'a \text{ option}) \text{ prog}) \leq \{\langle \text{True} \rangle\}, \text{UNIV} \vdash \text{Id}, \{\text{search-post}\}$
unfolding *search-def* **by** (*rule ag.prog.local ag-amain*)**+**

Refinement abbreviation $A \equiv \text{ag.assm } (\text{Id} \times_R \text{Id} \times_R \text{UNIV})$

abbreviation $\text{absfn } c \equiv \text{prog.sinvmap } (\text{map-prod } \text{pending.absfn } \text{id}) c$

abbreviation $\text{p2s-absfn } c \equiv \text{prog.p2s } (\text{absfn } c)$

— visited set: reflexive

lemma *ref-set-init*:

shows $\text{prog.p2s } (\text{set.init } \text{set.abstract}) \leq \{\lambda s. \text{True}\}, A \Vdash \text{p2s-absfn } (\text{set.init } \text{set.abstract}), \{\lambda v s. \text{True}\}$
by (*auto simp: set.abstract-def intro: rair.prog.action stable.intro*)

lemma *ref-set-ins*:

shows $\text{prog.p2s } (\text{set.ins } \text{set.abstract } x) \leq \{\lambda s. \text{True}\}, A \Vdash \text{p2s-absfn } (\text{set.ins } \text{set.abstract } x), \{\lambda v s. \text{True}\}$
by (*auto simp: set.abstract-def intro: rair.prog.action stable.intro*)

lemma *ref-set-mem*:

shows $\text{prog.p2s } (\text{set.mem } \text{set.abstract } x) \leq \{\lambda s. \text{True}\}, A \Vdash \text{p2s-absfn } (\text{set.mem } \text{set.abstract } x), \{\lambda v s. \text{True}\}$
by (*auto simp: set.abstract-def intro: rair.prog.action stable.intro*)

— queue

lemma *ref-queue-init*:

shows $\text{prog.p2s } (\text{pending.init } \text{pending.concrete}) \leq \{\lambda s. \text{True}\}, A \Vdash \text{p2s-absfn } (\text{pending.init } \text{pending.abstract}), \{\lambda v s. \text{True}\}$
by (*auto simp: pending.abstract-def pending.concrete-def pending.absfn-def intro: rair.prog.action stable.intro*)

lemma *ref-queue-enq*:

shows $\text{prog.p2s } (\text{pending.enq } \text{pending.concrete } x) \leq \{\lambda s. \text{True}\}, A \Vdash \text{p2s-absfn } (\text{pending.enq } \text{pending.abstract } x), \{\lambda v s. \text{True}\}$
by (*auto simp: pending.abstract-def pending.concrete-def pending.absfn-def intro: rair.prog.action stable.intro*)

lemma *ref-queue-deq*:

shows $\text{prog.p2s } (\text{pending.deq } \text{pending.concrete}) \leq \{\lambda s. \text{True}\}, A \Vdash \text{p2s-absfn } (\text{pending.deq } \text{pending.abstract}), \{\lambda v s. \text{True}\}$
by (*auto simp: pending.abstract-def pending.concrete-def pending.absfn-def*
intro: rair.prog.action stable.intro elim!: pending.cdeq-update.elims[OF sym])

— program

lemma *ref-bfs-loop*:

shows $prog.p2s (loop\ pending.concrete\ f\ P) \leq \{\lambda s. True\}, A \Vdash p2s-absfn (loop\ pending.abstract\ f\ P), \{\lambda v\ s. True\}$

apply (*simp add: loop-def*)

apply (*rule rair.prog.while*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-queue-deq*)

apply (*rule refinement.pre-pre[OF rair.prog.case-option]*)

apply (*rule rair.prog.return*)

apply (*((simp only: stable-simps)?; (rule stable.intro)+; fail)*)

apply (*rule rair.prog.if*)

apply (*rule rair.prog.return*)

apply (*((simp only: stable-simps)?; (rule stable.intro)+; fail)*)

apply (*rule rair.prog.rev-bind*)

apply (*rule rair.prog.app*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-set-mem*)

apply (*rule refinement.pre-pre[OF rair.prog.if]*)

apply (*rule rair.prog.return*)

apply (*((simp only: stable-simps)?; (rule stable.intro)+; fail)*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-set-ins*)

apply (*rule ref-queue-enq*)

apply (*simp; fail*)

apply (*((simp only: stable-simps)?; (rule stable.intro)+; fail)*)

apply (*rule refinement.pre-pre[OF rair.prog.return]*)

apply (*((simp only: stable-simps)?; (rule stable.intro)+; fail)*)

apply (*auto intro: stable.intro split: option.split*)

done

lemma *ref-bfs-main*:

shows $prog.p2s (main\ pending.concrete\ f\ P\ x) \leq \{\langle True \rangle\}, A \Vdash p2s-absfn (amain\ f\ P\ x), \{\lambda v\ s. True\}$

apply (*simp add: main-def*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-set-init*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-queue-init*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-set-ins*)

apply (*rule rair.prog.rev-bind*)

apply (*rule ref-queue-enq*)

apply (*rule ref-bfs-loop*)

done

theorem *ref-bfs*:

shows $bfs\ f\ P\ x \leq asearch\ f\ P\ x$

unfolding *search-def*

apply (*intro refinement.prog.leI refinement.prog.data[where sf=id]*)

apply (*simp add: spec.invmap.id spec.localizeA.top*)

apply (*rule refinement.prog.data[where sf=pending.absfn]*)

apply (*simp flip: prog.p2s.invmap*)

apply (*rule refinement.pre-a[OF ref-bfs-main]*)

apply (*auto simp: spec.localizeA-def spec.invmap.rel*
simp flip: spec.rel.inf
intro: spec.rel.mono)

done

theorem *bfs-post-le*:

shows $\text{prog.p2s } (bfs\ f\ P\ x_0) \leq \text{spec.post } (\text{search-post})$
apply (*strengthen ord-to-strengthen*[*OF ref-bfs*])
apply (*strengthen ord-to-strengthen*(1)[*OF ag-asearch*])
apply (*simp add: ag-def spec.rel.UNIV flip: Sigma-Un-distrib1*)
done

end

24 Observations about safety closure

We demonstrate that *Sup* does not distribute in $(\prime a, \prime s, \prime v)$ *tls* as it does in $(\prime a, \prime s, \prime v)$ *spec*: specifically a *Sup* of a set of safety properties in the former need not be a safety property, whereas in the latter it is (see §8.2).

corec *bnats* :: $\text{nat} \Rightarrow (\prime a \times \text{nat}, \prime v)$ *tlist* **where**
bnats *i* = *TCons* (*undefined*, *i*) (*bnats* (*Suc* *i*))

definition *bnat* :: $(\prime a, \text{nat}, \prime v)$ *behavior.t* **where**
bnat = *behavior.B* 0 (*bnats* 1)

definition *tnats* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow (\prime a \times \text{nat})$ *list* **where**
tnats *i* *j* = *map* (*Pair* *undefined*) (*upt* *i* *j*)

definition *tnat* :: $\text{nat} \Rightarrow (\prime a, \text{nat}, \prime v)$ *trace.t* **where**
tnat *i* = *trace.T* 0 (*tnats* (*Suc* 0) (*Suc* *i*)) *None*

lemma *tnat-simps*[*simp*]:

shows *tnats* *i* *i* = []
and *trace.init* (*tnat* *i*) = 0
and *trace.rest* (*tnat* *i*) = *tnats* 1 (*Suc* *i*)
and *length* (*tnats* *i* *j*) = *j* - *i*
by (*simp-all add: tnats-def tnat-def*)

lemma *take-tnats*:

shows *take* *i* (*tnats* *j* *k*) = *tnats* *j* (*min* (*i* + *j*) *k*)
by (*simp add: tnats-def take-map add commute split: split-min*)

lemma *take-tnat*:

shows *trace.take* *i* (*tnat* *j*) = *tnat* (*min* *i* *j*)
by (*simp add: trace.take-def take-tnats tnat-def*)

lemma *mono-tnat*:

shows *mono* *tnat*
by (*rule monoI*) (*auto simp: trace.less-eq-take-def take-tnat split: split-min*)

lemma *final'-tnats*:

shows *trace.final'* *i* (*tnats* *j* *k*) = (*if* *j* < *k* *then* *k* - 1 *else* *i*)
by (*simp add: tnats-def trace.final'-def comp-def*)

lemma *sset-tnat*:

shows *trace.sset* (*tnat* *i*) = {*j*. *j* ≤ *i*}
by (*force simp: tnat-def tnats-def trace.sset.simps*)

lemma *natural'-tnats*:

shows *trace.natural'* *i* (*tnats* (*Suc* *i*) (*Suc* *j*)) = *tnats* (*Suc* *i*) (*Suc* *j*)
proof -
have *trace.natural'* *i* (*map* (*Pair* *undefined*) (*upt* (*Suc* *i*) (*Suc* *j*)))
= *map* (*Pair* *undefined*) (*upt* (*Suc* *i*) (*Suc* *j*)) **for** *j*

by (induct j arbitrary: i) (simp-all add: trace.natural'.append)
 from this show ?thesis unfolding tnats-def .
 qed

lemma natural-tnat:

shows $\Downarrow(\text{tnat } i) = \text{tnat } i$

by (simp add: tnat-def trace.natural-def natural'-tnats del: upt-Suc)

lemma ttake-bnats:

shows $\text{ttake } i (\text{bnats } j) = (\text{tnats } j (i + j), \text{None})$

by (induct i arbitrary: j) (subst bnats.code; simp add: tnats-def upt-rec)+

lemma take-bnat-tnat:

shows $\text{behavior.take } i \text{ bnat} = \text{tnat } i$

by (simp add: bnat-def tnat-def behavior.take-def ttake-bnats)

unbundle tls.extra-syntax

definition bnat-approx :: (unit, nat, unit) spec set **where**

$\text{bnat-approx} = \{\Downarrow(\text{behavior.take } i \text{ bnat}) \mid i. \text{True}\}$

lemma bnat-approx-alt-def:

shows $\text{bnat-approx} = \{\Downarrow(\text{tnat } i) \mid i. \text{True}\}$

by (simp add: bnat-approx-def take-bnat-tnat)

lemma not-tls-from-spec-Sup-distrib: — tls.from-spec is not *Sup*-distributive

shows $\neg \text{tls.from-spec} (\bigsqcup \text{bnat-approx}) \leq \bigsqcup (\text{tls.from-spec } ' \text{ bnat-approx})$ (is $\neg ?lhs \leq ?rhs$)

proof –

have $\Downarrow(\text{bnat})_T \leq ?lhs$

proof –

have *: $\exists j. \text{behavior.take } i \omega \in \text{raw.spec.cl } \{\text{behavior.take } j \text{ bnat}\}$ if $\text{bnat} \simeq_T \omega$

for i and $\omega :: ('a, \text{nat}, 'b) \text{behavior.t}$

by (rule behavior.stuttering.equiv.takeE[OF sym[OF that], where $i=i$])

(force simp: raw.spec.cl-def simp flip: trace.stuttering.cl.downwards.cl)

note spec.singleton.transfer[transfer-rule]

show ?thesis

unfolding bnat-approx-def

by transfer

(force dest: * simp: TLS.raw.singleton-def raw.from-spec-def Safety-Logic.raw.singleton-def
 simp flip: ex-simps elim!: behavior.stuttering.clE)

qed

moreover

have $\neg(\forall j. \text{tnat } j \leq \text{tnat } i)$ for i

by (auto intro!: exI[**where** $x=\text{Suc } i$] dest!: monoD[OF trace.sset.mono] simp: sset-tnat)

then have $\neg \Downarrow(\text{bnat})_T \leq ?rhs$

by (fastforce simp: bnat-approx-def tls.singleton.le-conv spec.singleton-le-conv take-bnat-tnat natural-tnat)

ultimately show ?thesis

by (blast dest: order.trans)

qed

definition bnat' :: (unit, nat, unit) tls set **where**

$\text{bnat}' = \text{tls.from-spec } ' \text{ bnat-approx}$

lemma not-tls-safety-cl-Sup-distrib: — tls.safety.cl is not *Sup*-distributive

shows $\neg \text{tls.safety.cl} (\bigsqcup \text{bnat}') \leq \bigsqcup (\text{tls.safety.cl } ' \text{ bnat}')$

proof –

have $(\bigsqcup x \in \text{bnat-approx}. \text{tls.to-spec } (\text{tls.from-spec } x)) = \bigsqcup \text{bnat-approx}$ (is $?lhs = ?rhs$)

proof(rule antisym)

```

show ?lhs ≤ ?rhs
  by (simp add: Sup-upper2 tls.safety.lower-upper-contractive)
have ∃ω ia ib. (∀i. † (behavior.take i ω) ≤ tnat ib) ∧ tnat i = behavior.take ia ω
for i
  by (rule exI[where x=behavior.B 0 (tshift2 (tnats (Suc 0) (Suc i), None) (trepeat (undefined, i)))]])
    (force simp: behavior.take.tshift ttake-trepeat trace.take.continue take-tnat
      trace.natural.continue trace.natural'.replicate natural-tnat not-le final'-tnats
      simp flip: tnat-def
      split: split-min
      intro: monoD[OF mono-tnat])
then show ?rhs ≤ ?lhs
  by (clarsimp simp: bnat-approx-alt-def spec.singleton.le-conv tls.singleton.le-conv
    spec.singleton-le-conv natural-tnat
    simp flip: ex-simps;
    fast)
qed
then show ?thesis
  by (simp add: bnat'-def tls.safety.cl-def tls.safety.upper-lower-upper tls.to-spec.Sup
    not-tls-from-spec-Sup-distrib image-image)
qed

```

definition *cl-bnat'* :: (unit, nat, unit) tls set **where**
cl-bnat' = tls.safety.cl ' bnat'

lemma *not-tls-safety-closed-Sup*:

shows *cl-bnat'* ⊆ *tls.safety.closed*
and ⊔ *cl-bnat'* ∉ *tls.safety.closed*
unfolding *cl-bnat'-def*

using *not-tls-safety-cl-Sup-distrib*

by (blast intro: tls.safety.expansive complete-lattice-class.Sup-mono
 dest: tls.safety.least[rotated, **where** x=⊔ bnat'])+

Negation does not preserve *tls.safety.closed* **notepad**
begin

have *tls.always (tls.state-prop id) ∈ tls.safety.closed*
by (simp add: tls.safety.closed.always tls.safety.closed.state-prop)

have \neg *tls.always (tls.state-prop id) ∈ tls.safety.closed*

proof –

let ?P = behavior.B True (trepeat (undefined, True)) :: ('a, bool, 'c) behavior.t

have ∃ω'. behavior.take i ?P = behavior.take i ω'
 ∧ (∃j ω''. behavior.dropn j ω' = Some ω'' ∧ ¬ behavior.init ω'')

for i

by (auto simp: behavior.dropn.continue behavior.take.continue behavior.take.trepeat
 trace.take.replicate case-tllist-trepeat
 exI[**where** x=behavior.take i ?P @-_B trepeat (undefined, False)]
 exI[**where** x=Suc i])

then have ⟨?P⟩_T ≤ *tls.safety.cl* (\neg *tls.always (tls.state-prop id)*)

by (clarsimp simp: tls.singleton.le-conv; fast)

moreover

have behavior.init ω'

if behavior.dropn i (behavior.B True (trepeat (undefined, True))) = Some ω'

for i **and** ω' :: ('a, bool, 'c) behavior.t

using that

by (cases i) (auto simp: behavior.dropn-alt-def tdropn-trepeat case-tllist-trepeat)

then have \neg ⟨?P⟩_T ≤ \neg *tls.always (tls.state-prop id)*

by (auto simp: tls.singleton.le-conv)

ultimately show *?thesis*
 using *tls.safety.le-closedE* by *blast*
 qed
 end

24.1 Liveness

Famously arbitrary properties on infinite sequences can be decomposed into *safety* and *liveness* properties. The latter have been identified with the topologically dense sets.

References:

- [Alpern and Schneider \(1985\)](#); [Schneider \(1987\)](#): topological account
- [Kindler \(1994\)](#): overview
- [Lynch \(1996, §8.5.3\)](#)
- [Manolios and Trefler \(2003\)](#): lattice-theoretic account
- [Maier \(2004\)](#): an intuitionistic semantics for LTL (including next/X/⊙) over finite and infinite sequences

setup $\langle \text{Sign.mandatory-path raw.safety} \rangle$

lemma *dense-alt-def*: — [Lynch \(1996, §8.5.3 Liveness Property\)](#)

shows $(\text{raw.safety.dense} :: ('a, 's, 'v) \text{behavior.t set set})$
 $= \{P. \forall \sigma. \exists xsv. \sigma @_{-B} xsv \in P\}$ (is *?lhs = ?rhs*)

proof(*rule antisym*)

have $\exists xsv. \sigma @_{-B} xsv \in P$ if *raw.safety.cl* $P = \text{UNIV}$ for P and $\sigma :: ('a, 's, 'v) \text{trace.t}$
 using *that*

by (*auto simp: behavior.take.continue*
simp flip: trace.take.Ex-all

elim!: *raw.safety.cl-altE*[**where** $i = \text{Suc} (\text{length} (\text{trace.rest } \sigma))$]

dest!: *subsetD*[**where** $c = \sigma @_{-B} \text{TNil undefined, OF Set.equalityD2}$]

(*metis behavior.continue.take-drop-id behavior.continue.tshift2*)

then show *?lhs* \subseteq *?rhs*

by (*clarsimp simp: raw.safety.dense-def*)

next

have $\omega \in \text{raw.safety.cl } P$ if $\forall \sigma. \exists xsv. \sigma @_{-B} xsv \in P$ for P and $\omega :: ('a, 's, 'v) \text{behavior.t}$

proof(*rule raw.safety.cl-altI*)

fix i

from *spec*[*OF that*, **where** $x = \text{behavior.take } i \omega$]

obtain xsv **where** $\text{behavior.take } i \omega @_{-B} xsv \in P$..

moreover

have $\text{behavior.take } i \omega = \text{behavior.take } i (\text{behavior.take } i \omega @_{-B} xsv)$

by (*clarsimp simp: behavior.take.continue behavior.take.all-continue*
trace.take.behavior.take length-ttake not-le

split: enat.split split-min)

ultimately show $\exists \omega' \in P. \text{behavior.take } i \omega = \text{behavior.take } i \omega' ..$

qed

then show *?rhs* \subseteq *?lhs*

by (*auto simp: raw.safety.dense-def*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path tls} \rangle$

definition *live* :: $('a, 's, 'v) \text{tls set}$ **where**

$live = tls.safety.dense$

setup $\langle Sign.mandatory-path\ live \rangle$

lemma *not-bot*:

shows $\perp \notin tls.live$

by (*simp add: tls.live-def tls.safety.dense-def tls.bot-not-top*)

lemma *top*:

shows $\top \in tls.live$

by (*simp add: tls.live-def tls.safety.dense-top*)

lemma *live-le*:

assumes $P \in tls.live$

assumes $P \leq Q$

shows $Q \in tls.live$

using *assms* **by** (*simp add: tls.live-def tls.safety.dense-le*)

lemma *inf-safety-eq-top*: — Lynch (1996, Theorem 8.8)

shows $tls.live \sqcap tls.safety.closed = \{\top\}$

unfolding *tls.live-def* **by** (*rule tls.safety.dense-inf-closed*)

lemma *terminating*:

shows $tls.eventually\ tls.terminated \in tls.live$

by (*simp add: tls.live-def tls.safety.dense-def tls.safety.cl.eventually[OF tls.terminated.not-bot]*)

However this definition of liveness does not endorse traditional *response* properties.

corec *alternating* $:: bool \Rightarrow ('a \times bool, 'b)\ tlist$ **where**

alternating $b = TCons\ (undefined, b)\ (alternating\ (\neg b))$

abbreviation (*input*) $A\ b \equiv behavior.B\ b\ (tls.live.alternating\ (\neg b))$

lemma *dropn-alternating*:

shows $behavior.dropn\ i\ (tls.live.A\ b) = Some\ (tls.live.A\ (if\ even\ i\ then\ b\ else\ \neg b))$

proof(*induct i arbitrary: b*)

case (*Suc i*) **show** *?case*

by (*subst tls.live.alternating.code*) (*simp add: behavior.dropn.Suc Suc[of $\neg b$, simplified]*)

qed *simp*

notepad

begin

let $?P = tls.leads-to\ (tls.state-prop\ id)\ (tls.state-prop\ Not) :: ('a, bool, unit)\ tls$

let $?w = \langle behavior.B\ True\ (TNil\ ()) \rangle_T :: ('a, bool, unit)\ tls$

have $\neg ?w \leq ?P$

by (*auto simp: tls.singleton.le-conv split: nat.split*)

then **have** $\neg ?w \leq tls.safety.cl\ ?P$

by (*simp add: tls.safety.cl.terminated-iff tls.singleton.terminated-le-conv behavior.sset.simps*)

then **have** $?P \notin tls.live$

by (*auto simp: tls.live-def tls.safety.dense-def*)

*dest: order.trans[**where** $a = ?w$, OF top-greatest eq-refl[OF sym]]*)

— non-triviality

let $?w' = \langle tls.live.A\ True \rangle_T :: ('a, bool, unit)\ tls$

have $?w' \leq ?P$

by (*clarsimp simp: tls.singleton.le-conv tls.live.dropn-alternating[**where** $b = True$, simplified]*) *presburger*

— intuition: there's some safety in these response properties

let $?Q = \text{tls.always } (\text{tls.terminated} \longrightarrow_B \text{tls.state-prop Not}) :: ('a, \text{bool}, \text{unit}) \text{tls}$

have $?Q \in \text{tls.safety.closed}$

by (*simp add: tls.safety.closed.always tls.safety.closed.boolean-implication
tls.safety.closed.not-terminated tls.safety.closed.state-prop*)

moreover have $\neg ?\omega \leq ?Q$

by (*auto simp: tls.singleton.le-conv behavior.sset.simps split: nat.split*)

then have $?Q \neq \top$

by (*auto dest: order.trans[where a=? ω , OF top-greatest eq-refl[OF sym]]*)

ultimately have $?Q \notin \text{tls.live}$

using *tls.live.inf-safety-eq-top* **by** *auto*

moreover

have $\text{tls.terminated} \sqcap (\text{tls.state-prop id} \longrightarrow_B \text{tls.eventually } (\text{tls.state-prop Not})) \leq \text{tls.state-prop Not}$

by (*simp add: boolean-implication.conv-sup inf-sup-distrib tls.state-prop.simps tls.terminated.inf-eventually*)

then have $?P \leq ?Q$

by — (*rule tls.always.mono;
simp add: tls.terminated.inf-always flip: boolean-implication.shunt2*)

ultimately have $?P \notin \text{tls.live}$

by (*blast dest: tls.live.live-le*)

end

setup $\langle \text{Sign.parent-path} \rangle$

The famous decomposition definition $\text{Safe} :: ('a, 's, 'v) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

$\text{Safe } P = \text{tls.safety.cl } P$

definition $\text{Live} :: ('a, 's, 'v) \text{tls} \Rightarrow ('a, 's, 'v) \text{tls}$ **where**

$\text{Live } P = P \sqcup \neg \text{tls.safety.cl } P$

lemma *decomp*:

shows $P = \text{tls.Safe } P \sqcap \text{tls.Live } P$

by (*simp add: tls.Safe-def tls.Live-def boolean-algebra.conj-disj-distrib inf.absorb2 tls.safety.expansive*)

setup $\langle \text{Sign.mandatory-path safety.closed} \rangle$

lemma *Safe*:

shows $\text{tls.Safe } P \in \text{tls.safety.closed}$

by (*simp add: tls.Safe-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path live} \rangle$

lemma *Live*:

shows $\text{tls.Live } P \in \text{tls.live}$

by (*simp add: tls.live-def tls.safety.dense-def tls.Live-def sup-shunt tls.safety.cl.sup tls.safety.expansive*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

24.2 A Haskell-like Ix class

We allow arbitrary indexing schemes for user-facing arrays via the Ix class, which essentially represents a bijection between a subset of an arbitrary type and an initial segment of the naturals.

Source materials:

- Haskell 2010 report: <https://www.haskell.org/onlinereport/haskell2010/haskellch19.html>
- GHC implementation: <https://hackage.haskell.org/package/base-4.16.0.0/docs/src/GHC.Ix.html>
- Haskell pure arrays (just for colour): <https://www.haskell.org/onlinereport/haskell2010/haskellch14.html>
- SML 2D arrays: <https://smlfamily.github.io/Basis/array2.html>

Observations:

- follow Haskell convention here: include the bounds
- could alternatively use an array of one-dimensional arrays but those are not necessarily rectangular
- we can't use *enum* as that requires the whole type to be enumerable

Limitations:

- the basic design assumes laziness; we don't ever want to build the list of indices
 - can be improved either by tweaking the code generator setup or changing the constants here
- array indices typically have partial predecessor and successor operations and are totally ordered on their domain
- no guarantee the *interval* is correct (does not prevent off-by-one errors in instances)

```
class Ix =
  fixes interval :: 'a × 'a ⇒ 'a list
  fixes index :: 'a × 'a ⇒ 'a ⇒ nat
  assumes index: i ∈ set (interval b) ⇒ interval b ! index b i = i
  assumes interval: map (index b) (interval b) = [0..<length (interval b)]
```

lemma *index-length*:

```
  assumes i ∈ set (interval b)
  shows index b i < length (interval b)
```

proof –

```
  from assms[unfolded in-set-conv-nth]
  obtain j where j < length (interval b) and interval b ! j = i
  by blast
  with arg-cong[where f=λx. List.nth x j, OF interval[of b]] show ?thesis
  by simp
```

qed

lemma *distinct-interval*:

```
  shows distinct (interval b)
```

by (metis distinct-map distinct-upt interval)

lemma *inj-on-index*:

```
  shows inj-on (index b) (set (interval b))
```

by (metis distinct-map distinct-upt interval)

lemma *index-eq-conv*:

```
  assumes i ∈ set (interval b)
```

```
  assumes j ∈ set (interval b)
```

```
  shows index b i = index b j ⟷ i = j
```

by (metis assms index)

lemma *index-inv-into*:

```
  assumes i < length (interval b)
```

shows *inv-into* (*set* (*interval b*)) (*index b*) $i \in \text{set } (\text{interval } b)$
by (*metis* *assms* *add.left-neutral* *inv-into-into* *length-map* *list.set-map* *interval* *nth-mem* *nth-upt*)

lemma *linear-order-on*:

shows *linear-order-on* (*set* (*interval b*)) $\{(i, j). \{i, j\} \subseteq \text{set } (\text{interval } b) \wedge \text{index } b \ i \leq \text{index } b \ j\}$
by (*force simp*: *linear-order-on-def* *partial-order-on-def* *preorder-on-def* *refl-on-def* *total-on-def*
intro: *transI* *antisymI*
dest: *index*)

lemma *interval-map*:

shows *map* ($\lambda i. f (\text{interval } b \ ! \ i)$) $[0..<\text{length } (\text{interval } b)] = \text{map } f (\text{interval } b)$
by (*simp* *add*: *map-equality-iff*)

lemma *index-forE*:

assumes $i < \text{length } (\text{interval } b)$
obtains j **where** $j \in \text{set } (\text{interval } b)$ **and** $\text{index } b \ j = i$
using *assms* *index* *index-length* *nth-eq-iff-index-eq*[*OF distinct-interval*] *nth-mem*[*OF assms*] **by** *blast*

instantiation *unit* :: *Ix*

begin

definition *interval-unit* = $(\lambda(x::\text{unit}, y::\text{unit}). [()])$

definition *index-unit* = $(\lambda(x::\text{unit}, y::\text{unit}) \ -::\text{unit}. \ 0::\text{nat})$

instance **by** *standard* (*auto simp*: *interval-unit-def* *index-unit-def*)

end

instantiation *nat* :: *Ix*

begin

definition *interval-nat* = $(\lambda(l, u::\text{nat}). [l..<\text{Suc } u])$ — bounds are inclusive

definition *index-nat* = $(\lambda(l, u::\text{nat}) \ i::\text{nat}. \ i - l)$

lemma *upt-minus*:

shows *map* ($\lambda i. \ i - l$) $[l..<u] = [0..<u - l]$

by (*induct* *u*) (*auto simp*: *Suc-diff-le*)

instance **by** *standard* (*auto simp*: *interval-nat-def* *index-nat-def* *upt-minus* *nth-append*)

end

instantiation *int* :: *Ix*

begin

definition *interval-int* = $(\lambda(l, u::\text{int}). [l..u])$ — bounds are inclusive

definition *index-int* = $(\lambda(l, u::\text{int}) \ i::\text{int}. \ \text{nat } (i - l))$

lemma *upto-minus*:

shows *map* ($\lambda i. \ \text{nat } (i - l)$) $[l..u] = [0..<\text{nat } (u - l + 1)]$

proof(*induct* $\text{nat}(u - l + 1)$ *arbitrary*: *u*)

case (*Suc i*)

from *Suc.hyps*(1)[*of* $u - 1$] *Suc.hyps*(2) **show** *?case*

by (*simp* *add*: *upto-rec2* *ac-simps* *Suc-nat-eq-nat-zadd1* *flip*: *upt-Suc-append*)

qed *simp*

instance **by** *standard* (*auto simp*: *interval-int-def* *index-int-def* *upto-minus*)

end

type-synonym ('i, 'j) two-dim = ('i × 'j) × ('i × 'j)

instantiation prod :: (Ix, Ix) Ix

begin

definition interval-prod = (λ((l, l'), (u, u')). List.product (interval (l, u)) (interval (l', u')))

definition index-prod = (λ((l, l'), (u, u')) (i, i'). index (l, u) i * length (interval (l', u')) + index (l', u') i')

abbreviation (input) fst-bounds :: ('a × 'b) × ('a × 'b) ⇒ ('a × 'a) **where**

fst-bounds b ≡ (fst (fst b), fst (snd b))

abbreviation (input) snd-bounds :: ('a × 'b) × ('a × 'b) ⇒ ('b × 'b) **where**

snd-bounds b ≡ (snd (fst b), snd (snd b))

lemma inj-on-index-prod:

shows inj-on (index ((l, l'), (u, u'))) (set (interval ((l, l'), (u, u'))))

by (clarsimp simp: inj-on-def interval-prod-def index-prod-def)

(metis index index-length length-pos-if-in-set add-diff-cancel-right'

div-mult-self-is-m mod-less mod-mult-self3)

instance

proof

show interval b ! index b i = i **if** i ∈ set (interval b) **for** b **and** i :: 'a × 'b

proof –

have *: i * n + j < m * n **if** i < m **and** j < n

for i j m n :: nat

using that **by** (metis bot-nat-0.extremum-strict div-less div-less-iff-less-mult div-mult-self3 nat-arith.rule0 not-gr-zero)

from that

have index (fst-bounds b) (fst i) * length (interval (snd-bounds b))

+ index (snd-bounds b) (snd i)

< length (interval (fst-bounds b)) * length (interval (snd-bounds b))

by (clarsimp simp: interval-prod-def index-prod-def * dest!: index-length)

then show ?thesis

using that length-pos-if-in-set

by (fastforce simp: interval-prod-def index-prod-def List.product-nth index index-length)

qed

show map (index b) (interval b) = [0..<length (interval b)] **for** b :: ('a × 'b) × ('a × 'b)

by (rule iffD2[OF list-eq-iff-nth-eq])

(clarsimp simp: interval-prod-def index-prod-def split-def product-nth ac-simps;

metis (no-types, lifting) distinct-interval index index-length length-pos-if-in-set nth-mem

less-mult-imp-div-less mod-div-mult-eq mod-less-divisor mult.commute nth-eq-iff-index-eq)

qed

end

setup ‹Sign.mandatory-path Ix›

setup ‹Sign.mandatory-path prod›

lemma interval-conv:

shows (x, y) ∈ set (interval b) ↔ x ∈ set (interval (fst-bounds b)) ∧ y ∈ set (interval (snd-bounds b))

by (force simp: interval-prod-def)

setup ‹Sign.parent-path›

type-synonym $'i$ square = ($'i, 'i$) two-dim

definition square :: $'i::Ix$ Ix.square \Rightarrow bool **where**

square = $(\lambda((l, l'), (u, u')). Ix.interval (l, u) = Ix.interval (l', u'))$

setup \langle Sign.mandatory-path square \rangle

lemma conv:

assumes Ix.square b

shows $i \in \text{set } (Ix.interval (\text{fst-bounds } b)) \longleftrightarrow i \in \text{set } (Ix.interval (\text{snd-bounds } b))$

using *assms* **by** (*clarsimp simp: Ix.square-def*)

setup \langle Sign.parent-path \rangle

setup \langle Sign.parent-path \rangle

hide-const (**open**) interval index

25 A polymorphic heap

We model a heap as a partial map from opaque addresses to structured objects.

- we use this extra structure to handle buffered writes (see §27)
- allocation is non-deterministic and partial
- supports explicit deallocation

Limitations:

- does not support polymorphic sum types such as $'a + 'b$ and $'a$ option or products or lists
- the class of representable types is too small to represent processes

Source materials:

- \$ISABELLE_HOME/src/HOL/Imperative_HOL/Heap.thy
 - that model of heaps includes a *lim* field to support deterministic allocation
 - it uses distinct heaps for arrays and references

setup \langle Sign.mandatory-path heap \rangle

type-synonym addr = nat — untyped heap addresses

datatype rep — the concrete representation of heap values

= Addr nat heap.addr — metadata paired with an address

| Val nat

datatype write = Write heap.addr nat heap.rep

type-synonym t = heap.addr \rightarrow heap.rep list — partial map from addresses to structured encoded values

abbreviation empty :: heap.t **where**

empty \equiv Map.empty

primrec apply-write :: heap.write \Rightarrow heap.t \Rightarrow heap.t **where**

apply-write (heap.Write addr i x) s = s(addr \mapsto (the (s addr))[i := x])

```

class rep = — the class of representable types
  assumes ex-inj:  $\exists to\text{-heap}\text{-rep} :: 'a \Rightarrow \text{heap.rep. inj } to\text{-heap}\text{-rep}$ 

setup  $\langle \text{Sign.mandatory-path } rep \rangle$ 

lemma countable-classI[intro!]:
  shows OFCLASS('a::countable, heap.rep-class)
by intro-classes (simp add: inj-on-def exI[where  $x = \text{heap.Val} \circ to\text{-nat}$ ])

definition to :: 'a::heap.rep  $\Rightarrow$  heap.rep where
  to = (SOME f. inj f)

definition from :: heap.rep  $\Rightarrow$  'a::heap.rep where
  from = inv (heap.rep.to :: 'a  $\Rightarrow$  heap.rep)

lemmas inj-to[simp] = someI-ex[OF heap.ex-inj, folded heap.rep.to-def]

lemma inj-on-to[simp, intro]: inj-on heap.rep.to S
using heap.rep.inj-to by (auto simp: inj-on-def)

lemma surj-from[simp]: surj heap.rep.from
unfolding heap.rep.from-def by (simp add: inj-imp-surj-inv)

lemma to-split[simp]: heap.rep.to x = heap.rep.to y  $\longleftrightarrow$  x = y
using injD[OF heap.rep.inj-to] by auto

lemma from-to[simp]:
  shows heap.rep.from (heap.rep.to x) = x
by (simp add: heap.rep.from-def)

instance unit :: heap.rep ..

instance bool :: heap.rep ..

instance nat :: heap.rep ..

instance int :: heap.rep ..

instance char :: heap.rep ..

instance String.literal :: heap.rep ..

instance typerep :: heap.rep ..

setup  $\langle \text{Sign.parent-path} \rangle$ 

```

User-facing heap types typically carry more information than an (untyped) address, such as (phantom) typing and a representation invariant that guarantees the soundness of the encoding (for the given value at the given address only). We abstract over that here and provide some generic operations.

Notes:

- intuitively *addr-of* should be surjective but we do not enforce this
- we use sets here but these are not very flexible: all refs must have the same type
 - this means some intuitive facts involving *UNIV* cannot be stated

class *addr-of* =

fixes *addr-of* :: 'a ⇒ heap.addr

fixes *rep-val-inv* :: 'a ⇒ heap.rep list pred

definition *obj-at* :: heap.rep list pred ⇒ heap.addr ⇒ heap.t pred **where**

obj-at P r s = (case s r of None ⇒ False | Some v ⇒ P v)

abbreviation (*input*) *present* :: 'a::heap.addr-of ⇒ heap.t pred **where**

present r ≡ heap.obj-at ⟨True⟩ (heap.addr-of r)

abbreviation (*input*) *rep-inv* :: 'a::heap.addr-of ⇒ heap.t pred **where**

rep-inv r ≡ heap.obj-at (heap.rep-val-inv r) (heap.addr-of r)

abbreviation (*input*) *rep-inv-set* :: 'a::heap.addr-of ⇒ heap.t set **where**

rep-inv-set r ≡ Collect (heap.rep-inv r)

— allows arbitrary transitions provided the *rep-inv* of *r* is respected

abbreviation (*input*) *rep-inv-rel* :: 'a::heap.addr-of ⇒ heap.t rel **where**

rep-inv-rel r ≡ heap.rep-inv-set r × heap.rep-inv-set r

— totality models the idea that all dereferences are “valid” but only some are reasonable

definition *get* :: 'a::heap.addr-of ⇒ heap.t ⇒ 'v::heap.rep list **where**

get r s = map heap.rep.from (the (s (heap.addr-of r)))

definition *set* :: 'a::heap.addr-of ⇒ 'v::heap.rep list ⇒ heap.t ⇒ heap.t **where**

set r v s = s(heap.addr-of r ↦ map heap.rep.to v)

definition *dealloc* :: 'a::heap.addr-of ⇒ heap.t ⇒ heap.t **where**

dealloc r s = s |' {heap.addr-of r}

— allows no changes to *rs*, asserts the *rep-inv* of *rs* is valid, arbitrary changes to $\neg rs$

definition *Id-on* :: 'a::heap.addr-of set ⇒ heap.t rel (⟨heap.Id.⟩) **where**

heap.Id_{rs} = (∩ r∈rs. heap.rep-inv-rel r ∩ Id_{λs. s (heap.addr-of r)})

— allows arbitrary changes to *rs* provided the *rep-inv* of *rs* is respected. requires addresses in $\neg heap.addr-of$ ' *rs* to be unchanged

definition *modifies* :: 'a::heap.addr-of set ⇒ heap.t rel (⟨heap.modifies.⟩) **where**

heap.modifies_{rs} = (∩ r∈rs. heap.rep-inv-rel r) ∩ {(s, s'). ∀ r∈ $\neg heap.addr-of$ ' rs. s r = s' r}

setup ⟨Sign.mandatory-path get⟩

lemma *cong*:

assumes s (heap.addr-of r) = s' (heap.addr-of r')

shows heap.get r s = heap.get r' s'

by (simp add: assms heap.get-def)

lemma *Id-on-proj-cong*:

assumes (s, s') ∈ heap.Id_{r}

shows heap.get r s = heap.get r s'

using assms **by** (simp add: heap.Id-on-def heap.get-def)

lemma *fun-upd*:

shows heap.get r (fun-upd s a (Some w))

= (if heap.addr-of r = a then map heap.rep.from w else heap.get r s)

by (simp add: heap.get-def)

lemma *set-eq*:

shows heap.get r (heap.set r v s) = v

by (simp add: heap.get-def heap.set-def comp-def)

lemma set-neq:

assumes heap.addr-of $r \neq$ heap.addr-of r'

shows heap.get r (heap.set $r' v s$) = heap.get $r s$

by (simp add: heap.get-def heap.set-def assms)

setup <Sign.parent-path>

setup <Sign.mandatory-path set>

lemma cong:

assumes heap.addr-of $r =$ heap.addr-of r'

assumes $v = v'$

assumes $\bigwedge r'. r' \neq$ heap.addr-of $r \implies s r' = s' r'$

shows heap.set $r v s =$ heap.set $r' v' s'$

by (simp add: assms heap.set-def fun-eq-iff)

lemma empty:

shows heap.set $r v$ (heap.empty) = [heap.addr-of $r \mapsto$ map heap.rep.to v]

by (simp add: heap.set-def)

lemma fun-upd:

shows heap.set $r v$ (fun-upd $s a w$) = (fun-upd $s a w$)(heap.addr-of $r \mapsto$ map heap.rep.to v)

by (simp add: heap.set-def)

lemma same:

shows heap.set $r v$ (heap.set $r w s$) = heap.set $r v s$

by (simp add: heap.set-def)

lemma twist:

assumes heap.addr-of $r \neq$ heap.addr-of r'

shows heap.set $r v$ (heap.set $r' w s$) = heap.set $r' w$ (heap.set $r v s$)

using assms by (simp add: heap.set-def fun-eq-iff)

setup <Sign.parent-path>

setup <Sign.mandatory-path obj-at>

lemma cong[cong]:

fixes $P ::$ heap.rep list pred

assumes $\bigwedge v. s r =$ Some $v \implies P v = P' v$

assumes $r = r'$

assumes $s r = s' r'$

shows heap.obj-at $P r s \longleftrightarrow$ heap.obj-at $P' r' s'$

using assms by (simp add: heap.obj-at-def cong: option.case-cong)

lemma split:

shows Q (heap.obj-at $P r s$) \longleftrightarrow ($s r =$ None $\longrightarrow Q$ False) \wedge ($\forall v. s r =$ Some $v \longrightarrow Q$ ($P v$))

by (simp add: heap.obj-at-def split: option.splits)

lemma split-asm:

shows Q (heap.obj-at $P r s$) \longleftrightarrow \neg (($s r =$ None $\wedge \neg Q$ False) \vee ($\exists v. s r =$ Some $v \wedge \neg Q$ ($P v$)))

by (simp add: heap.obj-at-def split: option.splits)

lemmas splits = heap.obj-at.split heap.obj-at.split-asm

lemma empty:

shows $\neg \text{heap.obj-at } P \ r \ \text{heap.empty}$
by (*simp add: heap.obj-at-def*)

lemma *set*:

shows $\text{heap.obj-at } P \ r \ (\text{heap.set } r' \ v \ s)$
 $\longleftrightarrow (r = \text{heap.addr-of } r' \wedge P \ (\text{map } \text{heap.rep.to } v)) \vee (r \neq \text{heap.addr-of } r' \wedge \text{heap.obj-at } P \ r \ s)$
by (*simp add: comp-def heap.set-def split: heap.obj-at.split*)

lemma *fun-upd*:

shows $\text{heap.obj-at } P \ r \ (\text{fun-upd } s \ a \ (\text{Some } w)) = (\text{if } r = a \ \text{then } P \ w \ \text{else } \text{heap.obj-at } P \ r \ s)$
by (*simp split: heap.obj-at.split*)

setup $\langle \text{Sign.parent-path} \rangle$

lemmas *simps* = — objective: reduce manifest heaps

heap.get.set-eq
heap.get.fun-upd
heap.set.empty
heap.set.same
heap.set.fun-upd
heap.obj-at.empty
heap.obj-at.fun-upd

setup $\langle \text{Sign.mandatory-path Id-on} \rangle$

lemma *empty[simp]*:

shows $\text{heap.Id}_{\{\}} = \text{UNIV}$
by (*simp add: heap.Id-on-def*)

lemma *sup*:

shows $\text{heap.Id}_{X \cup Y} = \text{heap.Id}_X \cap \text{heap.Id}_Y$
unfolding *heap.Id-on-def* **by** *blast*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path modifies} \rangle$

lemma *empty[simp]*:

shows $\text{heap.modifies}_{\{\}} = \text{Id}$
by (*auto simp: heap.modifies-def*)

lemma *rep-inv-rel-le*:

shows $\text{heap.modifies}_{rs} \subseteq (\bigcap r \in rs. \text{heap.rep-inv-rel } r)$
by (*simp add: heap.modifies-def*)

lemma *rep-inv*:

assumes $(s, s') \in \text{heap.modifies}_{\{a\}}$
shows $\text{heap.rep-inv } a \ s$
and $\text{heap.rep-inv } a \ s'$
using *assms* **by** (*simp-all add: heap.modifies-def split: heap.obj-at.split*)

lemma *Id-conv*:

shows $(s, s) \in \text{heap.modifies}_{rs} \longleftrightarrow (\forall r \in rs. (s, s) \in \text{heap.rep-inv-rel } r)$
by (*simp add: heap.modifies-def*)

lemma *eqI*:

assumes $(s, s') \in \text{heap.modifies}_{rs}$
assumes $\bigwedge r. \llbracket r \in rs; \text{heap.rep-inv } r \ s; \text{heap.rep-inv } r \ s' \rrbracket \implies s \ (\text{heap.addr-of } r) = s' \ (\text{heap.addr-of } r)$

shows $s = s'$
using *assms* **by** (*simp add: heap.modifies-def*) *blast*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable.heap} \rangle$

lemma *Id-on-frame-cong*:

assumes $\bigwedge s s'. (\bigwedge r. r \in rs \implies \text{heap.rep-inv } r s \wedge \text{heap.rep-inv } r s' \wedge s (\text{heap.addr-of } r) = s' (\text{heap.addr-of } r))$
 $\implies P s \longleftrightarrow P' s'$

shows $\text{stable heap.Id}_{rs} P \longleftrightarrow \text{stable heap.Id}_{rs} P'$

using *assms* **by** (*auto 10 0 simp: stable-def monotone-def heap.Id-on-def*)

lemma *Id-on-frameI*:

assumes $\bigwedge s s'. (\bigwedge r. r \in rs \implies \text{heap.rep-inv } r s \wedge \text{heap.rep-inv } r s' \wedge s (\text{heap.addr-of } r) = s' (\text{heap.addr-of } r))$
 $\implies P s \longleftrightarrow P' s'$

shows $\text{stable heap.Id}_{rs} P$

using *assms* **by** (*auto simp: stable-def monotone-def heap.Id-on-def*)

lemma *Id-on-rep-invI[stable.intro]*:

assumes $r \in rs$

shows $\text{stable heap.Id}_{rs} (\text{heap.rep-inv } r)$

using *assms* **by** (*blast intro: stable.heap.Id-on-frameI*)

setup $\langle \text{Sign.parent-path} \rangle$

25.1 References

datatype $'a \text{ ref} = \text{Ref} (\text{addr-of}: \text{heap.addr})$

instantiation $\text{ref} :: (\text{heap.rep}) \text{heap.addr-of}$
begin

definition $\text{addr-of-ref} :: 'a \text{ ref} \Rightarrow \text{heap.addr}$ **where**
 $\text{addr-of-ref} = \text{ref.addr-of}$

definition $\text{rep-val-inv-ref} :: 'a \text{ ref} \Rightarrow \text{heap.rep list pred}$ **where**
 $\text{rep-val-inv-ref } r \text{ vs} \longleftrightarrow (\text{case vs of } [v] \Rightarrow \text{heap.rep.to } (\text{heap.rep.from } v :: 'a) = v \mid _ \Rightarrow \text{False})$

instance ..

end

instance $\text{ref} :: (\text{heap.rep}) \text{heap.rep}$
by *standard* (*simp add: inj-on-def ref.expand exI[where x=heap.Addr 0 \circ ref.addr-of]*)

setup $\langle \text{Sign.mandatory-path Ref} \rangle$

definition $\text{get} :: 'a::\text{heap.rep } \text{ref} \Rightarrow \text{heap.t} \Rightarrow 'a$ **where**
 $\text{get } r \text{ s} = \text{hd } (\text{heap.get } r \text{ s})$

definition $\text{set} :: 'a::\text{heap.rep } \text{ref} \Rightarrow 'a \Rightarrow \text{heap.t} \Rightarrow \text{heap.t}$ **where**
 $\text{set } r \text{ v s} = \text{heap.set } r [v] \text{ s}$

definition $\text{alloc} :: 'a \Rightarrow \text{heap.t} \Rightarrow ('a::\text{heap.rep } \text{ref} \times \text{heap.t}) \text{ set}$ **where**
 $\text{alloc } v \text{ s} = \{(r, \text{Ref.set } r \text{ v s}) \mid r. \neg \text{heap.present } r \text{ s}\}$

lemma *addr-of*:

shows $\text{heap.addr-of } (\text{Ref } r) = r$
by (*simp add: addr-of-ref-def*)

setup $\langle \text{Sign.mandatory-path get} \rangle$

lemma *fun-upd*:

shows $\text{Ref.get } r (\text{fun-upd } s \ a \ (\text{Some } [w]))$
 $= (\text{if } \text{heap.addr-of } r = a \ \text{then } \text{heap.rep.from } w \ \text{else } \text{Ref.get } r \ s)$
by (*simp add: Ref.get-def heap.simps*)

lemma *set-eq*:

shows $\text{Ref.get } r (\text{Ref.set } r \ v \ s) = v$
by (*simp add: Ref.get-def Ref.set-def heap.simps*)

lemma *set-neq*:

fixes $r :: 'a::\text{heap.rep } \text{ref}$
fixes $r' :: 'b::\text{heap.rep } \text{ref}$
assumes $\text{addr-of } r \neq \text{addr-of } r'$
shows $\text{Ref.get } r (\text{Ref.set } r' \ v \ s) = \text{Ref.get } r \ s$
using *assms* **by** (*simp add: Ref.get-def Ref.set-def addr-of-ref-def heap.get.set-neq*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path set} \rangle$

lemma *empty*:

shows $\text{Ref.set } r \ v \ (\text{heap.empty}) = [\text{heap.addr-of } r \mapsto [\text{heap.rep.to } v]]$
by (*simp add: Ref.set-def heap.simps*)

lemma *fun-upd*:

shows $\text{Ref.set } r \ v \ (\text{fun-upd } s \ a \ w) = (\text{fun-upd } s \ a \ w)(\text{heap.addr-of } r \mapsto [\text{heap.rep.to } v])$
by (*simp add: Ref.set-def heap.simps*)

lemma *same*:

shows $\text{Ref.set } r \ v \ (\text{Ref.set } r \ w \ s) = \text{Ref.set } r \ v \ s$
by (*simp add: Ref.set-def heap.set-def*)

lemma *obj-at-conv*:

fixes $a :: \text{heap.addr}$
fixes $r :: 'a::\text{heap.rep } \text{ref}$
fixes $v :: 'a$
fixes $P :: \text{heap.rep } \text{list } \text{pred}$
shows $\text{heap.obj-at } P \ a \ (\text{Ref.set } r \ v \ s) \longleftrightarrow (a = \text{heap.addr-of } r \wedge P \ [\text{heap.rep.to } v])$
 $\vee (a \neq \text{heap.addr-of } r \wedge \text{heap.obj-at } P \ a \ s)$
by (*simp add: Ref.set-def heap.set-def split: heap.obj-at.split*)

setup $\langle \text{Sign.parent-path} \rangle$

lemmas *simps*[*simp*] =

Ref.addr-of
Ref.get.set-eq
Ref.get.set-neq
Ref.get.fun-upd
Ref.set.same
Ref.set.empty
Ref.set.fun-upd

Ref.set.obj-at-conv

setup $\langle \text{Sign.parent-path} \rangle$

25.2 Arrays

25.2.1 Code generation constants: one-dimensional arrays

We ask that targets of the code generator provide implementations of one-dimensional arrays and the associated operations.

Notes:

- user-facing arrays make use of Ix
- due to the lack of bounds there is no $rep\text{-}val\text{-}inv$

datatype $'a\ one\text{-}dim\text{-}array = Array\ (addr\text{-}of:\ heap.addr)$

instantiation $one\text{-}dim\text{-}array :: (type)\ heap.addr\text{-}of$
begin

definition $addr\text{-}of\text{-}one\text{-}dim\text{-}array :: 'a\ one\text{-}dim\text{-}array \Rightarrow heap.addr$ **where**
 $addr\text{-}of\text{-}one\text{-}dim\text{-}array = addr\text{-}of$

definition $rep\text{-}val\text{-}inv\text{-}one\text{-}dim\text{-}array :: 'a\ one\text{-}dim\text{-}array \Rightarrow heap.rep\ list\ pred$ **where**
 $[simp]: rep\text{-}val\text{-}inv\text{-}one\text{-}dim\text{-}array\ a\ vs \longleftrightarrow True$

instance ..

end

setup $\langle \text{Sign.mandatory-path}\ ODArray \rangle$

definition $get :: 'a::heap.rep\ one\text{-}dim\text{-}array \Rightarrow nat \Rightarrow heap.t \Rightarrow 'a$ **where**
 $get\ a\ i\ s = heap.get\ a\ s\ !\ i$

definition $set :: 'a::heap.rep\ one\text{-}dim\text{-}array \Rightarrow nat \Rightarrow 'a \Rightarrow heap.t \Rightarrow heap.t$ **where**
 $set\ a\ i\ v\ s = heap.set\ a\ ((heap.get\ a\ s)[i:=v])\ s$

definition $alloc :: 'a\ list \Rightarrow heap.t \Rightarrow ('a::heap.rep\ one\text{-}dim\text{-}array \times heap.t)\ set$ **where**
 $alloc\ av\ s = \{(a, heap.set\ a\ av\ s) \mid a. \neg heap.present\ a\ s\}$

definition $list\text{-}for :: 'a::heap.rep\ one\text{-}dim\text{-}array \Rightarrow heap.t \Rightarrow 'a\ list$ **where**
 $list\text{-}for\ a = heap.get\ a$

setup $\langle \text{Sign.mandatory-path}\ get \rangle$

lemma *weak-cong*:

assumes $i = i'$

assumes $a = a'$

assumes $s\ (heap.addr\text{-}of\ a) = s'\ (heap.addr\text{-}of\ a')$

shows $ODArray.get\ a\ i\ s = ODArray.get\ a'\ i'\ s'$

using *assms* **by** (*simp* *add: ODArray.get-def* *cong: heap.get.cong*)

lemma *weak-Id-on-proj-cong*:

assumes $i = i'$

assumes $a = a'$

assumes $(s, s') \in heap.Id_{\{a\}}$

shows $ODArray.get\ a\ i\ s = ODArray.get\ a'\ i'\ s'$
using *assms* **by** (*simp* *add*: $ODArray.get-def\ cong:\ heap.get.Id-on-proj-cong$)

lemma *set-eq*:

assumes $i < length\ (the\ (s\ (heap.addr-of\ a)))$

shows $ODArray.get\ a\ i\ (ODArray.set\ a\ i\ v\ s) = v$

using *assms* **by** (*simp* *add*: $ODArray.get-def\ ODArray.set-def\ heap.get.set-eq$) (*simp* *add*: $heap.get-def$)

lemma *set-neq*:

assumes $i \neq j$

shows $ODArray.get\ a\ i\ (ODArray.set\ a\ j\ v\ s) = ODArray.get\ a\ i\ s$

using *assms* **by** (*simp* *add*: $ODArray.get-def\ ODArray.set-def\ heap.get.set-eq$)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

25.2.2 User-facing arrays

datatype $(i, 'a)\ array = Array\ (bounds:\ (i \times 'i))\ (arr:\ 'a\ one-dim-array)$

hide-const (**open**) $bounds\ arr$

instantiation $array :: (Ix, heap.rep)\ heap.addr-of$

begin

definition $addr-of-array :: ('a, 'b)\ array \Rightarrow heap.addr$ **where**

$addr-of-array = addr-of \circ array.arr$

definition $rep-val-inv-array :: ('a, 'b)\ array \Rightarrow heap.rep\ list\ pred$ **where**

$rep-val-inv-array\ a\ vs \longleftrightarrow$

$length\ vs = length\ (Ix.interval\ (array.bounds\ a))$

$\wedge (\forall v \in set\ vs.\ heap.rep.to\ (heap.rep.from\ v :: 'b) = v)$

instance ..

end

instance $array :: (countable, type)\ heap.rep$

by *standard*

(*rule* *exI*[**where** $x = \lambda a.\ heap.Addr\ (to-nat\ (array.bounds\ a))\ (addr-of\ (array.arr\ a))$],

rule *injI*, *simp* *add*: $array.expand\ one-dim-array.expand$)

setup $\langle Sign.mandatory-path\ Array \rangle$

abbreviation (*input*) $square :: (i :: Ix \times 'i, 'a)\ array \Rightarrow bool$ **where**

$square\ a \equiv Ix.square\ (array.bounds\ a)$

abbreviation (*input*) $index :: (i :: Ix, 'a)\ array \Rightarrow 'i \Rightarrow nat$ **where**

$index\ a \equiv Ix.index\ (array.bounds\ a)$

abbreviation (*input*) $interval :: (i :: Ix, 'a)\ array \Rightarrow 'i\ list$ **where**

$interval\ a \equiv Ix.interval\ (array.bounds\ a)$

definition $get :: (i :: Ix, 'a :: heap.rep)\ array \Rightarrow 'i \Rightarrow heap.t \Rightarrow 'a$ **where**

$get\ a\ i = ODArray.get\ (array.arr\ a)\ (Array.index\ a\ i)$

definition $set :: (i :: Ix, 'a :: heap.rep)\ array \Rightarrow 'i \Rightarrow 'a \Rightarrow heap.t \Rightarrow heap.t$ **where**

set a i v = OArray.set (array.arr a) (Array.index a i) v

definition list-for :: ('i::Ix, 'a::heap.rep) array ⇒ heap.t ⇒ 'a list **where**
list-for a = OArray.list-for (array.arr a)

— can coerce any indexing regime into any other provided the contents fit

definition coerce :: ('i::Ix, 'a::heap.rep) array ⇒ ('j × 'j) ⇒ ('j::Ix, 'a::heap.rep) array option **where**
coerce a b = (if length (Array.interval a) = length (Ix.interval b)
then Some (Array b (array.arr a))
else None)

definition Id-on :: ('i::Ix, 'a::heap.rep) array ⇒ 'i set ⇒ heap.t rel (⟨Array.Id_, _⟩) **where**
Array.Id_{a, is} = heap.rep-inv-rel a ∩ {(s, s'). ∀ i ∈ is. Array.get a i s = Array.get a i s'}

definition modifies :: ('i::Ix, 'a::heap.rep) array ⇒ 'i set ⇒ heap.t rel (⟨Array.modifies_, _⟩) **where**
Array.modifies_{a, is}
= heap.modifies_{a} ∩ {(s, s'). ∀ i ∈ set (Array.interval a) - is. Array.get a i s = Array.get a i s'}

lemma_simps[simp]:
shows heap.addr-of (array.arr a) = heap.addr-of a
and heap.addr-of ∘ array.arr = heap.addr-of
by (simp-all add: addr-of-array-def addr-of-one-dim-array-def)

setup ⟨Sign.mandatory-path get⟩

lemma set-eq:
assumes heap.rep-inv a s
assumes i ∈ set (Array.interval a)
shows Array.get a i (Array.set a i v s) = v
using assms
by (simp add: Array.get-def Array.set-def OArray.get.set-eq index-length rep-val-inv-array-def
split: heap.obj-at.split-asm)

lemma set-neq:
assumes i ∈ set (Array.interval a)
assumes j ∈ set (Array.interval a)
assumes i ≠ j
shows Array.get a j (Array.set a i v s) = Array.get a j s
using assms **by** (simp add: Array.get-def Array.set-def OArray.get.set-neq index-eq-conv)

lemma Id-on-proj-cong:
assumes a = a'
assumes i = i'
assumes (s, s') ∈ Array.Id_{a', {i'}}
assumes i' ∈ set (Array.interval a)
shows Array.get a i s = Array.get a' i' s'
using assms **by** (simp add: Array.get-def Array.Id-on-def)

lemma weak-cong:
assumes a = a'
assumes i = i'
assumes s (heap.addr-of a) = s' (heap.addr-of a')
shows Array.get a i s = Array.get a' i' s'
using assms **unfolding** Array.get-def **by** (simp cong: OArray.get.weak-cong)

lemma weak-Id-on-proj-cong:
assumes i = i'
assumes a = a'

assumes $(s, s') \in \text{heap.Id}_{\{a\}}$
shows $\text{Array.get } a \ i \ s = \text{Array.get } a' \ i' \ s'$
using *assms* **unfolding** *Array.get-def*
by (*simp add: heap.Id-on-def OArray.get.weak-Id-on-proj-cong split: heap.obj-at.splits*)

lemma *ext*:

assumes *heap.rep-inv a s*
assumes *heap.rep-inv a s'*
assumes $\forall i \in \text{set } (\text{Ix-class.interval } (\text{array.bounds } a)). \text{Array.get } a \ i \ s = \text{Array.get } a \ i \ s'$
shows $s \ (\text{heap.addr-of } a) = s' \ (\text{heap.addr-of } a)$
using *assms*
by (*simp add: Array.get-def OArray.get-def heap.get-def rep-val-inv-array-def split: heap.obj-at.splits*)
(rule nth-equalityI, simp, metis index-forE nth-map nth-mem)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path set} \rangle$

lemma *cong-deref*:

assumes $a = a'$
assumes $i = i'$
assumes $v = v'$
assumes $s \ r = s' \ r'$
assumes $r = r'$
shows $\text{Array.set } a \ i \ v \ s \ r = \text{Array.set } a' \ i' \ v' \ s' \ r'$
using *assms* **by** (*clarsimp simp: Array.set-def OArray.set-def heap.set-def heap.get-def*)

lemma *same*:

shows $\text{Array.set } a \ i \ v \ (\text{Array.set } a \ i \ v' \ s) = \text{Array.set } a \ i \ v \ s$
by (*simp add: Array.set-def OArray.set-def heap.simps*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path coerce} \rangle$

lemma *ex-bij-betw*:

fixes $a :: ('i::\text{Ix}, 'a::\text{heap.rep}) \text{array}$
fixes $b :: 'j::\text{Ix} \times 'j$
assumes $\text{Array.coerce } a \ b = \text{Some } a'$
obtains f **where** $\text{map } f \ (\text{Array.interval } a) = \text{Ix.interval } b$
using *assms* **unfolding** *Array.coerce-def* **by** (*metis interval map-map map-nth not-None-eq*)

lemma *ex-bij-betw2*:

fixes $a :: ('i::\text{Ix}, 'a::\text{heap.rep}) \text{array}$
fixes $b :: 'j::\text{Ix} \times 'j$
assumes $\text{Array.coerce } a \ b = \text{Some } a'$
obtains f **where** $\text{map } f \ (\text{Ix.interval } b) = \text{Array.interval } a$
using *assms* **by** (*metis Array.coerce-def Array.coerce.ex-bij-betw array.sel(1) option.distinct(1)*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path rep-inv} \rangle$

lemma *set*:

assumes *heap.rep-inv a s*
shows *heap.rep-inv a (Array.set a i v s)*
using *assms*

by (*simp add: Array.set-def OArray.set-def rep-val-inv-array-def heap.set-def heap.get-def split: heap.obj-at.splits*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path modifies} \rangle$

lemma *heap-modifies-le*:

shows $\text{Array.modifies}_{a, is} \subseteq \text{heap.modifies}_{\{a\}}$

by (*simp add: Array.modifies-def*)

lemma *heap-rep-inv-rel-le*:

shows $\text{Array.modifies}_{a, is} \subseteq \text{heap.rep-inv-rel } a$

using *heap.modifies.rep-inv-rel-le* [**where** $rs = \{a\}$] **by** (*auto simp: Array.modifies-def*)

lemma *empty*:

shows $\text{Array.modifies}_{a, \{ \}} = \text{Id} \cap \text{heap.rep-inv-rel } a$ (**is** $?lhs = ?rhs$)

by (*auto simp: Array.modifies-def heap.modifies.Id-conv heap.modifies.rep-inv elim: heap.modifies.eqI Array.get.ext*)

lemma *mono*:

assumes $is \subseteq js$

shows $\text{Array.modifies}_{a, is} \subseteq \text{Array.modifies}_{a, js}$

using *assms* **by** (*auto simp: Array.modifies-def*)

lemma *INTER*:

shows $\text{Array.modifies}_{a, \bigcap_{x \in X} f x} = (\bigcap_{x \in X} \text{Array.modifies}_{a, f x}) \cap \text{heap.modifies}_{\{a\}}$

by (*auto simp: Array.modifies-def*)

lemma *Inter*:

shows $\text{Array.modifies}_{a, \bigcap X} = (\bigcap_{x \in X} \text{Array.modifies}_{a, x}) \cap \text{heap.modifies}_{\{a\}}$

by (*auto simp: Array.modifies-def*)

lemma *inter*:

shows $\text{Array.modifies}_{a, is} \cap \text{Array.modifies}_{a, js} = \text{Array.modifies}_{a, is \cap js}$

by (*auto simp: Array.modifies-def*)

lemma *UNION-subseteq*:

shows $(\bigcup_{x \in X} \text{Array.modifies}_{a, I x}) \subseteq \text{Array.modifies}_{a, (\bigcup_{x \in X} I x)}$

by (*simp add: Array.modifies.mono Sup-upper UN-least*)

lemma *union-subseteq*:

shows $\text{Array.modifies}_{a, is} \cup \text{Array.modifies}_{a, js} \subseteq \text{Array.modifies}_{a, is \cup js}$

by (*simp add: Array.modifies.mono*)

lemma *Diag-subseteq*:

assumes $\bigwedge s. P s \implies \text{heap.rep-inv } a s$

shows $\text{Diag } P \subseteq \text{Array.modifies}_{a, is}$

using *assms* **by** (*auto simp: Array.modifies-def heap.modifies-def Diag-def*)

lemma *get*:

assumes $(s, s') \in \text{Array.modifies}_{a, is}$

assumes $i \in \text{set } (\text{Array.interval } a) - is$

shows $\text{Array.get } a i s' = \text{Array.get } a i s$

using *assms* **by** (*simp add: Array.modifies-def*)

lemma *set*:

assumes $\text{heap.rep-inv } a s$

shows $(s, \text{Array.set } a \ i \ v \ s) \in \text{heap.modifies}_{\{a\}}$
using *assms*
by (*simp add: heap.modifies-def Array.set-def OArray.set-def heap.set-def heap.get-def rep-val-inv-array-def*
split: heap.obj-at.splits)

lemma *Array-set*:
assumes *heap.rep-inv a s*
assumes $i \in \text{set } (\text{Array.interval } a) \cap \text{is}$
shows $(s, \text{Array.set } a \ i \ v \ s) \in \text{Array.modifies}_{a, \text{is}}$
using *assms*
by (*auto simp: Array.modifies-def Array.rep-inv.set Array.modifies.set*
intro: Array.get.set-neq[symmetric])

lemma *Array-set-conv*:
assumes $i \in \text{set } (\text{Array.interval } a) \cap \text{is}$
shows $(s, \text{Array.set } a \ i \ v \ s) \in \text{Array.modifies}_{a, \text{is}} \longleftrightarrow \text{heap.rep-inv } a \ s \ (\mathbf{is} \ ?lhs \longleftrightarrow \ ?rhs)$
proof(*rule iffI*)
show $?lhs \implies ?rhs$
using *heap.modifies.rep-inv-rel-le[of {a}, simplified]* **by** (*auto simp: Array.modifies-def*)
from *assms* **show** $?rhs \implies ?lhs$
by (*simp add: Array.modifies.Array-set*)
qed

setup $\langle \text{Sign.parent-path} \rangle$

lemmas *simps'* =
Array.rep-inv.set
Array.get.set-eq

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path heap.Id-on.Array} \rangle$

lemma *Id-on-le*:
shows $\text{heap.Id}_{\{a\}} \subseteq \text{Array.Id}_{a, \text{is}}$
by (*auto simp: Array.Id-on-def heap.Id-on-def Array.get-def OArray.get-def heap.get-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path Array.Id-on} \rangle$

lemma *empty*:
shows $\text{Array.Id}_{a, \{\}} = \text{heap.rep-inv-rel } a$
by (*simp add: Array.Id-on-def*)

lemma *mono*:
assumes $\text{is} \subseteq \text{js}$
shows $\text{Array.Id}_{a, \text{js}} \subseteq \text{Array.Id}_{a, \text{is}}$
using *assms* **by** (*auto simp: Array.Id-on-def*)

lemma *insert*:
shows $\text{Array.Id}_{a, \text{insert } i \ \text{is}} = \text{Array.Id}_{a, \{i\}} \cap \text{Array.Id}_{a, \text{is}}$
by (*fastforce simp: Array.Id-on-def*)

lemma *union*:
shows $\text{Array.Id}_{a, \text{is} \cup \text{js}} = \text{Array.Id}_{a, \text{is}} \cap \text{Array.Id}_{a, \text{js}}$
by (*fastforce simp: Array.Id-on-def*)

lemma *rep-inv-rel*:

shows $\text{Array.Id}_{a, is} \subseteq \text{heap.rep-inv-rel } a$
by (*simp add: Array.Id-on-def*)

lemma *eq-heap-Id-on*:

assumes $\text{set } (\text{Array.interval } a) \subseteq is$
shows $\text{Array.Id}_{a, is} = \text{heap.Id}_{\{a\}}$
by (*rule antisym[OF - heap.Id-on.Array.Id-on-le]*)
(*use assms in <force simp: Array.Id-on-def heap.Id-on-def elim: Array.get.ext>*)

setup $\langle \text{Sign.parent-path} \rangle$

25.2.3 Stability

setup $\langle \text{Sign.mandatory-path stable.heap.Id-on.Array} \rangle$

lemma *get[stable.intro]*:

assumes $a \in as$
shows $\text{stable heap.Id}_{as} (\lambda s. P (\text{Array.get } a \ i \ s))$
using *assms by (auto simp: stable-def monotone-def heap.Id-on-def cong: Array.get.weak-cong)*

lemma *get-chain*: — difficult to apply

assumes $\bigwedge v. \text{stable heap.Id}_{as} (P \ v)$
assumes $a \in as$
shows $\text{stable heap.Id}_{as} (\lambda s. P (\text{Array.get } a \ i \ s) \ s)$
using *assms by (auto simp: stable-def monotone-def heap.Id-on-def cong: Array.get.weak-cong)*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable.Array.Id-on.Array} \rangle$

lemma *get[stable.intro]*:

assumes $i \in is$
shows $\text{stable Array.Id}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s))$
using *assms by (auto simp: stable-def monotone-def Array.Id-on-def)*

lemma *get-chain*: — difficult to apply

assumes $\bigwedge v. \text{stable Array.Id}_{a, is} (P \ v)$
assumes $i \in is$
shows $\text{stable Array.Id}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s) \ s)$
using *assms by (auto simp: stable-def monotone-def Array.Id-on-def)*

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable.heap.Array.Id-on.heap} \rangle$

lemma *rep-inv[stable.intro]*:

shows $\text{stable Array.Id}_{a, is} (\text{heap.rep-inv } a)$
by (*simp add: stable-def monotone-def Array.Id-on-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable.heap.Array.modifies.heap} \rangle$

lemma *rep-inv[stable.intro]*:

shows $\text{stable Array.modifies}_{a, is} (\text{heap.rep-inv } a)$
by (*simp add: stable-def monotone-def Array.modifies-def heap.modifies-def*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path stable.heap.Array.modifies.Array} \rangle$

lemma *get[stable.intro]*:

assumes $i \in \text{set } (\text{Array.interval } a) - is$

shows $\text{stable Array.modifies}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s))$

using *assms* **by** (*simp add: stable-def monotone-def Array.modifies-def*)

lemma *get-chain*: — difficult to apply

assumes $\bigwedge v. \text{stable Array.modifies}_{a, is} (P \ v)$

assumes $i \in \text{set } (\text{Array.interval } a) - is$

shows $\text{stable Array.modifies}_{a, is} (\lambda s. P (\text{Array.get } a \ i \ s) \ s)$

using *assms* **by** (*simp add: stable-def monotone-def Array.modifies-def*)

setup $\langle \text{Sign.parent-path} \rangle$

26 A concurrent variant of Imperative HOL

We model programs operating on sequentially-consistent memory with the type $(\text{heap.t}, 'v) \text{ prog}$.

Source materials:

- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Heap_Monad.thy`
- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Array.thy`
- `$ISABELLE_HOME/src/HOL/Imperative_HOL/Ref.thy`

– note that ImperativeHOL is deterministic and sequential

type-synonym $'v \text{ imp} = (\text{heap.t}, 'v) \text{ prog}$

setup $\langle \text{Sign.mandatory-path prog} \rangle$

definition *raise* $:: \text{String.literal} \Rightarrow 'a \text{ imp}$ **where** — the literal is just decoration

raise $s = \perp$

definition *assert* $:: \text{bool} \Rightarrow \text{unit imp}$ **where**

assert $P = (\text{if } P \text{ then prog.return } () \text{ else prog.raise STR "assert"})$

setup $\langle \text{Sign.mandatory-path Ref} \rangle$

definition *ref* $:: 'a::\text{heap.rep} \Rightarrow 'a \text{ ref imp}$ **where**

ref $v = \text{prog.action } \{(r, s, s'). (r, s') \in \text{Ref.alloc } v \ s\}$

definition *lookup* $:: 'a::\text{heap.rep} \text{ ref} \Rightarrow 'a \text{ imp}$ ($\langle !- \rangle$ 61) **where**

lookup $r = \text{prog.read } (\text{Ref.get } r)$

definition *update* $:: 'a \text{ ref} \Rightarrow 'a::\text{heap.rep} \Rightarrow \text{unit imp}$ ($\langle - := - \rangle$ 62) **where**

update $r \ v = \text{prog.write } (\text{Ref.set } r \ v)$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path Array} \rangle$

definition *new* $:: ('i \times 'i) \Rightarrow 'a \Rightarrow ('i::Ix, 'a::\text{heap.rep}) \text{ array imp}$ **where**

new $b \ v = \text{prog.action } \{(\text{Array } b \ a, s, s') \mid a \ s \ s'. (a, s') \in \text{ODArray.alloc } (\text{replicate } (\text{length } (Ix.interval \ b)) \ v) \ s\}$

definition *make* :: ('i × 'i) ⇒ ('i ⇒ 'a) ⇒ ('i::Ix, 'a::heap.rep) array imp **where**
make b f = prog.action {(Array b a, s, s') | a s s'. (a, s') ∈ OArray.alloc (map f (Ix.interval b)) s}

— Approximately Haskell’s *listArray*: “Construct an array from a pair of bounds and a list of values in index order.”

definition *of-list* :: ('i × 'i) ⇒ 'a list ⇒ ('i::Ix, 'a::heap.rep) array imp **where**
of-list b xs = prog.action {(Array b a, s, s') | a s s'. length (Ix.interval b) ≤ length xs ∧ (a, s') ∈ OArray.alloc xs s}

definition *nth* :: ('i::Ix, 'a::heap.rep) array ⇒ 'i ⇒ 'a imp **where**
nth a i = prog.read (λs. Array.get a i s)

definition *upd* :: ('i::Ix, 'a::heap.rep) array ⇒ 'i ⇒ 'a ⇒ unit imp **where**
upd a i v = prog.write (Array.set a i v)

— derived operations; observe the lack of atomicity

definition *freeze* :: ('i::Ix, 'a::heap.rep) array ⇒ 'a list imp **where**
freeze a = prog.fold-mapM (prog.Array.nth a) (Array.interval a)

definition *swap* :: ('i::Ix, 'a::heap.rep) array ⇒ 'i ⇒ 'i ⇒ unit imp **where**

swap a i j =
do {
x ← prog.Array.nth a i;
y ← prog.Array.nth a j;
prog.Array.upd a i y;
prog.Array.upd a j x;
prog.return ()
}

declare prog.raise-def[code del]
declare prog.Ref.ref-def[code del]
declare prog.Ref.lookup-def[code del]
declare prog.Ref.update-def[code del]
declare prog.Array.new-def[code del]
declare prog.Array.make-def[code del]
declare prog.Array.of-list-def[code del]
declare prog.Array.nth-def[code del]
declare prog.Array.upd-def[code del]
declare prog.Array.freeze-def[code del]

Operations on two-dimensional arrays **definition** *fst-app-chaotic* :: ('a::Ix, 'b::Ix) two-dim ⇒ ('a ⇒ ('s, unit) prog) ⇒ ('s, unit) prog **where**

fst-app-chaotic b f = prog.set-app f (set (Ix.interval (fst-bounds b)))

definition *fst-app* :: ('a::Ix, 'b::Ix) two-dim ⇒ ('a ⇒ ('s, unit) prog) ⇒ ('s, unit) prog **where**
fst-app b f = prog.app f (Ix.interval (fst-bounds b))

lemma *fst-app-fst-app-chaotic-le*:

shows prog.Array.fst-app b f ≤ prog.Array.fst-app-chaotic b f

unfolding prog.Array.fst-app-chaotic-def prog.Array.fst-app-def

by (strengthen ord-to-strengthen(1)[OF prog.app.set-app-le]) (auto simp: distinct-interval)

setup <Sign.parent-path>

setup <Sign.parent-path>

```
setup <Sign.mandatory-path ag.prog>
```

```
lemmas fst-app-chaotic =
```

```
  ag.prog.app-set[where X=set (Ix.interval (fst-bounds b)) for b, folded prog.Array.fst-app-chaotic-def]
```

```
lemmas fst-app =
```

```
  ag.prog.app[where xs=Ix.interval (fst-bounds b) for b, folded prog.Array.fst-app-def]
```

```
setup <Sign.parent-path>
```

26.1 Code generator setup

26.1.1 Haskell

```
code-printing code-module Heap → (Haskell)
```

```
<
-- Sequentially-consistent primitives
-- Arrays:
-- https://hackage.haskell.org/package/array-0.5.4.0/docs/Data-Array-IO.html
-- https://hackage.haskell.org/package/array-0.5.4.0/docs/src/Data.Array.Base.html
module Heap (
  Prog
  , Ref, newIORef, readIORef, writeIORef
  , Array, newArray, newListArray, newFunArray, readArray, writeArray
  , parallel
  ) where
```

```
import Control.Concurrent (forkIO)
import qualified Control.Concurrent.MVar as MVar
import qualified Data.Array.IO as Array
import Data.IORef (IORef, newIORef, readIORef, atomicWriteIORef)
import Data.List (genericLength)
```

```
type Prog a b = IO b
type Array a = Array.IOArray Integer a
type Ref a = Data.IORef.IORef a
```

```
writeIORef :: IORef a -> a -> IO ()
writeIORef = atomicWriteIORef -- could use the strict variant
```

```
newArray :: Integer -> a -> IO (Array a)
newArray k = Array.newArray (0, k - 1)
```

```
newFunArray :: Integer -> (Integer -> a) -> IO (Array a)
newFunArray k f = Array.newListArray (0, k - 1) (map f [0..k-1])
```

```
newListArray :: Integer -> [a] -> IO (Array a)
newListArray k xs = Array.newListArray (0, k) xs
```

```
readArray :: Array a -> Integer -> IO a
readArray = Array.readArray
```

```
writeArray :: Array a -> Integer -> a -> IO ()
writeArray = Array.writeArray -- probably should be the WMM atomic op
```

```
{-
-- 'forkIO' is reputedly cheap, but other papers imply the use of worker threads, perhaps for other reasons
-- note we don't want forkFinally as we don't model exceptions
parallel' :: IO a -> IO b -> IO (a, b)
parallel' p q = do
```

```

mvar <- MVar.newEmptyMVar
forkIO (p >>= MVar.putMVar mvar) -- note putMVar is lazy
b <- q
a <- MVar.takeMVar mvar
return (a, b)
-}

```

```

parallel :: IO () -> IO () -> IO ()
parallel p q = do
  mvar <- MVar.newEmptyMVar
  forkIO (p >> MVar.putMVar mvar ()) -- note putMVar is lazy
  b <- q
  a <- MVar.takeMVar mvar
  return ()
>

```

code-reserved (*Haskell*) *Ix*

code-printing type-constructor *prog* \rightarrow (*Haskell*) *Heap.Prog* - -
code-monad *prog.bind Haskell*
code-printing constant *prog.return* \rightarrow (*Haskell*) *return*
code-printing constant *prog.raise* \rightarrow (*Haskell*) *error*
code-printing constant *prog.parallel* \rightarrow (*Haskell*) *Heap.parallel*

Intermediate operation avoids invariance problem in *Scala* (similar to value restriction)

setup \langle *Sign.mandatory-path Ref* \rangle

definition *ref'* **where**

[*code del*]: *ref'* = *prog.Ref.ref*

lemma [*code*]:

prog.Ref.ref x = *Ref.ref' x*

by (*simp add: Ref.ref'-def*)

setup \langle *Sign.parent-path* \rangle

code-printing type-constructor *ref* \rightarrow (*Haskell*) *Heap.Ref* -
code-printing constant *Ref* \rightarrow (*Haskell*) *error/ bare Ref*
code-printing constant *Ref.ref'* \rightarrow (*Haskell*) *Heap.newIORef*
code-printing constant *prog.Ref.lookup* \rightarrow (*Haskell*) *Heap.readIORef*
code-printing constant *prog.Ref.update* \rightarrow (*Haskell*) *Heap.writeIORef*
code-printing constant *HOL.equal* :: '*a ref* \Rightarrow '*a ref* \Rightarrow *bool* \rightarrow (*Haskell*) **infix** 4 ==
code-printing class-instance *ref* :: *HOL.equal* \rightarrow (*Haskell*) -

The target language only has to provide one-dimensional arrays indexed by *integer*.

setup \langle *Sign.mandatory-path prog.Array* \rangle

definition *new'* :: *integer* \Rightarrow '*a* \Rightarrow '*a::heap.rep one-dim-array imp* **where**

new' k v = *prog.action* $\{(a, s, s') \mid a \ s \ s'. (a, s') \in ODArray.alloc (replicate (nat-of-integer k) v) s\}$

declare *prog.Array.new'-def*[*code del*]

lemma *new-new'*[*code*]:

shows *prog.Array.new b v* = *prog.Array.new' (of-nat (length (Ix.interval b))) v* \gg *prog.return* \circ *Array b*
by (*force simp: prog.Array.new-def prog.Array.new'-def prog.vmap.action*
simp flip: prog.vmap.eq-return
intro: arg-cong[where f=prog.action])

definition $make' :: integer \Rightarrow (integer \Rightarrow 'a) \Rightarrow 'a :: heap.rep\ one-dim-array\ imp$ **where**
 $make' k f = prog.action \{(a, s, s') \mid a\ s\ s'. (a, s') \in ODArry.alloc (map (f \circ of-nat) [0..<nat-of-integer\ k])\ s\}$

declare $prog.Array.make'-def[code\ del]$

lemma $make-make'[code]:$

shows $prog.Array.make\ b\ f$
 $= prog.Array.make' (of-nat (length (Ix.interval\ b))) (\lambda i. f (Ix.interval\ b\ !\ nat-of-integer\ i))$
 $\ggg prog.return \circ Array\ b$

by ($force\ simp: interval-map\ prog.Array.make-def\ prog.Array.make'-def\ prog.vmap.action\ comp-def$
 $simp\ flip: prog.vmap.eq-return$
 $intro: arg-cong[where\ f=prog.action]$)

definition $of-list' :: integer \Rightarrow 'a\ list \Rightarrow 'a :: heap.rep\ one-dim-array\ imp$ **where**

$of-list' k xs = prog.action \{(a, s, s') \mid a\ s\ s'. nat-of-integer\ k \leq length\ xs \wedge (a, s') \in ODArry.alloc\ xs\ s\}$

declare $prog.Array.of-list'-def[code\ del]$

lemma $of-list-of-list'[code]:$

shows $prog.Array.of-list\ b\ xs$
 $= prog.Array.of-list' (of-nat (length (Ix.interval\ b))) xs \ggg prog.return \circ Array\ b$

by ($force\ simp: prog.Array.of-list-def\ prog.Array.of-list'-def\ prog.vmap.action$
 $simp\ flip: prog.vmap.eq-return$
 $intro: arg-cong[where\ f=prog.action]$)

definition $nth' :: 'a :: heap.rep\ one-dim-array \Rightarrow integer \Rightarrow 'a\ imp$ **where**

$nth' a i = prog.read (ODArray.get\ a (nat-of-integer\ i))$

declare $prog.Array.nth'-def[code\ del]$

lemma $nth-nth'[code]:$

shows $prog.Array.nth\ a\ i = prog.Array.nth' (array.arr\ a) (of-nat (Array.index\ a\ i))$

by ($simp\ add: prog.Array.nth-def\ prog.Array.nth'-def\ Array.get-def$)

definition $upd' :: 'a :: heap.rep\ one-dim-array \Rightarrow integer \Rightarrow 'a :: heap.rep \Rightarrow unit\ imp$ **where**

$upd' a i v = prog.write (ODArray.set\ a (nat-of-integer\ i)\ v)$

declare $prog.Array.upd'-def[code\ del]$

lemma $upd-upd'[code]:$

shows $prog.Array.upd\ a\ i\ v = prog.Array.upd' (array.arr\ a) (of-nat (Array.index\ a\ i))\ v$

by ($simp\ add: prog.Array.upd-def\ prog.Array.upd'-def\ Array.set-def$)

setup $\langle Sign.parent-path \rangle$

code-printing type-constructor $one-dim-array \rightarrow (Haskell)\ Heap.Array / -$

code-printing constant $one-dim.array.Array \rightarrow (Haskell)\ error / bare\ Array$

code-printing constant $prog.Array.new' \rightarrow (Haskell)\ Heap.newArray$

code-printing constant $prog.Array.make' \rightarrow (Haskell)\ Heap.newFunArray$

code-printing constant $prog.Array.of-list' \rightarrow (Haskell)\ Heap.newListArray$

code-printing constant $prog.Array.nth' \rightarrow (Haskell)\ Heap.readArray$

code-printing constant $prog.Array.upd' \rightarrow (Haskell)\ Heap.writeArray$

code-printing constant $HOL.equal :: ('i, 'a)\ array \Rightarrow ('i, 'a)\ array \Rightarrow bool \rightarrow (Haskell)\ infix\ 4\ ==$

code-printing class-instance $array :: HOL.equal \rightarrow (Haskell)\ -$

26.2 Value-returning parallel

definition $parallelP' :: 'a :: heap.rep\ imp \Rightarrow 'b :: heap.rep\ imp \Rightarrow ('a \times 'b)\ imp$ **where**

```

parallelP' P1 P2 = do {
  r1 ← prog.Ref.ref undefined
; r2 ← prog.Ref.ref undefined
; ((P1 ≧≧ prog.Ref.update r1) || (P2 ≧≧ prog.Ref.update r2))
; v1 ← prog.Ref.lookup r1
; v2 ← prog.Ref.lookup r2
; prog.return (v1, v2)
}

```

27 Total store order (TSO)

The total store order (TSO) memory model (Owens, Sarkar, and Sewell (2009); valid on multicore x86) can be modelled as a closure as demonstrated by Jagadeesan, Petri, and Riely (2012, p182). Essentially this is done by incorporating a write buffer into each thread’s local state and adding buffer draining opportunities before and after every command. The only subtlety is that the all threads involved in a parallel composition need to start and end with empty write buffers (see §27).

We configure the code generator in §27.3.

Comparison with Jagadeesan et al. (2012):

- We ignore mumbling-related issues and it doesn’t make any difference
 - in our model we commit writes one at a time; mumbling allows several to be committed at once (p182) which we model as an uninterrupted sequence of individual writes
 - if we allowed *commit-writes* to commit multiple writes in a single step then *tso-closure* would not be idempotent
- their semantics is for terminating computations only; ours is for safety only
- their language is deterministic, ours is non-deterministic
- They do not provide many general laws for TSO
- Their claims that the semantics allows them to prove things (§5) is not substantiated

type-synonym *write-buffer* = *heap.write list*

definition *apply-writes* :: *write-buffer* ⇒ *heap.t* ⇒ *heap.t* **where**
apply-writes ws = *fold* (λ*w*. (○) (*heap.apply-write w*)) *ws id*

lemma *apply-write-present*:

assumes *heap.present r s*

shows *heap.present r (heap.apply-write w s)*

using *assms* **by** (*cases w*) (*simp split: heap.obj-at.splits*)

lemma *apply-writes-present*:

assumes *heap.present r s*

shows *heap.present r (apply-writes wb s)*

using *assms* **by** (*induct wb arbitrary: s*) (*simp-all add: apply-writes-def fold-fun apply-write-present*)

setup ⟨*Sign.mandatory-path raw*⟩

type-synonym *'v tso* = *write-buffer* ⇒ (*heap.t*, *'v × write-buffer*) *prog*

definition *bind* :: *'a raw.tso* ⇒ (*'a* ⇒ *'b raw.tso*) ⇒ *'b raw.tso* **where**

bind f g = (λ*wb*. *f wb* ≧≧ *uncurry g*)

ad hoc-overloading

Monad-Syntax.bind ≡ *raw.bind*

definition *prim-return* :: 'a ⇒ 'a *raw.tso* **where**

prim-return v = (λwb. *prog.return* (v, wb))

setup <*Sign.mandatory-path* *bind*>

lemma *mono*:

assumes $f \leq f'$

assumes $\bigwedge x. g\ x \leq g'\ x$

shows *raw.bind* f g ≤ *raw.bind* f' g'

using *assms* **by** (*fastforce simp: raw.bind-def prog.bind.mono le-fun-def intro: prog.bind.mono*)

lemma *strengthen[stg]*:

assumes *st-ord* F f f'

assumes $\bigwedge x. \text{st-ord } F (g\ x) (g'\ x)$

shows *st-ord* F (*raw.bind* f g) (*raw.bind* f' g')

using *assms* **by** (*cases* F; *clarsimp intro!: raw.bind.mono*)

lemma *mono2mono[cont-intro, partial-function-mono]*:

assumes *monotone orda* (≤) F

assumes $\bigwedge x. \text{monotone orda } (\leq) (\lambda y. G\ y\ x)$

shows *monotone orda* (≤) (λf. *raw.bind* (F f) (G f))

using *assms* **unfolding** *monotone-def* **by** (*meson raw.bind.mono*)

lemma *botL*:

shows *raw.bind* ⊥ g = ⊥

by (*simp add: raw.bind-def fun-eq-iff prog.bind.botL*)

lemma *bind*:

fixes f :: - *raw.tso*

shows $f \ggg g \ggg h = f \ggg (\lambda x. g\ x \ggg h)$

by (*simp add: raw.bind-def fun-eq-iff split-def prog.bind.bind*)

lemma *prim-return*:

shows *prim-returnL*: *raw.bind* (*raw.prim-return* v) = (λg. g v)

and *prim-returnR*: $f \ggg \text{raw.prim-return} = f$

by (*simp-all add: fun-eq-iff raw.prim-return-def raw.bind-def split-def prog.bind.return*)

lemma *supL*:

fixes g :: - ⇒ - *raw.tso*

shows $f_1 \sqcup f_2 \ggg g = (f_1 \ggg g) \sqcup (f_2 \ggg g)$

by (*simp add: raw.bind-def fun-eq-iff prog.bind.supL*)

lemma *supR*:

fixes f :: - *raw.tso*

shows $f \ggg (\lambda v. g_1\ v \sqcup g_2\ v) = (f \ggg g_1) \sqcup (f \ggg g_2)$

by (*simp add: raw.bind-def fun-eq-iff split-def prog.bind.supR*)

lemma *SUPL*:

fixes X :: - *set*

fixes f :: - ⇒ - *raw.tso*

shows $(\bigsqcup_{x \in X}. f\ x) \ggg g = (\bigsqcup_{x \in X}. f\ x \ggg g)$

by (*simp add: raw.bind-def fun-eq-iff prog.bind.SUPL image-image*)

lemma *SUPR*:

fixes X :: - *set*

fixes f :: - *raw.tso*

shows $f \ggg (\lambda v. \bigsqcup_{x \in X}. g\ x\ v) = (\bigsqcup_{x \in X}. f \ggg g\ x) \sqcup (f \ggg \perp)$

by (simp add: raw.bind-def split-def fun-eq-iff image-image bot-fun-def prog.bind.SUPR)

lemma SUPR-not-empty:

fixes $f :: - \text{raw.tso}$

assumes $X \neq \{\}$

shows $f \ggg (\lambda v. \bigsqcup_{x \in X}. g x v) = (\bigsqcup_{x \in X}. f \ggg g x)$

by (simp add: raw.bind-def split-def fun-eq-iff image-image prog.bind.SUPR-not-empty[OF assms])

lemma mcont2mcont[cont-intro]:

assumes mcont luba orda Sup (\leq) f

assumes $\bigwedge v. \text{mcont luba orda Sup } (\leq) (\lambda x. g x v)$

shows mcont luba orda Sup (\leq) $(\lambda x. \text{raw.bind } (f x) (g x))$

proof(rule cppo.mcont2mcont'[OF complete-lattice-ccpo - - assms(1)])

show mcont Sup (\leq) Sup (\leq) $(\lambda f. \text{raw.bind } f (g x))$ for x

by (intro mcontI contI monotoneI) (simp-all add: raw.bind.mono flip: raw.bind.SUPL)

show mcont luba orda Sup (\leq) $(\lambda x. \text{raw.bind } f (g x))$ for f

by (intro mcontI monotoneI contI)

(simp-all add: mcont-monoD[OF assms(2)] raw.bind.mono flip: raw.bind.SUPR-not-empty contD[OF mcont-cont[OF assms(2)]])

qed

setup $\langle \text{Sign.parent-path} \rangle$

interpretation kleene: kleene raw.prim-return () $\lambda x y. \text{raw.bind } x \langle y \rangle$

by standard (simp-all add: raw.bind.prim-return raw.bind.botL raw.bind.bind raw.bind.supL raw.bind.supR)

primrec commit-write :: unit raw.tso **where**

commit-write [] = prog.return ((), [])

| commit-write (w # wb) = prog.action {(((), wb), h, heap.apply-write w h) |h. True}

definition commit-writes :: unit raw.tso **where**

commit-writes = raw.kleene.star raw.commit-write

setup $\langle \text{Sign.mandatory-path tso} \rangle$

definition cl :: 'v raw.tso \Rightarrow 'v raw.tso **where**

cl P = raw.commit-writes \ggg P \ggg $(\lambda v. \text{raw.commit-writes } \ggg \text{raw.prim-return } v)$

setup $\langle \text{Sign.parent-path} \rangle$

definition action :: (write-buffer \Rightarrow ('v \times write-buffer \times heap.t \times heap.t) set) \Rightarrow 'v raw.tso **where**

action F = raw.tso.cl $(\lambda wb. \text{prog.action } \{((v, wb @ ws), ss') \mid v ss' ws. (v, ws, ss') \in F wb\})$

definition return :: 'v \Rightarrow 'v raw.tso **where**

return v = raw.action $\langle \{v\} \times \{\}\times \text{Id} \rangle$

definition guard :: (write-buffer \Rightarrow heap.t pred) \Rightarrow unit raw.tso **where**

guard g = raw.action $(\lambda wb. \{\} \times \{\}\times \text{Diag } (g wb))$

definition MFENCE :: unit raw.tso **where**

MFENCE = raw.guard $(\lambda wb s. wb = \{\})$

definition vmap :: ('v \Rightarrow 'w) \Rightarrow 'v raw.tso \Rightarrow 'w raw.tso **where**

vmap vf P = $(\lambda wb. \text{prog.vmap } (\text{map-prod } vf \text{ id}) (P wb))$

— Parallel composition

definition t2p :: 'v raw.tso \Rightarrow (heap.t, 'v) prog **where**

t2p P = P [] \ggg $(\lambda(v, wb). \text{raw.MFENCE } wb \gg \text{prog.return } v)$

— Jagadeesan et al. (2012, p184 rule PAR-CMD): perform MFENCE before fork

definition *parallel* :: *unit raw.tso* \Rightarrow *unit raw.tso* \Rightarrow *unit raw.tso* **where**
parallel P Q = *raw.MFENCE* \gg \langle (*raw.t2p P* || *raw.t2p Q*) \gg *prog.return* (*()*, *[]*) \rangle

lemma *return-alt-def*:

shows *raw.return* = (λv . *raw.tso.cl* (*raw.prim-return v*))

by (*fastforce simp*: *raw.return-def raw.action-def raw.prim-return-def prog.return-def*
intro: *arg-cong*[**where** *f*= λP . *raw.tso.cl P wb* **for** *wb*] *arg-cong*[**where** *f*=*prog.action*])

setup \langle *Sign.mandatory-path commit-writes* \rangle

lemma *return-le*:

shows *raw.prim-return* *()* \leq *raw.commit-writes*

unfolding *raw.commit-writes-def* **by** (*subst raw.kleene.star.simps*) *simp*

lemma *return-le'*:

shows *prog.return* *()*, *wb* \leq *raw.commit-writes wb*

using *raw.commit-writes.return-le* **by** (*simp add*: *raw.prim-return-def le-fun-def*)

lemma *commit-writes*:

shows *raw.commit-writes* \gg *raw.commit-writes* = *raw.commit-writes*

by (*simp add*: *raw.commit-writes-def raw.kleene.star-comp-star*)

lemma *Nil*:

shows *raw.commit-writes* *[]* = *prog.return* *()*, *[]* (**is** *?lhs* = *?rhs*)

proof(*rule antisym*)

show *?lhs* \leq *?rhs*

unfolding *raw.commit-writes-def*

by (*induct rule*: *raw.kleene.star.fixp-induct*)

(*simp-all add*: *raw.bind-def raw.prim-return-def prog.bind.returnL prog.p2s.bot spec.bind.mono*)

show *?rhs* \leq *?lhs*

unfolding *raw.commit-writes-def*

by (*subst raw.kleene.star.simps*) (*simp add*: *raw.bind-def raw.prim-return-def*)

qed

lemma *Cons*:

shows *raw.commit-writes* (*w # wb*)

= (*raw.commit-write* [*w*] \gg *raw.commit-writes wb*) \sqcup *raw.prim-return* *()* (*w # wb*)

apply (*simp add*: *raw.commit-writes-def*)

apply (*subst* (*1*) *raw.kleene.star.simps*)

apply (*subst* (*1*) *raw.bind-def*)

apply *simp*

apply (*subst prog.action.return-const*[**where** *F*={(*s*, *heap.apply-write w s*) |*s*. *True*} **and** *V*={(*()*, *wb*)} **and**
W={(*()*, *[]*)}, *simplified Pair-image*[*symmetric*] *image-def*, *simplified*])

apply (*simp add*: *prog.bind.bind prog.bind.returnL*)

done

lemma *Cons-le*:

shows *raw.commit-write* [*w*] \gg *raw.commit-writes wb* \leq *raw.commit-writes* (*w # wb*)

by (*simp add*: *raw.commit-writes.Cons*)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path spec.singleton.raw* \rangle

lemma *prim-return-Nil-le*:

shows $\langle s, [], \text{Some } ((), wb) \rangle \leq \text{prog.p2s } (\text{raw.prim-return } () \text{ } wb)$
by (*simp add*: *raw.prim-return-def prog.p2s.return spec.interference.cl.return*
spec.bind.continueI[where xs=[], simplified] spec.singleton.le-conv)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.singleton.raw.commit-writes} \rangle$

lemma *noop-le*:

shows $\langle s, [], \text{Some } ((), wb) \rangle \leq \text{prog.p2s } (\text{raw.commit-writes } wb)$
unfolding *raw.commit-writes-def*
by (*rule order.trans[OF spec.singleton.raw.prim-return-Nil-le*
prog.p2s.mono[OF le-funD[OF raw.kleene.epsilon-star-le]]])

lemma *wb-suffix*:

assumes $\langle s, xs, \text{Some } ((), wb') \rangle \leq \text{prog.p2s } (\text{raw.commit-writes } wb)$
shows *suffix wb' wb*
using *assms*
by (*induct wb arbitrary: s xs*)
(*auto simp: raw.commit-writes.Nil raw.commit-writes.Cons raw.prim-return-def*
prog.p2s.simps prog.p2s.return spec.interference.cl.return
trace.split-all spec.singleton.le-conv
suffix-ConsI
elim!: spec.singleton.bind-le)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path raw} \rangle$

setup $\langle \text{Sign.mandatory-path tso.cl} \rangle$

lemma *bind-commit-writes-absorbL*:

fixes $P :: 'v \text{ raw.tso}$
shows *raw.commit-writes* $\gg \text{raw.tso.cl } P = \text{raw.tso.cl } P$
by (*simp add: raw.tso.cl-def raw.commit-writes.commit-writes flip: raw.bind.bind*)

lemma *bind-commit-writes-absorb-unitR*:

fixes $P :: \text{unit raw.tso}$
shows *raw.tso.cl } P \gg \text{raw.commit-writes} = \text{raw.tso.cl } P
by (*simp add: raw.tso.cl-def raw.bind.bind raw.commit-writes.commit-writes raw.bind.prim-returnR*)*

lemma *bind-commit-writes-absorbR*:

fixes $P :: 'v \text{ raw.tso}$
shows *raw.tso.cl } P \gg (\lambda v. \text{raw.commit-writes} \gg \text{raw.prim-return } v) = \text{raw.tso.cl } P
by (*simp add: raw.tso.cl-def raw.bind.bind raw.commit-writes.commit-writes raw.bind.prim-returnL*)
(*simp add: raw.commit-writes.commit-writes flip: raw.bind.bind*)*

lemma *bot*:

shows *raw.tso.cl } \perp = \text{raw.commit-writes} \gg \perp
by (*simp add: raw.tso.cl-def raw.bind.bind raw.bind.botL flip: bot-fun-def*)*

lemma *prim-return*:

shows *raw.tso.cl } (\text{raw.prim-return } v) = \text{raw.commit-writes} \gg \text{raw.prim-return } v
by (*simp add: raw.tso.cl-def raw.bind.bind raw.bind.prim-returnL*)
(*simp add: raw.commit-writes.commit-writes flip: raw.bind.bind*)*

lemma *Nil*:

shows $raw.tso.cl P \sqcap = P \sqcap \ggg (\lambda v. raw.commit-writes (snd v) \ggg (\lambda w. prog.return (fst v, snd w)))$
by (*simp add: raw.tso.cl-def raw.bind-def raw.prim-return-def raw.commit-writes.Nil prog.bind.returnL split-def*)

lemma *commit:*

fixes $wb :: write-buffer$
shows $raw.commit-write [w] \ggg f wb \leq raw.tso.cl f (w \# wb)$
apply (*simp add: raw.tso.cl-def raw.bind-def raw.prim-return-def split-def*)
apply (*strengthen ord-to-strengthen[OF raw.commit-writes.Cons-le]*)
apply (*simp add: prog.bind.bind*)
apply (*rule prog.bind.mono[OF order.refl]*)
apply (*strengthen ord-to-strengthen[OF raw.commit-writes.return-le]*)
apply (*simp add: prog.bind.returnL prog.bind.returnR*)
done

setup $\langle Sign.parent-path \rangle$

interpretation *tso: closure-complete-distrib-lattice-distributive-class raw.tso.cl*

proof *standard*

show $(x \leq raw.tso.cl y) = (raw.tso.cl x \leq raw.tso.cl y)$ **for** $x y :: 'a raw.tso$
proof(*intro iffD2[OF order-class.order.closure-axioms-alt-def[unfolded closure-axioms-def], rule-format, simplified conj-explode] allI*)

show $P \leq raw.tso.cl P$ **for** $P :: 'a raw.tso$

unfolding *raw.tso.cl-def*

by (*strengthen ord-to-strengthen[OF raw.commit-writes.return-le]*)

(*simp add: raw.bind.prim-returnL raw.bind.prim-returnR*)

show *mono raw.tso.cl*

proof(*rule monotoneI*)

fix $P P' :: 'v raw.tso$

assume $P \leq P'$ **show** $raw.tso.cl P \leq raw.tso.cl P'$

unfolding *raw.tso.cl-def* **by** (*strengthen ord-to-strengthen(1)[OF $\langle P \leq P' \rangle$]*) *simp*

qed

show $raw.tso.cl (raw.tso.cl P) = raw.tso.cl P$ **for** $P :: 'a raw.tso$

by (*simp add: raw.tso.cl-def raw.bind.bind raw.commit-writes.commit-writes raw.bind.prim-returnL*)

(*simp add: raw.commit-writes.commit-writes flip: raw.bind.bind*)

qed

show $raw.tso.cl (\bigsqcup X) \leq \bigsqcup (raw.tso.cl ' X) \sqcup raw.tso.cl \perp$ **for** $X :: 'a raw.tso$ *set*

by (*simp add: raw.tso.cl-def raw.bind.bind raw.bind.botL flip: bot-fun-def raw.bind.SUPR raw.bind.SUPL*)

qed

setup $\langle Sign.mandatory-path tso.cl \rangle$

lemma *bind:*

fixes $f :: 'v raw.tso$

assumes $f \in raw.tso.closed$

shows $raw.tso.cl (f \ggg g) = f \ggg (\lambda v. raw.tso.cl (g v))$

apply (*simp add: raw.tso.cl-def raw.bind.bind*)

apply (*subst (1 2) raw.tso.closed-conv[OF assms(1)]*)

apply (*simp add: raw.tso.cl.bind-commit-writes-absorbL flip: raw.bind.bind*)

apply (*subst (1) raw.tso.cl.bind-commit-writes-absorbR[symmetric]*)

apply (*simp add: raw.bind.bind raw.bind.prim-returnL*)

done

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path bind \rangle$

lemma *commit-writes-absorbL:*

assumes $f \in raw.tso.closed$

shows $\text{raw.commit-writes} \gg f = f$
by (*metis* *assms* $\text{raw.tso.closed-conv}$ $\text{raw.tso.cl.bind-commit-writes-absorbL}$)

lemma *commit-writes-absorb-unitR*:
assumes $f \in \text{raw.tso.closed}$
shows $f \gg \text{raw.commit-writes} = f$
by (*metis* *assms* $\text{raw.tso.closed-conv}$ $\text{raw.tso.cl.bind-commit-writes-absorb-unitR}$)

lemma *returnL*:
assumes $g \ v \in \text{raw.tso.closed}$
shows $\text{raw.return } v \ggg g = g \ v$
by (*simp* *add*: $\text{raw.return-alt-def}$ $\text{raw.bind.commit-writes-absorbL}$
 $\text{raw.tso.cl.prim-return}$ raw.bind.bind $\text{raw.bind.prim-returnL}$)

lemma *returnR*:
assumes $f \in \text{raw.tso.closed}$
shows $f \ggg \text{raw.return} = f$
by (*simp* *add*: $\text{raw.return-alt-def}$ $\text{raw.tso.cl.prim-return}$
 $\text{raw.tso.cl.bind-commit-writes-absorbR}$ [*of* f , *simplified* $\text{raw.tso.closed-conv}$ [*OF* *assms*, *symmetric*]])

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path } \text{tso.closed} \rangle$

lemma *commit-writes*:
shows $\text{raw.commit-writes} \in \text{raw.tso.closed}$
by (*rule* $\text{raw.tso.closed-clI}$)
(*simp* *add*: raw.tso.cl-def $\text{raw.commit-writes.commit-writes}$ $\text{raw.bind.prim-returnR}$
 flip : raw.bind.bind)

lemma *bind[intro]*:
fixes $f :: 'v \ \text{raw.tso}$
fixes $g :: 'v \Rightarrow 'w \ \text{raw.tso}$
assumes $f \in \text{raw.tso.closed}$
assumes $\bigwedge x. g \ x \in \text{raw.tso.closed}$
shows $f \ggg g \in \text{raw.tso.closed}$
by (*simp* *add*: assms $\text{raw.tso.closed-clI}$ raw.tso.cl.bind flip : $\text{raw.tso.closed-conv}$)

lemma *action[intro]*:
shows $\text{raw.action } F \in \text{raw.tso.closed}$
by (*simp* *add*: raw.action-def)

lemma *guard[intro]*:
shows $\text{raw.guard } g \in \text{raw.tso.closed}$
by (*simp* *add*: raw.guard-def $\text{raw.tso.closed.action}$)

lemma *MFENCE[intro]*:
shows $\text{raw.MFENCE} \in \text{raw.tso.closed}$
by (*simp* *add*: raw.MFENCE-def $\text{raw.tso.closed.guard}$)

lemma *parallel[intro]*:
assumes $P \in \text{raw.tso.closed}$
assumes $Q \in \text{raw.tso.closed}$
shows $\text{raw.parallel } P \ Q \in \text{raw.tso.closed}$
apply (*rule* $\text{raw.tso.closed-clI}$)
apply (*clarsimp* *simp*: raw.parallel-def raw.tso.cl-def $\text{raw.bind.prim-returnR}$ *le-fun-def*)
apply (*subst* (2) $\text{raw.bind.commit-writes-absorbL}$ [*OF* $\text{raw.tso.closed.MFENCE}$, *symmetric*])
apply (*simp* *add*: raw.bind-def split-def prog.bind.bind prog.bind.mono [*OF* *order.refl*])

prog.bind.returnL raw.commit-writes.Nil)

done

lemma *vmap*[*intro*]:

assumes $P \in \text{raw.tso.closed}$

shows $\text{raw.vmap } vf \ P \in \text{raw.tso.closed}$

proof(*rule raw.tso.closed-clI*)

have $\text{raw.tso.cl } (\text{raw.vmap } vf \ P) \leq \text{raw.vmap } vf \ (\text{raw.tso.cl } P)$

by (*simp add: le-funI raw.tso.cl-def raw.vmap-def raw.bind-def raw.prim-return-def split-def comp-def prog.vmap.eq-return prog.bind.bind prog.bind.returnL*)

then show $\text{raw.tso.cl } (\text{raw.vmap } vf \ P) \leq \text{raw.vmap } vf \ P$

by (*simp flip: raw.tso.closed-conv[OF assms]*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path action} \rangle$

lemma *bot*:

shows $\text{raw.action } \perp = \text{raw.tso.cl } \perp$

by (*simp add: raw.action-def prog.action.empty bot-fun-def*)

lemma *monotone*:

shows *mono raw.action*

unfolding *raw.action-def*

by (*fastforce simp: le-fun-def intro: monoI prog.action.mono raw.tso.mono-cl*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF raw.action.monotone*]

lemmas *mono* = *monotoneD*[*OF raw.action.monotone*]

lemma *Sup*:

shows $\text{raw.action } (\bigsqcup F_s) = \bigsqcup (\text{raw.action } \text{' } F_s) \sqcup \text{raw.tso.cl } \perp$ (**is** *?lhs = ?rhs*)

proof –

have *?rhs* = $\bigsqcup (\text{raw.tso.cl } \text{' } (\lambda F \text{ wb. } \text{prog.action } \{((v, \text{wb } @ \text{ws}), s, s') \mid v \ s \ s' \ \text{ws. } (v, \text{ws}, s, s') \in F \ \text{wb}\}) \text{' } F_s)$
 $\sqcup \text{raw.tso.cl } \perp$

by (*simp add: raw.action-def image-comp*)

also have $\dots = \text{raw.tso.cl } (\bigsqcup F \in F_s. (\lambda \text{wb. } \text{prog.action } \{((v, \text{wb } @ \text{ws}), s, s') \mid v \ s \ s' \ \text{ws. } (v, \text{ws}, s, s') \in F \ \text{wb}\}))$

by (*simp add: raw.tso.cl-Sup*)

also have $\dots = \text{raw.tso.cl } (\lambda \text{wb. } \bigsqcup (\text{prog.action } \text{' } (\lambda F. \{((v, \text{wb } @ \text{ws}), s, s') \mid v \ s \ s' \ \text{ws. } (v, \text{ws}, s, s') \in F \ \text{wb}\}) \text{' } F_s))$

by (*simp add: Sup-fun-def image-comp*)

also have $\dots = \text{?lhs}$

by (*force simp: raw.action-def*

simp flip: prog.action.Sup

intro: arg-cong[where f=raw.tso.cl] arg-cong[where f=prog.action])

finally show *?thesis ..*

qed

lemma *sup*:

shows $\text{raw.action } (F \sqcup G) = \text{raw.action } F \sqcup \text{raw.action } G$

using *raw.action.Sup*[**where** $F_s = \{F, G\}$]

by (*simp add: sup.absorb1 le-supI1 raw.action.mono flip: raw.action.bot*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path guard} \rangle$

lemma *return-le*:

shows $raw.guard\ g \leq raw.return\ ()$
by (*fastforce simp add: raw.guard-def raw.return-def intro: le-funI raw.action.mono*)

lemma *monotone*:

shows $mono\ (raw.guard\ ::\ (write-buffer\ \Rightarrow\ heap.t\ pred)\ \Rightarrow\ -)$

proof(*rule monoI*)

show $raw.guard\ g \leq raw.guard\ h$ **if** $g \leq h$ **for** $g\ h\ ::\ write-buffer\ \Rightarrow\ heap.t\ pred$

unfolding *raw.guard-def Diag-def*

by (*blast intro: raw.action.mono le-funI predicate2D[OF that]*)

qed

lemmas *strengthen[stg] = st-monotone[OF raw.guard.monotone]*

lemmas *mono = monotoneD[OF raw.guard.monotone]*

lemma *less*: — Non-triviality; essentially replay *prog.guard.less*

assumes $g < g'$

shows $raw.guard\ g < raw.guard\ g'$

proof(*rule le-neq-trans*)

show $raw.guard\ g \leq raw.guard\ g'$

by (*strengthen ord-to-strengthen(1)[OF order-less-imp-le[OF assms]] simp*

from *assms* **obtain** $wb\ s$ **where** $g' wb\ s \neg g wb\ s$ **by** (*metis leD predicate2I*)

from $\langle \neg g wb\ s \rangle$ **have** $\neg \langle trace.T\ s\ []\ (Some\ ((),\ wb)) \rangle \leq prog.p2s\ (raw.guard\ g\ wb)$

by (*auto simp: raw.guard-def raw.action-def raw.tso.cl-def raw.bind-def raw.prim-return-def split-def trace.split-all prog.p2s.simps prog.p2s.action prog.p2s.return spec.interference.cl.action spec.interference.cl.return spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv suffix-order.antisym-conv*

elim!: *spec.singleton.bind-le*

dest!: *spec.singleton.raw.commit-writes.wb-suffix*)

moreover

from $\langle g' wb\ s \rangle$ **have** $\langle trace.T\ s\ []\ (Some\ ((),\ wb)) \rangle \leq prog.p2s\ (raw.guard\ g'\ wb)$

by (*force simp: raw.guard-def raw.action-def raw.prim-return-def raw.tso.cl-def raw.bind-def spec.bind.bind spec.singleton.le-conv prog.p2s.bind prog.p2s.action prog.p2s.return spec.interference.cl.action spec.interference.cl.return*

intro: spec.bind.continueI[where xs=[], simplified] spec.action.stutterI spec.singleton.raw.commit-writes.noop-le)

ultimately show $raw.guard\ g \neq raw.guard\ g'$ **by** *metis*

qed

setup $\langle Sign.parent-path \rangle$

lemma *MFENCE-alt-def*:

shows $raw.MFENCE = raw.commit-writes \gg (\lambda wb. prog.action\ (\{((),\ wb)\} \times Diag\ \langle wb = [] \rangle))$

proof —

have $*$: $prog.action\ \{x. (\exists a. x = (((),\ wb),\ a,\ a)) \wedge wb = []\} \gg (\lambda p. raw.commit-writes\ (snd\ p))$
 $= prog.action\ (\{((),\ wb)\} \times Diag\ (\lambda s. wb = [])) \gg prog.return$

for wb

proof(*induct rule: refinement.prog.eqI[case-names l2r r2l]*)

case *l2r* **show** *?case*

apply (*rule refinement.prog.rev-bind*)

apply (*rule refinement.prog.action[where Q= $\lambda v\ s. snd\ v = []$;*

simp add: stable-def monotone-def; fail)

apply (*rule refinement.gen-asm; clarsimp simp: raw.commit-writes.Nil*)

apply (*rule refinement.sort-of-refl*)

apply (*subst refinement.top, simp; fail*)

apply (*simp add: spec.idle.p2s-le; fail*)

```

  done
next
case r2l show ?case
  apply (rule refinement.prog.rev-bind)
  apply (rule refinement.prog.action[where Q= $\lambda v s. \text{snd } v = []$ ];
    simp add: stable-def monotone-def; fail)
  apply (rule refinement.gen-asm; clarsimp simp: raw.commit-writes.Nil)
  apply (rule refinement.sort-of-refl)
  apply (subst refinement.top, simp; fail)
  apply (simp add: spec.idle.p2s-le; fail)
  done
qed
show ?thesis
  by (simp add: raw.MFENCE-def raw.guard-def raw.action-def raw.tso.cl-def
    raw.bind.bind raw.bind.prim-returnR)
    (simp add: * raw.bind-def raw.prim-return-def split-def prog.bind.return)
qed

setup <Sign.mandatory-path MFENCE>

lemma Nil:
  shows raw.MFENCE [] = prog.return ((), [])
by (simp add: raw.MFENCE-alt-def raw.bind-def raw.commit-writes.Nil prog.bind.returnL
  flip: Id-def prog.return-def)

setup <Sign.parent-path>

setup <Sign.parent-path>

setup <Sign.mandatory-path refinement.raw>

lemma MFENCE:
  shows prog.p2s (raw.MFENCE wb)  $\leq \{P\}$ ,  $A \Vdash \text{prog.p2s } (raw.MFENCE wb), \{\lambda v s. \text{snd } v = []\}$ 
  apply (simp add: raw.MFENCE-alt-def raw.bind-def split-def)
  apply (rule refinement.prog.rev-bind)
  apply (rule refinement.sort-of-refl)
  apply (subst refinement.top, simp; fail)
  apply (rule refinement.prog.action; simp add: stable-def monotone-def)
  done

setup <Sign.parent-path>

setup <Sign.mandatory-path raw>

setup <Sign.mandatory-path bind>

lemma MFENCEL:
  shows raw.MFENCE wb  $\ggg g = raw.MFENCE wb \ggg g$  ((), []) (is ?lhs = ?rhs)
proof(induct rule: refinement.prog.eqI[case-names l2r r2l])
  case l2r show ?case
    apply (rule refinement.prog.rev-bind)
    apply (rule refinement.raw.MFENCE)
    apply (rule refinement.gen-asm; clarsimp)
    apply (rule refinement.sort-of-refl)
    apply (subst refinement.top, simp; fail)
    apply (rule spec.idle.p2s-le)
  done
  case r2l show ?case

```

```

apply (rule refinement.prog.rev-bind)
apply (rule refinement.raw.MFENCE)
apply (rule refinement.gen-asm; clarsimp)
apply (rule refinement.sort-of-refl)
apply (subst refinement.top, simp; fail)
apply (rule spec.idle.p2s-le)
done
qed

```

lemma *MFENCE-return*:

```

shows raw.MFENCE wb  $\gg$  prog.return ((), []) = raw.MFENCE wb
by (simp add: prog.bind.returnR flip: raw.bind.MFENCEL)

```

lemma *MFENCE-MFENCE*:

```

shows raw.MFENCE  $\gg$  raw.MFENCE = raw.MFENCE
by (simp add: raw.bind-def raw.prim-return-def raw.MFENCE.Nil raw.bind.MFENCE-return
      raw.bind.MFENCEL[where g=( $\lambda(-, y). raw.MFENCE y$ )])

```

setup \langle Sign.parent-path \rangle

setup \langle Sign.mandatory-path t2p \rangle

lemma *bot*:

```

shows raw.t2p  $\perp = \perp$ 
by (simp add: raw.t2p-def prog.bind.botL)

```

lemma *cl-bot*:

```

shows raw.t2p (raw.tso.cl  $\perp$ ) =  $\perp$ 
by (simp add: raw.t2p-def raw.tso.cl.bot raw.bind-def raw.commit-writes.Nil
      prog.bind.bind prog.bind.botL prog.bind.returnL)

```

lemma *monotone*:

```

shows mono raw.t2p
by (rule monotoneI) (simp add: raw.t2p-def le-fun-def prog.bind.mono)

```

lemmas *strengthen*[strg] = *st-monotone*[OF raw.t2p.monotone]

lemmas *mono* = *monotoneD*[OF raw.t2p.monotone]

lemmas *mono2mono*[cont-intro, partial-function-mono] = *monotone2monotone*[OF raw.t2p.monotone, simplified]

lemma *Sup*:

```

shows raw.t2p ( $\sqcup X$ ) =  $\sqcup$ (raw.t2p ‘ X)
by (simp add: raw.t2p-def prog.bind.SUPL)

```

lemma *sup*:

```

shows raw.t2p (P  $\sqcup$  Q) = raw.t2p P  $\sqcup$  raw.t2p Q
using raw.t2p.Sup[where X={P, Q}] by simp

```

lemma *mcont2mcont*[cont-intro]:

```

fixes P :: -  $\Rightarrow$  - raw.tso
assumes mcont luba orda Sup ( $\leq$ ) F
shows mcont luba orda Sup ( $\leq$ ) ( $\lambda x. raw.t2p (F x)$ )

```

proof –

```

from assms have mcont luba orda Sup ( $\leq$ ) ( $\lambda x. F x$  [])
by (fastforce intro!: mcontI contI monotoneI
      dest: mcont-contD mcont-monoD
      simp: le-funD
      simp flip: SUP-apply)

```

then show ?thesis

by (*simp add: raw.t2p-def split-def*)
qed

lemma return:

shows *raw.t2p (raw.return v) = prog.return v*
by (*simp add: raw.t2p-def raw.return-alt-def raw.tso.cl-def raw.bind-def raw.prim-return-def*
prog.p2s.simps prog.p2s.return
spec.interference.cl.action spec.interference.cl.return
spec.bind.bind spec.bind.return
raw.commit-writes.Nil raw.MFENCE.Nil
flip: prog.p2s-inject)
(*simp add: spec.rel.wind-bind flip: spec.bind.bind*)

lemma MFENCE-bind:

shows *raw.t2p (raw.MFENCE \ggg P) = raw.t2p (P ())*
by (*simp add: raw.t2p-def raw.bind-def split-def prog.bind.returnL raw.MFENCE.Nil*)

lemma bind-return-unit:

shows *raw.t2p ($\lambda w b. prog.bind P (\lambda :: unit. prog.return ((), [])) = P$)*
by (*simp add: raw.t2p-def raw.bind-def split-def*
prog.bind.bind prog.bind.returnL prog.bind.returnR raw.MFENCE.Nil)

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path parallel* \rangle

lemma commute: — Jagadeesan et al. (2012, §5 (3))

shows *raw.parallel P Q = raw.parallel Q P*
by (*simp add: raw.parallel-def prog.parallel commute*)

lemma assoc: — Jagadeesan et al. (2012, §5 (4))

shows *raw.parallel P (raw.parallel Q R) = raw.parallel (raw.parallel P Q) R*
by (*simp add: raw.parallel-def raw.t2p.MFENCE-bind raw.t2p.bind-return-unit prog.parallel.assoc*)

lemma mono:

assumes $P \leq P'$
assumes $Q \leq Q'$
shows *raw.parallel P Q \leq raw.parallel P' Q'*
unfolding *raw.parallel-def*
apply (*strengthen ord-to-strengthen(1)[OF assms(1)]*)
apply (*strengthen ord-to-strengthen(1)[OF assms(2)]*)
apply (*rule order.refl*)
done

lemma botL:

shows *raw.parallel (raw.tso.cl \perp) f = raw.MFENCE \ggg f \ggg raw.MFENCE \ggg raw.tso.cl \perp*
apply (*simp add: raw.parallel-def raw.t2p.cl-bot prog.parallel.bot prog.bind.bind prog.bind.botL*)
apply (*simp add: raw.t2p-def split-def raw.bind-def prog.bind.bind prog.bind.returnL*)
apply (*subst (3 4) raw.bind.MFENCEL*)
apply (*simp add: raw.tso.cl.Nil prog.bind.botL*)
done

lemma returnL:

shows *raw.parallel (raw.return ()) P = raw.MFENCE \ggg ($\lambda -. P$) \ggg ($\lambda -. raw.MFENCE$)*
apply (*simp add: raw.parallel-def raw.t2p.return prog.parallel.return*)
apply (*simp add: raw.t2p-def split-def raw.bind-def prog.bind.bind prog.bind.returnL raw.bind.MFENCE-return*)
apply (*subst (2) raw.bind.MFENCEL*)
apply *simp*

done

lemma *SupL-not-empty*:

assumes $\forall x \in X. x \in \text{raw.tso.closed}$

assumes $Q \in \text{raw.tso.closed}$

assumes $X \neq \{\}$

shows $\text{raw.parallel} (\bigsqcup X \sqcup \text{raw.tso.cl } \perp) Q = (\bigsqcup P \in X. \text{raw.parallel } P Q) \sqcup \text{raw.tso.cl } \perp$

proof –

from $\langle X \neq \{\} \rangle$

have $\text{raw.parallel} (\bigsqcup X) Q = (\bigsqcup P \in X. \text{raw.parallel } P Q)$

by (*simp add: raw.parallel-def raw.t2p.Sup raw.t2p.sup*
prog.parallel.SupL-not-empty prog.parallel.supL prog.bind.SUPL prog.bind.supL)
(*simp add: raw.bind-def split-def fun-eq-iff prog.bind.SUPR-not-empty image-image*)

moreover

from *assms* have $\text{raw.tso.cl } \perp \leq (\bigsqcup P \in X. \text{raw.parallel } P Q)$

by (*force intro: less-eq-Sup raw.tso.least[OF - raw.tso.closed.parallel]*)

moreover note *assms*

ultimately show *?thesis*

by (*simp add: sup.absorb1 less-eq-Sup raw.tso.least*)

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.parent-path} \rangle$

typedef $'v \text{ tso} = \text{raw.tso.closed} :: 'v \text{ raw.tso set}$

morphisms $t2p'$ *Abs-tso*

by *blast*

setup-lifting *type-definition-tso*

instantiation *tso* :: (type) *complete-distrib-lattice*

begin

lift-definition *bot-tso* :: $'v \text{ tso is raw.tso.cl } \perp ..$

lift-definition *top-tso* :: $'v \text{ tso is } \top ..$

lift-definition *sup-tso* :: $'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow 'v \text{ tso is sup} ..$

lift-definition *inf-tso* :: $'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow 'v \text{ tso is inf} ..$

lift-definition *less-eq-tso* :: $'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow \text{bool is less-eq} .$

lift-definition *less-tso* :: $'v \text{ tso} \Rightarrow 'v \text{ tso} \Rightarrow \text{bool is less} .$

lift-definition *Inf-tso* :: $'v \text{ tso set} \Rightarrow 'v \text{ tso is Inf} ..$

lift-definition *Sup-tso* :: $'v \text{ tso set} \Rightarrow 'v \text{ tso is } \lambda X. \text{Sup } X \sqcup \text{raw.tso.cl } \perp ..$

instance by (*standard; transfer; auto simp: InfI Inf-lower le-supI1 SupI SupE raw.tso.least*)

end

setup $\langle \text{Sign.mandatory-path } \textit{tso} \rangle$

lift-definition *bind* :: $'v \text{ tso} \Rightarrow ('v \Rightarrow 'w \text{ tso}) \Rightarrow 'w \text{ tso is raw.bind} ..$

lift-definition *action* :: $(\text{write-buffer} \Rightarrow ('v \times \text{write-buffer} \times \text{heap.t} \times \text{heap.t}) \text{ set}) \Rightarrow 'v \text{ tso is raw.action} ..$

lift-definition *MFENCE* :: $\text{unit tso is raw.MFENCE} ..$

lift-definition *parallel* :: $\text{unit tso} \Rightarrow \text{unit tso} \Rightarrow \text{unit tso is raw.parallel} ..$

lift-definition *vmap* :: $('v \Rightarrow 'w) \Rightarrow 'v \text{ tso} \Rightarrow 'w \text{ tso is raw.vmap} ..$

lift-definition *t2p* :: $'v \text{ tso} \Rightarrow (\text{heap.t}, 'v) \text{ prog is raw.t2p} .$

adhoc-overloading

Monad-Syntax.bind \equiv *tso.bind*

ad hoc-overloading

parallel \equiv *tso.parallel*

definition *return* :: 'v \Rightarrow 'v *tso* **where**

return v = *tso.action* $\langle \{v\} \times \{\square\} \times Id \rangle$

definition *guard* :: (*write-buffer* \Rightarrow *heap.t pred*) \Rightarrow *unit tso* **where**

guard g = *tso.action* $\langle \lambda wb. \{()\} \times \{\square\} \times Diag (g wb) \rangle$

abbreviation (*input*) *read* :: (*heap.t* \Rightarrow 'v) \Rightarrow 'v *tso* **where**

read f \equiv *tso.action* $\langle \lambda wb. \{f (apply-writes wb s), [], s, s\} | s. True \rangle$

abbreviation (*input*) *write* :: (*heap.t* \Rightarrow *heap.write*) \Rightarrow *unit tso* **where**

write f \equiv *tso.action* $\langle \{(), [f s], s, s\} | s. True \rangle$

lemma *return-alt-def*:

shows *tso.return* v = *tso.read* $\langle v \rangle$

by (*auto simp: tso.return-def intro: arg-cong[where f=tso.action]*)

declare *tso.bind-def*[code del]

declare *tso.action-def*[code del]

declare *tso.return-def*[code del]

declare *tso.MFENCE-def*[code del]

declare *tso.parallel-def*[code del]

declare *tso.vmap-def*[code del]

setup \langle *Sign.mandatory-path return* \rangle

lemma *transfer*[transfer-rule]:

shows *rel-fun* (=) *cr-tso raw.return tso.return*

unfolding *raw.return-def tso.return-def* **by** *transfer-prover*

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path action* \rangle

lemma *empty*:

shows *bot: tso.action* $\perp = \perp$

and *tso.action* $\langle \lambda-. \{ \} \rangle = \perp$

by (*simp-all add: raw.action.bot[transferred, unfolded bot-fun-def] bot-fun-def*)

lemmas *monotone* = *raw.action.monotone*[transferred]

lemmas *strengthen*[strg] = *st-monotone*[OF *tso.action.monotone*]

lemmas *mono* = *monotoneD*[OF *tso.action.monotone*]

lemmas *mono2mono*[cont-intro, partial-function-mono] = *monotone2monotone*[OF *tso.action.monotone*, *simplified*]

lemma *Sup*:

shows *tso.action* $\langle \bigsqcup Fs \rangle = \bigsqcup (tso.action \text{ ' } Fs)$

by *transfer* (*simp add: raw.action.Sup*)

lemmas *sup* = *tso.action.Sup*[**where** *Fs*={*F*, *G*} **for** *F G*, *simplified*]

setup \langle *Sign.parent-path* \rangle

setup \langle *Sign.mandatory-path bind* \rangle

lemmas *if-distrL* = *if-distrib*[**where** $f = \lambda f. \text{tso.bind } f \text{ } g$ **for** g] — Jagadeesan et al. (2012, §5 (5))

lemmas *mono* = *raw.bind.mono*[*transferred*]

lemma *strengthen*[*strg*]:

assumes *st-ord* $F f f'$

assumes $\bigwedge x. \text{st-ord } F (g x) (g' x)$

shows *st-ord* $F (\text{tso.bind } f g) (\text{tso.bind } f' g')$

using *assms* **by** (*cases* F ; *clarsimp* *intro!*: *tso.bind.mono*)

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *raw.bind.mono2mono*[*transferred*]

lemma *bind*: — Jagadeesan et al. (2012, §5 (2))

shows $f \ggg g \ggg h = \text{tso.bind } f (\lambda x. g x \ggg h)$

by *transfer* (*simp* *add*: *raw.bind.bind*)

lemma *return*: — Jagadeesan et al. (2012, §5 (1))

shows *returnL*: $\text{tso.return } v \ggg g = g v$

and *returnR*: $f \ggg \text{tso.return} = f$

by (*transfer*; *simp* *add*: *raw.bind.returnL* *raw.bind.returnR*)⁺

lemma *botL*:

shows $\text{tso.bind } \perp g = \perp$

by *transfer* (*simp* *add*: *raw.tso.cl.bot* *raw.bind.bind* *raw.bind.botL* *flip*: *bot-fun-def*)

lemma *botR-le*:

shows $\text{tso.bind } f \langle \perp \rangle \leq f$ (**is** *?thesis1*)

and $\text{tso.bind } f \perp \leq f$ (**is** *?thesis2*)

proof —

show *?thesis1*

by (*metis* *bot.extremum* *dual-order.refl* *tso.bind.mono* *tso.bind.returnR*)

then show *?thesis2*

by (*simp* *add*: *bot-fun-def*)

qed

lemma

fixes $f :: - \text{tso}$

fixes $f_1 :: - \text{tso}$

shows *supL*: $(f_1 \sqcup f_2) \ggg g = (f_1 \ggg g) \sqcup (f_2 \ggg g)$

and *supR*: $f \ggg (\lambda x. g_1 x \sqcup g_2 x) = (f \ggg g_1) \sqcup (f \ggg g_2)$

by (*transfer*; *blast* *intro*: *raw.bind.supL* *raw.bind.supR*)⁺

lemma *SUPL*:

fixes $X :: - \text{set}$

fixes $f :: - \Rightarrow - \text{tso}$

shows $(\bigsqcup x \in X. f x) \ggg g = (\bigsqcup x \in X. f x \ggg g)$

by *transfer*

(*simp* *add*: *raw.bind.supL* *raw.bind.SUPL* *raw.tso.cl.bot* *raw.bind.bind* *raw.bind.botL*
flip: *bot-fun-def*)

lemma *SUPR*:

fixes $X :: - \text{set}$

fixes $f :: - \text{tso}$

shows $f \ggg (\lambda v. \bigsqcup x \in X. g x v) = (\bigsqcup x \in X. f \ggg g x) \sqcup (f \ggg \perp)$

unfolding *bot-fun-def*

by *transfer*

(*simp* *add*: *raw.bind.supR* *raw.bind.SUPR* *ac-simps*
sup.absorb2 *le-supI1* *raw.bind.mono* *raw.tso.closed.bind* *raw.tso.least*)

lemma *SupR*:

fixes $X :: - \text{ set}$

fixes $f :: - \text{ tso}$

shows $f \gg (\bigsqcup X) = (\bigsqcup_{x \in X}. f \gg x) \sqcup (f \gg \perp)$

by (*simp add: tso.bind.SUPR[where g= $\lambda x v. x$, simplified]*)

lemma *SUPR-not-empty*:

fixes $f :: - \text{ tso}$

assumes $X \neq \{\}$

shows $f \gg (\lambda v. \bigsqcup_{x \in X}. g x v) = (\bigsqcup_{x \in X}. f \gg g x)$

using *iffD2[OF ex-in-conv assms]*

by (*subst trans[OF tso.bind.SUPR sup.absorb1]; force intro: SUPI tso.bind.mono*)

lemma *mcont2mcont[cont-intro]*:

assumes $mcont \text{ luba } orda \text{ Sup } (\leq) f$

assumes $\bigwedge v. mcont \text{ luba } orda \text{ Sup } (\leq) (\lambda x. g x v)$

shows $mcont \text{ luba } orda \text{ Sup } (\leq) (\lambda x. \text{tso.bind } (f x) (g x))$

proof(*rule ccpo.mcont2mcont'[OF complete-lattice-ccpo - - assms(1)]*)

show $mcont \text{ Sup } (\leq) \text{ Sup } (\leq) (\lambda f. \text{tso.bind } f (g x))$ **for** x

by (*intro mcontI contI monotoneI*) (*simp-all add: tso.bind.mono flip: tso.bind.SUPL*)

show $mcont \text{ luba } orda \text{ Sup } (\leq) (\lambda x. \text{tso.bind } f (g x))$ **for** f

by (*intro mcontI monotoneI contI*)

(*simp-all add: mcont-monoD[OF assms(2)] tso.bind.mono*)

flip: tso.bind.SUPR-not-empty contD[OF mcont-cont[OF assms(2)]])

qed

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path guard} \rangle$

lemma *transfer[transfer-rule]*:

shows $\text{rel-fun } (=) \text{ cr-tso raw.guard tso.guard}$

unfolding *raw.guard-def tso.guard-def* **by** *transfer-prover*

lemma *bot*:

shows $\text{tso.guard } \perp = \perp$

and $\text{tso.guard } (\lambda - . \text{False}) = \perp$

by (*simp-all add: tso.guard-def tso.action.empty*)

lemma *top*:

shows $\text{tso.guard } \top = \text{tso.return } ()$ (**is** *?thesis1*)

and $\text{tso.guard } (\lambda - . \top) = \text{tso.return } ()$ (**is** *?thesis2*)

and $\text{tso.guard } (\lambda - . \text{True}) = \text{tso.return } ()$ (**is** *?thesis3*)

proof –

show *?thesis1*

by (*simp add: tso.guard-def tso.return-def flip: Id-def*)

then show *?thesis2* **and** *?thesis3*

by (*simp-all add: top-fun-def*)

qed

lemma *return-le*:

shows $\text{tso.guard } g \leq \text{tso.return } ()$

by *transfer (rule raw.guard.return-le)*

lemma *monotone*:

shows $\text{mono } \text{tso.guard}$

by *transfer (rule raw.guard.monotone)*

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.guard.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.guard.monotone*]
lemmas *mono2mono*[*cont-intro, partial-function-mono*] = *monotone2monotone*[*OF tso.guard.monotone, simplified*]

lemma *less*: — Non-triviality
assumes $g < g'$
shows $tso.guard\ g < tso.guard\ g'$
using *assms* **by** *transfer* (*rule raw.guard.less*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.mandatory-path\ parallel \rangle$

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))
shows $tso.parallel\ P\ Q = tso.parallel\ Q\ P$
by *transfer* (*rule raw.parallel.commute*)

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))
shows $tso.parallel\ P\ (tso.parallel\ Q\ R) = tso.parallel\ (tso.parallel\ P\ Q)\ R$
by *transfer* (*rule raw.parallel.assoc*)

lemmas *mono* = *raw.parallel.mono*[*transferred*]

lemma *strengthen*[*strg*]:
assumes *st-ord* $F\ P\ P'$
assumes *st-ord* $F\ Q\ Q'$
shows *st-ord* $F\ (tso.parallel\ P\ Q)\ (tso.parallel\ P'\ Q')$
using *assms* **by** (*cases F; simp add: tso.parallel.mono*)

lemma *mono2mono*[*cont-intro, partial-function-mono*]:
assumes *monotone orda* $(\leq)\ F$
assumes *monotone orda* $(\leq)\ G$
shows *monotone orda* $(\leq)\ (\lambda f. tso.parallel\ (F\ f)\ (G\ f))$
using *assms* **by** (*simp add: monotone-def tso.parallel.mono*)

lemma *bot*:
shows *parallel-botL*: $tso.parallel\ \perp\ f = tso.MFENCE\ \gg\ f\ \gg\ tso.MFENCE\ \gg\ \perp$ (**is** *?thesis1*)
and *parallel-botR*: $tso.parallel\ f\ \perp = tso.MFENCE\ \gg\ f\ \gg\ tso.MFENCE\ \gg\ \perp$ (**is** *?thesis2*)
proof –
show *?thesis1*
unfolding *bot-fun-def* **by** *transfer* (*simp add: raw.parallel.botL raw.bind.bind*)
then show *?thesis2*
by (*simp add: tso.parallel.commute*)
qed

lemma *return*: — Jagadeesan et al. (2012, unnumbered)
shows *returnL*: $tso.return\ ()\ \parallel\ P = tso.MFENCE\ \gg\ P\ \gg\ tso.MFENCE$ (**is** *?thesis1*)
and *returnR*: $P\ \parallel\ tso.return\ () = tso.MFENCE\ \gg\ P\ \gg\ tso.MFENCE$ (**is** *?thesis2*)
proof –
show *?thesis1*
by *transfer* (*rule raw.parallel.returnL*)
then show *?thesis2*
by (*simp add: tso.parallel.commute*)
qed

lemma *Sup-not-empty*:

```

fixes  $X :: \text{unit tso set}$ 
assumes  $X \neq \{\}$ 
shows SupL-not-empty:  $\sqcup X \parallel Q = (\sqcup P \in X. P \parallel Q)$  (is ?thesis1 Q)
  and SupR-not-empty:  $P \parallel \sqcup X = (\sqcup Q \in X. P \parallel Q)$  (is ?thesis2)
proof –
  from assms show ?thesis1 Q for  $Q$ 
    by transfer (rule raw.parallel.SupL-not-empty)
  then show ?thesis2
    by (simp add: tso.parallel commute)
qed

lemma sup:
  fixes  $P :: \text{unit tso}$ 
  shows supL:  $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$ 
    and supR:  $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$ 
using tso.parallel.SupL-not-empty[where  $X=\{P, Q\}$ ] tso.parallel.SupR-not-empty[where  $X=\{Q, R\}$ ]
by simp-all

lemma mcont2mcont[cont-intro]:
  assumes mcont luba orda Sup  $(\leq) P$ 
  assumes mcont luba orda Sup  $(\leq) Q$ 
  shows mcont luba orda Sup  $(\leq) (\lambda x. \text{tso.parallel } (P x) (Q x))$ 
proof(rule ccpo.mcont2mcont'[OF complete-lattice-ccpo - - assms(1)])
  show mcont Sup  $(\leq) \text{Sup} (\leq) (\lambda y. \text{tso.parallel } y (Q x))$  for  $x$ 
    by (intro mcontI contI monotoneI) (simp-all add: tso.parallel.mono tso.parallel.SupL-not-empty)
  show mcont luba orda Sup  $(\leq) (\lambda x. \text{tso.parallel } y (Q x))$  for  $y$ 
    by (simp add: mcontI monotoneI contI mcont-monoD[OF assms(2)]
      spec.parallel.mono mcont-contD[OF assms(2)] tso.parallel.SupR-not-empty image-image)
qed

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.mandatory-path bind} \rangle$ 

lemmas MFENCE-MFENCE = raw.bind.MFENCE-MFENCE[transferred]

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.mandatory-path } t2p' \rangle$ 

lemma monotone:
  shows mono  $(\lambda t. t2p' t wb)$ 
by (simp add: le-fun-def less-eq-tso.rep-eq monotone-def)

lemmas strengthen[strg] = st-monotone[OF tso.t2p'.monotone]
lemmas mono = monotoneD[OF tso.t2p'.monotone]

lemmas action = tso.action.rep-eq
lemma return:
  shows  $t2p' (tso.return v) = \text{raw.return } v$ 
by transfer simp

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.parent-path} \rangle$ 

Combinators setup  $\langle \text{Sign.mandatory-path } tso \rangle$ 

```

abbreviation $guardM :: bool \Rightarrow unit\ tso\ \mathbf{where}$

$guardM\ b \equiv \text{if } b \text{ then } \perp \text{ else } tso.\text{return } ()$

abbreviation $unlessM :: bool \Rightarrow unit\ tso \Rightarrow unit\ tso\ \mathbf{where}$

$unlessM\ b\ c \equiv \text{if } b \text{ then } tso.\text{return } () \text{ else } c$

abbreviation $whenM :: bool \Rightarrow unit\ tso \Rightarrow unit\ tso\ \mathbf{where}$

$whenM\ b\ c \equiv \text{if } b \text{ then } c \text{ else } tso.\text{return } ()$

definition $app :: ('a \Rightarrow unit\ tso) \Rightarrow 'a\ list \Rightarrow unit\ tso\ \mathbf{where}$ — Haskell's $mapM$ -

$app\ f\ xs = \text{foldr } (\lambda x\ m.\ f\ x \gg m)\ xs\ (tso.\text{return } ())$

primrec $fold\text{-}mapM :: ('a \Rightarrow 'b\ tso) \Rightarrow 'a\ list \Rightarrow 'b\ list\ tso\ \mathbf{where}$

$fold\text{-}mapM\ f\ [] = tso.\text{return } []$

| $fold\text{-}mapM\ f\ (x \# xs) = \text{do } \{$

$y \leftarrow f\ x;$

$ys \leftarrow fold\text{-}mapM\ f\ xs;$

$tso.\text{return } (y \# ys)$

$\}$

— Jagadeesan et al. (2012, §5 (6) is $tso.\text{while}.\text{simps}$)

partial-function $(lfp)\ \text{while} :: ('k \Rightarrow ('k + 'v)\ tso) \Rightarrow 'k \Rightarrow 'v\ tso\ \mathbf{where}$

$\text{while}\ c\ k = c\ k \gg (\lambda rv.\ \text{case } rv \text{ of } Inl\ k' \Rightarrow \text{while}\ c\ k' \mid Inr\ v \Rightarrow tso.\text{return } v)$

abbreviation $(input)\ \text{while}' :: ((unit + 'v)\ tso) \Rightarrow 'v\ tso\ \mathbf{where}$

$\text{while}'\ c \equiv tso.\text{while } \langle c \rangle ()$

definition $\text{raise} :: String.\text{literal} \Rightarrow 'v\ tso\ \mathbf{where}$

$\text{raise } s = \perp$

definition $\text{assert} :: bool \Rightarrow unit\ tso\ \mathbf{where}$

$\text{assert } P = (\text{if } P \text{ then } tso.\text{return } () \text{ else } tso.\text{raise } STR\ \text{"assert"})$

declare $tso.\text{raise}\text{-}def[\text{code } del]$

setup $\langle Sign.\text{mandatory}\text{-}path\ fold\text{-}mapM \rangle$

lemma $bot:$

shows $tso.\text{fold}\text{-}mapM\ \perp = (\lambda xs.\ \text{case } xs \text{ of } [] \Rightarrow tso.\text{return } [] \mid - \Rightarrow \perp)$

by ($\text{simp add: fun}\text{-}eq\text{-}iff\ tso.\text{bind}.\text{botL split: list}.\text{split}$)

lemma $\text{append}:$

shows $tso.\text{fold}\text{-}mapM\ f\ (xs\ @\ ys) = tso.\text{fold}\text{-}mapM\ f\ xs \gg (\lambda xs.\ tso.\text{fold}\text{-}mapM\ f\ ys \gg (\lambda ys.\ tso.\text{return } (xs\ @\ ys)))$

by ($\text{induct } xs$) ($\text{simp}\text{-}all\ \text{add: } tso.\text{bind}.\text{bind}\ tso.\text{bind}.\text{returnL}\ tso.\text{bind}.\text{returnR}$)

setup $\langle Sign.\text{parent}\text{-}path \rangle$

setup $\langle Sign.\text{mandatory}\text{-}path\ app \rangle$

lemma $bot:$

shows $tso.\text{app}\ \perp = (\lambda xs.\ \text{case } xs \text{ of } [] \Rightarrow tso.\text{return } () \mid - \Rightarrow \perp)$

and $tso.\text{app}\ (\lambda\cdot.\ \perp) = (\lambda xs.\ \text{case } xs \text{ of } [] \Rightarrow tso.\text{return } () \mid - \Rightarrow \perp)$

by ($\text{simp}\text{-}all\ \text{add: fun}\text{-}eq\text{-}iff\ tso.\text{app}\text{-}def\ tso.\text{bind}.\text{botL split: list}.\text{split}$)

lemma $Nil:$

shows $tso.\text{app}\ f\ [] = tso.\text{return } ()$

by ($\text{simp add: } tso.\text{app}\text{-}def$)

lemma *Cons*:

shows $tso.app\ f\ (x\ \# \ xs) = f\ x \gg tso.app\ f\ xs$
by (*simp add: tso.app-def*)

lemmas *simps* =

tso.app.bot
tso.app.Nil
tso.app.Cons

lemma *append*:

shows $tso.app\ f\ (xs\ @\ ys) = tso.app\ f\ xs \gg tso.app\ f\ ys$
by (*induct xs arbitrary: ys*) (*simp-all add: tso.app.simps tso.bind.returnL tso.bind.bind*)

lemma *monotone*:

shows *mono* $(\lambda f. tso.app\ f\ xs)$
by (*induct xs*) (*simp-all add: tso.app.simps le-fun-def monotone-on-def tso.bind.mono*)

lemmas *strengthen[stg]* = *st-monotone[OF tso.app.monotone]*

lemmas *mono* = *monotoneD[OF tso.app.monotone]*

lemmas *mono2mono[cont-intro, partial-function-mono]* = *monotone2monotone[OF tso.app.monotone, simplified, of orda P for orda P]*

lemma *Sup-le*:

shows $(\bigsqcup f \in X. tso.app\ f\ xs) \leq tso.app\ (\bigsqcup X)\ xs$
by (*simp add: SUP-le-iff SupI tso.app.mono*)

setup $\langle Sign.parent-path \rangle$

setup $\langle Sign.parent-path \rangle$

27.1 References

Observe that allocation is global in this model. We allow the memory location to have an arbitrary value and enqueue the initialising write in the TSO buffer.

setup $\langle Sign.mandatory-path\ tso.Ref \rangle$

definition *ref* :: $'a::heap.rep \Rightarrow 'a\ ref\ tso$ **where**

$ref\ v = tso.action\ (\lambda wb. \{(r, [heap.Write\ (ref.addr-of\ r)\ 0\ (heap.rep.to\ v)]), s, s'\}$
 $\mid r\ s\ s'\ v'. (r, s') \in Ref.alloc\ v'\ s\})$

definition *lookup* :: $'a::heap.rep\ ref \Rightarrow 'a\ tso$ ($\langle !- \rangle$ 61) **where**

$lookup\ r = tso.read\ (Ref.get\ r)$

definition *update* :: $'a\ ref \Rightarrow 'a::heap.rep \Rightarrow unit\ tso$ ($\langle - := - \rangle$ 62) **where**

$update\ r\ v = tso.write\ \langle heap.Write\ (ref.addr-of\ r)\ 0\ (heap.rep.to\ v) \rangle$

declare *tso.Ref.ref-def*[*code del*]

declare *tso.Ref.lookup-def*[*code del*]

declare *tso.Ref.update-def*[*code del*]

setup $\langle Sign.parent-path \rangle$

27.2 Inhabitation

In order to obtain compositional rules we need to make the write buffer explicit.

setup $\langle Sign.mandatory-path\ tso \rangle$

definition $t2s :: \text{write-buffer} \Rightarrow 'v \text{ tso} \Rightarrow (\text{sequential}, \text{heap.t}, 'v \times \text{write-buffer}) \text{ spec}$ **where**
 $t2s \text{ wb } P = \text{prog.p2s } (tso.t2p' P \text{ wb})$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.singleton.tso} \rangle$

lemma $t2s\text{-commit}$:

assumes $\langle \text{heap.apply-write } w \ s, \ xs, \ v \rangle \leq tso.t2s \text{ wb } f$

shows $\langle s, (\text{self}, \text{heap.apply-write } w \ s) \# \ xs, \ v \rangle \leq tso.t2s (w \# \text{wb}) f$

unfolding $tso.t2s\text{-def}$

by $(\text{subst raw.tso.closed-conv}[OF \ tso.t2p'])$

$(\text{fastforce simp: prog.p2s.action simp add: prog.p2s.simps simp flip: tso.t2s-def}$

$\text{intro: order.trans}[OF - \text{prog.p2s.mono}[OF \ \text{raw.tso.cl.commit}]$

$\text{spec.bind.continueI}[\mathbf{where} \ xs=[(\text{self}, \text{heap.apply-write } w \ s)] \ \mathbf{and} \ v=((), \ []), \ \text{simplified}, \ OF - \text{assms}]$

$\text{order.trans}[OF \ \text{spec.action.stepI spec.interference.expansive}]$)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.idle.tso} \rangle$

lemma $t2s\text{-le}$:

shows $\text{spec.idle} \leq tso.t2s \text{ wb } P$

by $(\text{simp add: tso.t2s-def spec.idle.p2s-le})$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.t2s} \rangle$

lemmas $\text{minimal}[iff] = \text{order.trans}[OF \ \text{spec.idle.minimal-le spec.idle.tso.t2s-le}]$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path spec.interference.tso} \rangle$

lemma $t2s\text{-le}$:

shows $\text{spec.rel } (\{env\} \times UNIV) \ggg (\lambda :: \text{unit. } tso.t2s \text{ wb } P) \leq tso.t2s \text{ wb } P$

by $(\text{simp add: tso.t2s-def prog.p2s.interference-wind-bind})$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path prog.p2s} \rangle$

lemma $t2p[\text{prog.p2s.simps}]$:

shows $\text{prog.p2s } (tso.t2p P)$

$= tso.t2s \ [] \ P \ggg (\lambda vwb. \text{prog.p2s } (\text{raw.MFENCE } (\text{snd } vwb) \ggg \text{prog.return } (\text{fst } vwb)))$

by $\text{transfer } (\text{simp add: tso.t2p-def tso.t2s-def raw.t2p-def prog.p2s.simps split-def})$

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path tso.t2s} \rangle$

lemma bind :

shows $tso.t2s \text{ wb } (f \ggg g) = tso.t2s \text{ wb } f \ggg (\lambda x. \text{tso.t2s } (\text{snd } x) (g (\text{fst } x)))$

unfolding $tso.t2s\text{-def}$ **by** $\text{transfer } (\text{simp add: raw.bind-def split-def prog.p2s.simps})$

lemma parallel :

shows $tso.t2s [] (P \parallel Q) = prog.p2s ((tso.t2p P \parallel tso.t2p Q) \gg prog.return ((), []))$
unfolding $tso.t2s-def$
by $transfer (simp add: raw.parallel-def raw.bind-def raw.MFENCE.Nil prog.bind.returnL)$

lemma $return$:

shows $tso.t2s [] (tso.return v) = prog.p2s (prog.return (v, []))$
unfolding $tso.t2s-def$
by $transfer$
 $(simp add: raw.return-alt-def raw.tso.cl.Nil raw.prim-return-def prog.bind.returnL raw.commit-writes.Nil)$

setup $\langle Sign.parent-path \rangle$

Inhabitation rules. **setup** $\langle Sign.mandatory-path inhabits.tso \rangle$

lemma $bind$:

assumes $tso.t2s wb f -s, xs \rightarrow tso.t2s wb' f'$
shows $tso.t2s wb (f \gg g) -s, xs \rightarrow tso.t2s wb' (f' \gg g)$
by $(simp add: tso.t2s.bind inhabits.spec.bind assms)$

lemma $commit$:

shows $tso.t2s (w \# wb) f -s, [(self, heap.apply-write w s)] \rightarrow tso.t2s wb f$
by $(clarsimp simp: inhabits-def spec.bind.singletonL spec.term.none.singleton trace.split-all spec.singleton.tso.t2s-commit)$

setup $\langle Sign.mandatory-path Ref \rangle$

lemma ref :

fixes $r :: 'a::heap.rep ref$
fixes $s :: heap.t$
fixes $v :: 'a$
fixes $v' :: 'a$
assumes $\neg heap.present r s$
shows $tso.t2s wb (tso.Ref.ref v)$
 $-s, [(self, Ref.set r v' s)] \rightarrow$
 $tso.t2s (wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]) (tso.return r) (is ?lhs -s, ?step \rightarrow ?rhs)$

proof –

have $rhs: ?rhs = prog.p2s (raw.commit-writes (wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)])$
 $\gg (\lambda v. raw.prim-return r (snd v)))$
apply $(simp add: tso.t2s-def tso.t2p'.return raw.return-def raw.action-def)$
apply $(subst (1) prog.return.cong)$
apply $(simp-all add: image-iff split-def Sup-fst fst-image raw.tso.cl.prim-return raw.bind-def$
 $flip: raw.prim-return-def)$
done
note $* = order.trans[OF - spec.bind.mono[OF prog.p2s.mono[OF$
 $raw.commit-writes.return-le[unfolded le-fun-def raw.prim-return-def, rule-format]]$
 $order.refl]]$
note $** = spec.bind.mono[OF spec.action.stepI[where a=self and s=s$
and $v=(r, wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)])$
and $s'=Ref.set r v' s$
and $w=Some (r, wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)])]$
 $order.refl]$
have $lhs: \langle s, [(self, Ref.set r v' s)], Some (r, wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]) \rangle$
 $\gg (\lambda v. prog.p2s (raw.commit-writes (snd v)))$
 $\gg (\lambda x. prog.p2s (raw.prim-return (fst v) (snd x)))$
 $\leq ?lhs$
apply $(simp add: tso.Ref.ref-def tso.t2s-def split-def tso.t2p'.action$
 $raw.action-def raw.tso.cl-def raw.bind-def prog.p2s.bind prog.bind.bind)$
apply $(rule *)$

```

apply (simp flip: prog.p2s.bind)
apply (force simp: assms Ref.alloc-def prog.p2s.simps prog.p2s.action prog.bind.returnL
      intro: ** order.trans[OF - spec.bind.mono[OF spec.interference.expansive order.refl]])
done
show ?thesis
unfolding inhabits-def
by (rule order.trans[OF - lhs])
      (simp add: rhs prog.p2s.simps spec.bind.singletonL spec.term.none.singleton)
qed

lemma lookup:
  fixes r :: 'a::heap.rep ref
  shows tso.t2s wb (!r) -s, [] → tso.t2s wb (tso.return (Ref.get r (apply-writes wb s))))
apply (clarsimp simp: tso.Ref.lookup-def inhabits-def trace.split-all
      tso.t2s-def tso.t2p'.action tso.t2p'.return
      raw.action-def raw.return-alt-def raw.tso.cl.prim-return
      spec.bind.singletonL spec.term.none.singleton)
apply (clarsimp simp: raw.tso.cl-def raw.bind-def split-def prog.bind.bind)
apply (rule order.trans[OF - prog.p2s.mono[OF
      prog.bind.mono[OF raw.commit-writes.return-le[unfolded raw.prim-return-def le-fun-def, rule-format]
      order.refl]]])
apply (force simp: prog.bind.return prog.p2s.bind prog.p2s.action
      intro: order.trans[OF spec.bind.continueI[where xs=[], simplified, OF spec.action.stutterI]
      spec.bind.mono[OF spec.interference.expansive order.refl]])
done

lemma update:
  fixes r :: 'a::heap.rep ref
  shows tso.t2s wb (r := v)
      -s, [] →
      tso.t2s (wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]) (tso.return ()))
proof –
  have *: (λp. raw.prim-return () (snd p) = prog.return)
    by (simp add: raw.prim-return-def fun-eq-iff)
  have raw.tso.cl (λwb. prog.return ((, wb)) (wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]))
    ≤ raw.tso.cl (λwb. prog.action {(((), wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]), s, s) |s. True}) wb
  — LHS
  apply (simp add: raw.tso.cl-def raw.bind.prim-returnR raw.commit-writes.commit-writes
      flip: raw.prim-return-def
      cong: order.assms-cong)
  — RHS
  apply (simp add: * raw.tso.cl-def raw.bind-def split-def prog.bind.bind)
  apply (rule order.trans[OF -
      prog.bind.mono[OF raw.commit-writes.return-le[unfolded raw.prim-return-def le-fun-def, rule-format]
      order.refl]]])
  apply (simp add: prog.bind.returnL flip: prog.p2s.bind)
  apply (subst (1) prog.return.cong, force, force)
  apply (simp add: split-def Sup-fst fst-image prog.bind.return)
  done
from prog.p2s.mono[OF this] show ?thesis
  by (fastforce simp: tso.Ref.update-def raw.action-def tso.t2s-def inhabits-def
      tso.t2p'.return tso.t2p'.action
      raw.return-alt-def raw.prim-return-def
      spec.bind.singletonL spec.term.none.singleton)
qed

setup ‹Sign.parent-path›

```

```
lemmas bind' = inhabits.trans[OF inhabits.tso.bind]
lemmas commit' = inhabits.trans[OF inhabits.tso.commit]
```

```
setup <Sign.parent-path>
```

27.3 Code generator setup for TSO

The following is only sound if the generated code runs on a machine with a TSO memory model such as:

- x86
- x86 code running on macOS under Rosetta 2 (ask Google)

Notes:

- Haskell: GHC exposes unfenced operations for references and some kinds of arrays
 - GHC has a zoo of arrays; for now we use the general but inefficient boxed array type
- SML: Poly/ML appears to have committed to release/acquire (see email with subject “Git master update: ARM64, PIE and new bootstrap process”)
 - on x86 this is TSO
- Scala: beyond the scope of this work

TODO:

- support a CAS-like operation
 - Haskell: <https://stackoverflow.com/questions/10102881/haskell-how-does-atomicmodifyio-ref-work>

27.3.1 Haskell

Adaption layer

```
code-printing code-module TSOHeap  $\rightarrow$  (Haskell)
```

```
<
module TSOHeap (
  TSO
  , IORef, newIORef, readIORef, writeIORef
  , Array, newArray, newListArray, newFunArray, lengthArray, readArray, writeArray
  , parallel
) where

import Control.Concurrent (forkIO)
import qualified Control.Concurrent.MVar as MVar
import qualified Data.Array.IO as Array -- FIXME boxed, contemplate the menagerie of other arrays; perhaps
type families might help here
import Data.IORef (IORef, newIORef, readIORef, writeIORef)
import Data.List (genericLength)

type TSO a = IO a
type Array a = Array.IOArray Integer a
type Ref a = Data.IORef.IORef a

writeIORef :: IORef a -> a -> IO ()
writeIORef = writeIORef -- FIXME strict variant?
```

```

newArray :: Integer -> a -> IO (Array a)
newArray k = Array.newArray (0, k - 1)

newListArray :: [a] -> IO (Array a)
newListArray xs = Array.newListArray (0, genericLength xs - 1) xs

newFunArray :: Integer -> (Integer -> a) -> IO (Array a)
newFunArray k f = Array.newListArray (0, k - 1) (map f [0..k-1])

lengthArray :: Array a -> IO Integer
lengthArray a = Array.getBounds a >>= return . (\(-, l) -> l + 1)

readArray :: Array a -> Integer -> IO a
readArray = Array.readArray

writeArray :: Array a -> Integer -> a -> IO ()
writeArray = Array.writeArray

-- note we don't want forkFinally as we don't model exceptions
parallel :: IO () -> IO () -> IO ()
parallel p q = do
  mvar <- MVar.newEmptyMVar
  forkIO (p >> MVar.putMVar mvar ()) -- FIXME putMVar is lazy
  b <- q
  a <- MVar.takeMVar mvar
  return ()
>

code-reserved (Haskell) TSOHeap

Monad

code-printing type-constructor tso -> (Haskell) TSOHeap.TSO -
code-monad tso.bind Haskell
code-printing constant tso.return -> (Haskell) return
code-printing constant tso.raise -> (Haskell) error
code-printing constant tso.parallel -> (Haskell) TSOHeap.parallel

Intermediate operation avoids invariance problem in Scala (similar to value restriction)
setup <Sign.mandatory-path tso.Ref>

definition ref' where
  [code del]: ref' = tso.Ref.ref

lemma [code]:
  tso.Ref.ref x = tso.Ref.ref' x
  by (simp add: tso.Ref.ref'-def)

setup <Sign.parent-path>

Haskell

code-printing type-constructor ref -> (Haskell) TSOHeap.Ref -
code-printing constant Ref -> (Haskell) error/ bare Ref
code-printing constant tso.Ref.ref' -> (Haskell) TSOHeap.newIORef
code-printing constant tso.Ref.lookup -> (Haskell) TSOHeap.readIORef
code-printing constant tso.Ref.update -> (Haskell) TSOHeap.writeIORef
code-printing constant HOL.equal :: 'a ref => 'a ref => bool -> (Haskell) infix 4 ==
code-printing class-instance ref :: HOL.equal -> (Haskell) -

```

27.4 A TSO litmus test

The classic TSO litmus test [Owens et al. \(2009, §1\)](#): write buffering allows both threads to read zero, which is impossible under sequential consistency.

definition *iwp2-3-a* :: (nat × nat) tso where

```
iwp2-3-a = do {
  x ← tso.Ref.ref 0
; y ← tso.Ref.ref 0
; xvr ← tso.Ref.ref 0
; yvr ← tso.Ref.ref 0
; ( ( do { x := 1 ; yv ← !y ; yvr := yv } )
  || ( do { y := 1 ; xv ← !x ; xvr := xv } ) )
; xv <- !xvr
; yv <- !yvr
; tso.return (xv, yv)
}
```

code-thms *iwp2-3-a*

export-code *iwp2-3-a* in *Haskell*

schematic-goal *iwp2-3-a*: — “Can terminate with both threads reading 0”

shows $\langle \text{heap.empty}, ?xs, \text{Some } (0, 0) \rangle \leq \text{prog.p2s } (tso.t2p \text{ } iwp2-3-a)$

supply *heap.simps*[*simp*]

apply (*rule inhabits.I*)

unfolding *iwp2-3-a-def*

apply (*simp add: prog.p2s.t2p*)

apply (*rule inhabits.spec.bind'*)

apply (*rule inhabits.tso.bind'*)

apply (*rule inhabits.tso.Ref.ref*[**where** *r=Ref 0*], *simp*; *fail*)

apply (*simp add: tso.bind.returnL*)

apply (*rule inhabits.tso.bind'*)

apply (*rule inhabits.tso.Ref.ref*[**where** *r=Ref 1*], *simp*)

apply (*simp add: tso.bind.returnL*)

apply (*rule inhabits.tso.bind'*)

apply (*rule inhabits.tso.Ref.ref*[**where** *r=Ref 2*], *simp*)

apply (*simp add: tso.bind.returnL*)

apply (*rule inhabits.tso.bind'*)

apply (*rule inhabits.tso.Ref.ref*[**where** *r=Ref 3*], *simp*)

apply (*simp add: tso.bind.returnL*)

apply (*rule inhabits.tso.commit'*)⁺

apply *simp*

apply (*rule inhabits.tso.bind'*)

apply (*simp add: tso.t2s.parallel prog.p2s.bind prog.p2s.parallel prog.p2s.t2p*)

apply (*rule inhabits.spec.bind'*)

apply (*rule inhabits.spec.parallelL'*)

apply (*rule inhabits.spec.bind'*)

apply (*rule inhabits.tso.bind'*)

apply (*rule inhabits.tso.Ref.update*)

apply (*simp add: tso.bind.returnL*)

apply (*rule inhabits.tso.bind'*)

apply (*rule inhabits.tso.Ref.lookup*)

apply (*simp add: tso.bind.returnL*)

apply (*rule inhabits.tso.Ref.update*)

apply (*rule inhabits.tau*)

apply (*simp add: spec.idle.bind-le-conv spec.idle.tso.t2s-le*; *fail*)

apply (*simp add: spec.bind.mono spec.interference.tso.t2s-le flip: spec.bind.bind*; *fail*)

```

apply (rule inhabits.spec.parallelR')
  apply (rule inhabits.spec.bind')
    apply (rule inhabits.tso.bind')
      apply (rule inhabits.tso.Ref.update)
    apply (simp add: tso.bind.returnL)
  apply (rule inhabits.tso.bind')
    apply (rule inhabits.tso.Ref.lookup)
  apply (simp add: tso.bind.returnL)
  apply (rule inhabits.tso.Ref.update)
apply (rule inhabits.tau)
apply (simp add: spec.idle.bind-le-conv spec.idle.tso.t2s-le; fail)
apply (simp add: spec.bind.mono spec.interference.tso.t2s-le flip: spec.bind.bind; fail)
apply clarsimp

apply (rule inhabits.spec.parallelL')
  apply (rule inhabits.spec.bind'[OF inhabits.tso.commit])+
  apply (simp add: tso.t2s.return split-def prog.bind.returnL raw.MFENCE.Nil flip: prog.p2s.bind)
  apply (rule inhabits.tau)
  apply (simp add: spec.idle.p2s-le; fail)
apply (simp add: spec.bind.mono spec.interference.tso.t2s-le flip: spec.bind.bind; fail)

apply (rule inhabits.spec.parallelR')
  apply (rule inhabits.spec.bind'[OF inhabits.tso.commit])+
  apply (simp add: tso.t2s.return split-def prog.bind.returnL raw.MFENCE.Nil flip: prog.p2s.bind)
  apply (rule inhabits.tau)
  apply (simp add: spec.idle.p2s-le; fail)
apply (simp add: prog.p2s.interference-wind-bind; fail)

apply (simp add: prog.parallel.return flip: prog.p2s.parallel)
apply (rule inhabits.tau)
apply (simp add: spec.idle.p2s-le; fail)
apply (simp add: prog.bind.returnL flip: prog.p2s.bind)
apply (subst tso.t2s.return[symmetric])
apply (rule inhabits.tau)
apply (simp add: spec.idle.tso.t2s-le; fail)

apply (simp add: tso.bind.return)
apply (rule inhabits.tso.bind')
  apply (rule inhabits.tso.Ref.lookup)
apply (simp add: Ref.get-def apply-writes-def tso.bind.return)
apply (rule inhabits.tso.bind')
  apply (rule inhabits.tso.Ref.lookup)
apply (simp add: Ref.get-def apply-writes-def tso.bind.return tso.t2s.return)
apply (rule inhabits.prog.return)
apply (simp add: spec.bind.returnL spec.idle.p2s-le raw.MFENCE.Nil prog.bind.returnL)
apply (rule inhabits.prog.return)
done

thm iw2-3-a[simplified apply-writes-def, simplified]

```

28 Floyd-Warshall all-pairs shortest paths

The Floyd-Warshall algorithm computes the lengths of the shortest paths between all pairs of nodes by updating an adjacency (square) matrix that represents the edge weights. Our goal here is to present it at a very abstract level to exhibit the data dependencies.

Source materials:

- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- \$AFP/Floyd_Warshall/Floyd_Warshall.thy
 - a proof by refinement yielding a thorough correctness result including negative weights but not the absence of edges
- Dingel (2002, §6.2)
 - Overly parallelised, which is not practically useful but does reveal the data dependencies
 - the refinement is pretty much the same as the direct partial correctness proof here
 - the equivalent to *fw-update* is a single expression

We are not very ambitious here. This theory:

- does not track the actual shortest paths here but it is easy to add another array to do so
- ignores numeric concerns
- assumes the graph is complete

A further step would be to refine the parallel program to the classic three-loop presentation.

definition *fw-update* :: ('i::Ix × 'i, nat) array ⇒ 'i × 'i ⇒ 'i ⇒ unit imp **where**
fw-update = (λa (i, j) k. do {
 ij ← prog.Array.nth a (i, j);
 ik ← prog.Array.nth a (i, k);
 kj ← prog.Array.nth a (k, j);
 prog.whenM (ik + kj < ij) (prog.Array.upd a (i, j) (ik + kj))
 })

— top-level specification: we can process the nodes in an arbitrary order

definition *fw-chaotic* :: ('i::Ix × 'i, nat) array ⇒ unit imp **where**
fw-chaotic a =
 (let b = array.bounds a in
 prog.Array.fst-app-chaotic b (λk. ||(i, j)∈set (Ix.interval b). *fw-update* a (i, j) k))

— executable version

definition *fw* :: ('i::Ix × 'i, nat) array ⇒ unit imp **where**
fw a =
 (let b = array.bounds a in
 prog.Array.fst-app b (λk. ||(i, j)∈set (Ix.interval b). *fw-update* a (i, j) k))

lemma *fw-fw-chaotic-le*: — the executable program refines the specification

shows *fw* a ≤ *fw-chaotic* a

unfolding *fw-chaotic-def* *fw-def*

by (*strengthen ord-to-strengthen* (1)[OF *prog.Array.fst-app-fst-app-chaotic-le*]) *simp*

Safety proof **type-synonym** 'i matrix = 'i × 'i ⇒ nat

— The weight of the given path

fun *path-weight* :: 'i matrix ⇒ 'i × 'i ⇒ 'i list ⇒ nat **where**
path-weight m ij [] = m ij
| *path-weight* m ij (k # xs) = m (fst ij, k) + *path-weight* m (k, snd ij) xs

— The set of acyclic paths from *i* to *j* using the nodes *ks*

definition *paths* :: 'i × 'i ⇒ 'i set ⇒ 'i list set **where**

$paths\ ij\ ks = \{p. set\ p \subseteq ks \wedge fst\ ij \notin set\ p \wedge snd\ ij \notin set\ p \wedge distinct\ p\}$

— The minimum weight of a path from i to j using the nodes ks . See \$AFP/Floyd_Warshall/Floyd_Warshall.thy for proof that these are minimal amongst all paths.

definition $min\text{-}path\text{-}weight :: 'i\ matrix \Rightarrow 'i \times 'i \Rightarrow 'i\ set \Rightarrow nat$ **where**
 $min\text{-}path\text{-}weight\ m\ ij\ ks = Min\ (path\text{-}weight\ m\ ij\ \text{'}\ paths\ ij\ ks)$

context

fixes $a :: ('i::Ix \times 'i, nat)$ array

fixes $m :: 'i\ matrix$

begin

definition $fw\text{-}p\text{-}inv :: 'i \times 'i \Rightarrow 'i\ set \Rightarrow heap.t\ pred$ **where** — process invariant
 $fw\text{-}p\text{-}inv\ ij\ ks = (heap.rep\text{-}inv\ a \wedge Array.get\ a\ ij = \langle min\text{-}path\text{-}weight\ m\ ij\ ks \rangle)$

definition $fw\text{-}inv :: 'i\ set \Rightarrow heap.t\ pred$ **where** — loop invariant
 $fw\text{-}inv\ ks = (\forall\ ij. \langle ij \in set\ (Array.interval\ a) \rangle \longrightarrow fw\text{-}p\text{-}inv\ ij\ ks)$

definition $fw\text{-}pre :: heap.t\ pred$ **where** — overall precondition
 $fw\text{-}pre = (\langle Array.square\ a \rangle \wedge heap.rep\text{-}inv\ a$
 $\wedge (\forall\ ij. \langle ij \in set\ (Array.interval\ a) \rangle \longrightarrow Array.get\ a\ ij = \langle m\ ij \rangle))$

definition $fw\text{-}post :: unit \Rightarrow heap.t\ pred$ **where** — overall postcondition
 $fw\text{-}post = fw\text{-}inv\ (set\ (Ix.interval\ (fst\ bounds\ (array.bounds\ a))))$

end

setup $\langle Sign.mandatory\text{-}path\ paths \rangle$

lemma I :

assumes $set\ p \subseteq ks$

assumes $i \notin set\ p$

assumes $j \notin set\ p$

assumes $distinct\ p$

shows $p \in paths\ (i, j)\ ks$

using $assms$ **by** $(simp\ add: paths\text{-}def)$

lemma Nil :

shows $\square \in paths\ ij\ ks$

by $(simp\ add: paths\text{-}def)$

lemma $empty$:

shows $paths\ ij\ \{\} = \{\square\}$

by $(fastforce\ simp: paths\text{-}def)$

lemma $not\text{-}empty$:

shows $paths\ ij\ ks \neq \{\}$

by $(metis\ empty\text{-}iff\ paths.Nil)$

lemma $monotone$:

shows $mono\ (paths\ ij)$

by $(rule\ monoI)\ (auto\ simp\ add: paths\text{-}def)$

lemmas $mono = monoD[OF\ paths.monotone]$

lemmas $strengthen[strg] = st\text{-}monotone[OF\ paths.monotone]$

lemma $finite$:

assumes $finite\ ks$

shows *finite* (*paths ij ks*)
unfolding *paths-def* **by** (*rule finite-subset[OF - iffD1[OF finite-distinct-conv assms]]*) *auto*

lemma *unused*:

assumes $p \in \text{paths } ij$ (*insert k ks*)
assumes $k \notin \text{set } p$
shows $p \in \text{paths } ij \text{ ks}$

using *assms* **unfolding** *paths-def* **by** *blast*

lemma *decompE*:

assumes $p \in \text{paths } (i, j)$ (*insert k ks*)
assumes $k \in \text{set } p$
obtains $r \ s$
where $p = r @ k \# s$
and $r \in \text{paths } (i, k) \text{ ks}$ **and** $s \in \text{paths } (k, j) \text{ ks}$
and *distinct* ($r @ s$) **and** $i \notin \text{set } (r @ k \# s)$ **and** $j \notin \text{set } (r @ k \# s)$

using *assms* **by** (*fastforce simp: paths-def dest: split-list*)

setup $\langle \text{Sign.parent-path} \rangle$

setup $\langle \text{Sign.mandatory-path path-weight} \rangle$

lemma *append*:

shows $\text{path-weight } m \ ij \ (xs @ y \# ys) = \text{path-weight } m \ (fst \ ij, y) \ xs + \text{path-weight } m \ (y, snd \ ij) \ ys$
by (*induct xs arbitrary: ij*) *simp-all*

setup $\langle \text{Sign.parent-path} \rangle$

lemmas *min-path-weightI* = *trans[OF min-path-weight-def Min-eqI]*

setup $\langle \text{Sign.mandatory-path min-path-weight} \rangle$

lemma *fw-update*:

assumes $m: \text{min-path-weight } m \ (i, k) \text{ ks} + \text{min-path-weight } m \ (k, j) \text{ ks} < \text{min-path-weight } m \ (i, j) \text{ ks}$
assumes *finite ks*
shows $\text{min-path-weight } m \ (i, j)$ (*insert k ks*)
 $= \text{min-path-weight } m \ (i, k) \text{ ks} + \text{min-path-weight } m \ (k, j) \text{ ks}$ (**is** *?lhs = ?rhs*)

proof(*rule min-path-weightI*)

from $\langle \text{finite ks} \rangle$ **show** *finite* ($\text{path-weight } m \ (i, j) \text{ ' paths } (i, j) \text{ (insert k ks)}$)
by (*simp add: paths.finite*)

next

fix w

assume $w: w \in \text{path-weight } m \ (i, j) \text{ ' paths } (i, j) \text{ (insert k ks)}$

then obtain p **where** $p: w = \text{path-weight } m \ (i, j) \ p \ p \in \text{paths } (i, j) \text{ (insert k ks)}$..

show $?rhs \leq w$

proof(*cases k \in set p*)

case *True* **with** $m \ \langle \text{finite ks} \rangle \ w \ p$ **show** *?thesis*

by (*clarsimp simp: min-path-weight-def path-weight.append elim!: paths.decompE*)

(*auto simp: Min-plus paths.finite paths.not-empty finite-image-set2 intro!: Min-le*)

next

case *False* **with** $m \ \langle \text{finite ks} \rangle \ w \ p$ **show** *?thesis*

unfolding *min-path-weight-def*

by (*fastforce simp: paths.finite paths.not-empty dest: paths.unused*)

qed

next

from $\langle \text{finite ks} \rangle$ **obtain** pik

where $pik: pik \in \text{paths } (i, k) \text{ ks}$

and $mpik: \text{Min} \ (\text{path-weight } m \ (i, k) \text{ ' paths } (i, k) \text{ ks}) = \text{path-weight } m \ (i, k) \ pik$

by (meson finite-set Min-in finite-imageI paths.finite image-iff image-is-empty paths.not-empty)
 from ⟨finite ks⟩ obtain pkj
 where pkj: $pkj \in paths\ k, j\ ks$
 and mpkj: $Min\ (path\text{-}weight\ m\ (k, j)\ \langle paths\ (k, j)\ ks\rangle) = path\text{-}weight\ m\ (k, j)\ pkj$
 by (meson finite-set Min-in finite-imageI paths.finite image-iff image-is-empty paths.not-empty)
 let ?p = pik @ k # pkj
 have ?p ∈ paths (i, j) (insert k ks)
 proof(rule paths.I)
 from pik pkj
 show set ?p ⊆ insert k ks by (auto simp: paths-def)
 show i ∉ set ?p
 proof(rule notI)
 assume i ∈ set ?p
 with m pik have i ∈ set pkj by (fastforce simp: paths-def)
 then obtain p' zs where *: $pkj = zs\ @\ i\ \#\ p'$ by (meson split-list)
 moreover from pkj * have p' ∈ paths (i, j) ks by (simp add: paths-def)
 moreover note m ⟨finite ks⟩ mpkj
 ultimately show False by (simp add: paths.finite leD min-path-weight-def path-weight.append trans-le-add2)
 qed
 show j ∉ set ?p
 proof(rule notI)
 assume j ∈ set ?p
 with m pkj have j ∈ set pik by (fastforce simp: paths-def)
 then obtain p' zs where *: $pik = p'\ @\ j\ \#\ zs$ by (meson split-list)
 moreover from pik * have p' ∈ paths (i, j) ks by (simp add: paths-def)
 moreover note m ⟨finite ks⟩ mpik
 ultimately show False
 by (fastforce simp: min-path-weight-def path-weight.append paths.finite paths.not-empty)
 qed
 show distinct ?p
 proof(rule ccontr)
 let ?p1 = takeWhile (λx. x ∉ set pkj) pik
 let ?l = hd (drop (length ?p1) pik)
 let ?p2 = tl (dropWhile (λx. x ≠ ?l) pkj)
 let ?p' = ?p1 @ ?l # ?p2
 assume ¬distinct (pik @ k # pkj)
 from pik pkj ⟨¬distinct (pik @ k # pkj)⟩ have strict-prefix ?p1 pik
 by (auto simp: paths-def strict-prefix-def takeWhile-is-prefix)
 from pik pkj ⟨¬distinct (pik @ k # pkj)⟩ ⟨strict-prefix ?p1 pik⟩ have strict-suffix ?p2 pkj
 by (fastforce simp: dropWhile-eq-drop tl-drop
 intro: drop-strict-suffix[OF strict-suffix-tl]
 dest: prefix-length-less nth-length-takeWhile)
 from ⟨strict-prefix ?p1 pik⟩ have ?l ∈ set pkj
 by (fastforce simp: hd-drop-conv-nth dest: prefix-length-less nth-length-takeWhile)
 have ?p' ∈ paths (i, j) ks
 proof(rule paths.I)
 from pik pkj ⟨strict-prefix ?p1 pik⟩ ⟨strict-suffix ?p2 pkj⟩ ⟨?l ∈ set pkj⟩ show set ?p' ⊆ ks
 by (force dest: set-takeWhileD strict-suffix-set-subset simp: paths-def)
 from ⟨i ∉ set ?p⟩ ⟨strict-suffix ?p2 pkj⟩ ⟨?l ∈ set pkj⟩ show i ∉ set ?p'
 by (auto dest: set-takeWhileD strict-suffix-set-subset)
 from ⟨j ∉ set ?p⟩ ⟨strict-suffix ?p2 pkj⟩ ⟨?l ∈ set pkj⟩ show j ∉ set ?p'
 by (auto dest: set-takeWhileD strict-suffix-set-subset)
 from pik pkj ⟨strict-suffix ?p2 pkj⟩ ⟨?l ∈ set pkj⟩ show distinct ?p'
 by (auto simp: paths-def distinct-tl dest!: set-takeWhileD strict-suffix-set-subset
 simp flip: arg-cong[where f=set, OF takeWhile-neq-rev, simplified])
 qed
 have path-weight m (i, j) ?p' ≤ path-weight m (i, k) pik + path-weight m (k, j) pkj
 unfolding path-weight.append

```

proof(induct rule: add-le-mono[case-names l r])
  case l from  $\langle \text{strict-prefix } ?p1 \text{ pik} \rangle$  show ?case
    by (metis append.right-neutral append-take-drop-id fst-conv linorder-le-less-linear
      list.collapse not-add-less1 path-weight.append prefix-order.less-le takeWhile-eq-take)
  next
    case r from  $\langle ?l \in \text{set } pkj \rangle$  show ?case
      by (smt (verit) append.right-neutral hd-dropWhile le-add2 list.collapse path-weight.append
        set-takeWhileD snd-conv takeWhile-dropWhile-id)
  qed
with m  $\langle \text{finite } ks \rangle$  mpik mpkj  $\langle ?p' \in \text{paths } (i, j) \text{ ks} \rangle$  show False
  by (fastforce simp: min-path-weight-def paths.finite paths.not-empty)
qed
with m mpik mpkj
show  $?rhs \in \text{path-weight } m (i, j) \text{ 'paths } (i, j) \text{ (insert } k \text{ ks)}$ 
  by (force simp: min-path-weight-def path-weight.append)
qed

lemma return:
  assumes m:  $\neg(\text{min-path-weight } m (i, k) \text{ ks} + \text{min-path-weight } m (k, j) \text{ ks} < \text{min-path-weight } m (i, j) \text{ ks})$ 
  assumes finite ks
  shows  $\text{min-path-weight } m (i, j) \text{ (insert } k \text{ ks)} = \text{min-path-weight } m (i, j) \text{ ks}$ 
unfolding min-path-weight-def
proof(rule Min-eqI)
  from  $\langle \text{finite } ks \rangle$  show finite ( $\text{path-weight } m (i, j) \text{ 'paths } (i, j) \text{ (insert } k \text{ ks)}$ )
  by (simp add: paths.finite)
next
  fix w
  assume w:  $w \in \text{path-weight } m (i, j) \text{ 'paths } (i, j) \text{ (insert } k \text{ ks)}$ 
  then obtain p where p:  $w = \text{path-weight } m (i, j) \text{ p } p \in \text{paths } (i, j) \text{ (insert } k \text{ ks)} \dots$ 
  with m  $\langle \text{finite } ks \rangle$  show Min ( $\text{path-weight } m (i, j) \text{ 'paths } (i, j) \text{ ks}$ )  $\leq w$ 
  proof(cases k \in set p)
    case True with m  $\langle \text{finite } ks \rangle$  w p show ?thesis
      by (auto simp: not-less min-path-weight-def path-weight.append paths.finite
        intro: order.trans[OF add-mono[OF Min-le Min-le]]
        elim!: order.trans paths.decompE)
  next
    case False with m  $\langle \text{finite } ks \rangle$  w p show ?thesis
      by (meson Min-le finite-imageI paths.finite image-eqI paths.unused)
  qed
next
  from  $\langle \text{finite } ks \rangle$ 
  show  $\text{Min} (\text{path-weight } m (i, j) \text{ 'paths } (i, j) \text{ ks}) \in \text{path-weight } m (i, j) \text{ 'paths } (i, j) \text{ (insert } k \text{ ks)}$ 
  by (fastforce simp: paths.finite paths.not-empty intro: subsetD[OF - Min-in] subsetD[OF paths.mono])
qed

setup  $\langle \text{Sign.parent-path} \rangle$ 

setup  $\langle \text{Sign.mandatory-path stable} \rangle$ 

lemma Id-on-fw-inv:
  shows stable heap.Id{a} (fw-inv a m ys)
  by (auto simp: fw-inv-def fw-p-inv-def intro!: stable.intro stable.impliesI)

lemma Id-on-fw-p-inv:
  shows stable heap.Id{a} (fw-p-inv a m ij ks)
  by (auto simp: fw-p-inv-def intro: stable.intro)

```

lemma *modifies-fw-p-inv*:

assumes $ij \in \text{set } (\text{Array.interval } a) - is$
shows $\text{stable Array.modifies}_{a, is} (\text{fw-p-inv } a \ m \ ij \ ks)$
using *assms* **by** (*auto simp: fw-p-inv-def intro: stable.intro*)

setup $\langle \text{Sign.parent-path} \rangle$

lemma *fw-p-inv-cong*:

assumes $a = a'$
assumes $m = m'$
assumes $ij = ij'$
assumes $ks = ks'$
assumes $s (\text{heap.addr-of } a) = s' (\text{heap.addr-of } a')$
shows $\text{fw-p-inv } a \ m \ ij \ ks \ s = \text{fw-p-inv } a' \ m' \ ij' \ ks' \ s'$
using *assms* **by** (*simp add: fw-p-inv-def cong: heap.obj-at.cong Array.get.weak-cong*)

lemma *fw-p-invD*:

assumes $\text{fw-p-inv } a \ m \ ij \ ks \ s$
shows $\text{heap.rep-inv } a \ s$
and $\text{Array.get } a \ ij \ s = \text{min-path-weight } m \ ij \ ks$
using *assms* **unfolding** *fw-p-inv-def* **by** *blast+*

lemma *fw-p-inv-fw-update*:

assumes *finite* ks
assumes $ij \in \text{set } (\text{Array.interval } a)$
assumes $\text{fw-p-inv } a \ m \ ij \ ks \ s$
assumes $\text{min-path-weight } m \ (\text{fst } ij, k) \ ks + \text{min-path-weight } m \ (k, \text{snd } ij) \ ks < \text{min-path-weight } m \ ij \ ks$
shows $\text{fw-p-inv } a \ m \ ij \ (\text{insert } k \ ks) \ (\text{Array.set } a \ ij \ (\text{min-path-weight } m \ (\text{fst } ij, k) \ ks + \text{min-path-weight } m \ (k, \text{snd } ij) \ ks) \ s)$
using *assms* **by** (*cases ij*) (*simp add: fw-p-inv-def Array.simps' min-path-weight.fw-update*)

lemma *fw-p-inv-return*:

assumes *finite* ks
assumes $\text{fw-p-inv } a \ m \ ij \ ks \ s$
assumes $\neg(\text{min-path-weight } m \ (\text{fst } ij, k) \ ks + \text{min-path-weight } m \ (k, \text{snd } ij) \ ks < \text{min-path-weight } m \ ij \ ks)$
shows $\text{fw-p-inv } a \ m \ ij \ (\text{insert } k \ ks) \ s$
using *assms* **by** (*cases ij*) (*simp add: fw-p-inv-def min-path-weight.return*)

setup $\langle \text{Sign.mandatory-path } ag \rangle$

Dingel (2000, p109) key intuition: when processing index k , neither $a[i, k]$ and $a[k, j]$ change.

- his argument is bogus: it is enough to observe that shortest paths never get shorter by adding edges
- he unnecessarily assumes that $\delta(i, i) = 0$ for all i

lemma *fw-update*:

assumes $\text{insert } k \ ks \subseteq \text{set } (\text{Ix.interval } (\text{fst-bounds } (\text{array.bounds } a)))$
assumes $\text{Array.square } a$
assumes $ij: ij \in \text{set } (\text{Array.interval } a)$
defines $\bigwedge ij. G \ ij \equiv \text{Array.modifies}_{a, \{ij \mid \text{unit. } k \notin \{\text{fst } ij, \text{snd } ij\}\}}$
defines $A \equiv \text{heap.Id}_{\{a\}} \cup \bigcup (G \ ' (\text{set } (\text{Array.interval } a) - \{ij\}))$
shows $\text{prog.p2s } (\text{fw-update } a \ ij \ k)$
 $\leq \{\{\text{fw-p-inv } a \ m \ ij \ ks \wedge \text{fw-p-inv } a \ m \ (\text{fst } ij, k) \ ks \wedge \text{fw-p-inv } a \ m \ (k, \text{snd } ij) \ ks\}, A$
 $\vdash G \ ij, \{\lambda-. \text{fw-p-inv } a \ m \ ij \ (\text{insert } k \ ks)\}$

proof –

from *assms*(1) **have** *finite* ks

using *finite-subset* **by** *auto*

from *assms*(1–3) **have** $ijk: (\text{fst } ij, k) \in \text{set } (\text{Array.interval } a) \ (k, \text{snd } ij) \in \text{set } (\text{Array.interval } a)$

by (*auto simp: Ix.square-def interval-prod-def*)
 show *?thesis*
 apply (*simp add: fw-update-def split-def*)
 apply (*rule ag.pre-pre*)
 apply (*rule ag.prog.bind*)+
 apply (*rule ag.prog.if*)
 apply (*rename-tac v_{ij} v_{ik} v_{kj}*)
 apply (*subst prog.Array.upd-def*)
 apply (*rule-tac P= $\lambda s. fw-p-inv a m ij ks s \wedge fw-p-inv a m (fst ij, k) ks s \wedge fw-p-inv a m (k, snd ij) ks s$*
 $\wedge v_{ij} = Array.get a ij s \wedge v_{ik} = Array.get a (fst ij, k) s \wedge v_{kj} = Array.get a (k, snd ij) s$)
 in *ag.prog.action*)
 apply (*clarsimp simp: <finite ks> fw-p-invD(2) fw-p-inv-fw-update ij; fail*)
 using *ij* apply (*fastforce simp: G-def intro: Array.modifies.Array-set dest: fw-p-invD(1)*)
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 apply (*rename-tac v_{ij} v_{ik} v_{kj}*)
 apply (*rule-tac Q= $\lambda s. v_{ij} = Array.get a ij s \wedge v_{ik} = Array.get a (fst ij, k) s \wedge v_{kj} = Array.get a (k, snd ij) s$*
 $ij) s$)
 in *ag.augment-post*)
 apply (*rule ag.prog.return*)
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 apply (*rename-tac v_{ij} v_{ik}*)
 apply (*rule-tac Q= $\lambda v s. fw-p-inv a m ij ks s \wedge fw-p-inv a m (fst ij, k) ks s \wedge fw-p-inv a m (k, snd ij) ks s$*
 $\wedge v_{ij} = Array.get a ij s \wedge v_{ik} = Array.get a (fst ij, k) s \wedge v = Array.get a (k, snd ij) s$)
 in *ag.post-imp*)
 apply (*force simp: <finite ks> fw-p-invD(2) fw-p-inv-return*)
 apply (*subst prog.Array.nth-def*)
 apply (*rule ag.prog.action*)
 apply (*clarsimp split del: if-split; assumption*)
 apply *fast*
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 apply (*subst prog.Array.nth-def*)
 apply (*rule ag.prog.action*)
 apply (*clarsimp; assumption*)
 apply *fast*
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 apply (*subst prog.Array.nth-def*)
 apply (*rule ag.prog.action*)
 apply (*clarsimp; assumption*)
 apply *fast*
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 using *ij ijk* apply (*fastforce simp: A-def G-def intro: stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
 apply *blast*
 done
 qed

lemma *fw-chaotic*:

fixes *a* :: (*'i::Ix* \times *'i*, *nat*) *array*
 fixes *m* :: *'i* *matrix*
 shows *prog.p2s (fw-chaotic a) \leq {fw-pre a m}, heap.Id_{a} \vdash heap.modifies_{a}, {fw-post a m}*
 unfolding *fw-chaotic-def fw-pre-def*
 apply (*simp add: prog.p2s.simps Let-def split-def*)
 apply (*rule ag.gen-asm*)
 apply (*rule ag.pre-pre-post*)
 apply (*rule ag.prog.fst-app-chaotic[where P= $fw-inv a m$]*)
 apply (*rule ag.pre*)

```

apply (rule ag.prog.Parallel)
apply (rule ag.fw-update[where m=m])
  apply (simp; fail)
  apply (simp; fail)
apply (simp; fail)
apply (fastforce simp: fw-inv-def split-def Ix.prod.interval-conv Ix.square.conv)
apply blast
using Array.modifies.heap-modifies-le apply blast
apply (simp add: fw-inv-def; fail)
apply (simp add: stable.Id-on-fw-inv; fail)
apply (fastforce simp: fw-pre-def fw-inv-def fw-p-inv-def min-path-weight-def paths.empty)
apply (fastforce simp: fw-post-def split-def stable.Id-on-fw-inv)
done

setup ‹Sign.parent-path›

```

References

- M. Abadi. An axiomatization of lamport's temporal logic of actions. In *CONCUR '90*, volume 458 of *LNCS*, pages 57–69. Springer, 1990. doi: 10.1007/BFB0039051.
- M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi: 10.1016/0304-3975(91)90224-P.
- M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995. doi: 10.1145/203095.201069.
- M. Abadi and S. Merz. An abstract account of composition. In *MFCS'95*, volume 969 of *LNCS*, pages 499–508. Springer, 1995. doi: 10.1007/3-540-60246-1_155.
- M. Abadi and S. Merz. On TLA as a logic. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 235–271. IOS Press, 1996. ISBN 3-540-60947-4.
- M. Abadi and G. D. Plotkin. A logical view of composition and refinement. In *POPL'1991*, pages 323–332. ACM Press, 1991. doi: 10.1145/99583.99626.
- M. Abadi and G. D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993. doi: 10.1016/0304-3975(93)90151-I.
- B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. doi: 10.1016/0020-0190(85)90056-0.
- B. Alpern, A. J. Demers, and F. B. Schneider. Safety without stuttering. *Information Processing Letters*, 23(4):177–180, 1986. doi: 10.1016/0020-0190(86)90132-8.
- K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009. ISBN 978-1-84882-744-8. doi: 10.1007/978-1-84882-745-5.
- A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014. doi: 10.1007/978-3-319-06410-9_6.
- R. C. Backhouse. Galois connections and fixed point calculus. In R.C. Backhouse, R. L. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 89–148. Springer, 2000. doi: 10.1007/3-540-47797-7_4.
- S. D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. doi: 10.1006/inco.1996.0056.
- A. Cau and P. Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33(2):153–176, 1996. doi: 10.1007/s002360050039.

- K. M. Chandy and J. Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989. ISBN 978-0-201-05866-6.
- B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. ISBN 978-0-521-78451-1. doi: 10.1017/CBO9780511809088.
- G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13*, pages 854–860. IJCAI/AAAI, 2013.
- W.-P. de Roeper and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
- W.-P. de Roeper, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001. ISBN 0-521-80608-9.
- J. Dingel. Modular verification for shared-variable concurrent programs. In *CONCUR '96*, volume 1119 of *LNCS*, pages 703–718. Springer, 1996. doi: 10.1007/3-540-61604-7_85.
- J. Dingel. *Systematic parallel programming*. PhD thesis, Carnegie Mellon University, May 2000. CMU Tech Report CS-99-172.
- J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14(2):123–197, 2002. doi: 10.1007/s001650200032.
- E. A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983. doi: 10.1016/0304-3975(83)90082-8.
- L. Esakia, G. Bezhanishvili, W. H. Holliday, and A. Evseev. *Heyting Algebras: Duality Theory*. Springer, 1st edition, 2019. ISBN 3030120953.
- S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in isabelle/utp. *Science of Computer Programming*, 197:102510, 2020. ISSN 0167-6423. doi: 10.1016/j.scico.2020.102510.
- G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. doi: 10.1017/CBO9780511542725.
- R. Goldblatt. *Logics of Time and Computation*. Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, 2 edition, 1992.
- I. J. Hayes. Generalised rely-guarantee concurrency: an algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, 2016. doi: 10.1007/s00165-016-0384-0.
- I. J. Hayes and C. B. Jones. A guide to rely/guarantee thinking. In *SETSS 2017*, volume 11174 of *LNCS*, pages 1–38. Springer, 2017. doi: 10.1007/978-3-030-02928-9_1.
- C. A. R. Hoare and J. He. The weakest prespecification. *Information Processing Letters*, 24(2):127–132, 1987. doi: 10.1016/0020-0190(87)90106-2. Oxford Technical Monograph PRG-44.
- C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987a. doi: 10.1145/27651.27653.
- C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, 1987b. doi: 10.1016/0020-0190(87)90224-9.
- C. A. R. Hoare, J. He, and A. Sampaio. Algebraic derivation of an operational semantics. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 77–98. The MIT Press, 2000.
- T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra and its foundations. *Journal of Logic and Algebraic Programming*, 80(6):266–296, 2011. doi: 10.1016/j.jlap.2011.04.005.

- P. B. Jackson. Verifying a garbage collection algorithm. In *TPHOLs*, volume 1479 of *LNCS*, pages 225–244. Springer, 1998. doi: 10.1007/BFb0055139.
- R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In Lars Birkedal, editor, *FOSSACS 2012*, volume 7213 of *LNCS*, pages 180–194. Springer, 2012. doi: 10.1007/978-3-642-28729-9_12.
- C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. doi: 10.1145/69575.69577.
- B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1&2):47–72, 1996. doi: 10.1016/0304-3975(96)00069-2.
- R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi: 10.1016/S0022-0000(69)80011-5.
- E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(30):268–272, 6 1994.
- D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994. doi: 10.1006/inco.1994.1037.
- F. Kröger and S. Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. ISBN 978-3-540-67401-6.
- L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994. doi: 10.1145/177492.177726.
- L. Lamport. Specifying concurrent systems in TLA⁺. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series, III: Computer and Systems Sciences*, pages 183–247. IOS Press, January 2000. ISBN 9789051994599. Proceedings of Marktoberdorf 1998.
- H. Liang, X. Feng, and M. Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Transactions on Programming Languages and Systems*, 36(1):3:1–3:55, 2014. doi: 10.1145/2576235.
- N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.
- P. Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP’2001*, volume 2076 of *LNCS*, pages 821–834. Springer, 2001. doi: 10.1007/3-540-48224-5_67.
- P. Maier. Intuitionistic LTL and a new characterization of safety and liveness. In *CSL 2004*, volume 3210 of *LNCS*, pages 295–309. Springer, 2004. doi: 10.1007/978-3-540-30124-0_24.
- Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *LNCS*, pages 201–284. Springer, 1988. doi: 10.1007/BFb0013024.
- Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 121–156. ACM Press and Addison-Wesley, 1991. Also Technical Report STAN-CS-90-1321.
- P. Manolios and R. J. Treffer. A lattice-theoretic characterization of safety and liveness. In E. Borowsky and S. Rajsbaum, editors, *PODC’2003*, pages 325–333. ACM, 2003. doi: 10.1145/872035.872083.
- A. Melton, D. A. Schmidt, and G. E. Strecker. Calois connections and computer science applications. In *Category Theory and Computer Programming*, volume 240 of *LNCS*, pages 299–312. Springer, 1985. doi: 10.1007/3-540-17162-2_130.
- M. Müller-Olm. *Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS*. Springer, 1997. doi: 10.1007/BFb0027453.
- H. Ono. *Proof Theory and Algebra in Logic*. Short Textbooks in Logic. Springer, 2019. doi: 10.1007/978-981-13-7997-0.

- S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs'2009*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009. doi: 10.1007/978-3-642-03359-9_27. URL <https://www.cl.cam.ac.uk/~pes20/weakmemory/x86tso-paper.pdf>.
- S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. doi: 10.1007/BF00268134.
- S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. doi: 10.1145/357172.357178.
- J. L. Pfaltz and J. Šlapal. Transformations of discrete closure systems. *Acta Mathematica Hungarica*, 138(4): 386–405, 2013. doi: 10.1007/s10474-012-0262-z.
- V. R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI, European Workshop, JELIA '90, Amsterdam, The Netherlands, September 10-14, 1990, Proceedings*, volume 478 of *LNCS*, pages 97–120. Springer, 1990. doi: 10.1007/BFb0018436.
- L. Prensa Nieto. The rely-guarantee method in isabelle/hol. In *ESOP 2003*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003. doi: 10.1007/3-540-36575-3_24.
- B. K. Rosen. Correctness of parallel programs: The church-rosser approach. *Theoretical Computer Science*, 2(2): 183–207, 1976. doi: 10.1016/0304-3975(76)90032-3.
- F. B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, October 1987.
- D. S. Scott. *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, chapter Lambda Calculus: Some Models, Some Philosophy, pages 223–265. Elsevier, 1980. doi: 10.1016/S0049-237X(08) 71262-X.
- A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994. doi: 10.1007/BF01211865.
- M. H. Stone. Topological representations of distributive lattices and Brouwerian logics. *Časopis pro pěstování matematiky a fyziky*, 67(1):1–25, 1938. doi: 10.21136/CPMF.1938.124080.
- V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK, 2008.
- D. van Dalen. *Logic and structure (4. ed.)*. Universitext. Springer, 2004. ISBN 978-3-540-57839-0.
- R. van Glabbeek and P. Höfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52(4):69:1–69:38, 2019. doi: 10.1145/3329125.
- S. van Staden. Constructing the views framework. In *UTP 2014*, volume 8963 of *LNCS*, pages 62–83. Springer, 2014. doi: 10.1007/978-3-319-14806-9_4.
- S. van Staden. On rely-guarantee reasoning. In *MPC 2015*, volume 9129 of *LNCS*, pages 30–49. Springer, 2015. doi: 10.1007/978-3-319-19797-5_2.
- S. Vickers. *Topology via Logic*. Cambridge University Press, 1989. ISBN 0521360625.
- J. S. Warford, D. Vega, and S. M. Staley. A calculational deductive system for linear temporal logic. *ACM Computing Surveys*, 53(3):53:1–53:38, 2020. doi: 10.1145/3387109.
- J. Wickerson, M. Dodds, and M. J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP 2010*, volume 6012 of *LNCS*, pages 610–629. Springer, 2010. doi: 10.1007/978-3-642-11957-6_32. URL <https://johnwickerson.github.io/expstab.thy.html>. Extended version in UCAM-CL-TR-774.
- Q. Xu and J. He. A theory of state-based parallel programming: Part 1. In Joseph M. Morris and Roger C. Shaw, editors, *4th Refinement Workshop*, pages 326–359. Springer, 1991.
- Q. Xu and J. He. Laws of parallel programming with shared variables. In David Till, editor, *6th Refinement Workshop, Proceedings of the 6th Refinement Workshop, organised by BCS-FACS, London, UK, 5-7 January 1994*, Workshops in Computing, pages 205–216. Springer, 1994. doi: 10.1007/978-1-4471-3240-0_11.

- Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR '94*, volume 836 of *LNCS*, pages 267–282. Springer, 1994. doi: 10.1007/978-3-540-48654-1_22.
- Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997. doi: 10.1007/BF01211617.
- Y. Zakowski, D. Cachera, D. Demange, G. Petri, D. Pichardie, S. Jagannathan, and J. Vitek. Verifying a concurrent garbage collector with a rely-guarantee methodology. *Journal of Automated Reasoning*, 63(2):489–515, 2019. doi: 10.1007/s10817-018-9489-x.
- J. Zwiers. *Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *LNCS*. Springer, 1989. ISBN 3-540-50845-7. doi: 10.1007/BFb0020836.