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

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 Lamport (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 Lamport's approach to separating liveness and safety properties (Abadi and Lamport 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 Lamport (§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 Lamport (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 $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \ldots (\rightarrow v)?$ 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 $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \ldots -a_n\rightarrow s_n \rightarrow v$ for $n \geq 0$ with optional return value v (see §8). States, agents and return values are of arbitrary type.

2.1 Traces

⟨*ML*⟩

datatype (*aset*: *'a*, *sset*: *'s*, *vset*: *'v*) *t* =
  T (*init*: *'s*) (*rest*: (*'a* × *'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 *trace.t* :: (*countable*, *countable*, *countable*) *countable* ⟨*proof*⟩

lemma *split-all*[*no-atp*]: — imitate the setup for *'a* × *'b* without the automation
  shows ($\bigwedge x$. *PROP P x*) ≡ ($\bigwedge s$ *xs v*. *PROP P* (*trace.T s xs v*))
⟨*proof*⟩

lemma *split-All*[*no-atp*]:
  shows (∀ *x*. *P x*) ⟷ (∀ *s xs v*. *P* (*trace.T s xs v*)) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *split-Ex*[*no-atp*]:
  shows (∃ *x*. *P x*) ⟷ (∃ *s xs v*. *P* (*trace.T s xs v*)) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

2.2 Combinators on traces

definition *final'* :: *'s* ⇒ (*'a* × *'s*) *list* ⇒ *'s* **where**
  final' s xs = *last* (*s* # *map snd xs*)

abbreviation (*input*) *final* :: (*'a*, *'s*, *'v*) *trace.t* ⇒ *'s* **where**
  final σ ≡ *trace.final'* (*trace.init* σ) (*trace.rest* σ)

definition *continue* :: (*'a*, *'s*, *'v*) *trace.t* ⇒ (*'a* × *'s*) *list* × *'v option* ⇒ (*'a*, *'s*, *'v*) *trace.t* (**infixl** ‹$@-_S$› *64*) **where**
  σ $@-_S$ *xsv* = (*case trace.term* σ *of None* ⇒ *trace.T* (*trace.init* σ) (*trace.rest* σ @ *fst xsv*) (*snd xsv*) | *Some v* ⇒ σ)

definition *tl* :: (*'a*, *'s*, *'v*) *trace.t* ⇀ (*'a*, *'s*, *'v*) *trace.t* **where**
  tl σ = (*case trace.rest* σ *of* [] ⇒ *None* | *x* # *xs* ⇒ *Some* (*trace.T* (*snd x*) *xs* (*trace.term* σ)))

definition *dropn* :: *nat* ⇒ (*'a*, *'s*, *'v*) *trace.t* ⇀ (*'a*, *'s*, *'v*) *trace.t* **where**
  dropn = ($\frown\frown$) *trace.tl*

definition *take* :: *nat* ⇒ (*'a*, *'s*, *'v*) *trace.t* ⇒ (*'a*, *'s*, *'v*) *trace.t* **where**
take i σ = (*if i* ≤ *length* (*trace.rest σ*) *then trace.T* (*trace.init σ*) (*List.take i* (*trace.rest σ*)) *None else σ*)

type-synonym (*'a*, *'s*) *transitions* = (*'a* × *'s* × *'s*) *list*

primrec *transitions'* :: *'s* ⇒ (*'a* × *'s*) *list* ⇒ (*'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* ⇒ (*'a*, *'s*) *trace.transitions* **where**
transitions σ ≡ *trace.transitions'* (*trace.init σ*) (*trace.rest σ*)

⟨*ML*⟩

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*: *snd* ‘ *set xs* ⊆ {*s*} ⟹ *trace.final' s xs* = *s*
and *snd* ‘ *set xs* ⊆ {*s*} ⟹ *trace.final' s* (*xs* @ *ys*) = *trace.final' s ys*
and *snd* ‘ *set ys* ⊆ {*trace.final' s xs*} ⟹ *trace.final' s* (*xs* @ *ys*) = *trace.final' s xs*
⟨*proof*⟩

lemma *map*:
shows *trace.final'* (*sf s*) (*map* (*map-prod af sf*) *xs*) = *sf* (*trace.final' s xs*)
⟨*proof*⟩

lemma *replicate*:
shows *trace.final' s* (*replicate i as*) = (*if i* = *0 then s else snd as*)
⟨*proof*⟩

lemma *map-idle*:
assumes (*λx. sf* (*snd x*)) ‘ *set xs* ⊆ {*sf s*}
shows *sf* (*trace.final' s xs*) = *sf s*
⟨*proof*⟩

⟨*ML*⟩

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*)
⟨*proof*⟩

⟨*ML*⟩

lemma *dropn-alt-def*:
shows *trace.dropn i σ*
= (*case drop i* ((*undefined*, *trace.init σ*) # *trace.rest σ*) *of*
[] ⇒ *None*
| *x* # *xs* ⇒ *Some* (*trace.T* (*snd x*) *xs* (*trace.term σ*)))
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*[*simp*]:
shows *0*: *trace.dropn 0* = *Some*
and *Suc*: *trace.dropn* (*Suc i*) *σ* = *Option.bind* (*trace.tl σ*) (*trace.dropn i*)
and *dropn*: *Option.bind* (*trace.dropn i σ*) (*trace.dropn j*) = *trace.dropn* (*i* + *j*) *σ*

⟨*proof*⟩

lemma *Suc-right*:
  shows $trace.dropn\ (Suc\ i)\ \sigma = Option.bind\ (trace.dropn\ i\ \sigma)\ trace.tl$
⟨*proof*⟩

lemma *eq-none-length-conv*:
  shows $trace.dropn\ i\ \sigma = None \longleftrightarrow length\ (trace.rest\ \sigma) < i$
⟨*proof*⟩

lemma *eq-Some-length-conv*:
  shows $(\exists \sigma'.\ trace.dropn\ i\ \sigma = Some\ \sigma') \longleftrightarrow i \leq length\ (trace.rest\ \sigma)$
⟨*proof*⟩

lemma *eq-Some-lengthD*:
  assumes $trace.dropn\ i\ \sigma = Some\ \sigma'$
  shows $i \leq length\ (trace.rest\ \sigma)$
⟨*proof*⟩

⟨*ML*⟩

lemma *sel*:
  shows $trace.init\ (trace.take\ i\ \sigma) = trace.init\ \sigma$
    and $trace.rest\ (trace.take\ i\ \sigma) = List.take\ i\ (trace.rest\ \sigma)$
    and $trace.term\ (trace.take\ i\ \sigma) = (if\ i \leq length\ (trace.rest\ \sigma)\ then\ None\ else\ trace.term\ \sigma)$
⟨*proof*⟩

lemma *0*:
  shows $trace.take\ 0\ \sigma = trace.T\ (trace.init\ \sigma)\ []\ None$
⟨*proof*⟩

lemma *Nil*:
  shows $trace.take\ i\ (trace.T\ s\ []\ None) = trace.T\ s\ []\ None$
⟨*proof*⟩

lemmas *simps*[*simp*] =
  trace.take.sel
  trace.take.0
  trace.take.Nil

lemma *map*:
  shows $trace.take\ i\ (trace.map\ af\ sf\ vf\ \sigma) = trace.map\ af\ sf\ vf\ (trace.take\ i\ \sigma)$
⟨*proof*⟩

lemma *append*:
  shows $trace.take\ i\ (trace.T\ s\ (xs\ @\ ys)\ v) = trace.T\ s\ (List.take\ i\ (xs\ @\ ys))\ (if\ length\ (xs\ @\ ys) < i\ then\ v\ else\ None)$
⟨*proof*⟩

lemma *take*:
  shows $trace.take\ i\ (trace.take\ j\ \sigma) = trace.take\ (min\ i\ j)\ \sigma$
⟨*proof*⟩

lemma *continue*:
  shows $trace.take\ i\ (\sigma\ @-_S\ xsv)$
    $= trace.take\ i\ \sigma\ @-_S\ (List.take\ (i - length\ (trace.rest\ \sigma))\ (fst\ xsv),$
      $if\ i \leq length\ (trace.rest\ \sigma) + length\ (fst\ xsv)\ then\ None\ else\ snd\ xsv)$
⟨*proof*⟩

lemma *all-iff*:
shows $trace.take\ i\ \sigma = \sigma \longleftrightarrow (case\ trace.term\ \sigma\ of\ None \Rightarrow length\ (trace.rest\ \sigma)\ |\ Some\ \text{-} \Rightarrow Suc\ (length\ (trace.rest\ \sigma))) \leq i$ (**is** *?thesis1*)
and $\sigma = trace.take\ i\ \sigma \longleftrightarrow (case\ trace.term\ \sigma\ of\ None \Rightarrow length\ (trace.rest\ \sigma)\ |\ Some\ \text{-} \Rightarrow Suc\ (length\ (trace.rest\ \sigma))) \leq i$ (**is** *?thesis2*)
⟨*proof*⟩

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

lemma *Ex-all*:
shows $\sigma = trace.take\ (Suc\ (length\ (trace.rest\ \sigma)))\ \sigma$
⟨*proof*⟩

lemma *replicate*:
shows $trace.take\ i\ (trace.T\ s\ (replicate\ j\ as)\ v)$
$= trace.T\ s\ (replicate\ (min\ i\ j)\ as)\ (if\ i \leq j\ then\ None\ else\ v)$
⟨*proof*⟩

⟨*ML*⟩

lemma *sel*[*simp*]:
shows $trace.init\ (\sigma\ @-_S\ xs) = trace.init\ \sigma$
and $trace.rest\ (\sigma\ @-_S\ xsv) = (case\ trace.term\ \sigma\ of\ None \Rightarrow trace.rest\ \sigma\ @\ fst\ xsv\ |\ Some\ v \Rightarrow trace.rest\ \sigma)$
and $trace.term\ (\sigma\ @-_S\ xsv) = (case\ trace.term\ \sigma\ of\ None \Rightarrow snd\ xsv\ |\ Some\ v \Rightarrow trace.term\ \sigma)$
⟨*proof*⟩

lemma *simps*[*simp*]:
shows $trace.T\ s\ xs\ None\ @-_S\ ysv = trace.T\ s\ (xs\ @\ fst\ ysv)\ (snd\ ysv)$
and $trace.T\ s\ xs\ (Some\ v)\ @-_S\ ysv = trace.T\ s\ xs\ (Some\ v)$
and $\sigma\ @-_S\ ([], None) = \sigma$
⟨*proof*⟩

lemma *Nil*:
shows $\sigma\ @-_S\ ([], trace.term\ \sigma) = \sigma$
and $trace.T\ (trace.init\ \sigma)\ []\ None\ @-_S\ (trace.rest\ \sigma, trace.term\ \sigma) = \sigma$
⟨*proof*⟩

lemma *map*:
shows $trace.map\ af\ sf\ vf\ (\sigma\ @-_S\ xsv) = trace.map\ af\ sf\ vf\ \sigma\ @-_S\ map\text{-}prod\ (map\ (map\text{-}prod\ af\ sf))\ (map\text{-}option\ vf)\ xsv$
⟨*proof*⟩

lemma *eq-trace-conv*:
shows $\sigma\ @-_S\ xsv = trace.T\ s\ xs\ v \longleftrightarrow trace.init\ \sigma = s \land (case\ trace.term\ \sigma\ of\ None \Rightarrow trace.rest\ \sigma\ @\ fst\ xsv = xs \land v = snd\ xsv\ |\ Some\ v' \Rightarrow trace.rest\ \sigma = xs \land v = Some\ v')$
and $trace.T\ s\ xs\ v = \sigma\ @-_S\ xsv \longleftrightarrow trace.init\ \sigma = s \land (case\ trace.term\ \sigma\ of\ None \Rightarrow trace.rest\ \sigma\ @\ fst\ xsv = xs \land v = snd\ xsv\ |\ Some\ v' \Rightarrow trace.rest\ \sigma = xs \land v = Some\ v')$
⟨*proof*⟩

lemma *self-conv*:
shows $(\sigma = \sigma\ @-_S\ xsv) \longleftrightarrow (case\ trace.term\ \sigma\ of\ None \Rightarrow xsv = ([], None)\ |\ Some\ \text{-} \Rightarrow True)$
and $(\sigma\ @-_S\ xsv = \sigma) \longleftrightarrow (case\ trace.term\ \sigma\ of\ None \Rightarrow xsv = ([], None)\ |\ Some\ \text{-} \Rightarrow True)$
⟨*proof*⟩

lemma *same-eq*:
shows $(\sigma\ @-_S\ xsv = \sigma\ @-_S\ ysv) \longleftrightarrow (case\ trace.term\ \sigma\ of\ None \Rightarrow xsv = ysv\ |\ Some\ \text{-} \Rightarrow True)$
⟨*proof*⟩

lemma *continue*:
  shows σ @$-_S$ *xsv* @$-_S$ *ysv* = σ @$-_S$ (*case snd xsv of None* $\Rightarrow$ (*fst xsv* @ *fst ysv*, *snd ysv*) | *Some* - $\Rightarrow$ *xsv*)
⟨*proof*⟩

lemma *take-drop-id*:
  shows *trace.take i* σ @$-_S$ *case-option* ([], *None*) ($\lambda\sigma'$. (*trace.rest* σ', *trace.term* σ')) (*trace.dropn i* σ) = σ
⟨*proof*⟩

⟨*ML*⟩

Prefix ordering **instantiation** *trace.t* :: (*type*, *type*, *type*) *order*
begin

definition *less-eq-t* :: ('*a*, '*s*, '*v*) *trace.t relp* **where**
  less-eq-t σ_1 σ_2 $\longleftrightarrow$ ($\exists$ *xsv*. σ_2 = σ_1 @$-_S$ *xsv*)

definition *less-t* :: ('*a*, '*s*, '*v*) *trace.t relp* **where**
  less-t σ_1 σ_2 $\longleftrightarrow$ $\sigma_1 \leq \sigma_2 \land \sigma_1 \neq \sigma_2$

instance
⟨*proof*⟩

end

lemma *less-eqE*[*consumes 1*, *case-names prefix maximal*]:
  assumes $\sigma_1 \leq \sigma_2$
  assumes ⟦*trace.term* σ_1 = *None*; *trace.init* σ_1 = *trace.init* σ_2; *prefix* (*trace.rest* σ_1) (*trace.rest* σ_2)⟧ $\Longrightarrow$ *P*
  assumes $\bigwedge$*v*. ⟦*trace.term* σ_1 = *Some v*; $\sigma_1 = \sigma_2$⟧ $\Longrightarrow$ *P*
  shows *P*
⟨*proof*⟩

lemmas *less-eq-extE*[*consumes 1*, *case-names prefix maximal*]
  = *trace.less-eqE*[*of trace.T* s_1 xs_1 v_1 *trace.T* s_2 xs_2 v_2, *simplified*, *simplified conj-explode*]
    for s_1 xs_1 v_1 s_2 xs_2 v_2

lemma *less-eq-self-continue*:
  shows $\sigma \leq \sigma$ @$-_S$ *xsv*
⟨*proof*⟩

lemma *less-eq-same-append-conv*:
  shows *trace.T s xs v* $\leq$ *trace.T s'* (*xs* @ *ys*) *v'* $\longleftrightarrow$ *s* = *s'* $\land$ ($\forall$ *v''*. *v* = *Some v''* $\longrightarrow$ *ys* = [] $\land$ *v* = *v'*)
⟨*proof*⟩

lemma *less-same-append-conv*:
  shows *trace.T s xs v* < *trace.T s'* (*xs* @ *ys*) *v'* $\longleftrightarrow$ *s* = *s'* $\land$ *v* = *None* $\land$ (*ys* $\neq$ [] $\lor$ ($\exists$ *v''*. *v'* = *Some v''*))
⟨*proof*⟩

lemma *less-eq-Some*[*simp*]:
  shows *trace.T s xs* (*Some v*) $\leq \sigma$ $\longleftrightarrow$ *trace.init* σ = *s* $\land$ *trace.rest* σ = *xs* $\land$ *trace.term* σ = *Some v*
⟨*proof*⟩

lemma *less-eq-None*:
  shows $\sigma \leq$ *trace.T s xs None* $\longleftrightarrow$ *trace.init* σ = *s* $\land$ *prefix* (*trace.rest* σ) *xs* $\land$ *trace.term* σ = *None*
    and *trace.T s xs None* $\leq \sigma$ $\longleftrightarrow$ *trace.init* σ = *s* $\land$ *prefix xs* (*trace.rest* σ)
⟨*proof*⟩

lemma *less*:

shows *trace.T s xs v* $<$ $\sigma \longleftrightarrow$ *trace.init* $\sigma = s \wedge (\exists\, ys.$ *trace.rest* $\sigma = xs$ @ $ys \wedge$ (*trace.term* $\sigma = None \longrightarrow ys \neq [])) \wedge v = None$
and $\sigma <$ *trace.T s xs v* $\longleftrightarrow$ *trace.init* $\sigma = s \wedge (\exists\, ys.\ xs =$ *trace.rest* σ @ $ys \wedge (v = None \longrightarrow ys \neq [])) \wedge$ *trace.term* $\sigma = None$
⟨*proof*⟩

lemma *less-eq-take*[*iff*]:
shows *trace.take i* $\sigma \leq \sigma$
⟨*proof*⟩

lemma *less-eq-takeE*:
assumes $\sigma_1 \leq \sigma_2$
obtains *i* **where** $\sigma_1 =$ *trace.take i* σ_2
⟨*proof*⟩

lemma *less-eq-take-def*:
shows $\sigma_1 \leq \sigma_2 \longleftrightarrow (\exists\, i.\ \sigma_1 =$ *trace.take i* $\sigma_2)$
⟨*proof*⟩

lemma *less-take-less-eq*:
assumes $\sigma <$ *trace.take* (*Suc i*) σ'
shows $\sigma \leq$ *trace.take i* σ'
⟨*proof*⟩

lemma *wfP-less*:
shows *wfP* ((<) :: (-, -, -) *trace.t relp*)
⟨*proof*⟩

lemma *less-eq-same-cases*:
fixes *ys* :: (-, -, -) *trace.t*
assumes $xs_1 \leq ys$
assumes $xs_2 \leq ys$
shows $xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
⟨*proof*⟩

⟨*ML*⟩

lemma *mono*:
assumes $\sigma_1 \leq \sigma_2$
assumes $i \leq j$
shows *trace.take i* $\sigma_1 \leq$ *trace.take j* σ_2
⟨*proof*⟩

⟨*ML*⟩

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

lemma *monotone*:
shows *mono* (*trace.map af sf vf*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *less-eqR*:
  assumes $\sigma_1 \leq$ *trace.map af sf vf* σ_2
  obtains $\sigma_2{}'$ **where** $\sigma_2{}' \leq \sigma_2$ **and** $\sigma_1 =$ *trace.map af sf vf* $\sigma_2{}'$
⟨*proof*⟩

⟨*ML*⟩

lemmas *eq* = *trace.t.rel-eq*

lemmas *mono* = *trace.t.rel-mono-strong*[*of ar sr vr* σ_1 σ_2 *ar′ sr′ vr′*] **for** *ar sr vr* σ_1 σ_2 *ar′ sr′ vr′*

lemma *strengthen*[*strg*]:
  assumes *st-ord F ar ar′*
  assumes *st-ord F sr sr′*
  assumes *st-ord F vr vr′*
  shows *st-ord F* (*trace.rel ar sr vr* σ_1 σ_2) (*trace.rel ar′ sr′ vr′* σ_1 σ_2)
⟨*proof*⟩

lemma *length-rest*:
  assumes *trace.rel ar sr vr* σ_1 σ_2
  shows *length* (*trace.rest* σ_1)
        = *length* (*trace.rest* σ_2) ∧ (∀ *i*<*length* (*trace.rest* σ_1). *rel-prod ar sr* (*trace.rest* σ_1 ! *i*) (*trace.rest* σ_2 ! *i*))
⟨*proof*⟩

⟨*ML*⟩

lemma *rel*:
  assumes *trace.rel ar sr vr* σ_1 σ_2
  shows *trace.rel ar sr vr* (*trace.take i* σ_1) (*trace.take i* σ_2)
⟨*proof*⟩

⟨*ML*⟩

lemma *prefix-conv*:
  shows *prefix* (*trace.transitions′ s xs*) (*trace.transitions′ s ys*) ⟷ *prefix xs ys*
⟨*proof*⟩

lemma *monotone*:
  shows *monotone prefix prefix* (*trace.transitions′ s*)
⟨*proof*⟩

lemma *append*:
  shows *trace.transitions′ s* (*xs* @ *ys*) = *trace.transitions′ s xs* @ *trace.transitions′* (*trace.final′ s xs*) *ys*
⟨*proof*⟩

lemma *eq-Nil-conv*:
  shows *trace.transitions′ s xs* = [] ⟷ *xs* = []
    and [] = *trace.transitions′ s xs* ⟷ *xs* = []
⟨*proof*⟩

lemma *eq-Cons-conv*:
  shows *trace.transitions′ s xs* = *y* # *ys* ⟷ (∃ *a s′ xs′*. *xs* = (*a*, *s′*) # *xs′* ∧ *y* = (*a*, *s*, *s′*) ∧ *ys* = *trace.transitions′ s′ xs′*)
    and *y* # *ys* = *trace.transitions′ s xs* ⟷ (∃ *a s′ xs′*. *xs* = (*a*, *s′*) # *xs′* ∧ *y* = (*a*, *s*, *s′*) ∧ *ys* = *trace.transitions′ s′ xs′*)
⟨*proof*⟩

lemma *inj-conv*:
shows $trace.transitions'\ s\ xs = trace.transitions'\ s\ ys \longleftrightarrow xs = ys$
⟨*proof*⟩

lemma *continue*:
shows $trace.transitions\ (\sigma\ @{-_S}\ xsv)$
$= trace.transitions\ \sigma\ @\ (case\ trace.term\ \sigma\ of\ None \Rightarrow trace.transitions'\ (trace.final\ \sigma)\ (fst\ xsv)\ |\ Some\ v \Rightarrow$
$[])$
⟨*proof*⟩

lemma *idle-conv*:
shows $set\ (trace.transitions'\ s\ xs) \subseteq UNIV \times Id \longleftrightarrow snd\ `\ set\ xs \subseteq \{s\}$
⟨*proof*⟩

lemma *map*:
shows $trace.transitions'\ (sf\ s)\ (map\ (map\text{-}prod\ af\ sf)\ xs)$
$= map\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf))\ (trace.transitions'\ s\ xs)$
⟨*proof*⟩

⟨*ML*⟩

lemma *monotone*:
shows $monotone\ (\leq)\ prefix\ trace.transitions$
⟨*proof*⟩

lemmas $mono = monotoneD[OF\ trace.transitions.monotone]$

lemma *subseq*:
assumes $\sigma \leq \sigma'$
shows $subseq\ (trace.transitions\ \sigma)\ (trace.transitions\ \sigma')$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

definition $steps' :: 's \Rightarrow ('a \times 's)\ list \Rightarrow ('a, 's)\ steps$ **where**
$steps'\ s\ xs = set\ (trace.transitions'\ s\ xs) - UNIV \times Id$

abbreviation $(input)\ steps :: ('a, 's, 'v)\ trace.t \Rightarrow ('a, 's)\ steps$ **where**
$steps\ \sigma \equiv trace.steps'\ (trace.init\ \sigma)\ (trace.rest\ \sigma)$

⟨*ML*⟩

lemma *simps*[*simp*]:
shows $trace.steps'\ s\ [] = \{\}$
and $trace.steps'\ s\ ((a, s)\ \#\ xs) = trace.steps'\ s\ xs$
and $s \neq snd\ x \Longrightarrow trace.steps'\ s\ (x\ \#\ xs) = insert\ (fst\ x, s, snd\ x)\ (trace.steps'\ (snd\ x)\ xs)$
and $(a, s', s') \notin trace.steps'\ s\ xs$
and $snd\ `\ set\ xs \subseteq \{s\} \Longrightarrow trace.steps'\ s\ xs = \{\}$
and $trace.steps'\ s\ [x] = (if\ s = snd\ x\ then\ \{\}\ else\ \{(fst\ x, s, snd\ x)\})$
⟨*proof*⟩

lemma *Cons-eq-if*:
shows $trace.steps'\ s\ (x\ \#\ xs)$
$= (if\ s = snd\ x\ then\ trace.steps'\ s\ xs\ else\ insert\ (fst\ x, s, snd\ x)\ (trace.steps'\ (snd\ x)\ xs))$

⟨*proof*⟩

lemma *stuttering*:
  shows $trace.steps'\ s\ xs \subseteq r \cup A \times Id \longleftrightarrow trace.steps'\ s\ xs \subseteq r$
    and $trace.steps'\ s\ xs \subseteq A \times Id \cup r \longleftrightarrow trace.steps'\ s\ xs \subseteq r$
⟨*proof*⟩

lemma *empty-conv*[*simp*]:
  shows $trace.steps'\ s\ xs = \{\} \longleftrightarrow snd\ `\ set\ xs \subseteq \{s\}$ (**is** *?thesis1*)
    and $\{\} = trace.steps'\ s\ xs \longleftrightarrow snd\ `\ set\ xs \subseteq \{s\}$ (**is** *?thesis2*)
⟨*proof*⟩

lemma *append*:
  shows $trace.steps'\ s\ (xs\ @\ ys)$
      $= trace.steps'\ s\ xs \cup trace.steps'\ (trace.final'\ s\ xs)\ ys$
⟨*proof*⟩

lemma *map*:
  shows $trace.steps'\ (sf\ s)\ (map\ (map\text{-}prod\ af\ sf)\ xs) = map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ `\ trace.steps'\ s\ xs - UNIV \times Id$
    and $trace.steps'\ s\ (map\ (map\text{-}prod\ af\ id)\ xs) = map\text{-}prod\ af\ id\ `\ trace.steps'\ s\ xs - UNIV \times Id$
⟨*proof*⟩

lemma *memberD*:
  assumes $(a, s, s') \in trace.steps'\ s_0\ xs$
  shows $(a, s') \in set\ xs$
⟨*proof*⟩

⟨*ML*⟩

lemma *monotone*:
  shows *mono trace.steps*
⟨*proof*⟩

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

⟨*ML*⟩

lemma *simps*:
  shows $trace.aset\ (trace.T\ s\ xs\ v) = fst\ `\ set\ xs$
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*:
  shows $trace.sset\ (trace.T\ s\ xs\ v) = insert\ s\ (snd\ `\ set\ xs)$
⟨*proof*⟩

lemma *dropn-le*:
  assumes $trace.dropn\ i\ \sigma = Some\ \sigma'$
  shows $trace.sset\ \sigma' \subseteq trace.sset\ \sigma$
⟨*proof*⟩

lemma *take-le*:
  shows $trace.sset\ (trace.take\ i\ \sigma) \subseteq trace.sset\ \sigma$
⟨*proof*⟩

lemma *mono*:
  shows *mono trace.sset*
⟨*proof*⟩

⟨*ML*⟩

2.3 Behaviors

⟨*ML*⟩

datatype (*aset*: *'a*, *sset*: *'s*, *vset*: *'v*) *t* =
  B (*init*: *'s*) (*rest*: (*'a* × *'s*, *'v*) *tllist*)
for
  map: *map*

definition *term* :: (*'a*, *'s*, *'v*) *behavior.t* ⇒ *'v option* **where**
  term ω = (*if tfinite* (*behavior.rest* ω) *then Some* (*terminal* (*behavior.rest* ω)) *else 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* × *'b* without the automation
  shows (⋀*x*. *PROP P x*) ≡ (⋀*s xs*. *PROP P* (*behavior.B s xs*))
⟨*proof*⟩

lemma *split-All*[*no-atp*]:
  shows (∀ *x*. *P x*) ⟷ (∀ *s xs*. *P* (*behavior.B s xs*)) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *split-Ex*[*no-atp*]:
  shows (∃ *x*. *P x*) ⟷ (∃ *s xs*. *P* (*behavior.B s xs*)) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

2.4 Combinators on behaviors

definition *continue* :: (*'a*, *'s*, *'v*) *trace.t* ⇒ (*'a* × *'s*, *'v*) *tllist* ⇒ (*'a*, *'s*, *'v*) *behavior.t* (**infix** ‹$@-_B$› *64*) **where**
  σ $@-_B$ *xs* = *behavior.B* (*trace.init* σ) (*tshift2* (*trace.rest* σ, *trace.term* σ) *xs*)

definition *tl* :: (*'a*, *'s*, *'v*) *behavior.t* ⇀ (*'a*, *'s*, *'v*) *behavior.t* **where**
  tl ω = (*case behavior.rest* ω *of TNil v* ⇒ *None* | *TCons x xs* ⇒ *Some* (*behavior.B* (*snd x*) *xs*))

definition *dropn* :: *nat* ⇒ (*'a*, *'s*, *'v*) *behavior.t* ⇀ (*'a*, *'s*, *'v*) *behavior.t* **where**
  dropn = ($\frown\frown$) *behavior.tl*

definition *take* :: *nat* ⇒ (*'a*, *'s*, *'v*) *behavior.t* ⇒ (*'a*, *'s*, *'v*) *trace.t* **where**
  take i ω = *uncurry* (*trace.T* (*behavior.init* ω)) (*ttake i* (*behavior.rest* ω))

⟨*ML*⟩

lemma *simps*:
  shows *trace.T s xs None* $@-_B$ *ys* = *behavior.B s* (*tshift xs ys*)
    and *trace.T s xs* (*Some v*) $@-_B$ *ys* = *behavior.B s* (*tshift xs* (*TNil v*))
    and *trace.T s* (*x* # *xs*) *w* $@-_B$ *ys* = *behavior.B s* (*TCons x* (*tshift2* (*xs*, *w*) *ys*))
⟨*proof*⟩

lemma *sel*[*simp*]:
  shows *init*: *behavior.init* $(\sigma$ @$-_B$ *xs*$)$ = *trace.init* σ
    and *rest*: *behavior.rest* $(\sigma$ @$-_B$ *xs*$)$ = *tshift2* (*trace.rest* σ, *trace.term* σ) *xs*
⟨*proof*⟩

lemma *term-None*:
  assumes *trace.term* σ = *None*
  shows σ @$-_B$ *xs* = *behavior.B* (*trace.init* σ) (*tshift* (*trace.rest* σ) *xs*)
⟨*proof*⟩

lemma *term-Some*:
  assumes *trace.term* σ = *Some v*
  shows σ @$-_B$ *xs* = *behavior.B* (*trace.init* σ) (*tshift* (*trace.rest* σ) (*TNil v*))
⟨*proof*⟩

lemma *tshift2*:
  shows σ @$-_B$ *tshift2 xsv ys* = (σ @$-_S$ *xsv*) @$-_B$ *ys*
⟨*proof*⟩

⟨*ML*⟩

lemma *TNil*:
  shows *behavior.tl* (*behavior.B s* (*TNil v*)) = *None*
⟨*proof*⟩

lemma *TCons*:
  shows *behavior.tl* (*behavior.B s* (*TCons x xs*)) = *Some* (*behavior.B* (*snd x*) *xs*)
⟨*proof*⟩

lemma *eq-None-conv*:
  shows *behavior.tl* ω = *None* $\longleftrightarrow$ *is-TNil* (*behavior.rest* ω)
⟨*proof*⟩

lemma *continue-Cons*:
  shows *behavior.tl* (*trace.T s* (*x* # *xs*) *v* @$-_B$ *ys*) = *Some* (*trace.T* (*snd x*) *xs v* @$-_B$ *ys*)
⟨*proof*⟩

lemmas *simps*[*simp*] =
  behavior.tl.TNil
  behavior.tl.TCons
  behavior.tl.eq-None-conv
  behavior.tl.continue-Cons

lemma *tfiniteD*:
  assumes *behavior.tl* ω = *Some* ω'
  shows *tfinite* (*behavior.rest* ω') $\longleftrightarrow$ *tfinite* (*behavior.rest* ω)
⟨*proof*⟩

⟨*ML*⟩

lemma *dropn-alt-def*:
  shows *behavior.dropn i* ω
      = (*case tdropn i* (*TCons* (*undefined*, *behavior.init* ω) (*behavior.rest* ω)) *of*
          TNil - $\Rightarrow$ *None*
        | *TCons x xs* $\Rightarrow$ *Some* (*behavior.B* (*snd x*) *xs*))
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*[*simp*]:
shows *0*: $behavior.dropn\ 0\ \omega = Some\ \omega$
and *TNil*: $behavior.dropn\ i\ (behavior.B\ s\ (TNil\ v)) = (case\ i\ of\ 0 \Rightarrow Some\ (behavior.B\ s\ (TNil\ v))\ |\ \text{-} \Rightarrow None)$
⟨*proof*⟩

lemma *TCons*:
shows $behavior.dropn\ i\ (behavior.B\ s\ (TCons\ x\ xs))$
$= (case\ i\ of\ 0 \Rightarrow Some\ (behavior.B\ s\ (TCons\ x\ xs))\ |\ Suc\ j \Rightarrow behavior.dropn\ j\ (behavior.B\ (snd\ x)\ xs))$
⟨*proof*⟩

lemma *Suc*:
shows $behavior.dropn\ (Suc\ i)\ \omega = Option.bind\ (behavior.tl\ \omega)\ (behavior.dropn\ i)$
⟨*proof*⟩

lemma *bind-tl-commute*:
shows $behavior.tl\ \omega \ggg behavior.dropn\ i = behavior.dropn\ i\ \omega \ggg behavior.tl$
⟨*proof*⟩

lemma *Suc-right*:
shows $behavior.dropn\ (Suc\ i)\ \omega = Option.bind\ (behavior.dropn\ i\ \omega)\ behavior.tl$
⟨*proof*⟩

lemma *dropn*:
shows $Option.bind\ (behavior.dropn\ i\ \omega)\ (behavior.dropn\ j) = behavior.dropn\ (i + j)\ \omega$
⟨*proof*⟩

lemma *add*:
shows $behavior.dropn\ (i + j) = (\lambda\omega.\ Option.bind\ (behavior.dropn\ i\ \omega)\ (behavior.dropn\ j))$
⟨*proof*⟩

lemma *tfiniteD*:
assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
shows $tfinite\ (behavior.rest\ \omega') \longleftrightarrow tfinite\ (behavior.rest\ \omega)$
⟨*proof*⟩

lemma *shorterD*:
assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
assumes $j \leq i$
shows $\exists\omega''.\ behavior.dropn\ j\ \omega = Some\ \omega''$
⟨*proof*⟩

lemma *eq-None-tlength-conv*:
shows $behavior.dropn\ i\ \omega = None \longleftrightarrow tlength\ (behavior.rest\ \omega) < enat\ i$
⟨*proof*⟩

lemma *eq-Some-tlength-conv*:
shows $(\exists\omega'.\ behavior.dropn\ i\ \omega = Some\ \omega') \longleftrightarrow enat\ i \leq tlength\ (behavior.rest\ \omega)$
⟨*proof*⟩

lemma *eq-Some-tlengthD*:
assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
shows $enat\ i \leq tlength\ (behavior.rest\ \omega)$
⟨*proof*⟩

lemma *tlength-eq-SomeD*:

assumes *enat i* $\leq$ *tlength* (*behavior.rest* ω)
shows $\exists\,\omega'$. *behavior.dropn i* ω = *Some* ω'
⟨*proof*⟩

lemma *eq-Some-tdropnD*:
assumes *behavior.dropn i* ω = *Some* ω'
shows *tdropn i* (*behavior.rest* ω) = *behavior.rest* ω'
⟨*proof*⟩

lemma *continue-shorter*:
assumes *i* $\leq$ *length* (*trace.rest* σ)
shows *behavior.dropn i* (σ @$-_B$ *xs*) = *Option.bind* (*trace.dropn i* σ) ($\lambda\sigma'$. *Some* (σ' @$-_B$ *xs*))
⟨*proof*⟩

lemma *continue-Some*:
assumes *length* (*trace.rest* σ) < *i*
assumes *trace.term* σ = *Some v*
shows *behavior.dropn i* (σ @$-_B$ *xs*) = *None*
⟨*proof*⟩

lemma *continue-None*:
assumes *length* (*trace.rest* σ) < *i*
assumes *trace.term* σ = *None*
shows *behavior.dropn i* (σ @$-_B$ *xs*)
= (*case tdropn* (*i* − *Suc* (*length* (*trace.rest* σ))) *xs of*
TNil - ⇒ *None*
| *TCons y ys* ⇒ *Some* (*behavior.B* (*snd y*) *ys*))
⟨*proof*⟩

lemma *continue*:
shows *behavior.dropn i* (σ @$-_B$ *xs*)
= (*if i* $\leq$ *length* (*trace.rest* σ)
then Option.bind (*trace.dropn i* σ) ($\lambda\sigma'$. *Some* (σ' @$-_B$ *xs*))
else if trace.term σ = *None*
then case tdropn (*i* − *Suc* (*length* (*trace.rest* σ))) *xs of*
TNil - ⇒ *None*
| *TCons y ys* ⇒ *Some* (*behavior.B* (*snd y*) *ys*)
else None)
⟨*proof*⟩

⟨*ML*⟩

lemma *take*:
shows *trace.take i* (*behavior.take j* ω) = *behavior.take* (*min i j*) ω
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*[*simp*]:
shows *0*: *behavior.take 0* ω = *trace.T* (*behavior.init* ω) [] *None*
and *Suc-TNil*: *behavior.take* (*Suc i*) (*behavior.B s* (*TNil v*)) = *trace.T s* [] (*Some v*)
⟨*proof*⟩

lemma *sel*[*simp*]:
shows *trace.init* (*behavior.take i* ω) = *behavior.init* ω
and *trace.rest* (*behavior.take i* ω) = *fst* (*ttake i* (*behavior.rest* ω))
and *trace.term* (*behavior.take i* ω) = *snd* (*ttake i* (*behavior.rest* ω))
⟨*proof*⟩

lemma *monotone*:
  shows *mono* ($\lambda i.$ *behavior.take* *i* ω)
⟨*proof*⟩

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

lemma *map*:
  shows *behavior.take i* (*behavior.map af sf vf* ω) = *trace.map af sf vf* (*behavior.take i* ω)
⟨*proof*⟩

lemma *continue*:
  shows *behavior.take i* (σ $@\!-_B$ ω) = *trace.take i* σ $@\!-_S$ *ttake* (*i* − *length* (*trace.rest* σ)) ω
⟨*proof*⟩

lemma *all-continue*:
  assumes *tlength* (*behavior.rest* ω) < *enat i*
  shows *behavior.take i* ω $@\!-_S$ *xsv* = *behavior.take i* ω
⟨*proof*⟩

lemma *continue-same*:
  shows *behavior.take i* (*behavior.take i* ω $@\!-_B$ *xsv*) = *behavior.take i* ω
⟨*proof*⟩

lemma *treplicate*:
  shows *behavior.take i* (*behavior.B s* (*treplicate j as v*))
    = *trace.T s* (*List.replicate* (*min i j*) *as*) (*if j* < *i then Some v else None*)
⟨*proof*⟩

lemma *trepeat*:
  shows *behavior.take i* (*behavior.B s* (*trepeat as*)) = *trace.T s* (*List.replicate i as*) *None*
⟨*proof*⟩

lemma *tshift*:
  shows *behavior.take i* (*behavior.B s* (*tshift xs ys*)) = *trace.take i* (*trace.T s xs None*) $@\!-_S$ *ttake* (*i* − *length xs*) *ys*
⟨*proof*⟩

lemma *length*:
  shows *length* (*trace.rest* (*behavior.take j* ω))
    = (*case tlength* (*behavior.rest* ω) *of enat i* ⇒ *min i j* | ∞ ⇒ *j*)
⟨*proof*⟩

lemma *add*:
  shows *behavior.take* (*i* + *j*) ω
    = *behavior.take i* ω $@\!-_S$ (*case behavior.dropn i* ω *of Some* ω' ⇒ *ttake j* (*behavior.rest* ω'))
⟨*proof*⟩

lemma *term-Some-conv*:
  shows *trace.term* (*behavior.take j* ω) = *Some v*
  ⟷ (*tlength* (*behavior.rest* ω) < *enat j* ∧ *Some v* = *behavior.term* ω)
⟨*proof*⟩

lemma *dropn*:
  assumes *behavior.dropn i* ω = *Some* ω'
  shows *behavior.take j* ω' = *the* (*trace.dropn i* (*behavior.take* (*i* + *j*) ω))
⟨*proof*⟩

lemma *continue-id*:
  assumes $tlength\ (behavior.rest\ \omega) < enat\ i$
  shows $behavior.take\ i\ \omega\ @-_B\ xs = \omega$
⟨*proof*⟩

lemma *flat*:
  assumes $tlength\ (behavior.rest\ \omega) < enat\ i$
  assumes $i \leq j$
  shows $behavior.take\ i\ \omega = behavior.take\ j\ \omega$
⟨*proof*⟩

lemma *eqI*:
  assumes $\bigwedge i.\ behavior.take\ i\ \omega_1 = behavior.take\ i\ \omega_2$
  shows $\omega_1 = \omega_2$
⟨*proof*⟩

⟨*ML*⟩

lemma *take-drop-shorter*:
  assumes $i \leq j$
  shows $behavior.take\ i\ \omega\ @-_S\ apfst\ (drop\ i)\ (ttake\ j\ (behavior.rest\ \omega)) = behavior.take\ j\ \omega$
⟨*proof*⟩

lemma *take-drop-id*:
  shows $behavior.take\ i\ \omega\ @-_B\ behavior.rest\ (the\ (behavior.dropn\ i\ \omega)) = \omega$
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*:
  shows $behavior.aset\ (behavior.B\ s\ xs) = fst\ `\ tset\ xs$
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*:
  shows $behavior.sset\ (behavior.B\ s\ xs) = insert\ s\ (snd\ `\ tset\ xs)$
⟨*proof*⟩

lemma *dropn-le*:
  assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
  shows $behavior.sset\ \omega' \subseteq behavior.sset\ \omega$
⟨*proof*⟩

lemma *take-le*:
  shows $trace.sset\ (behavior.take\ i\ \omega) \subseteq behavior.sset\ \omega$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

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 $'s\ pred = 's \Rightarrow bool$

abbreviation (*input*)
pred-K :: $'b \Rightarrow 'a \Rightarrow 'b$ (‹⟨-⟩›) **where**
$\langle f \rangle \equiv \lambda s.\ f$

abbreviation (*input*)
pred-not :: $'a\ pred \Rightarrow 'a\ pred$ (‹¬ -› [*40*] *40*) **where**
$\neg a \equiv \lambda s.\ \neg a\ s$

abbreviation (*input*)
pred-conj :: $'a\ pred \Rightarrow 'a\ pred \Rightarrow 'a\ pred$ (**infixr** ‹∧› *35*) **where**
$a \wedge b \equiv \lambda s.\ a\ s \wedge b\ s$

abbreviation (*input*)
pred-disj :: $'a\ pred \Rightarrow 'a\ pred \Rightarrow 'a\ pred$ (**infixr** ‹∨› *30*) **where**
$a \vee b \equiv \lambda s.\ a\ s \vee b\ s$

abbreviation (*input*)
pred-implies :: $'a\ pred \Rightarrow 'a\ pred \Rightarrow 'a\ pred$ (**infixr** ‹⟶› *25*) **where**
$a \longrightarrow b \equiv \lambda s.\ a\ s \longrightarrow b\ s$

abbreviation (*input*)
pred-iff :: $'a\ pred \Rightarrow 'a\ pred \Rightarrow 'a\ pred$ (**infixr** ‹⟷› *25*) **where**
$a \longleftrightarrow b \equiv \lambda s.\ a\ s \longleftrightarrow b\ s$

abbreviation (*input*)
pred-eq :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹=› *40*) **where**
$a = b \equiv \lambda s.\ a\ s = b\ s$

abbreviation (*input*)
pred-neq :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹≠› *40*) **where**
$a \neq b \equiv \lambda s.\ a\ s \neq b\ s$

abbreviation (*input*)
pred-If :: $'a\ pred \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (‹(*If* (-)/ *Then* (-)/ *Else* (-))› [*0*, *0*, *10*] *10*)
where *If P Then x Else y* $\equiv \lambda s.$ *if P s then x s else y s*

abbreviation (*input*)
pred-less :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹<› *40*) **where**
$a < b \equiv \lambda s.\ a\ s < b\ s$

abbreviation (*input*)
pred-less-eq :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹≤› *40*) **where**
$a \leq b \equiv \lambda s.\ a\ s \leq b\ s$

abbreviation (*input*)
pred-greater :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹>› *40*) **where**
$a > b \equiv \lambda s.\ a\ s > b\ s$

abbreviation (*input*)
pred-greater-eq :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹≥› *40*) **where**
$a \geq b \equiv \lambda s.\ a\ s \geq b\ s$

abbreviation (*input*)
$\mathit{pred\text{-}plus} :: ('a \Rightarrow 'b{::}plus) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** ‹+› *65*) **where**
$a + b \equiv \lambda s.\ a\ s + b\ s$

abbreviation (*input*)
$\mathit{pred\text{-}minus} :: ('a \Rightarrow 'b{::}minus) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** ‹−› *65*) **where**
$a - b \equiv \lambda s.\ a\ s - b\ s$

abbreviation (*input*)
$\mathit{pred\text{-}times} :: ('a \Rightarrow 'b{::}times) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** ‹∗› *65*) **where**
$a * b \equiv \lambda s.\ a\ s * b\ s$

abbreviation (*input*)
$\mathit{pred\text{-}all} :: ('b \Rightarrow 'a\ pred) \Rightarrow 'a\ pred$ (**binder** ‹∀› *10*) **where**
$\forall x.\ P\ x \equiv \lambda s.\ \forall x.\ P\ x\ s$

abbreviation (*input*)
$\mathit{pred\text{-}ex} :: ('b \Rightarrow 'a\ pred) \Rightarrow 'a\ pred$ (**binder** ‹∃› *10*) **where**
$\exists x.\ P\ x \equiv \lambda s.\ \exists x.\ P\ x\ s$

abbreviation (*input*)
$\mathit{pred\text{-}app} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ‹$› *100*) **where**
$f\ \mathbf{\$}\ g \equiv \lambda s.\ f\ s\ (g\ s)$

abbreviation (*input*)
$\mathit{pred\text{-}app'} :: ('b \Rightarrow 'a \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ‹$$› *100*) **where**
$f\ \mathbf{\$\$}\ g \equiv \lambda s.\ f\ (g\ s)\ s$

abbreviation (*input*)
$\mathit{pred\text{-}member} :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a\ pred$ (**infix** ‹∈› *40*) **where**
$a \in b \equiv \lambda s.\ a\ s \in b\ s$

abbreviation (*input*)
$\mathit{pred\text{-}subseteq} :: ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a\ pred$ (**infix** ‹⊆› *50*) **where**
$A \subseteq B \equiv \lambda s.\ A\ s \subseteq B\ s$

abbreviation (*input*)
$\mathit{pred\text{-}union} :: ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow 'b\ set$ (**infixl** ‹∪› *65*) **where**
$a \cup b \equiv \lambda s.\ a\ s \cup b\ s$

abbreviation (*input*)
$\mathit{pred\text{-}inter} :: ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow 'b\ set$ (**infixl** ‹∩› *65*) **where**
$a \cap b \equiv \lambda s.\ a\ s \cap b\ s$

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

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

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

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

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

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

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

abbreviation (*input*)
  SIZE :: ($'a \Rightarrow 'b$::*size*) $\Rightarrow$ $'a \Rightarrow$ *nat* **where**
  SIZE a $\equiv \lambda s.$ *size* (*a s*)

abbreviation (*input*)
  SET :: ($'a \Rightarrow 'b$ *list*) $\Rightarrow$ $'a \Rightarrow 'b$ *set* **where**
  SET a $\equiv \lambda s.$ *set* (*a s*)

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

abbreviation (*input*)
  pred-list-nth :: ($'a \Rightarrow 'b$ *list*) $\Rightarrow$ ($'a \Rightarrow$ *nat*) $\Rightarrow$ $'a \Rightarrow 'b$ (**infixl** ‹**!**› *150*) **where**
  xs **!** *i* $\equiv \lambda s.$ *xs s* ! *i s*

abbreviation (*input*)
  pred-list-append :: ($'a \Rightarrow 'b$ *list*) $\Rightarrow$ ($'a \Rightarrow 'b$ *list*) $\Rightarrow$ $'a \Rightarrow 'b$ *list* (**infixr** ‹**@**› *65*) **where**
  xs **@** *ys* $\equiv \lambda s.$ *xs s* @ *ys s*

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

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

abbreviation (*input*)
  pred-pair :: ($'a \Rightarrow 'b$) $\Rightarrow$ ($'a \Rightarrow 'c$) $\Rightarrow$ $'a \Rightarrow 'b \times 'c$ (**infixr** ‹$\otimes$› *60*) **where**
  $a \otimes b \equiv \lambda s.$ (*a s*, *b s*)

4 More lattice

lemma (**in** *semilattice-sup*) *sup-iff-le*:
  shows $x \sqcup y = y \longleftrightarrow x \leq y$
    and $y \sqcup x = y \longleftrightarrow x \leq y$
⟨*proof*⟩

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

shows $x \sqcap y = x \longleftrightarrow x \leq y$
and $y \sqcap x = x \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *if-sup-distr*:
fixes t e :: -::*semilattice-sup*
shows *if-sup-distrL*: (*if* b *then* $t_1 \sqcup t_2$ *else* e) = (*if* b *then* t_1 *else* e) $\sqcup$ (*if* b *then* t_2 *else* e)
and *if-sup-distrR*: (*if* b *then* t *else* $e_1 \sqcup e_2$) = (*if* b *then* t *else* e_1) $\sqcup$ (*if* b *then* t *else* e_2)
⟨*proof*⟩

lemma *INF-bot*:
assumes $F\ i = (\bot$::-::*complete-lattice*)
assumes $i \in X$
shows $(\bigsqcap i{\in}X.\ F\ i) = \bot$
⟨*proof*⟩

lemma *mcont-fun-app-const*[*cont-intro*]:
shows *mcont Sup* (≤) *Sup* (≤) (λ*f*. *f c*)
⟨*proof*⟩

declare *mcont-applyI*[*cont-intro*]

lemma *INF-rename-bij*:
assumes *bij-betw* π X Y
shows $(\bigsqcap y{\in}Y.\ F\ Y\ y) = (\bigsqcap x{\in}X.\ F\ (\pi\ `\ X)\ (\pi\ x))$
⟨*proof*⟩

lemma *Inf-rename-surj*:
assumes *surj* π
shows $(\bigsqcap x.\ F\ x) = (\bigsqcap x.\ F\ (\pi\ x))$
⟨*proof*⟩

lemma *INF-unwind-index*:
fixes A :: -⇒-:: *complete-lattice*
assumes $i \in I$
shows $(\bigsqcap x{\in}I.\ A\ x) = A\ i \sqcap (\bigsqcap x{\in}I - \{i\}.\ A\ x)$
⟨*proof*⟩

lemma *Sup-fst*:
shows $(\bigsqcup x{\in}X.\ P\ (fst\ x)) = (\bigsqcup x{\in}fst\ `\ X.\ P\ x)$
⟨*proof*⟩

⟨*ML*⟩

lemma *assms-cong*: — simplify assumptions only
assumes $x = x'$
shows $x \leq y \longleftrightarrow x' \leq y$
⟨*proof*⟩

lemma *concl-cong*: — simplify conclusions only
assumes $y = y'$
shows $x \leq y \longleftrightarrow x \leq y'$
⟨*proof*⟩

lemma *subgoal*: — cut for lattice logics
fixes P :: -::*semilattice-inf*
assumes $P \leq Q$
assumes $P \sqcap Q \leq R$

shows $P \leq R$
⟨*proof*⟩

⟨*ML*⟩

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

lemmas *SUPE* = *SUP-least*[**where** *u*=*z* **for** *z*]
lemmas *SupE* = *Sup-least*

lemmas *INFI* = *INF-greatest*
lemmas *InfI* = *Inf-greatest*
lemmas *infI* = *semilattice-inf-class.le-infI*

lemma *InfE*:
  fixes *R*::-::*complete-lattice*
  assumes $P\ x \leq R$
  shows $(\bigsqcap x.\ P\ x) \leq R$
⟨*proof*⟩

lemma *INFE*:
  fixes *R*::′*a*::*complete-lattice*
  assumes $P\ x \leq R$
  assumes $x \in A$
  shows $\bigsqcap (P\ `\ A) \leq R$
⟨*proof*⟩

lemmas *rev-INFE* = *INFE*[*rotated*]

lemma *Inf-inf-distrib*:
  fixes *P*::-::*complete-lattice*
  shows $(\bigsqcap x.\ P\ x \sqcap Q\ x) = (\bigsqcap x.\ P\ x) \sqcap (\bigsqcap x.\ Q\ x)$
⟨*proof*⟩

lemma *Sup-sup-distrib*:
  fixes *P*::-::*complete-lattice*
  shows $(\bigsqcup x.\ P\ x \sqcup Q\ x) = (\bigsqcup x.\ P\ x) \sqcup (\bigsqcup x.\ Q\ x)$
⟨*proof*⟩

lemma *Inf-inf*:
  fixes *Q* :: -::*complete-lattice*
  shows $(\bigsqcap x.\ P\ x \sqcap Q) = (\bigsqcap x.\ P\ x) \sqcap Q$
⟨*proof*⟩

lemma *inf-Inf*:
  fixes *P* :: -::*complete-lattice*
  shows $(\bigsqcap x.\ P \sqcap Q\ x) = P \sqcap (\bigsqcap x.\ Q\ x)$
⟨*proof*⟩

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

lemma *sup-SUP*:

fixes P :: -::*complete-lattice*
assumes $X \neq \{\}$
shows $(\bigsqcup x{\in}X.\ P \sqcup Q\ x) = P \sqcup (\bigsqcup x{\in}X.\ Q\ x)$
⟨*proof*⟩

4.1 Boolean lattices and implication

lemma
shows *minus-Not*[*simp*]: $-\ Not = id$
and *minus-id*[*simp*]: $-\ id = Not$
⟨*proof*⟩

definition *boolean-implication* :: $'a$::*boolean-algebra* $\Rightarrow\ 'a \Rightarrow\ 'a$ (**infixr** ‹$\longrightarrow_B$› *60*) **where**
$x \longrightarrow_B y = -x \sqcup y$

definition *boolean-eq* :: $'a$::*boolean-algebra* $\Rightarrow\ 'a \Rightarrow\ 'a$ (**infixr** ‹$\longleftrightarrow_B$› *60*) **where**
$x \longleftrightarrow_B y = x \longrightarrow_B y \sqcap y \longrightarrow_B x$

⟨*ML*⟩

lemma *bool-alt-def*[*simp*]:
shows $P \longrightarrow_B Q = (P \longrightarrow Q)$
⟨*proof*⟩

lemma *pred--alt-def*[*simp*]:
shows $(P \longrightarrow_B Q)\ x = (P\ x \longrightarrow_B Q\ x)$
⟨*proof*⟩

lemma *set-alt-def*:
shows $P \longrightarrow_B Q = \{x.\ x \in P \longrightarrow x \in Q\}$
⟨*proof*⟩

lemma *member*:
shows $x \in P \longrightarrow_B Q \longleftrightarrow x \in P \longrightarrow x \in Q$
⟨*proof*⟩

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

lemma *simps*[*simp*]:
shows
$\top \longrightarrow_B P = P$
$\bot \longrightarrow_B P = \top$
$P \longrightarrow_B \top = \top$
$P \longrightarrow_B P = \top$
$P \longrightarrow_B \bot = -P$
$P \longrightarrow_B -P = -P$
⟨*proof*⟩

lemma *Inf-simps*[*simp*]: — Miniscoping: pushing in universal quantifiers.
shows
$\bigwedge P$ (Q::-::*complete-boolean-algebra*). $(\bigsqcap x.\ P\ x \longrightarrow_B Q) = ((\bigsqcup x.\ P\ x) \longrightarrow_B Q)$
$\bigwedge P$ (Q::-::*complete-boolean-algebra*). $(\bigsqcap x{\in}X.\ P\ x \longrightarrow_B Q) = ((\bigsqcup x{\in}X.\ P\ x) \longrightarrow_B Q)$
$\bigwedge P$ (Q::-$\Rightarrow$-::*complete-boolean-algebra*). $(\bigsqcap x.\ P \longrightarrow_B Q\ x) = (P \longrightarrow_B (\bigsqcap x.\ Q\ x))$
$\bigwedge P$ (Q::-$\Rightarrow$-::*complete-boolean-algebra*). $(\bigsqcap x{\in}X.\ P \longrightarrow_B Q\ x) = (P \longrightarrow_B (\bigsqcap x{\in}X.\ Q\ x))$
⟨*proof*⟩

lemma *mono*:
assumes $x' \leq x$

assumes $y \leq y'$
shows $x \longrightarrow_B y \leq x' \longrightarrow_B y'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F)$ X X'
assumes *st-ord* F Y Y'
shows *st-ord* F $(X \longrightarrow_B Y)$ $(X' \longrightarrow_B Y')$
⟨*proof*⟩

lemma *eq-conv*:
shows $(P = Q) \longleftrightarrow (P \longrightarrow_B Q) \sqcap (Q \longrightarrow_B P) = \top$
⟨*proof*⟩

lemma *uminus-imp*[*simp*]:
shows $-(P \longrightarrow_B Q) = P \sqcap -Q$
⟨*proof*⟩

lemma *cases-simp*[*simp*]:
shows $(P \longrightarrow_B Q) \sqcap (-P \longrightarrow_B Q) = Q$
⟨*proof*⟩

lemma *conv-sup*:
shows $(P \longrightarrow_B Q) = -P \sqcup Q$
⟨*proof*⟩

lemma *infL*:
shows $P \sqcap Q \longrightarrow_B R = P \longrightarrow_B Q \longrightarrow_B R$
⟨*proof*⟩

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

lemma *shunt1*:
shows $x \sqcap y \leq z \longleftrightarrow x \leq y \longrightarrow_B z$
⟨*proof*⟩

lemma *shunt2*:
shows $x \sqcap y \leq z \longleftrightarrow y \leq x \longrightarrow_B z$
⟨*proof*⟩

lemma *mp*:
assumes $x \sqcap y \leq z$
shows $x \leq y \longrightarrow_B z$
⟨*proof*⟩

lemma *imp-trivialI*:
assumes $P \sqcap -R \leq -Q$
shows $P \leq Q \longrightarrow_B R$
⟨*proof*⟩

lemma *shunt-top*:
shows $P \longrightarrow_B Q = \top \longleftrightarrow P \leq Q$
⟨*proof*⟩

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*⟩

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*⟩

lemma *trans*:
  shows $(x \longrightarrow_B y) \sqcap (y \longrightarrow_B z) \leq (x \longrightarrow_B z)$
⟨*proof*⟩

⟨*ML*⟩

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

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 $\bot \in compact\text{-}points$
⟨*proof*⟩

lemma *compact-points-sup*: — Davey and Priestley (2002, Lemma 7.16)
  assumes $x \in compact\text{-}points$
  assumes $y \in compact\text{-}points$
  shows $x \sqcup y \in compact\text{-}points$
⟨*proof*⟩

lemma *compact-points-Sup*: — Davey and Priestley (2002, Lemma 7.16)
  assumes $X \subseteq compact\text{-}points$
  assumes *finite* X
  shows $\bigsqcup X \in compact\text{-}points$
⟨*proof*⟩

lemma *compact-points-are-ccpo-compact*: — converse should hold
  assumes $x \in compact\text{-}points$
  shows *ccpo.compact Sup* $(\leq)$ x
⟨*proof*⟩

definition *directed* :: *'a set* ⇒ *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 $X \neq \{\}$
⟨*proof*⟩

lemma *chain-directed*:

assumes *Complete-Partial-Order.chain* $(\leq)$ *Y*
assumes $Y \neq \{\}$
shows *directed Y*
⟨*proof*⟩

lemma *directed-alt-def*:
shows *directed X* $\longleftrightarrow$ $(\forall\, Y{\subseteq}X.\ \mathit{finite}\ Y \longrightarrow (\exists\, x{\in}X.\ \forall\, y{\in}Y.\ y \leq x))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *compact-points-alt-def*: — Davey and Priestley (2002, Definition 7.15(i) (finite points))
shows *compact-points* $= \{x{::}'a.\ \forall\, D.\ \mathit{directed}\ D \wedge x \leq \bigsqcup D \longrightarrow (\exists\, d{\in}D.\ x \leq d)\}$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemmas *compact-points-directedD*
= *subsetD*[*OF equalityD1*[*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*: $(x :: 'a) = \bigsqcup(\{Y.\ Y \leq x\} \cap \mathit{compact\text{-}points})$
begin

lemma *le-compact*:
shows $x \leq y \longleftrightarrow (\forall\, z{\in}\mathit{compact\text{-}points}.\ z \leq x \longrightarrow z \leq y)$
⟨*proof*⟩

end

lemma (**in** *ccpo*) *compact-alt-def*:
shows *ccpo.compact Sup* $(\leq)$ $x \longleftrightarrow (\forall\, Y.\ Y \neq \{\} \wedge \mathit{Complete\text{-}Partial\text{-}Order.chain}\ (\leq)\ Y \wedge x \leq \mathit{Sup}\ Y \longrightarrow (\exists\, y{\in}Y.\ x \leq y))$
⟨*proof*⟩

lemma *compact-points-eq-finite-sets*: — Davey and Priestley (2002, Examples 7.17)
shows *compact-points* = *Collect finite* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

instance *set* :: (*type*) *algebraic-lattice*
⟨*proof*⟩

context *semilattice-sup*
begin

definition *sup-irreducible-on* :: $'a\ \mathit{set} \Rightarrow\ 'a \Rightarrow \mathit{bool}$ **where** — Davey and Priestley (2002, Definition 2.42)
sup-irreducible-on A x $\longleftrightarrow (\forall\, y{\in}A.\ \forall\, z{\in}A.\ x = y \sqcup z \longrightarrow x = y \vee x = z)$

abbreviation *sup-irreducible* :: $'a \Rightarrow \mathit{bool}$ **where**
sup-irreducible ≡ *sup-irreducible-on UNIV*

lemma *sup-irreducible-onI*:
assumes $\bigwedge y\ z.\ [\![y \in A;\ z \in A;\ x = y \sqcup z]\!] \Longrightarrow x = y \vee x = z$
shows *sup-irreducible-on A x*
⟨*proof*⟩

lemma *sup-irreducible-onD*:
assumes *sup-irreducible-on A x*
assumes $x = y \sqcup z$

assumes $y \in A$
assumes $z \in A$
shows $x = y \vee x = z$
⟨*proof*⟩

lemma *sup-irreducible-on-less*: — Davey and Priestley (2002, Definition 2.42 (alt))
shows *sup-irreducible-on* $A\ x \longleftrightarrow (\forall y{\in}A.\ \forall z{\in}A.\ y < x \wedge z < x \longrightarrow y \sqcup z < x)$
⟨*proof*⟩

end

lemma *sup-irreducible-bot*:
assumes $\bot \in A$
shows *sup-irreducible-on* A ($\bot$::-::*bounded-semilattice-sup-bot*)
⟨*proof*⟩

lemma *sup-irreducible-le-conv*:
fixes x::-::*distrib-lattice*
assumes *sup-irreducible* x
shows $x \le y \sqcup z \longleftrightarrow x \le y \vee x \le z$
⟨*proof*⟩

lemma *set-sup-irreducible*:
shows *sup-irreducible* $X \longleftrightarrow (X = \{\} \vee (\exists y.\ X = \{y\}))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

definition *Sup-irreducible-on* :: *'a::complete-lattice set* ⇒ *'a* ⇒ *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::complete-lattice* ⇒ *bool* **where**
Sup-irreducible ≡ *Sup-irreducible-on UNIV*

definition *Sup-prime-on* :: *'a::complete-lattice set* ⇒ *'a* ⇒ *bool* **where** — Davey and Priestley (2002, Definition 10.26)
Sup-prime-on $A\ x \longleftrightarrow (\forall S{\subseteq}A.\ x \le \bigsqcup S \longrightarrow (\exists s{\in}S.\ x \le s))$

abbreviation *Sup-prime* :: *'a::complete-lattice* ⇒ *bool* **where**
Sup-prime ≡ *Sup-prime-on UNIV*

lemma *Sup-irreducible-onI*:
assumes $\bigwedge S.\ [\![S \subseteq A;\ x = \bigsqcup S]\!] \Longrightarrow x \in S$
shows *Sup-irreducible-on* $A\ x$
⟨*proof*⟩

lemma *Sup-irreducible-onD*:
assumes $x = \bigsqcup S$
assumes $S \subseteq A$
assumes *Sup-irreducible-on* $A\ x$
shows $x \in S$
⟨*proof*⟩

lemma *Sup-prime-onI*:
assumes $\bigwedge S.\ [\![S \subseteq A;\ x \le \bigsqcup S]\!] \Longrightarrow \exists s{\in}S.\ x \le s$
shows *Sup-prime-on* $A\ x$
⟨*proof*⟩

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** $x \leq s$
⟨*proof*⟩

lemma *Sup-prime-on-conv*:
assumes *Sup-prime-on A x*
assumes $S \subseteq A$
shows $x \leq \bigsqcup S \longleftrightarrow (\exists s \in S.\ x \leq s)$
⟨*proof*⟩

lemma *Sup-prime-not-bot*:
assumes *Sup-prime-on A x*
shows $x \neq \bot$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Sup-irreducible-on-imp-sup-irreducible-on*:
assumes *Sup-irreducible-on A x*
assumes $x \in A$
shows *sup-irreducible-on A x*
⟨*proof*⟩

lemma *Sup-prime-is-compact*:
assumes *Sup-prime x*
shows $x \in$ *compact-points*
⟨*proof*⟩

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 requre strictness (i.e., it need not be the case that the closure maps ⊥ to ⊥). Davey and Priestley (2002, §2.33) term these *topped intersection structures*; see also Pfaltz and Šlapal (2013) for additional useful results.

locale *closure* =
ordering (≤) (<) — We use a partial order as a preorder does not ensure that the closure is idempotent
for *less-eq* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹≤› *50*)
and *less* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹<› *50*)
\+ **fixes** *cl* :: *'a* ⇒ *'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*
⟨*proof*⟩

lemma *expansive*:
shows $x \leq cl\ x$

⟨*proof*⟩

lemma *idempotent*[*simp*]:
  shows *cl* (*cl* *x*) = *cl* *x*
    and *cl* ∘ *cl* = *cl*
⟨*proof*⟩

lemma *monotone-cl*:
  shows *monotone* (≤) (≤) *cl*
⟨*proof*⟩

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* $x \leq y$
⟨*proof*⟩

lemma *least-conv*:
  assumes $y \in$ *closed*
  shows *cl* $x \leq y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *closed*[*iff*]:
  shows *cl* $x \in$ *closed*
⟨*proof*⟩

lemma *le-closedE*:
  assumes $x \leq$ *cl* y
  assumes $y \in$ *closed*
  shows $x \leq y$
⟨*proof*⟩

lemma *closed-conv*: — Typically used to manifest the closure using *subst*
  assumes $X \in$ *closed*
  shows $X =$ *cl* X
⟨*proof*⟩

end

lemma (**in** *ordering*) *closure-axioms-alt-def*: — Equivalence with the Kuratowski closure axioms
  shows *closure-axioms* (≤) *cl* ⟷ (∀ *x*. *x* ≤ *cl* *x*) ∧ *monotone* (≤) (≤) *cl* ∧ (∀ *x*. *cl* (*cl* *x*) = *cl* *x*)
⟨*proof*⟩

lemma (**in** *ordering*) *closureI*:
  assumes ⋀*x*. *x* ≤ *cl* *x*
  assumes *monotone* (≤) (≤) *cl*
  assumes ⋀*x*. *cl* (*cl* *x*) = *cl* *x*
  shows *closure* (≤) (<) *cl*
⟨*proof*⟩

lemma *closure-inf-closure*:
  fixes cl_1 :: *'a*::*semilattice-inf* ⇒ *'a*
  assumes *closure-axioms* (≤) cl_1
  assumes *closure-axioms* (≤) cl_2
  shows *closure-axioms* (≤) ($\lambda X.\ cl_1\ X \sqcap cl_2\ X$)

⟨*proof*⟩

5.1 Complete lattices and algebraic closures

locale *closure-complete-lattice* =
complete-lattice $\bigsqcap$ $\bigsqcup$ ($\sqcap$) ($\leq$) ($<$) ($\sqcup$) $\bot$ $\top$
+ *closure* ($\leq$) ($<$) *cl*
for *less-eqa* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ‹$\leq$› *50*)
and *lessa* (**infix** ‹$<$› *50*)
and *infa* (**infixl** ‹$\sqcap$› *70*)
and *supa* (**infixl** ‹$\sqcup$› *65*)
and *bota* (‹$\bot$›)
and *topa* (‹$\top$›)
and *Inf* (‹$\bigsqcap$›)
and *Sup* (‹$\bigsqcup$›)
and *cl* :: $'a \Rightarrow 'a$
begin

lemma *cl-bot-least*:
shows *cl* $\bot \leq$ *cl X*
⟨*proof*⟩

lemma *cl-Inf-closed*:
shows *cl x* = $\bigsqcap\{y \in$ *closed*. $x \leq y\}$
⟨*proof*⟩

lemma *cl-top*:
shows *cl* $\top = \top$
⟨*proof*⟩

lemma *closed-top*[*iff*]:
shows $\top \in$ *closed*
⟨*proof*⟩

lemma *Sup-cl-le*:
shows $\bigsqcup$(*cl* ‘ *X*) $\leq$ *cl* ($\bigsqcup X$)
⟨*proof*⟩

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

lemma *cl-Inf-le*:
shows *cl* ($\bigsqcap X$) $\leq \bigsqcap$(*cl* ‘ *X*)
⟨*proof*⟩

lemma *cl-inf-le*:
shows *cl* ($x \sqcap y$) $\leq$ *cl x* $\sqcap$ *cl y*
⟨*proof*⟩

lemma *closed-Inf*:
assumes $X \subseteq$ *closed*
shows $\bigsqcap X \in$ *closed*
⟨*proof*⟩

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*
⟨*proof*⟩

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

definition *dense* :: ′*a set* **where**
dense = {*x*. *cl x* = ⊤}

lemma *dense-top*:
shows ⊤ ∈ *dense*
⟨*proof*⟩

lemma *dense-Sup*:
assumes $X \subseteq$ *dense*
assumes $X \neq \{\}$
shows $\bigsqcup X \in$ *dense*
⟨*proof*⟩

lemma *dense-sup*:
assumes $P \in$ *dense*
assumes $Q \in$ *dense*
shows $P \sqcup Q \in$ *dense*
⟨*proof*⟩

lemma *dense-le*:
assumes $P \in$ *dense*
assumes $P \leq Q$
shows $Q \in$ *dense*
⟨*proof*⟩

lemma *dense-inf-closed*:
shows *dense* ∩ *closed* = {⊤}
⟨*proof*⟩

end

locale *closure-complete-lattice-class* =
closure-complete-lattice (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-lattice* ⊤ *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* ≤ ⨆(*cl* ‘ ({*y*. *y* ≤ *x*} ∩ *compact-points*)) — The converse is given by monotonicity
begin

lemma *algebraic*:
shows *cl x* = ⨆(*cl* ‘ ({*y*. *y* ≤ *x*} ∩ *compact-points*))
⟨*proof*⟩

lemma *cont-cl*: — Equivalent to *algebraic-le* Davey and Priestley (2002, Theorem 7.14)
shows *cont* ⨆ (≤) ⨆ (≤) *cl*
⟨*proof*⟩

lemma *mcont-cl*:
shows *mcont* ⨆ (≤) ⨆ (≤) *cl*

⟨*proof*⟩

lemma *mcont2mcont-cl*[*cont-intro*]:
  assumes *mcont luba orda* $\bigsqcup$ ($\leq$) *P*
  shows *mcont luba orda* $\bigsqcup$ ($\leq$) ($\lambda x.$ *cl* (*P x*))
⟨*proof*⟩

end

locale *closure-complete-lattice-algebraic-class* =
  closure-complete-lattice-algebraic ($\leq$) (<) ($\sqcap$) ($\sqcup$) $\bot$ :: - :: *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* ($\bigsqcup X$) $\leq$ $\bigsqcup$(*cl* ‘ *X*) $\sqcup$ *cl* $\bot$
begin

lemma *cl-Sup*:
  shows *cl* ($\bigsqcup X$) = $\bigsqcup$(*cl* ‘ *X*) $\sqcup$ *cl* $\bot$
⟨*proof*⟩

lemma *cl-Sup-not-empty*:
  assumes $X \neq \{\}$
  shows *cl* ($\bigsqcup X$) = $\bigsqcup$(*cl* ‘ *X*)
⟨*proof*⟩

lemma *cl-sup*:
  shows *cl* ($X \sqcup Y$) = *cl X* $\sqcup$ *cl Y*
⟨*proof*⟩

lemma *closed-sup*[*intro*]:
  assumes $P \in$ *closed*
  assumes $Q \in$ *closed*
  shows $P \sqcup Q \in$ *closed*
⟨*proof*⟩

lemma *closed-Sup*: — Alexandrov: https://en.wikipedia.org/wiki/Alexandrov_topology
  assumes $X \subseteq$ *closed*
  shows $\bigsqcup X \sqcup$ *cl* $\bot \in$ *closed*
⟨*proof*⟩

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

lemma *cont-cl*:
  shows *cont* $\bigsqcup$ ($\leq$) $\bigsqcup$ ($\leq$) *cl*
⟨*proof*⟩

lemma *mcont-cl*:
  shows *mcont* $\bigsqcup$ ($\leq$) $\bigsqcup$ ($\leq$) *cl*
⟨*proof*⟩

lemma *mcont2mcont-cl*[*cont-intro*]:
  assumes *mcont luba orda* $\bigsqcup$ ($\leq$) *F*
  shows *mcont luba orda* $\bigsqcup$ ($\leq$) ($\lambda x.$ *cl* (*F x*))
⟨*proof*⟩

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

assumes *sup-irreducible-on closed* (*cl x*)
shows *sup-irreducible-on closed x*
⟨*proof*⟩

end

locale *closure-complete-lattice-distributive-class* =
closure-complete-lattice-distributive (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-lattice* ⊤ *Inf Sup*

locale *closure-complete-distrib-lattice-distributive-class* =
closure-complete-lattice-distributive (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-distrib-lattice* ⊤ *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 $\forall A \in A.\ \forall x \in A.\ x \in \mathit{closed}$
shows $(\bigsqcap X \in A.\ \bigsqcup X \sqcup \mathit{cl}\ \bot)$
$\le \bigsqcup(\mathit{Inf}\ \text{`}\ \{f\ \text{`}\ A\ |f.\ (\forall X \subseteq \mathit{closed}.\ f\ X \in \mathit{closed}) \wedge (\forall Y \in A.\ f\ Y \in Y)\}) \sqcup \mathit{cl}\ \bot$
⟨*proof*⟩

lemma *closed-complete-distrib-lattice-axiomI*[*intro*]:
assumes $\forall A \in A.\ \forall x \in A.\ x \in \mathit{closed}$
shows $(\bigsqcap X \in A.\ \bigsqcup X \sqcup \mathit{cl}\ \bot)$
$\le \bigsqcup(\mathit{Inf}\ \text{`}\ \{B.\ (\exists f.\ (\forall x.\ (\forall x \in x.\ x \in \mathit{closed}) \longrightarrow f\ x \in \mathit{closed})$
$\wedge B = f\ \text{`}\ A \wedge (\forall Y \in A.\ f\ Y \in Y)) \wedge (\forall x \in B.\ x \in \mathit{closed})\})$
$\sqcup\ \mathit{cl}\ \bot$
⟨*proof*⟩

lemma *closed-strict-complete-distrib-lattice-axiomI*[*intro*]:
assumes $\mathit{cl}\ \bot = \bot$
assumes $\forall A \in A.\ \forall x \in A.\ x \in \mathit{closed}$
shows $(\bigsqcap X \in A.\ \bigsqcup X)$
$\le \bigsqcup(\mathit{Inf}\ \text{`}\ \{x.\ (\exists f.\ (\forall x.\ (\forall x \in x.\ x \in \mathit{closed}) \longrightarrow f\ x \in \mathit{closed})$
$\wedge x = f\ \text{`}\ A \wedge (\forall Y \in A.\ f\ Y \in Y)) \wedge (\forall x \in x.\ x \in \mathit{closed})\})$
⟨*proof*⟩

end

5.2 Closures over powersets

locale *closure-powerset* =
closure-complete-lattice-class cl **for** *cl* :: *′a set* ⇒ *′a set*
begin

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

lemma *closedI*[*intro*]:
assumes $\bigwedge x.\ x \in \mathit{cl}\ X \Longrightarrow x \in X$
shows $X \in \mathit{closed}$
⟨*proof*⟩

lemma *closedE*:
assumes $x \in \mathit{cl}\ Y$
assumes $Y \in \mathit{closed}$
shows $x \in Y$
⟨*proof*⟩

lemma *cl-mono*:
  assumes $x \in cl\ X$
  assumes $X \subseteq Y$
  shows $x \in cl\ Y$
⟨*proof*⟩

lemma *cl-bind-le*:
  shows $X \gg= cl \circ f \leq cl\ (X \gg= f)$
⟨*proof*⟩

lemma *pointwise-distributive-iff*:
  shows $(\forall X.\ cl\ (\bigcup X) = \bigcup(cl\ `\ X) \cup cl\ \{\}) \longleftrightarrow (\forall X.\ cl\ X = (\bigcup x{\in}X.\ cl\ \{x\}) \cup cl\ \{\})$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *Sup-prime-on-singleton*:
  shows *Sup-prime-on closed* $(cl\ \{x\})$
⟨*proof*⟩

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 $cl\ x \subseteq \bigcup\ (cl\ `\ (\{y.\ y \subseteq x\} \cap local.compact\text{-}points))$
⟨*proof*⟩

lemma *cl-insert*:
  shows $cl\ (insert\ x\ X) = cl\ \{x\} \cup cl\ X$
⟨*proof*⟩

lemma *cl-UNION*:
  shows $cl\ (\bigcup i{\in}I.\ f\ i) = (\bigcup i{\in}I.\ cl\ (f\ i)) \cup cl\ \{\}$
⟨*proof*⟩

lemma *closed-UNION*:
  assumes $\bigwedge i.\ i \in I \Longrightarrow f\ i \in closed$
  shows $(\bigcup i{\in}I.\ f\ i) \cup cl\ \{\} \in closed$
⟨*proof*⟩

lemma *sort-of-inverse*: — Pfaltz and Šlapal (2013, Proposition 2.5)
  assumes $y \in cl\ X - cl\ \{\}$
  shows $\exists x{\in}X.\ y \in cl\ \{x\}$
⟨*proof*⟩

lemma *cl-diff-le*:
  shows $cl\ x - cl\ y \subseteq cl\ (x - y)$
⟨*proof*⟩

lemma *cl-bind*:
  shows $cl\ (X \gg= f) = (X \gg= cl \circ f) \cup cl\ \{\}$
⟨*proof*⟩

lemma *sup-irreducible-on-singleton*:
  shows *sup-irreducible-on closed* $(cl\ \{a\})$
⟨*proof*⟩

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* :: $('a\ set \Rightarrow 'a\ set) \Rightarrow bool$ **where**
  $anti\text{-}exchange\ cl \longleftrightarrow (\forall X\ x\ y.\ x \neq y \wedge y \in cl\ (insert\ x\ X) - cl\ X \longrightarrow x \notin cl\ (insert\ y\ X) - cl\ X)$

definition *exchange* :: $('a\ set \Rightarrow 'a\ set) \Rightarrow bool$ **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 $x \notin cl\ (insert\ y\ X) - cl\ X$
⟨*proof*⟩

lemma *exchange-Image*: — Some matroids arise from equivalence relations. Note $sym\ r \wedge trans\ r \longrightarrow Refl\ r$
  shows $exchange\ (Image\ r) \longleftrightarrow sym\ r \wedge trans\ r$
⟨*proof*⟩

locale *closure-powerset-distributive-exchange* =
  closure-powerset-distributive
+ **assumes** *exchange*: *exchange cl*
begin

lemma *exchange-exchange*:
  assumes $x \in cl\ \{y\}$
  assumes $x \notin cl\ \{\}$
  shows $y \in cl\ \{x\}$
⟨*proof*⟩

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*⟩

lemma *exchange-both-closed-inter*:

assumes $P \in closed$
assumes $Q \in closed$
shows $cl\ (P \cap Q) = P \cap Q$
⟨*proof*⟩

end

lemma *anti-exchange-Image*: — when r is asymmetric on distinct points
shows $anti\text{-}exchange\ (Image\ r) \longleftrightarrow (\forall x\ y.\ x \neq y \wedge (x, y) \in r \longrightarrow (y, x) \notin r)$
⟨*proof*⟩

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 $closure\ lesseqa\ lessa\ cl_1$
assumes $closure\ lesseqa\ lessa\ cl_2$
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows $closure\ lesseqa\ lessa\ (\lambda X.\ cl_1\ (cl_2\ X))$
⟨*proof*⟩

lemma *closure-complete-lattice-comp*:
assumes $closure\text{-}complete\text{-}lattice\ Infa\ Supa\ infa\ lesseqa\ lessa\ supa\ bota\ topa\ cl_1$
assumes $closure\text{-}complete\text{-}lattice\ Infa\ Supa\ infa\ lesseqa\ lessa\ supa\ bota\ topa\ cl_2$
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows $closure\text{-}complete\text{-}lattice\ Infa\ Supa\ infa\ lesseqa\ lessa\ supa\ bota\ topa\ (\lambda X.\ cl_1\ (cl_2\ X))$
⟨*proof*⟩

lemma *closure-powerset-comp*:
assumes $closure\text{-}powerset\ cl_1$
assumes $closure\text{-}powerset\ cl_2$
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows $closure\text{-}powerset\ (\lambda X.\ cl_1\ (cl_2\ X))$
⟨*proof*⟩

lemma *closure-powerset-distributive-comp*:
assumes $closure\text{-}powerset\text{-}distributive\ cl_1$
assumes $closure\text{-}powerset\text{-}distributive\ cl_2$
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows $closure\text{-}powerset\text{-}distributive\ (\lambda X.\ cl_1\ (cl_2\ X))$
⟨*proof*⟩

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* :: $('a \Rightarrow 'a) \Rightarrow bool$ **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*⟩

end

5.6 Some closures

interpretation *id-cl*: *closure-powerset-distributive id*
⟨*proof*⟩

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*
⟨*proof*⟩

interpretation *symcl-cl*: *closure-powerset-distributive-exchange* $\lambda X.\ X \cup X^{-1}$
⟨*proof*⟩

interpretation *trancl-cl*: *closure-powerset trancl*
⟨*proof*⟩

interpretation *rtrancl-cl*: *closure-powerset rtrancl*
⟨*proof*⟩

lemma *rtrancl-closed-Id*:
shows *Id* ∈ *rtrancl-cl.closed*
⟨*proof*⟩

lemma *rtrancl-closed-reflcl-closed*:
shows *rtrancl-cl.closed* ⊆ *reflcl-cl.closed*
⟨*proof*⟩

5.6.2 Relation image

lemma *idempotent-Image*:
assumes *refl-on Y r*
assumes *trans r*
assumes *X* ⊆ *Y*
shows *r* `` *r* `` *X* = *r* `` *X*
⟨*proof*⟩

lemmas *distributive-Image* = *Image-eq-UN*

lemma *closure-powerset-distributive-ImageI*:
assumes *cl* = *Image r*
assumes *refl r*
assumes *trans r*
shows *closure-powerset-distributive cl*
⟨*proof*⟩

lemma *closure-powerset-distributive-exchange-ImageI*:
assumes *cl* = *Image r*

assumes *equiv UNIV r* — symmetric, transitive and universal domain
shows *closure-powerset-distributive-exchange cl*
⟨*proof*⟩

interpretation *Image-rtrancl*: *closure-powerset-distributive Image* (r^*)
⟨*proof*⟩

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 $x \cdot \bot = \bot$ (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::complete-lattice* (‹ε›)
fixes *comp* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ‹$\cdot$› *60*)
assumes *comp-assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
assumes *weak-comp-unitL*: $\varepsilon \le x \Longrightarrow \varepsilon \cdot x = x$
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* (≤) *Sup* (≤) ($\lambda x.\ x \cdot y$)
assumes *mcont-compR*: *mcont Sup* (≤) *Sup* (≤) ($\lambda y.\ x \cdot y$)
assumes *comp-botL*: $\bot \cdot x = \bot$
begin

lemma *mcont2mcont-comp*:
assumes *mcont Supa orda Sup* (≤) *f*
assumes *mcont Supa orda Sup* (≤) *g*
shows *mcont Supa orda Sup* (≤) ($\lambda x.\ f\ x \cdot g\ x$)
⟨*proof*⟩

lemma *mono2mono-comp*:
assumes *monotone orda* (≤) *f*
assumes *monotone orda* (≤) *g*
shows *monotone orda* (≤) ($\lambda x.\ f\ x \cdot g\ x$)
⟨*proof*⟩

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* ⇒ *'a* **where**
star x = ($x \cdot$ *star x*) $\sqcup\ \varepsilon$

partial-function (*lfp*) *rev-star* :: *'a* ⇒ *'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* ⊥ = *ε*
⟨*proof*⟩

lemma *epsilon-star-le*:
shows *ε* ≤ *star P*
⟨*proof*⟩

lemma *monotone-star*:
shows *mono star*
⟨*proof*⟩

lemma *expansive-star*:
shows *x* ≤ *star x*
⟨*proof*⟩

lemma *star-comp-star*:
shows *star x* · *star x* = *star x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *idempotent-star*:
shows *star* (*star x*) = *star x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

interpretation *star*: *closure-complete-lattice-class star*
⟨*proof*⟩

lemma *star-epsilon*:
shows *star ε* = *ε*
⟨*proof*⟩

lemma *epsilon-rev-star-le*:
shows *ε* ≤ *rev-star P*
⟨*proof*⟩

lemma *rev-star-comp-rev-star*:
shows *rev-star x* · *rev-star x* = *rev-star x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *star-rev-star*:
shows *star* = *rev-star* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemmas *star-fixp-rev-induct* = *rev-star.fixp-induct*[*folded star-rev-star*]

interpretation *rev-star*: *closure-complete-lattice-class rev-star*
⟨*proof*⟩

lemma *rev-star-bot*:
shows *rev-star* ⊥ = *ε*
⟨*proof*⟩

lemma *rev-star-epsilon*:
shows *rev-star ε* = *ε*

⟨*proof*⟩

lemmas *star-unfoldL* = *star.simps*

lemma *star-unfoldR*:
  shows $star\ x = (star\ x \cdot x) \sqcup \varepsilon$
⟨*proof*⟩

lemmas *rev-star-unfoldR* = *rev-star.simps*

lemma *rev-star-unfoldL*:
  shows $rev\text{-}star\ x = (x \cdot rev\text{-}star\ x) \sqcup \varepsilon$
⟨*proof*⟩

lemma *fold-starL*:
  shows $x \cdot star\ x \leq star\ x$
⟨*proof*⟩

lemma *fold-starR*:
  shows $star\ x \cdot x \leq star\ x$
⟨*proof*⟩

lemma *fold-rev-starL*:
  shows $x \cdot rev\text{-}star\ x \leq rev\text{-}star\ x$
⟨*proof*⟩

lemma *fold-rev-starR*:
  shows $rev\text{-}star\ x \cdot x \leq rev\text{-}star\ x$
⟨*proof*⟩

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** ‹$\leq_a$› *50*)
    and *lessa* (**infix** ‹$<_a$› *50*)
    and *less-eqb* (**infix** ‹$\leq_b$› *50*)
    and *lessb* (**infix** ‹$<_b$› *50*)
\+ **fixes** $lower :: 'a \Rightarrow 'b$
  fixes $upper :: 'b \Rightarrow 'a$
  assumes *galois*: $lower\ x \leq_b y \longleftrightarrow x \leq_a upper\ y$

begin

lemma *monotone-lower*:
  shows *monotone* $(\leq_a)$ $(\leq_b)$ *lower*
⟨*proof*⟩

lemma *monotone-upper*:
  shows *monotone* $(\leq_b)$ $(\leq_a)$ *upper*
⟨*proof*⟩

lemmas *strengthen-lower*[*strg*] = *st-monotone*[*OF monotone-lower*]
lemmas *strengthen-upper*[*strg*] = *st-monotone*[*OF monotone-upper*]

lemma *upper-lower-expansive*:
  shows $x \leq_a$ *upper* (*lower* x)
⟨*proof*⟩

lemma *lower-upper-contractive*:
  shows *lower* (*upper* x) $\leq_b x$
⟨*proof*⟩

lemma *comp-galois*: — Backhouse (2000, Lemma 19). Observe that the roles of upper and lower have swapped.
  fixes *less-eqc* :: $'c \Rightarrow 'c \Rightarrow$ *bool* (**infix** ‹$\leq_c$› *50*)
  fixes *lessc* :: $'c \Rightarrow 'c \Rightarrow$ *bool* (**infix** ‹$<_c$› *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 $(\forall x.\ h$ (*upper* x) $\leq_c k\ x) \longleftrightarrow (\forall x.\ h\ x \leq_c k$ (*lower* x))
⟨*proof*⟩

lemma *lower-upper-le-iff*: — Backhouse (2000, Lemma 23)
  assumes $\forall x\ y.$ *lower'* $x \leq_b y \longleftrightarrow x \leq_a$ *upper'* y
  shows $(\forall x.$ *lower'* $x \leq_b$ *lower* $x) \longleftrightarrow (\forall y.$ *upper* $y \leq_a$ *upper'* $y)$
⟨*proof*⟩

lemma *lower-upper-unique*: — Backhouse (2000, Lemma 24)
  assumes $\forall x\ y.$ *lower'* $x \leq_b y \longleftrightarrow x \leq_a$ *upper'* y
  shows *lower'* = *lower* $\longleftrightarrow$ *upper'* = *upper*
⟨*proof*⟩

lemma *upper-lower-idem*:
  shows *upper* (*lower* (*upper* (*lower* x))) = *upper* (*lower* x)
⟨*proof*⟩

lemma *lower-upper-idem*:
  shows *lower* (*upper* (*lower* (*upper* x))) = *lower* (*upper* x)
⟨*proof*⟩

lemma *lower-upper-lower*: — Melton et al. (1985, Proposition 1.2(2))
  shows *lower* ∘ *upper* ∘ *lower* = *lower*
    and *lower* (*upper* (*lower* x)) = *lower* x
⟨*proof*⟩

lemma *upper-lower-upper*: — Melton et al. (1985, Proposition 1.2(2))
  shows *upper* ∘ *lower* ∘ *upper* = *upper*
    and *upper* (*lower* (*upper* x)) = *upper* x

⟨*proof*⟩

definition *cl* :: $'a \Rightarrow 'a$ **where** — The opposite composition yields a kernel operator
cl x = *upper* (*lower* x)

lemma *cl-axiom*:
shows $(x \leq_a cl\ y) = (cl\ x \leq_a cl\ y)$
⟨*proof*⟩

sublocale *closure* $(\leq_a)$ $(<_a)$ *cl* — incorporates definitions and lemmas into this namespace
⟨*proof*⟩

lemma *cl-upper*:
shows *cl* (*upper* P) = *upper* P
⟨*proof*⟩

lemma *closed-upper*:
shows *upper* $P \in$ *closed*
⟨*proof*⟩

lemma *inj-lower-iff-surj-upper*:
shows *inj lower* $\longleftrightarrow$ *surj upper*
⟨*proof*⟩

lemma *inj-lower-iff-upper-lower-id*:
shows *inj lower* $\longleftrightarrow$ *upper* $\circ$ *lower* = *id*
⟨*proof*⟩

lemma *upper-inj-iff-surj-lower*:
shows *inj upper* $\longleftrightarrow$ *surj lower*
⟨*proof*⟩

lemma *inj-upper-iff-lower-upper-id*:
shows *inj upper* $\longleftrightarrow$ *lower* $\circ$ *upper* = *id*
⟨*proof*⟩

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* $-`\ \{a.\ a \leq_b y\} = \{a.\ a \leq_a upper\ y\}$
⟨*proof*⟩

lemma *lower-downset*: — Davey and Priestley (2002, Lemma 7.32); equivalent to the Galois axiom
shows $\exists!x.$ *lower* $-`\ \{a.\ a \leq_b y\} = \{a.\ a \leq_a x\}$
⟨*proof*⟩

end

⟨*ML*⟩

lemma *axioms-alt*:
fixes *less-eqa* (**infix** ‹$\leq_a$› *50*)
fixes *less-eqb* (**infix** ‹$\leq_b$› *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
⟨*proof*⟩

lemma *compose*:
fixes $lower_1 :: 'b \Rightarrow 'c$
fixes $lower_2 :: 'a \Rightarrow 'b$
fixes *less-eqa* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
assumes *galois less-eqb lessb less-eqc lessc* $lower_1$ $upper_1$
assumes *galois less-eqa lessa less-eqb lessb* $lower_2$ $upper_2$
shows *galois less-eqa lessa less-eqc lessc* $(lower_1 \circ lower_2)$ $(upper_2 \circ upper_1)$
⟨*proof*⟩

locale *complete-lattice* =
cla: *complete-lattice* Inf_a Sup_a $(\sqcap_a)$ $(\leq_a)$ $(<_a)$ $(\sqcup_a)$ $\bot_a$ $\top_a$
+ *clb*: *complete-lattice* Inf_b Sup_b $(\sqcap_b)$ $(\leq_b)$ $(<_b)$ $(\sqcup_b)$ $\bot_b$ $\top_b$
+ *galois* $(\leq_a)$ $(<_a)$ $(\leq_b)$ $(<_b)$ *lower upper*
for *less-eqa* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ‹$\leq_a$› *50*)
and *lessa* (**infix** ‹$<_a$› *50*)
and *infa* (**infixl** ‹$\sqcap_a$› *70*)
and *supa* (**infixl** ‹$\sqcup_a$› *65*)
and *bota* (‹$\bot_a$›)
and *topa* (‹$\top_a$›)
and Inf_a Sup_a
and *less-eqb* :: $'b \Rightarrow 'b \Rightarrow$ *bool* (**infix** ‹$\leq_b$› *50*)
and *lessb* (**infix** ‹$<_b$› *50*)
and *infb* (**infixl** ‹$\sqcap_b$› *70*)
and *supb* (**infixl** ‹$\sqcup_b$› *65*)
and *botb* (‹$\bot_b$›)
and *topb* (‹$\top_b$›)
and Inf_b Sup_b
and *lower* :: $'a \Rightarrow 'b$
and *upper* :: $'b \Rightarrow 'a$
begin

lemma *lower-bot*:
shows *lower* $\bot_a = \bot_b$
⟨*proof*⟩

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*⟩

lemma *lower-SUP*:
shows *lower* $(Sup_a\ (f$ ‘ $X)) = Sup_b$ ((λ*x*. *lower* (*f x*)) ‘ X)
⟨*proof*⟩

lemma *lower-sup*:
shows *lower* $(X \sqcup_a Y) =$ *lower* $X \sqcup_b$ *lower* Y
⟨*proof*⟩

lemma *lower-Inf-le*:
shows *lower* $(Inf_a\ X) \leq_b Inf_b$ (*lower* ‘ X)
⟨*proof*⟩

lemma *lower-INF-le*:
  shows $lower\ (Inf_a\ (f\ `\ X)) \leq_b Inf_b\ ((\lambda x.\ lower\ (f\ x))\ `\ X)$
⟨*proof*⟩

lemma *lower-inf-le*:
  shows $lower\ (x \sqcap_a y) \leq_b lower\ x \sqcap_b lower\ y$
⟨*proof*⟩

lemma *mcont-lower*: — Backhouse (2000): fixed point theory based on Galois connections is less general than using countable chains
  shows $mcont\ Sup_a\ (\leq_a)\ Sup_b\ (\leq_b)\ lower$
⟨*proof*⟩

lemma *mcont2mcont-lower*[*cont-intro*]:
  assumes $mcont\ luba\ orda\ Sup_a\ (\leq_a)\ P$
  shows $mcont\ luba\ orda\ Sup_b\ (\leq_b)\ (\lambda x.\ lower\ (P\ x))$
⟨*proof*⟩

lemma *upper-top*:
  shows $upper\ \top_b = \top_a$
⟨*proof*⟩

lemma *Sup-upper-le*:
  shows $Sup_a\ (upper\ `\ X) \leq_a upper\ (Sup_b\ X)$
⟨*proof*⟩

lemma *sup-upper-le*:
  shows $upper\ x \sqcup_a upper\ y \leq_a upper\ (x \sqcup_b y)$
⟨*proof*⟩

lemma *upper-Inf*: — Melton et al. (1985, Proposition 1.2(6))
  shows $upper\ (Inf_b\ X) = Inf_a\ (upper\ `\ X)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *upper-INF*:
  shows $upper\ (Inf_b\ (f\ `\ X)) = Inf_a\ ((\lambda x.\ upper\ (f\ x))\ `\ X)$
⟨*proof*⟩

lemma *upper-inf*:
  shows $upper\ (X \sqcap_b Y) = upper\ X \sqcap_a upper\ Y$
⟨*proof*⟩

In a complete lattice *lower* is determined by *upper* and vice-versa.

lemma *lower-Inf-upper*:
  shows $lower\ X = Inf_b\ \{Y.\ X \leq_a upper\ Y\}$
⟨*proof*⟩

lemma *upper-Sup-lower*:
  shows $upper\ X = Sup_a\ \{Y.\ lower\ Y \leq_b X\}$
⟨*proof*⟩

lemma *upper-downwards-closure-lower*: — Melton et al. (1985, Lemma 2.1)
  shows $upper\ x = Sup_a\ (lower\ -`\ \{y.\ y \leq_b x\})$
⟨*proof*⟩

sublocale *closure-complete-lattice* $(\leq_a)\ (<_a)\ (\sqcap_a)\ (\sqcup_a)\ \bot_a\ \top_a\ Inf_a\ Sup_a\ cl$
⟨*proof*⟩

end

locale *complete-lattice-distributive* =
  galois.complete-lattice
\+ **assumes** *upper-Sup-le*: $upper\ (Sup_b\ X) \le_a Sup_a\ (upper\ `\ X)$ — Stronger than Scott continuity, which only asks for this for chain or directed *X*.
begin

lemma *upper-Sup*:
  shows $upper\ (Sup_b\ X) = Sup_a\ (upper\ `\ X)$
⟨*proof*⟩

lemma *upper-bot*:
  shows $upper\ \bot_b = \bot_a$
⟨*proof*⟩

lemma *upper-sup*:
  shows $upper\ (x \sqcup_b y) = upper\ x \sqcup_a upper\ y$
⟨*proof*⟩

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

lemma *mcont-upper*:
  shows $mcont\ Sup_b\ (\le_b)\ Sup_a\ (\le_a)\ upper$
⟨*proof*⟩

lemma *mcont2mcont-upper*[*cont-intro*]:
  assumes $mcont\ luba\ orda\ Sup_b\ (\le_b)\ P$
  shows $mcont\ luba\ orda\ Sup_a\ (\le_a)\ (\lambda x.\ upper\ (P\ x))$
⟨*proof*⟩

sublocale *closure-complete-lattice-distributive* $(\le_a)\ (<_a)\ (\sqcap_a)\ (\sqcup_a)\ \bot_a\ \top_a\ Inf_a\ Sup_a\ cl$
⟨*proof*⟩

lemma *cl-bot*:
  shows $cl\ \bot_a = \bot_a$
⟨*proof*⟩

lemma *closed-bot*[*iff*]:
  shows $\bot_a \in closed$
⟨*proof*⟩

end

locale *complete-lattice-class* =
  galois.complete-lattice
    $(\le)\ (<)\ (\sqcap)\ (\sqcup)\ \bot$ :: - :: *complete-lattice* $\top$ *Inf Sup*
    $(\le)\ (<)\ (\sqcap)\ (\sqcup)\ \bot$ :: - :: *complete-lattice* $\top$ *Inf Sup*
begin

sublocale *closure-complete-lattice-class cl* ⟨*proof*⟩

end

locale *complete-lattice-distributive-class* =
  galois.complete-lattice-distributive

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

sublocale *galois.complete-lattice-class* ⟨*proof*⟩
sublocale *closure-complete-lattice-distributive-class cl* ⟨*proof*⟩

end

lemma *existence-lower-preserves-Sup*: — Hoare and He (1987, p8 of Oxford TR PRG-44) amongst others
fixes *lower* :: -::*complete-lattice* ⇒ -::*complete-lattice*
assumes *mono lower*
shows $(\forall x\ y.\ lower\ x \leq y \longleftrightarrow x \leq \bigsqcup\{Y.\ lower\ Y \leq y\}) \longleftrightarrow (\forall X.\ lower\ (\bigsqcup X) \leq \bigsqcup(lower\ `\ X))$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lower-preserves-SupI*:
assumes *mono lower*
assumes $\bigwedge X.\ lower\ (\bigsqcup X) \leq \bigsqcup(lower\ `\ X)$
assumes $\bigwedge x.\ upper\ x = \bigsqcup\{X.\ lower\ X \leq x\}$
shows *galois.complete-lattice-class lower upper*
⟨*proof*⟩

lemma *existence-upper-preserves-Inf*:
fixes *upper* :: -::*complete-lattice* ⇒ -::*complete-lattice*
assumes *mono upper*
shows $(\forall x\ y.\ \bigsqcap\{Y.\ x \leq upper\ Y\} \leq y \longleftrightarrow x \leq upper\ y) \longleftrightarrow (\forall X.\ \bigsqcap(upper\ `\ X) \leq upper\ (\bigsqcap X))$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *upper-preserves-InfI*:
assumes *mono upper*
assumes $\bigwedge X.\ \bigsqcap(upper\ `\ X) \leq upper\ (\bigsqcap X)$
assumes $\bigwedge x.\ lower\ x = \bigsqcap\{X.\ x \leq upper\ X\}$
shows *galois.complete-lattice-class lower upper*
⟨*proof*⟩

locale *powerset* =
galois.complete-lattice-class lower upper
for *lower* :: *'a set* ⇒ *'b set*
and *upper* :: *'b set* ⇒ *'a set*
begin

lemma *lower-insert*:
shows $lower\ (insert\ x\ X) = lower\ \{x\} \cup lower\ X$
⟨*proof*⟩

lemma *lower-distributive*:
shows $lower\ X = (\bigcup x{\in}X.\ lower\ \{x\})$
⟨*proof*⟩

sublocale *closure-powerset cl* ⟨*proof*⟩

end

locale *powerset-distributive* =
galois.powerset
+ *galois.complete-lattice-distributive-class*

begin

lemma *upper-insert*:
  shows *upper* (*insert* x X) = *upper* $\{x\}$ ∪ *upper* X
⟨*proof*⟩

lemma *cl-distributive-axiom*:
  shows *cl* ($\bigcup$ X) ⊆ $\bigcup$ (*cl* ‘ X)
⟨*proof*⟩

sublocale *closure-powerset-distributive cl*
⟨*proof*⟩

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 × 'b) *set* ⇒ 'a *set* ⇒ 'b *set* **where**
  $lower_R$ R A = R ‘‘ A

definition $upper_R$ :: ('a × 'b) *set* ⇒ 'b *set* ⇒ '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
⟨*proof*⟩

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*⟩

end

⟨*ML*⟩

6.1 Some Galois connections

⟨*ML*⟩

locale *complete-lattice-class-monomorphic*
  = *galois.complete-lattice-class upper lower*
    for *upper* :: 'a::*complete-lattice* ⇒ 'a **and** *lower* :: 'a ⇒ 'a — Avoid 'a *itself* parameters

interpretation *conj-imp*: *galois.complete-lattice-class* (⊓) x ($\longrightarrow_B$) x **for** x :: -::*boolean-algebra* — Classic example
⟨*proof*⟩

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 ⇒ 'b
begin

definition *lower* :: 'a *set* ⇒ 'b *set* **where**

lower X = *f* ‘ *X*

definition *upper* :: *'b set* ⇒ *'a set* **where**
upper X = *f* −‘ *X*

lemma *upper-empty*[*iff*]:
shows *upper* {} = {}
⟨*proof*⟩

sublocale *galois.powerset-distributive lower upper*
⟨*proof*⟩

abbreviation *equivalent* :: *'a relp* **where**
equivalent x y ≡ *f x* = *f y*

lemma *equiv*:
shows *Equiv-Relations.equivp equivalent*
⟨*proof*⟩

lemma *equiv-cl-singleton*:
assumes *equivalent x y*
shows *cl* {*x*} = *cl* {*y*}
⟨*proof*⟩

lemma *cl-alt-def*:
shows *cl X* = {(*x*, *y*). *equivalent x y*} ‘‘ *X*
⟨*proof*⟩

sublocale *closure-powerset-distributive-exchange cl*
⟨*proof*⟩

lemma *closed-in*:
assumes *x* ∈ *P*
assumes *equivalent x y*
assumes *P*: *P* ∈ *closed*
shows *y* ∈ *P*
⟨*proof*⟩

lemma *clE*:
assumes *x* ∈ *cl P*
obtains *y* **where** *equivalent y x* **and** *y* ∈ *P*
⟨*proof*⟩

lemma *clI*[*intro*]:
assumes *x* ∈ *P*
assumes *equivalent x y*
shows *y* ∈ *cl P*
⟨*proof*⟩

lemma *closed-diff*[*intro*]:
assumes *X* ∈ *closed*
assumes *Y* ∈ *closed*
shows *X* − *Y* ∈ *closed*
⟨*proof*⟩

lemma *closed-uminus*[*intro*]:
assumes *X* ∈ *closed*
shows −*X* ∈ *closed*

⟨*proof*⟩

end

locale *image-vimage-monomorphic*
  = *galois.image-vimage f*
      for *f* :: *'a* ⇒ *'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 $f \circ f = f$
⟨*proof*⟩

lemma *idemI*:
  assumes $f\ x \in P$
  shows $x \in cl\ P$
⟨*proof*⟩

lemma *f-cl*:
  shows $f\ x \in cl\ P \longleftrightarrow x \in cl\ P$
⟨*proof*⟩

lemma *f-closed*:
  assumes $P \in closed$
  shows $f\ x \in P \longleftrightarrow x \in P$
⟨*proof*⟩

lemmas *f-closedI* = *iffD1*[*OF f-closed*]

end

⟨*ML*⟩

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*: $\bigwedge Y{::}'a\ set.\ \bigwedge x{::}'a.\ x \sqcap (\bigsqcup Y) = (\bigsqcup y{\in}Y.\ x \sqcap y)$
begin

definition *heyting* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ‹$\longrightarrow_H$› *53*) **where**
  $x \longrightarrow_H y = \bigsqcup\{z.\ x \sqcap z \leq y\}$

lemma *heyting*: — The Galois property for (⊓) and $\longrightarrow_H$

shows $z \le x \longrightarrow_H y \longleftrightarrow z \sqcap x \le y$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

end

⟨*ML*⟩

context *heyting-algebra*
begin

lemma *commute*:
shows $x \sqcap z \le y \longleftrightarrow z \le (x \longrightarrow_H y)$
⟨*proof*⟩

lemmas *uncurry* = *iffD1*[*OF heyting*]
lemmas *curry* = *iffD2*[*OF heyting*]

lemma *curry-conv*:
shows $(x \sqcap y \longrightarrow_H z) = (x \longrightarrow_H y \longrightarrow_H z)$
⟨*proof*⟩

lemma *swap*:
shows $P \longrightarrow_H Q \longrightarrow_H R = Q \longrightarrow_H P \longrightarrow_H R$
⟨*proof*⟩

lemma *absorb*:
shows $y \sqcap (x \longrightarrow_H y) = y$
and $(x \longrightarrow_H y) \sqcap y = y$
⟨*proof*⟩

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*⟩

lemma *discharge*:
assumes $x' \le x$
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*⟩

lemma *trans*:
shows $(x \longrightarrow_H y) \sqcap (y \longrightarrow_H z) \le x \longrightarrow_H z$
⟨*proof*⟩

lemma *rev-trans*:
shows $(y \longrightarrow_H z) \sqcap (x \longrightarrow_H y) \le x \longrightarrow_H z$
⟨*proof*⟩

lemma *discard*:
shows $Q \le P \longrightarrow_H Q$
⟨*proof*⟩

lemma *infR*:
shows $x \longrightarrow_H y \sqcap z = (x \longrightarrow_H y) \sqcap (x \longrightarrow_H z)$
⟨*proof*⟩

lemma *mono*:

assumes $x' \le x$
assumes $y \le y'$
shows $x \longrightarrow_H y \le x' \longrightarrow_H y'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F)$ X X'
assumes *st-ord* F Y Y'
shows *st-ord* F $(X \longrightarrow_H Y)$ $(X' \longrightarrow_H Y')$
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* $(\ge)$ F
assumes *monotone orda* $(\le)$ G
shows *monotone orda* $(\le)$ $(\lambda x.\ F\ x \longrightarrow_H G\ x)$
⟨*proof*⟩

lemma *mp*:
assumes $x \le y \longrightarrow_H z$
assumes $x \le y$
shows $x \le z$
⟨*proof*⟩

lemma *botL*:
shows $\bot \longrightarrow_H x = \top$
⟨*proof*⟩

lemma *top-conv*:
shows $x \longrightarrow_H y = \top \longleftrightarrow x \le y$
⟨*proof*⟩

lemma *refl*[*simp*]:
shows $x \longrightarrow_H x = \top$
⟨*proof*⟩

lemma *topL*[*simp*]:
shows $\top \longrightarrow_H x = x$
⟨*proof*⟩

lemma *topR*[*simp*]:
shows $x \longrightarrow_H \top = \top$
⟨*proof*⟩

lemma *K*[*simp*]:
shows $x \longrightarrow_H (y \longrightarrow_H x) = \top$
⟨*proof*⟩

subclass *distrib-lattice*
⟨*proof*⟩

lemma *supL*:
shows $(x \sqcup y) \longrightarrow_H z = (x \longrightarrow_H z) \sqcap (y \longrightarrow_H z)$
⟨*proof*⟩

subclass (**in** *complete-distrib-lattice*) *heyting-algebra* ⟨*proof*⟩

lemma *inf-Sup-distrib*:
shows $x \sqcap \bigsqcup Y = (\bigsqcup y{\in}Y.\ x \sqcap y)$

and $\bigsqcup Y \sqcap x = (\bigsqcup y{\in}Y.\ x \sqcap y)$
⟨*proof*⟩

lemma *inf-SUP-distrib*:
shows $x \sqcap (\bigsqcup i{\in}I.\ Y\ i) = (\bigsqcup i{\in}I.\ x \sqcap Y\ i)$
and $(\bigsqcup i{\in}I.\ Y\ i) \sqcap x = (\bigsqcup i{\in}I.\ Y\ i \sqcap x)$
⟨*proof*⟩

end

lemma *eq-boolean-implication*: — the implications coincide in *boolean-algebra*s
fixes x :: -::*boolean-algebra*
shows $x \longrightarrow_H y = x \longrightarrow_B y$
⟨*proof*⟩

lemmas *simp-thms* =
heyting.botL
heyting.topL
heyting.topR
heyting.refl

lemma *Sup-prime-Sup-irreducible-iff*:
fixes x :: -::*heyting-algebra*
shows *Sup-prime* $x \longleftrightarrow$ *Sup-irreducible* x
⟨*proof*⟩

Logical rules ala HOL **lemma** *bspec*:
fixes P :: - ⇒ (-::*heyting-algebra*)
shows $x \in X \Longrightarrow (\bigsqcap x{\in}X.\ P\ x \longrightarrow_H Q\ x) \sqcap P\ x \le Q\ x$ (**is** $?X \Longrightarrow ?thesis1$)
and $x \in X \Longrightarrow P\ x \sqcap (\bigsqcap x{\in}X.\ P\ x \longrightarrow_H Q\ x) \le Q\ x$ (**is** - $\Longrightarrow ?thesis2$)
and $(\bigsqcap x.\ P\ x \longrightarrow_H Q\ x) \sqcap P\ x \le Q\ x$ (**is** $?thesis3$)
and $P\ x \sqcap (\bigsqcap x.\ P\ x \longrightarrow_H Q\ x) \le Q\ x$ (**is** $?thesis4$)
⟨*proof*⟩

lemma *INFL*:
fixes Q :: -::*heyting-algebra*
shows $(\bigsqcap x{\in}X.\ P\ x \longrightarrow_H Q) = (\bigsqcup x{\in}X.\ P\ x) \longrightarrow_H Q$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemmas *SUPL* = *heyting.INFL*[*symmetric*]

lemma *INFR*:
fixes P :: -::*heyting-algebra*
shows $(\bigsqcap x{\in}X.\ P \longrightarrow_H Q\ x) = (P \longrightarrow_H (\bigsqcap x{\in}X.\ Q\ x))$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

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 :: -::*heyting-algebra*
shows $(\bigsqcup x{\in}X.\ P\ x \longrightarrow_H Q) \le (\bigsqcap x{\in}X.\ P\ x) \longrightarrow_H Q$
⟨*proof*⟩

lemma *SUPR-le*:
  fixes *P* :: -::*heyting-algebra*
  shows $(\bigsqcup x{\in}X.\ P \longrightarrow_H Q\ x) \leq P \longrightarrow_H (\bigsqcup x{\in}X.\ Q\ x)$
⟨*proof*⟩

lemma *SUP-inf*:
  fixes *Q* :: -::*heyting-algebra*
  shows $(\bigsqcup x{\in}X.\ P\ x \sqcap Q) = (\bigsqcup x{\in}X.\ P\ x) \sqcap Q$
⟨*proof*⟩

lemma *inf-SUP*:
  fixes *P* :: -::*heyting-algebra*
  shows $(\bigsqcup x{\in}X.\ P \sqcap Q\ x) = P \sqcap (\bigsqcup x{\in}X.\ Q\ x)$
⟨*proof*⟩

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* :: - ⇒ ′*a*::*heyting-algebra*
  fixes *G* :: - ⇒ ′*a*::*heyting-algebra*
  assumes *mcont luba orda Sup* (≤) *F*
  assumes *mcont luba orda Sup* (≤) *G*
  shows *mcont luba orda Sup* (≤) $(\lambda x.\ F\ x \sqcap G\ x)$
⟨*proof*⟩

lemma *closure-imp-distrib-le*: — Abadi and Plotkin (1993, Lemma 3.3), generalized
  fixes *P Q* :: - :: *heyting-algebra*
  assumes *cl*: *closure-axioms* (≤) *cl*
  assumes *cl-inf*: $\bigwedge x\ y.\ cl\ x \sqcap cl\ y \leq cl\ (x \sqcap y)$
  shows $P \longrightarrow_H Q \leq cl\ P \longrightarrow_H cl\ Q$
⟨*proof*⟩

⟨*ML*⟩

Pseudocomplements **definition** *pseudocomplement* :: ′*a*::*heyting-algebra* ⇒ ′*a* (‹$\neg_H$ -› [*75*] *75*) **where**
  $\neg_H x = x \longrightarrow_H \bot$

lemma *pseudocomplementI*:
  shows $x \leq \neg_H y \longleftrightarrow x \sqcap y \leq \bot$
⟨*proof*⟩

⟨*ML*⟩

lemma *monotone*:
  shows *antimono pseudocomplement*
⟨*proof*⟩

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 :: -::{*boolean-algebra*,*heyting-algebra*}
shows $\neg_H x = -x$
⟨*proof*⟩

lemma *heyting*:
shows $x \longrightarrow_H \neg_H x = \neg_H x$
⟨*proof*⟩

lemma *Inf*:
shows $x \sqcap \neg_H x = \bot$
and $\neg_H x \sqcap x = \bot$
⟨*proof*⟩

lemma *double-le*:
shows $x \leq \neg_H \neg_H x$
⟨*proof*⟩

interpretation *double*: *closure-complete-lattice-class pseudocomplement* ∘ *pseudocomplement*
⟨*proof*⟩

lemma *triple*:
shows $\neg_H \neg_H \neg_H x = \neg_H x$
⟨*proof*⟩

lemma *contrapos-le*:
shows $x \longrightarrow_H y \leq \neg_H y \longrightarrow_H \neg_H x$
⟨*proof*⟩

lemma *sup-inf*: — half of de Morgan
shows $\neg_H (x \sqcup y) = \neg_H x \sqcap \neg_H y$
⟨*proof*⟩

lemma *inf-sup-weak*: — the weakened other half of de Morgan
shows $\neg_H (x \sqcap y) = \neg_H \neg_H (\neg_H x \sqcup \neg_H y)$
⟨*proof*⟩

lemma *fix-triv*:
assumes $x = \neg_H x$
shows $x = y$
⟨*proof*⟩

lemma *double-top*:
shows $\neg_H \neg_H (x \sqcup \neg_H x) = \top$
⟨*proof*⟩

lemma *Inf-inf*:
fixes P :: - ⇒ (-::*heyting-algebra*)
shows $(\bigsqcap x.\ P\ x) \sqcap \neg_H P\ x = \bot$
⟨*proof*⟩

lemma *SUP-le*: — half of de Morgan
fixes P :: - ⇒ (-::*heyting-algebra*)
shows $(\bigsqcup x{\in}X.\ P\ x) \leq \neg_H (\bigsqcap x{\in}X.\ \neg_H P\ x)$
⟨*proof*⟩

lemma *SUP-INF-le*:
fixes P :: - ⇒ (-::*heyting-algebra*)
shows $(\bigsqcup x{\in}X.\ \neg_H P\ x) \leq \neg_H (\bigsqcap x{\in}X.\ P\ x)$

⟨*proof*⟩

lemma *SUP*:
  fixes P :: - ⇒ (-::*heyting-algebra*)
  shows $\neg_H(\bigsqcup x{\in}X.\ P\ x) = (\bigsqcap x{\in}X.\ \neg_H P\ x)$
⟨*proof*⟩

⟨*ML*⟩

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)

⟨*ML*⟩

definition *cl* :: *′a::preorder set* ⇒ *′a set* **where**
  $cl\ P = \{x\ |x\ y.\ y \in P \wedge x \leq y\}$

⟨*ML*⟩

interpretation *downwards*: *closure-powerset-distributive downwards.cl* — On preorders
⟨*proof*⟩

interpretation *downwards*: *closure-powerset-distributive-anti-exchange* (*downwards.cl*::-::*order set* ⇒ -)
  — On partial orders; see Pfaltz and Šlapal (2013)
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-empty*:
  shows *downwards.cl* {} = {}
⟨*proof*⟩

lemma *closed-empty*[*iff*]:
  shows {} ∈ *downwards.closed*
⟨*proof*⟩

lemma *clI*[*intro*]:
  assumes $y \in P$
  assumes $x \leq y$
  shows $x \in$ *downwards.cl* P
⟨*proof*⟩

lemma *clE*:
  assumes $x \in$ *downwards.cl* P
  obtains y **where** $y \in P$ **and** $x \leq y$
⟨*proof*⟩

lemma *closed-in*:
  assumes $x \in P$

assumes $y \leq x$
assumes $P \in$ *downwards.closed*
shows $y \in P$
⟨*proof*⟩

lemma *order-embedding*: — On preorders; see Davey and Priestley (2002, §1.35)
fixes x :: *-::preorder*
shows *downwards.cl* $\{x\} \subseteq$ *downwards.cl* $\{y\} \longleftrightarrow x \leq y$
⟨*proof*⟩

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::preorder set* ⇒ *'a set* ⇒ *'a set* **where**
imp P $Q = \{\sigma.\ \forall \sigma'{\leq}\sigma.\ \sigma' \in P \longrightarrow \sigma' \in Q\}$

lemma *imp-refl*:
shows *downwards.imp* P P = *UNIV*
⟨*proof*⟩

lemma *imp-contained*:
assumes $P \subseteq Q$
shows *downwards.imp* P Q = *UNIV*
⟨*proof*⟩

lemma *heyting-imp*:
assumes $P \in$ *downwards.closed*
shows $P \subseteq$ *downwards.imp* Q $R \longleftrightarrow P \cap Q \subseteq R$
⟨*proof*⟩

lemma *imp-mp'*:
assumes $\sigma \in$ *downwards.imp* P Q
assumes $\sigma \in P$
shows $\sigma \in Q$
⟨*proof*⟩

lemma *imp-mp*:
shows $P \cap$ *downwards.imp* P $Q \subseteq Q$
and *downwards.imp* P $Q \cap P \subseteq Q$
⟨*proof*⟩

lemma *imp-contains*:
assumes $X \subseteq Q$
assumes $X \in$ *downwards.closed*
shows $X \subseteq$ *downwards.imp* P Q
⟨*proof*⟩

lemma *imp-downwards*:
assumes $y \in$ *downwards.imp* P Q
assumes $x \leq y$
shows $x \in$ *downwards.imp* P Q
⟨*proof*⟩

lemma *closed-imp*:
  shows *downwards.imp* P $Q \in$ *downwards.closed*
⟨*proof*⟩

The set *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 *downwards.imp* P $Q \subseteq P \longrightarrow_B Q$
⟨*proof*⟩

lemma *downwards-closed-imp-greatest*:
  assumes $R \subseteq P \longrightarrow_B Q$
  assumes $R \in$ *downwards.closed*
  shows $R \subseteq$ *downwards.imp* P Q
⟨*proof*⟩

definition *kernel* :: *'a::order set* ⇒ *'a set* **where**
  kernel $X = \bigsqcup\{Q \in$ *downwards.closed*. $Q \subseteq X\}$

lemma *kernel-def2*:
  shows *downwards.kernel* $X = \{\sigma.\ \forall\sigma' \leq \sigma.\ \sigma' \in X\}$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *kernel-contractive*:
  shows *downwards.kernel* $X \subseteq X$
⟨*proof*⟩

lemma *kernel-idempotent*:
  shows *downwards.kernel* (*downwards.kernel* X) = *downwards.kernel* X
⟨*proof*⟩

lemma *kernel-monotone*:
  shows *mono downwards.kernel*
⟨*proof*⟩

lemma *closed-kernel-conv*:
  shows $X \in$ *downwards.closed* ⟷ *downwards.kernel* $X = X$
⟨*proof*⟩

lemma *closed-kernel*:
  shows *downwards.kernel* $X \in$ *downwards.closed*
⟨*proof*⟩

lemma *kernel-cl*:
  shows *downwards.kernel* (*downwards.cl* X) = *downwards.cl* X
⟨*proof*⟩

lemma *cl-kernel*:
  shows *downwards.cl* (*downwards.kernel* X) = *downwards.kernel* X
⟨*proof*⟩

lemma *kernel-boolean-implication*:
  fixes P :: *-::order*
  shows *downwards.kernel* $(P \longrightarrow_B Q)$ = *downwards.imp* P Q
⟨*proof*⟩

⟨*ML*⟩

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 (♮) on $('a, 's, 'v)$ *trace.t* and extract the (matroidal) closure over sets of these from the Galois connection *galois.image-vimage*.

⟨*ML*⟩

primrec *natural'* :: *'s* ⇒ (*'a* × *'s*) *list* ⇒ (*'a* × *'s*) *list* **where**
  natural' s [] = []
| *natural' s* (*x* # *xs*) = (*if snd x* = *s then natural' s xs else x* # *natural'* (*snd x*) *xs*)

⟨*ML*⟩

lemma *natural'*[*simp*]:
  shows *trace.final' s* (*trace.natural' s xs*) = *trace.final' s xs*
⟨*proof*⟩

lemma *natural'-cong*:
  assumes *s* = *s'*
  assumes *trace.natural' s xs* = *trace.natural' s xs'*
  shows *trace.final' s xs* = *trace.final' s' xs'*
⟨*proof*⟩

⟨*ML*⟩

lemma *natural'*:
  shows *trace.natural' s* (*trace.natural' s xs*) = *trace.natural' s xs*
⟨*proof*⟩

lemma *length*:
  shows *length* (*trace.natural' s xs*) ≤ *length xs*
⟨*proof*⟩

lemma *subseq*:
  shows *subseq* (*trace.natural' s xs*) *xs*
⟨*proof*⟩

lemma *remdups-adj*:
  shows *s* # *map snd* (*trace.natural' s xs*) = *remdups-adj* (*s* # *map snd xs*)
⟨*proof*⟩

lemma *append*:
  shows *trace.natural' s* (*xs* @ *ys*) = *trace.natural' s xs* @ *trace.natural'* (*trace.final' s xs*) *ys*
⟨*proof*⟩

lemma *eq-Nil-conv*:
  shows *trace.natural' s xs* = [] ⟷ *snd* ‘ *set xs* ⊆ {*s*}
    and [] = *trace.natural' s xs* ⟷ *snd* ‘ *set xs* ⊆ {*s*}
⟨*proof*⟩

lemma *eq-Cons-conv*:

shows $trace.natural'\ s\ xs = y\ \#\ ys$
$\longleftrightarrow (\exists xs'\ ys'.\ xs = xs'\ @\ y\ \#\ ys' \land snd\ `\ set\ xs' \subseteq \{s\} \land snd\ y \neq s \land trace.natural'\ (snd\ y)\ ys' = ys)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
and $y\ \#\ ys = trace.natural'\ s\ xs$
$\longleftrightarrow (\exists xs'\ ys'.\ xs = xs'\ @\ y\ \#\ ys' \land snd\ `\ set\ xs' \subseteq \{s\} \land snd\ y \neq s \land trace.natural'\ (snd\ y)\ ys' = ys)$ (**is** *?thesis1*)
⟨*proof*⟩

lemma *eq-append-conv*:
shows $trace.natural'\ s\ xs = ys\ @\ zs$
$\longleftrightarrow (\exists ys'\ zs'.\ xs = ys'\ @\ zs' \land trace.natural'\ s\ ys' = ys \land trace.natural'\ (trace.final'\ s\ ys)\ zs' = zs)$ (**is** *?lhs* = *?rhs*)
and $ys\ @\ zs = trace.natural'\ s\ xs$
$\longleftrightarrow (\exists ys'\ zs'.\ xs = ys'\ @\ zs' \land trace.natural'\ s\ ys' = ys \land trace.natural'\ (trace.final'\ s\ ys)\ zs' = zs)$ (**is** *?thesis1*)
⟨*proof*⟩

lemma *replicate*:
shows $trace.natural'\ s\ (replicate\ i\ as) = (if\ snd\ as = s \lor i = 0\ then\ []\ else\ [as])$
⟨*proof*⟩

lemma *map-natural'*:
shows $trace.natural'\ (sf\ s)\ (map\ (map\text{-}prod\ af\ sf)\ (trace.natural'\ s\ xs))$
$= trace.natural'\ (sf\ s)\ (map\ (map\text{-}prod\ af\ sf)\ xs)$
⟨*proof*⟩

lemma *map-inj-on-sf*:
assumes $inj\text{-}on\ sf\ (insert\ s\ (snd\ `\ set\ xs))$
shows $trace.natural'\ (sf\ s)\ (map\ (map\text{-}prod\ af\ sf)\ xs) = map\ (map\text{-}prod\ af\ sf)\ (trace.natural'\ s\ xs)$
⟨*proof*⟩

lemma *amap-noop*:
assumes $trace.natural'\ s\ xs = map\ (map\text{-}prod\ af\ id)\ zs$
shows $trace.natural'\ s\ zs = zs$
⟨*proof*⟩

lemma *take*:
shows $\exists j \le length\ xs.\ take\ i\ (trace.natural'\ s\ xs) = trace.natural'\ s\ (take\ j\ xs)$
⟨*proof*⟩

lemma *idle-prefix*:
assumes $snd\ `\ set\ xs \subseteq \{s\}$
shows $trace.natural'\ s\ (xs\ @\ ys) = trace.natural'\ s\ ys$
⟨*proof*⟩

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** $ys = xs'\ @\ xs'rest$
⟨*proof*⟩

lemma *aset-conv*:
shows $a \in trace.aset\ (trace.T\ s\ (trace.natural'\ s\ xs)\ v)$
$\longleftrightarrow (\exists s'\ s''.\ (a, s', s'') \in set\ (trace.transitions'\ s\ xs) \land s' \neq s'')$
⟨*proof*⟩

⟨*ML*⟩

definition $natural :: ('a, 's, 'v)\ trace.t \Rightarrow ('a, 's, 'v)\ trace.t$ (‹♮›) **where**

$\natural\sigma = trace.T\ (trace.init\ \sigma)\ (trace.natural'\ (trace.init\ \sigma)\ (trace.rest\ \sigma))\ (trace.term\ \sigma)$

⟨*ML*⟩

lemma *sel*[*simp*]:
shows $trace.init\ (\natural\sigma) = trace.init\ \sigma$
and $trace.rest\ (\natural\sigma) = trace.natural'\ (trace.init\ \sigma)\ (trace.rest\ \sigma)$
and $trace.term\ (\natural\sigma) = trace.term\ \sigma$
⟨*proof*⟩

lemma *simps*:
shows $\natural(trace.T\ s\ []\ v) = trace.T\ s\ []\ v$
and $\natural(trace.T\ s\ ((a,\ s)\ \#\ xs)\ v) = \natural(trace.T\ s\ xs\ v)$
and $\natural(trace.T\ s\ (trace.natural'\ s\ xs)\ v) = \natural(trace.T\ s\ xs\ v)$
⟨*proof*⟩

lemma *idempotent*[*simp*]:
shows $\natural(\natural\sigma) = \natural\sigma$
⟨*proof*⟩

lemma *idle*:
assumes $snd\ `\ set\ xs \subseteq \{s\}$
shows $\natural(trace.T\ s\ xs\ v) = trace.T\ s\ []\ v$
⟨*proof*⟩

lemma *trace-conv*:
shows $\natural(trace.T\ s\ xs\ v) = \natural\sigma \longleftrightarrow trace.init\ \sigma = s \land trace.natural'\ s\ xs = trace.natural'\ s\ (trace.rest\ \sigma) \land trace.term\ \sigma = v$
and $\natural\sigma = \natural(trace.T\ s\ xs\ v) \longleftrightarrow trace.init\ \sigma = s \land trace.natural'\ s\ xs = trace.natural'\ s\ (trace.rest\ \sigma) \land trace.term\ \sigma = v$
⟨*proof*⟩

lemma *map-natural*:
shows $\natural(trace.map\ af\ sf\ vf\ (\natural\sigma)) = \natural(trace.map\ af\ sf\ vf\ \sigma)$
⟨*proof*⟩

lemma *continue*:
shows $\natural(\sigma\ @-_S\ xsv) = \natural\sigma\ @-_S\ (trace.natural'\ (trace.final\ \sigma)\ (fst\ xsv),\ snd\ xsv)$
⟨*proof*⟩

lemma *replicate*:
shows $\natural(trace.T\ s\ (replicate\ i\ as)\ v)$
$= (trace.T\ s\ (if\ snd\ as = s \lor i = 0\ then\ []\ else\ [as])\ v)$
⟨*proof*⟩

lemma *monotone*:
shows *mono* $\natural$
⟨*proof*⟩

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 $t \le u$
assumes $\natural u' = \natural u$
obtains t' **where** $\natural t = \natural t'$ **and** $t' \le u'$

⟨proof⟩

lemma *less-eq-natural*:
assumes $\sigma_1 \leq \natural\sigma_2$
shows $\natural\sigma_1 = \sigma_1$
⟨proof⟩

lemma *map-le*:
assumes $\natural\sigma_1 \leq \natural\sigma_2$
shows $\natural(\mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ \sigma_1) \leq \natural(\mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ \sigma_2)$
⟨proof⟩

lemma *map-inj-on-sf*:
assumes $\mathit{inj\text{-}on}\ \mathit{sf}\ (\mathit{trace.sset}\ \sigma)$
shows $\natural(\mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ \sigma) = \mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ (\natural\sigma)$
⟨proof⟩

lemma *take*:
shows $\exists j.\ \natural(\mathit{trace.take}\ i\ \sigma) = \mathit{trace.take}\ j\ (\natural\sigma)$
⟨proof⟩

lemma *take-natural*:
shows $\natural(\mathit{trace.take}\ i\ (\natural\sigma)) = \mathit{trace.take}\ i\ (\natural\sigma)$
⟨proof⟩

lemma *takeE*:
shows $[\![\sigma_1 = \natural(\mathit{trace.take}\ i\ \sigma_2);\ \bigwedge j.\ [\![\sigma_1 = \mathit{trace.take}\ j\ (\natural\sigma_2)]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$
and $[\![\natural(\mathit{trace.take}\ i\ \sigma_2) = \sigma_1;\ \bigwedge j.\ [\![\sigma_1 = \mathit{trace.take}\ j\ (\natural\sigma_2)]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$
⟨proof⟩

⟨ML⟩

lemma *natural-conv*:
shows $a \in \mathit{trace.aset}\ (\natural\sigma) \longleftrightarrow (\exists s\ s'.\ (a,\ s,\ s') \in \mathit{trace.steps}\ \sigma)$
⟨proof⟩

⟨ML⟩

lemma *natural'*[*simp*]:
shows $\mathit{trace.sset}\ (\mathit{trace.T}\ s_0\ (\mathit{trace.natural'}\ s_0\ \mathit{xs})\ v) = \mathit{trace.sset}\ (\mathit{trace.T}\ s_0\ \mathit{xs}\ v)$
⟨proof⟩

lemma *natural*[*simp*]:
shows $\mathit{trace.sset}\ (\natural\sigma) = \mathit{trace.sset}\ \sigma$
⟨proof⟩

⟨ML⟩

lemma *natural*[*simp*]:
shows $\mathit{trace.vset}\ (\natural\sigma) = \mathit{trace.vset}\ \sigma$
⟨proof⟩

⟨ML⟩

lemma *natural*:
shows $\exists j \leq \mathit{Suc}\ (\mathit{length}\ (\mathit{trace.rest}\ \sigma)).\ \mathit{trace.take}\ i\ (\natural\sigma) = \natural(\mathit{trace.take}\ j\ \sigma)$
⟨proof⟩

lemma *naturalE*:
  shows $[\![\sigma_1 = \textit{trace.take } i\ (\natural\sigma_2);\ \bigwedge j.\ [\![j \leq \textit{Suc}\ (\textit{length}\ (\textit{trace.rest}\ \sigma_2));\ \sigma_1 = \natural(\textit{trace.take } j\ \sigma_2)]\!] \Longrightarrow \textit{thesis}]\!] \Longrightarrow \textit{thesis}$
    and $[\![\textit{trace.take } i\ (\natural\sigma_2) = \sigma_1;\ \bigwedge j.\ [\![j \leq \textit{Suc}\ (\textit{length}\ (\textit{trace.rest}\ \sigma_2));\ \natural(\textit{trace.take } j\ \sigma_2) = \sigma_1]\!] \Longrightarrow \textit{thesis}]\!] \Longrightarrow \textit{thesis}$
⟨*proof*⟩

⟨*ML*⟩

lemma *steps′-alt-def*:
  shows $\textit{trace.steps}'\ s\ xs = \textit{set}\ (\textit{trace.transitions}'\ s\ (\textit{trace.natural}'\ s\ xs))$
⟨*proof*⟩

⟨*ML*⟩

lemma *natural′*:
  shows $\textit{trace.steps}'\ s\ (\textit{trace.natural}'\ s\ xs) = \textit{trace.steps}'\ s\ xs$
⟨*proof*⟩

lemma *asetD*:
  assumes $\textit{trace.steps}\ \sigma \subseteq r$
  shows $\forall a.\ a \in \textit{trace.aset}\ (\natural\sigma) \longrightarrow a \in \textit{fst}\ `\ r$
⟨*proof*⟩

lemma *range-initE*:
  assumes $\textit{trace.steps}'\ s_0\ xs \subseteq \textit{range af} \times \textit{range sf} \times \textit{range sf}$
  assumes $(a, s, s') \in \textit{trace.steps}'\ s_0\ xs$
  obtains s_0' **where** $s_0 = \textit{sf}\ s_0'$
⟨*proof*⟩

lemma *map-range-conv*:
  shows $\textit{trace.steps}'\ (\textit{sf}\ s)\ xs \subseteq \textit{range af} \times \textit{range sf} \times \textit{range sf}$
    $\longleftrightarrow (\exists xs'.\ \textit{trace.natural}'\ (\textit{sf}\ s)\ xs = \textit{map}\ (\textit{map-prod af sf})\ xs')$ (**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma *step-conv*:
  shows $\textit{trace.steps}'\ s\ xs = \{x\}$
    $\longleftrightarrow \textit{fst}\ (\textit{snd}\ x) = s \wedge \textit{fst}\ (\textit{snd}\ x) \neq \textit{snd}\ (\textit{snd}\ x)$
      $\wedge (\exists ys\ zs.\ \textit{snd}\ `\ \textit{set}\ ys \subseteq \{s\} \wedge \textit{snd}\ `\ \textit{set}\ zs \subseteq \{\textit{snd}\ (\textit{snd}\ x)\}$
          $\wedge\ xs = ys\ @\ [(\textit{fst}\ x, \textit{snd}\ (\textit{snd}\ x))]\ @\ zs)$ (**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

⟨*ML*⟩

interpretation *stuttering*: *galois.image-vimage-idempotent* ♮
⟨*proof*⟩

abbreviation *stuttering-equiv-syn* :: $('a, 's, 'v)\ \textit{trace.t} \Rightarrow ('a, 's, 'v)\ \textit{trace.t} \Rightarrow \textit{bool}$ (**infix** ‹$\simeq_S$› *50*) **where**
  $\sigma_1 \simeq_S \sigma_2 \equiv \textit{trace.stuttering.equivalent}\ \sigma_1\ \sigma_2$

⟨*ML*⟩

lemma *cl*:
  shows $\textit{trace.stuttering.cl}\ (\textit{downwards.cl}\ P) = \textit{downwards.cl}\ (\textit{trace.stuttering.cl}\ P)$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *closed*:
  assumes $P \in \textit{downwards.closed}$

shows *trace.stuttering.cl P* $\in$ *downwards.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *downwards-imp*: — Abadi and Plotkin (1993, p13)
assumes $P \in$ *trace.stuttering.closed*
assumes $Q \in$ *trace.stuttering.closed*
shows *downwards.imp P Q* $\in$ *trace.stuttering.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*:
shows *snd* ‘ *set xs* $\subseteq \{s\} \Longrightarrow$ *trace.T s* (*xs* @ *ys*) *v* $\simeq_S$ *trace.T s ys v*
and *snd* ‘ *set ys* $\subseteq$ {*trace.final′ s xs*} $\Longrightarrow$ *trace.T s* (*xs* @ *ys*) *v* $\simeq_S$ *trace.T s xs v*
and *snd* ‘ *set xs* $\subseteq$ {*snd x*} $\Longrightarrow$ *trace.T s* (*x* # *xs* @ *ys*) *v* $\simeq_S$ *trace.T s* (*x* # *ys*) *v*
⟨*proof*⟩

lemma *append-cong*:
assumes $s = s'$
assumes *trace.natural′ s xs* = *trace.natural′ s xs′*
assumes *trace.natural′* (*trace.final′ s xs*) *ys* = *trace.natural′* (*trace.final′ s xs*) *ys′*
assumes $v = v'$
shows *trace.T s* (*xs* @ *ys*) *v* $\simeq_S$ *trace.T s′* (*xs′* @ *ys′*) *v′*
⟨*proof*⟩

lemma *E*:
assumes *trace.T s xs v* $\simeq_S$ *trace.T s′ xs′ v′*
obtains *trace.natural′ s xs* = *trace.natural′ s′ xs′* **and** $s = s'$ **and** $v = v'$
⟨*proof*⟩

lemma *append-conv*:
shows *trace.T s* (*xs* @ *ys*) *v* $\simeq_S \sigma$
$\longleftrightarrow$ ($\exists$ *xs′ ys′*. σ = *trace.T s* (*xs′* @ *ys′*) *v* $\wedge$ *trace.natural′ s xs* = *trace.natural′ s xs′*
$\wedge$ *trace.natural′* (*trace.final′ s xs*) *ys* = *trace.natural′* (*trace.final′ s xs*) *ys′*) (**is** *?thesis1*)
and $\sigma \simeq_S$ *trace.T s* (*xs* @ *ys*) *v*
$\longleftrightarrow$ ($\exists$ *xs′ ys′*. σ = *trace.T s* (*xs′* @ *ys′*) *v* $\wedge$ *trace.natural′ s xs* = *trace.natural′ s xs′*
$\wedge$ *trace.natural′* (*trace.final′ s xs*) *ys* = *trace.natural′* (*trace.final′ s xs*) *ys′*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *map*:
assumes $\sigma_1 \simeq_S \sigma_2$
shows *trace.map af sf vf* $\sigma_1 \simeq_S$ *trace.map af sf vf* σ_2
⟨*proof*⟩

lemma *steps*:
assumes $\sigma_1 \simeq_S \sigma_2$
shows *trace.steps* σ_1 = *trace.steps* σ_2
⟨*proof*⟩

⟨*ML*⟩

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".

⟨*ML*⟩

definition *cl* :: $('a, 's, 'v)$ *trace.t set* ⇒ $('a, 's, 'v)$ *trace.t set* **where**
  cl P = *downwards.cl* (*trace.stuttering.cl P*)

⟨*ML*⟩

interpretation *spec*: *closure-powerset-distributive raw.spec.cl*
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*[*simp*]:
  shows *raw.spec.cl* {} = {}
⟨*proof*⟩

⟨*ML*⟩

lemma *I*:
  assumes $P \in$ *downwards.closed*
  assumes $P \in$ *trace.stuttering.closed*
  shows $P \in$ *raw.spec.closed*
⟨*proof*⟩

lemma *empty*[*intro*]:
  shows {} ∈ *raw.spec.closed*
⟨*proof*⟩

lemma *downwards-closed*:
  assumes $P \in$ *raw.spec.closed*
  shows $P \in$ *downwards.closed*
⟨*proof*⟩

lemma *stuttering-closed*:
  assumes $P \in$ *raw.spec.closed*
  shows $P \in$ *trace.stuttering.closed*
⟨*proof*⟩

lemma *downwards-imp*:
  assumes $P \in$ *raw.spec.closed*
  assumes $Q \in$ *raw.spec.closed*
  shows *downwards.imp P Q* ∈ *raw.spec.closed*
⟨*proof*⟩

lemma *heyting-downwards-imp*:
  assumes $P \in$ *raw.spec.closed*
  shows $P \subseteq$ *downwards.imp Q R* $\longleftrightarrow P \cap Q \subseteq R$
⟨*proof*⟩

lemma *takeE*:
  assumes $\sigma \in P$
  assumes $P \in$ *raw.spec.closed*
  shows *trace.take i* $\sigma \in P$
⟨*proof*⟩

⟨*ML*⟩

typedef ('*a*, '*s*, '*v*) *spec* = *raw.spec.closed* :: ('*a*, '*s*, '*v*) *trace.t set set*
morphisms *unMkS MkS*
⟨*proof*⟩

setup-lifting *type-definition-spec*

instantiation *spec* :: (*type*, *type*, *type*) *complete-distrib-lattice*
begin

lift-definition *bot-spec* :: ('*a*, '*s*, '*v*) *spec* **is** *empty* ⟨*proof*⟩
lift-definition *top-spec* :: ('*a*, '*s*, '*v*) *spec* **is** *UNIV* ⟨*proof*⟩
lift-definition *sup-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *sup* ⟨*proof*⟩
lift-definition *inf-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *inf* ⟨*proof*⟩
lift-definition *less-eq-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩
lift-definition *less-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ *bool* **is** *less* ⟨*proof*⟩
lift-definition *Inf-spec* :: ('*a*, '*s*, '*v*) *spec set* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *Inf* ⟨*proof*⟩
lift-definition *Sup-spec* :: ('*a*, '*s*, '*v*) *spec set* ⇒ ('*a*, '*s*, '*v*) *spec* **is** λ*X*. *Sup X* ⊔ *raw.spec.cl* {} ⟨*proof*⟩

instance
⟨*proof*⟩

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 *Sup*s and *Inf*s 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

⟨*ML*⟩

lemmas *antisym* = *antisym*[**where** '*a*=('*a*, '*s*, '*v*) *spec*]
lemmas *eq-iff* = *order.eq-iff*[**where** '*a*=('*a*, '*s*, '*v*) *spec*]

⟨*ML*⟩

8.3 Irreducible elements

The irreducible elements of $('a, 's, 'v)$ *trace.t* are the closures of singletons.

⟨*ML*⟩

definition *singleton* :: $('a, 's, 'v)$ *trace.t* ⇒ $('a, 's, 'v)$ *trace.t set* **where**
singleton σ = *raw.spec.cl* $\{\sigma\}$

lemma *singleton-le-conv*:
shows *raw.singleton* $\sigma_1 \leq$ *raw.singleton* $\sigma_2 \longleftrightarrow \natural\sigma_1 \leq \natural\ \sigma_2$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lift-definition *singleton* :: $('a, 's, 'v)$ *trace.t* ⇒ $('a, 's, 'v)$ *spec* (‹⦇-⦈›) **is** *raw.singleton*
⟨*proof*⟩

abbreviation *singleton-trace-syn* :: $'s \Rightarrow ('a \times 's)$ *list* ⇒ $'v$ *option* ⇒ $('a, 's, 'v)$ *spec* (‹⦇-, -, -⦈›) **where**
⦇*s*, *xs*, *v*⦈ ≡ ⦇*trace.T s xs v*⦈

⟨*ML*⟩

lemma *Sup-prime*:
shows *Sup-prime* ⦇σ⦈
⟨*proof*⟩

lemma *nchotomy*:
shows $\exists X \in$ *raw.spec.closed*. $x = \bigsqcup$(*spec.singleton* ‘ X)
⟨*proof*⟩

lemmas *exhaust* = *bexE*[*OF spec.singleton.nchotomy*]

lemma *collapse*[*simp*]:
shows $\bigsqcup$(*spec.singleton* ‘ $\{\sigma.$ ⦇σ⦈ $\leq P\}) = P$
⟨*proof*⟩

lemmas *not-bot* = *Sup-prime-not-bot*[*OF spec.singleton.Sup-prime*] — Non-triviality

⟨*ML*⟩

lemma *singleton-le-ext-conv*:
shows $P \leq Q \longleftrightarrow (\forall \sigma.$ ⦇σ⦈ $\leq P \longrightarrow$ ⦇σ⦈ $\leq Q)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemmas *singleton-le-conv* = *raw.singleton-le-conv*[*transferred*]
lemmas *singleton-le-extI* = *iffD2*[*OF spec.singleton-le-ext-conv*, *rule-format*]

lemma *singleton-eq-conv*[*simp*]:
shows ⦇σ⦈ = ⦇σ'⦈ $\longleftrightarrow \sigma \simeq_S \sigma'$
⟨*proof*⟩

lemma *singleton-cong*:
assumes $\sigma \simeq_S \sigma'$
shows ⦇σ⦈ = ⦇σ'⦈
⟨*proof*⟩

⟨*ML*⟩

named-theorems *le-conv* ‹ *simplification rules for* ‹$\langle\!|\sigma|\!\rangle \leq$ *const* ...› ›

lemmas *antisym* = *antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*]

lemmas *top* = *spec.singleton.collapse*[*of* ⊤, *simplified*, *symmetric*]

lemma *monotone*:
  shows *mono spec.singleton*
⟨*proof*⟩

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 $\langle\!|s,\ xs,\ v|\!\rangle \leq \langle\!|s,\ trace.natural'\ s\ xs,\ v|\!\rangle$
    and snd ‘ $set\ xs \subseteq \{s\} \Longrightarrow \langle\!|s,\ xs\ @\ ys,\ v|\!\rangle = \langle\!|s,\ ys,\ v|\!\rangle$
    and snd ‘ $set\ ys \subseteq \{trace.final'\ s\ xs\} \Longrightarrow \langle\!|s,\ xs\ @\ ys,\ v|\!\rangle = \langle\!|s,\ xs,\ v|\!\rangle$
    and snd ‘ $set\ xs \subseteq \{snd\ x\} \Longrightarrow \langle\!|s,\ x\ \#\ xs\ @\ ys,\ v|\!\rangle = \langle\!|s,\ x\ \#\ ys,\ v|\!\rangle$
    and $\langle\!|s,\ (a,\ s)\ \#\ xs,\ v|\!\rangle = \langle\!|s,\ xs,\ v|\!\rangle$
⟨*proof*⟩

lemma *Cons*: — self-applies, not usable by *simp*
  assumes snd ‘ $set\ as \subseteq \{s'\}$
  shows $\langle\!|s,\ (a,\ s')\ \#\ as,\ v|\!\rangle = \langle\!|s,\ [(a,\ s')],\ v|\!\rangle$
⟨*proof*⟩

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 $\bigsqcap(spec.singleton$ ‘ $X) = \bigsqcup(spec.singleton$ ‘ $\{\sigma.\ \forall\sigma_1{\in}X.\ \sigma \leq \natural\sigma_1\})$
⟨*proof*⟩

lemmas *inf* = *spec.singleton.Inf*[**where** $X{=}\{\sigma_1,\ \sigma_2\}$, *simplified*] **for** $\sigma_1\ \sigma_2$

lemma *less-eq-Some*[*simp*]:
  shows $\langle\!|s,\ xs,\ Some\ v|\!\rangle \leq \langle\!|\sigma|\!\rangle$
    $\longleftrightarrow trace.term\ \sigma = Some\ v \wedge trace.init\ \sigma = s \wedge trace.natural'\ s\ (trace.rest\ \sigma) = trace.natural'\ s\ xs$
⟨*proof*⟩

lemma *less-eq-None*:
  shows [*iff*]: $\langle\!|s,\ xs,\ None|\!\rangle \leq \langle\!|s,\ xs,\ v'|\!\rangle$
⟨*proof*⟩

lemma *map-cong*:
  assumes $\bigwedge a.\ a \in trace.aset\ (\natural\sigma') \Longrightarrow af\ a = af'\ a$
  assumes $\bigwedge x.\ x \in trace.sset\ (\natural\sigma') \Longrightarrow sf\ x = sf'\ x$
  assumes $\bigwedge v.\ v \in trace.vset\ (\natural\sigma') \Longrightarrow vf\ v = vf'\ v$
  assumes $\natural\sigma = \natural\sigma'$
  shows $\langle\!|trace.map\ af\ sf\ vf\ \sigma|\!\rangle = \langle\!|trace.map\ af'\ sf'\ vf'\ \sigma'|\!\rangle$

⟨*proof*⟩

lemma *map-le*:
  assumes $\langle\!|\sigma|\!\rangle \le \langle\!|\sigma'|\!\rangle$
  shows $\langle\!|\mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ \sigma|\!\rangle \le \langle\!|\mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ \sigma'|\!\rangle$
⟨*proof*⟩

lemma *takeI*:
  assumes $\langle\!|\sigma|\!\rangle \le P$
  shows $\langle\!|\mathit{trace.take}\ i\ \sigma|\!\rangle \le P$
⟨*proof*⟩

⟨*ML*⟩

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*]

⟨*ML*⟩

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*.

⟨*ML*⟩

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v)\ \mathit{spec} \Rightarrow ('b, 't, 'w)\ \mathit{spec}$ **where**
  $\mathit{map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ P = \bigsqcup(\mathit{spec.singleton}\ `\ \mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ `\ \{\sigma.\ \langle\!|\sigma|\!\rangle \le P\})$

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w)\ \mathit{spec} \Rightarrow ('a, 's, 'v)\ \mathit{spec}$ **where**
  $\mathit{invmap}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ P = \bigsqcup(\mathit{spec.singleton}\ `\ \mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ -`\ \{\sigma.\ \langle\!|\sigma|\!\rangle \le P\})$

abbreviation *amap* ::$('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v)\ \mathit{spec} \Rightarrow ('b, 's, 'v)\ \mathit{spec}$ **where**
  $\mathit{amap}\ \mathit{af} \equiv \mathit{spec.map}\ \mathit{af}\ \mathit{id}\ \mathit{id}$
abbreviation *ainvmap* ::$('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v)\ \mathit{spec} \Rightarrow ('a, 's, 'v)\ \mathit{spec}$ **where**
  $\mathit{ainvmap}\ \mathit{af} \equiv \mathit{spec.invmap}\ \mathit{af}\ \mathit{id}\ \mathit{id}$
abbreviation *smap* ::$('s \Rightarrow 't) \Rightarrow ('a, 's, 'v)\ \mathit{spec} \Rightarrow ('a, 't, 'v)\ \mathit{spec}$ **where**
  $\mathit{smap}\ \mathit{sf} \equiv \mathit{spec.map}\ \mathit{id}\ \mathit{sf}\ \mathit{id}$
abbreviation *sinvmap* ::$('s \Rightarrow 't) \Rightarrow ('a, 't, 'v)\ \mathit{spec} \Rightarrow ('a, 's, 'v)\ \mathit{spec}$ **where**
  $\mathit{sinvmap}\ \mathit{sf} \equiv \mathit{spec.invmap}\ \mathit{id}\ \mathit{sf}\ \mathit{id}$
abbreviation *vmap* ::$('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v)\ \mathit{spec} \Rightarrow ('a, 's, 'w)\ \mathit{spec}$ **where** — aka *liftM*
  $\mathit{vmap}\ \mathit{vf} \equiv \mathit{spec.map}\ \mathit{id}\ \mathit{id}\ \mathit{vf}$
abbreviation *vinvmap* ::$('v \Rightarrow 'w) \Rightarrow ('a, 's, 'w)\ \mathit{spec} \Rightarrow ('a, 's, 'v)\ \mathit{spec}$ **where**
  $\mathit{vinvmap}\ \mathit{vf} \equiv \mathit{spec.invmap}\ \mathit{id}\ \mathit{id}\ \mathit{vf}$

interpretation *map-invmap*: *galois.complete-lattice-distributive-class*
  spec.map af sf vf
  spec.invmap af sf vf **for** *af sf vf*
⟨*proof*⟩

⟨*ML*⟩

lemma *map-le-conv*[*spec.singleton.le-conv*]:
  shows $\langle\!|\sigma|\!\rangle \le \mathit{spec.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ P \longleftrightarrow (\exists\sigma'.\ \langle\!|\sigma'|\!\rangle \le P \wedge \langle\!|\sigma|\!\rangle \le \langle\!|\mathit{trace.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ \sigma'|\!\rangle)$

⟨*proof*⟩

lemma *invmap-le-conv*[*spec.singleton.le-conv*]:
  shows $\langle\!\langle\sigma\rangle\!\rangle \le$ *spec.invmap af sf vf P* $\longleftrightarrow \langle\!\langle$*trace.map af sf vf* $\sigma\rangle\!\rangle \le P$
⟨*proof*⟩

⟨*ML*⟩

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* $\langle\!\langle\sigma\rangle\!\rangle = \langle\!\langle$*trace.map af sf vf* $\sigma\rangle\!\rangle$
⟨*proof*⟩

lemma *top*:
  assumes *surj af*
  assumes *surj sf*
  assumes *surj vf*
  shows *spec.map af sf vf* $\top = \top$
⟨*proof*⟩

lemma *id*:
  shows *spec.map id id id P* = *P*
    and *spec.map* $(\lambda x.\ x)$ $(\lambda x.\ x)$ $(\lambda x.\ x)$ $P = P$
⟨*proof*⟩

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 *spec.map af sf vf* (*spec.map ag sg vg P*) = *spec.map* $(\lambda a.$ *af* (*ag a*)) $(\lambda s.$ *sf* (*sg s*)) $(\lambda v.$ *vf* (*vg v*)) *P* (**is** *?thesis1*)
⟨*proof*⟩

lemmas *map* = *spec.map.comp*

lemma *inf-distr*:
  shows *spec.map af sf vf P* $\sqcap$ *Q* = *spec.map af sf vf* (*P* $\sqcap$ *spec.invmap af sf vf Q*) (**is** *?lhs* = *?rhs*)
    and *Q* $\sqcap$ *spec.map af sf vf P* = *spec.map af sf vf* (*spec.invmap af sf vf Q* $\sqcap$ *P*) (**is** *?thesis1*)
⟨*proof*⟩

⟨*ML*⟩

lemma *comp*:
  shows *spec.smap sf* $\circ$ *spec.smap sg* = *spec.smap* (*sf* $\circ$ *sg*)
    and *spec.smap sf* (*spec.smap sg P*) = *spec.smap* $(\lambda s.$ *sf* (*sg s*)) *P*
⟨*proof*⟩

⟨*ML*⟩

lemmas *bot* = *spec.map-invmap.upper-bot*
lemmas *top* = *spec.map-invmap.upper-top*

lemmas *monotone* = *spec.map-invmap.monotone-upper*
lemmas *mono* = *monotoneD*[*OF 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* $\langle\!|\sigma|\!\rangle = \bigsqcup($*spec.singleton* ‘ $\{\sigma'.\ \langle\!|$*trace.map af sf vf* $\sigma'|\!\rangle \leq \langle\!|\sigma|\!\rangle\})$
⟨*proof*⟩

lemma *id*:
  shows *spec.invmap id id id P* = *P*
    and *spec.invmap* $(\lambda x.\ x)$ $(\lambda x.\ x)$ $(\lambda x.\ x)$ *P* = *P*
⟨*proof*⟩

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* ∘ *spec.invmap ag sg vg* = *spec.invmap* (*ag* ∘ *af*) (*sg* ∘ *sf*) (*vg* ∘ *vf*) (**is** *?thesis1*)
⟨*proof*⟩

lemmas *invmap* = *spec.invmap.comp*

lemma *invmap-inf-distr-le*:
  fixes *af* :: *'a* ⇒ *'b*
  fixes *sf* :: *'s* ⇒ *'t*
  fixes *vf* :: *'v* ⇒ *'w*
  shows *spec.invmap af sf vf P* ⊓ *Q* ≤ *spec.invmap af sf vf* (*P* ⊓ *spec.map af sf vf Q*)
    and *Q* ⊓ *spec.invmap af sf vf P* ≤ *spec.invmap af sf vf* (*spec.map af sf vf Q* ⊓ *P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *invmap-le*: — *af* = *id* in *spec.invmap*
  shows *spec.amap af* (*spec.invmap id sf vf P*) ≤ *spec.invmap id sf vf* (*spec.amap af P*)
⟨*proof*⟩

lemma *surj-invmap*: — *af* = *id* in *spec.invmap*
  fixes *P* :: (*'a*, *'t*, *'w*) *spec*
  fixes *af* :: *'a* ⇒ *'b*
  fixes *sf* :: *'s* ⇒ *'t*
  fixes *vf* :: *'v* ⇒ *'w*
  assumes *surj af*
  shows *spec.amap af* (*spec.invmap id sf vf P*) = *spec.invmap id sf vf* (*spec.amap af P*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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).

⟨*ML*⟩

definition *idle* :: ($'a$, $'s$, $'v$) *spec* **where**
  $idle = (\bigsqcup s.\ \llparenthesis s,\ [],\ None\rrparenthesis)$

named-theorems *idle-le* ‹ *rules for* ‹*spec.idle* ≤ *const* ...› ›

⟨*ML*⟩

lemma *idle-le-conv*[*spec.singleton.le-conv*]:
  shows $\llparenthesis\sigma\rrparenthesis \le spec.idle \longleftrightarrow trace.steps\ \sigma = \{\} \wedge trace.term\ \sigma = None$
⟨*proof*⟩

⟨*ML*⟩

lemma *minimal-le*:
  shows $\llparenthesis s,\ [],\ None\rrparenthesis \le spec.idle$
⟨*proof*⟩

lemma *map-le*[*spec.idle-le*]:
  assumes $spec.idle \le P$
  assumes *surj sf*
  shows $spec.idle \le spec.map\ af\ sf\ vf\ P$
⟨*proof*⟩

lemma *invmap-le*:
  assumes $spec.idle \le P$
  shows $spec.idle \le spec.invmap\ af\ sf\ vf\ P$
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-alt-def*:
  shows *spec.map-invmap.cl - - - af sf vf P*
    $= \bigsqcup\{\llparenthesis\sigma\rrparenthesis\ |\sigma\ \sigma'.\ \llparenthesis\sigma'\rrparenthesis \le P \wedge \llparenthesis trace.map\ af\ sf\ vf\ \sigma\rrparenthesis \le \llparenthesis trace.map\ af\ sf\ vf\ \sigma'\rrparenthesis\}$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cl-le-conv*[*spec.singleton.le-conv*]:
  shows $\llparenthesis\sigma\rrparenthesis \le$ *spec.map-invmap.cl - - - af sf vf P* $\longleftrightarrow \llparenthesis trace.map\ af\ sf\ vf\ \sigma\rrparenthesis \le spec.map\ af\ sf\ vf\ P$
⟨*proof*⟩

⟨*ML*⟩

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 $idle \le action\ a\ F$; see §12.1.1.

⟨*ML*⟩

definition *action* :: ($'v \times 'a \times 's \times 's$) *set* ⇒ ($'a$, $'s$, $'v$) *spec* **where**
  $action\ F = (\bigsqcup(v,\ a,\ s,\ s')\in F.\ \llparenthesis s,\ [(a,\ s')],\ Some\ v\rrparenthesis) \sqcup spec.idle$

definition *guard* :: ('s ⇒ bool) ⇒ ('a, 's, unit) *spec* **where**
guard g = spec.action ({()} × *UNIV* × *Diag g*)

definition *return* :: 'v ⇒ ('a, 's, 'v) *spec* **where**
return v = spec.action ({v} × *UNIV* × *Id*)

abbreviation (*input*) *read* :: ('s ⇒ 'v option) ⇒ ('a, 's, 'v) *spec* **where**
read f ≡ spec.action {(v, a, s, s) |a s v. f s = Some v}

abbreviation (*input*) *write* :: 'a ⇒ ('s ⇒ 's) ⇒ ('a, 's, unit) *spec* **where**
write a f ≡ spec.action {((), a, s, f s) |s. True}

lemma *action-le*[*case-names idle step*]:
assumes *spec.idle* ≤ *P*
assumes ⋀*v a s s'*. (v, a, s, s') ∈ F ⟹ ⦉s, [(a, s')], Some v⦊ ≤ P
shows *spec.action F* ≤ *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *action-le*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.action F*
⟨*proof*⟩

lemma *guard-le*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.guard g*
⟨*proof*⟩

lemma *return-le*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.return v*
⟨*proof*⟩

⟨*ML*⟩

lemma *action-le*:
fixes *F* :: ('v × 'a × 's × 's) *set*
shows *spec.map af sf vf* (*spec.action F*) ≤ *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) ‘ *F*)
⟨*proof*⟩

lemma *action*:
fixes *F* :: ('v × 'a × 's × 's) *set*
shows *spec.map af sf vf* (*spec.action F*) ⊔ *spec.idle*
= *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) ‘ *F*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *surj-sf-action*:
assumes *surj sf*
shows *spec.map af sf vf* (*spec.action F*) = *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) ‘ *F*)
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
shows *spec.action* {} = *spec.idle*
⟨*proof*⟩

lemma *idleI*:

assumes *snd ‘ set xs* $\subseteq$ $\{s\}$
shows $\langle\!|$*s*, *xs*, *None*$|\!\rangle \leq$ *spec.action F*
⟨*proof*⟩

lemma *stepI*:
assumes $(v, a, s, s') \in F$
assumes $\forall v''.\ w =$ *Some* $v'' \longrightarrow v'' = v$
shows $\langle\!|$*s*, $[(a, s')]$, *w*$|\!\rangle \leq$ *spec.action F*
⟨*proof*⟩

lemma *stutterI*:
assumes $(v, a, s, s) \in F$
shows $\langle\!|$*s*, [], *Some v*$|\!\rangle \leq$ *spec.action F*
⟨*proof*⟩

lemma *stutter-stepI*:
assumes $(v, a, s, s) \in F$
shows $\langle\!|$*s*, $[(b, s)]$, *Some v*$|\!\rangle \leq$ *spec.action F*
⟨*proof*⟩

lemma *stutter-stepsI*:
assumes $(v, a, s, s) \in F$
assumes *snd ‘ set xs* $\subseteq$ $\{s\}$
shows $\langle\!|$*s*, *xs*, *Some v*$|\!\rangle \leq$ *spec.action F*
⟨*proof*⟩

lemma *monotone*:
shows *mono spec.action*
⟨*proof*⟩

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* $(\bigcup X) = (\bigsqcup F{\in}X.$ *spec.action F*$) \sqcup$ *spec.idle*
⟨*proof*⟩

lemma
shows *SUP*: *spec.action* $(\bigcup x{\in}X.\ F\ x) = (\bigsqcup x{\in}X.$ *spec.action* $(F\ x)) \sqcup$ *spec.idle*
and *SUP-not-empty*: $X \neq \{\} \Longrightarrow$ *spec.action* $(\bigcup x{\in}X.\ F\ x) = (\bigsqcup x{\in}X.$ *spec.action* $(F\ x))$
⟨*proof*⟩

lemma *sup*:
shows *spec.action* $(F \cup G) =$ *spec.action F* $\sqcup$ *spec.action G*
⟨*proof*⟩

lemma *Inf-le*:
shows *spec.action* $(\bigcap Fs) \leq \bigsqcap($*spec.action ‘ Fs*$)$
⟨*proof*⟩

lemma *inf-le*:
shows *spec.action* $(F \cap G) \leq$ *spec.action F* $\sqcap$ *spec.action G*
⟨*proof*⟩

lemma *stutter-agents-le*:

assumes $[\![A \neq \{\}; r \neq \{\}]\!] \Longrightarrow B \neq \{\}$
assumes $r \subseteq Id$
shows $spec.action\ (\{v\} \times A \times r) \leq spec.action\ (\{v\} \times B \times r)$
⟨*proof*⟩

lemma *read-agents*:
assumes $A \neq \{\}$
assumes $B \neq \{\}$
assumes $r \subseteq Id$
shows $spec.action\ (\{v\} \times A \times r) = spec.action\ (\{v\} \times B \times r)$
⟨*proof*⟩

lemma *invmap-le*: — A typical refinement
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
shows $spec.action\ (map\text{-}prod\ vf\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf))\ -`\ F) \leq spec.invmap\ af\ sf\ vf\ (spec.action\ F)$
⟨*proof*⟩

⟨*ML*⟩

lemma *action-le-conv*:
shows $\langle\!|\sigma|\!\rangle \leq spec.action\ F$
$\longleftrightarrow (trace.steps\ \sigma = \{\} \wedge case\text{-}option\ True\ (\lambda v.\ \exists a.\ (v,\ a,\ trace.init\ \sigma,\ trace.init\ \sigma) \in F)\ (trace.term\ \sigma))$
$\vee (\exists x{\in}F.\ trace.steps\ \sigma = \{snd\ x\} \wedge case\text{-}option\ True\ ((=)\ (fst\ x))\ (trace.term\ \sigma))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *action-Some-leE*:
assumes $\langle\!|\sigma|\!\rangle \leq spec.action\ F$
assumes $trace.term\ \sigma = Some\ v$
obtains x
where $x \in F$
and $trace.init\ \sigma = fst\ (snd\ (snd\ x))$
and $trace.final\ \sigma = snd\ (snd\ (snd\ x))$
and $trace.steps\ \sigma \subseteq \{snd\ x\}$
and $v = fst\ x$
⟨*proof*⟩

lemma *action-not-idle-leE*:
assumes $\langle\!|\sigma|\!\rangle \leq spec.action\ F$
assumes $\natural\sigma \neq trace.T\ (trace.init\ \sigma)\ []\ None$
obtains x
where $x \in F$
and $trace.init\ \sigma = fst\ (snd\ (snd\ x))$
and $trace.final\ \sigma = snd\ (snd\ (snd\ x))$
and $trace.steps\ \sigma \subseteq \{snd\ x\}$
and $case\text{-}option\ True\ ((=)\ (fst\ x))\ (trace.term\ \sigma)$
⟨*proof*⟩

lemma *action-not-idle-le-splitE*:
assumes $\langle\!|\sigma|\!\rangle \leq spec.action\ F$
assumes $\natural\sigma \neq trace.T\ (trace.init\ \sigma)\ []\ None$
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* σ
and *snd ' set ys* $\subseteq$ {*trace.init* σ}
and *snd ' set zs* $\subseteq$ {*trace.final* σ}
and *trace.rest* σ = *ys* @ [(*a*, *trace.final* σ)] @ *zs*
and *case-option True* ((=) *v*) (*trace.term* σ)
⟨*proof*⟩

lemma *guard-le-conv*[*spec.singleton.le-conv*]:
shows ⦇σ⦈ ≤ *spec.guard g* ⟷ *trace.steps* σ = {} ∧ (*case-option True* ⟨*g* (*trace.init* σ)⟩ (*trace.term* σ))
⟨*proof*⟩

lemma *return-le-conv*[*spec.singleton.le-conv*]:
shows ⦇σ⦈ ≤ *spec.return v*
⟷ *trace.steps* σ = {} ∧ (*case-option True* ((=) *v*) (*trace.term* σ))
⟨*proof*⟩

⟨*ML*⟩

lemma *mono-stronger*:
assumes ⋀*v a s s'*. ⟦(*v*, *a*, *s*, *s'*) ∈ *F*; *s* ≠ *s'*⟧ ⟹ (*v*, *a*, *s*, *s'*) ∈ *F'*
assumes ⋀*v a s*. (*v*, *a*, *s*, *s*) ∈ *F* ⟹ ∃ *a'*. (*v*, *a'*, *s*, *s*) ∈ *F'*
shows *spec.action F* ≤ *spec.action F'*
⟨*proof*⟩

lemma *cong*:
assumes ⋀*v a s s'*. *s* ≠ *s'* ⟹ (*v*, *a*, *s*, *s'*) ∈ *F* ⟷ (*v*, *a*, *s*, *s'*) ∈ *F'*
assumes ⋀*v a s*. (*v*, *a*, *s*, *s*) ∈ *F* ⟹ ∃ *a'*. (*v*, *a'*, *s*, *s*) ∈ *F'*
assumes ⋀*v a s*. (*v*, *a*, *s*, *s*) ∈ *F'* ⟹ ∃ *a'*. (*v*, *a'*, *s*, *s*) ∈ *F*
shows *spec.action F* = *spec.action F'*
⟨*proof*⟩

lemma *le-actionD*:
assumes *spec.action F* ≤ *spec.action F'*
shows ⟦(*v*, *a*, *s*, *s'*) ∈ *F*; *s* ≠ *s'*⟧ ⟹ (*v*, *a*, *s*, *s'*) ∈ *F'*
and (*v*, *a*, *s*, *s*) ∈ *F* ⟹ ∃ *a'*. (*v*, *a'*, *s*, *s*) ∈ *F'*
⟨*proof*⟩

lemma *eq-action-conv*:
shows *spec.action F* = *spec.action F'*
⟷ (∀ *v a s s'*. *s* ≠ *s'* ⟶ (*v*, *a*, *s*, *s'*) ∈ *F* ⟷ (*v*, *a*, *s*, *s'*) ∈ *F'*)
∧ (∀ *v a s*. (*v*, *a*, *s*, *s*) ∈ *F* ⟶ (∃ *a'*. (*v*, *a'*, *s*, *s*) ∈ *F'*))
∧ (∀ *v a s*. (*v*, *a*, *s*, *s*) ∈ *F'* ⟶ (∃ *a'*. (*v*, *a'*, *s*, *s*) ∈ *F*))
⟨*proof*⟩

⟨*ML*⟩

lemma *return-alt-def*:
assumes *A* ≠ {}
shows *spec.return v* = *spec.action* ({*v*} × *A* × *Id*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cong*:
assumes ⋀*v a s s'*. (*v*, *a*, *s*, *s'*) ∈ *F* ⟹ *s'* = *s*
assumes ⋀*v s*. *v* ∈ *fst ' F* ⟹ ∃ *a*. (*v*, *a*, *s*, *s*) ∈ *F*
shows *spec.action F* = ⨆(*spec.return ' fst ' F*) ⊔ *spec.idle*
⟨*proof*⟩

lemma *action-le*:
  assumes $Id \subseteq snd$ ‘ snd ‘ F
  shows *spec.return* () $\leq$ *spec.action F*
⟨*proof*⟩

⟨*ML*⟩

lemma *alt-def*:
  assumes $A \neq \{\}$
  shows *spec.guard g* = *spec.action* $(\{()\} \times A \times Diag\ g)$
⟨*proof*⟩

lemma *bot*:
  shows *spec.guard* $\bot$ = *spec.idle*
    and *spec.guard* ⟨*False*⟩ = *spec.idle*
⟨*proof*⟩

lemma *top*:
  shows *spec.guard* $\top$ = *spec.return* ()
    and *spec.guard* ⟨*True*⟩ = *spec.return* ()
⟨*proof*⟩

lemma *monotone*:
  shows *mono spec.guard*
⟨*proof*⟩

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* $(\bigsqcup X) = \bigsqcup(spec.guard$ ‘ $X) \sqcup spec.idle$
⟨*proof*⟩

lemma *sup*:
  shows *spec.guard* $(g \sqcup h)$ = *spec.guard g* $\sqcup$ *spec.guard h*
⟨*proof*⟩

lemma *return-le*:
  shows *spec.guard g* $\leq$ *spec.return* ()
⟨*proof*⟩

lemma *guard-less*: — Non-triviality
  assumes $g < g'$
  shows *spec.guard g* $<$ *spec.guard* g'
⟨*proof*⟩

lemma *cong*:
  assumes $\bigwedge v\ a\ s\ s'.\ (v,\ a,\ s,\ s') \in F \Longrightarrow s' = s$
  shows *spec.action F* = *spec.guard* $(\lambda s.\ s \in fst$ ‘ snd ‘ snd ‘ $F)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *action-le*:
  assumes $Diag\ g \subseteq snd$ ‘ snd ‘ F
  shows *spec.guard g* $\leq$ *spec.action F*
⟨*proof*⟩

⟨ML⟩

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.

⟨ML⟩

definition *none* :: $('a, 's, 'v)$ *spec* ⇒ $('a, 's, 'w)$ *spec* **where**
none $P = \bigsqcup\{⦇s,\ xs,\ None⦈ \mid s\ xs\ v.\ ⦇s,\ xs,\ v⦈ \le P\}$

definition *all* :: $('a, 's, 'v)$ *spec* ⇒ $('a, 's, 'w)$ *spec* **where**
all $P = \bigsqcup\{⦇s,\ xs,\ v⦈ \mid s\ xs\ v.\ ⦇s,\ xs,\ None⦈ \le P\}$

⟨ML⟩

interpretation *term*: *galois.complete-lattice-distributive-class spec.term.none spec.term.all*
⟨*proof*⟩

⟨ML⟩

lemma *none-le-conv*[*spec.singleton.le-conv*]:
shows $⦇\sigma⦈ \le$ *spec.term.none* $P \longleftrightarrow$ *trace.term* $\sigma = None \wedge ⦇$*trace.init* σ, *trace.rest* σ, *None*$⦈ \le P$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *all-le-conv*[*spec.singleton.le-conv*]:
shows $⦇\sigma⦈ \le$ *spec.term.all* $P \longleftrightarrow (\exists w.\ ⦇$*trace.init* σ, *trace.rest* σ, $w⦈ \le P)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨ML⟩

lemma *singleton*:
shows *spec.term.none* $⦇\sigma⦈ = ⦇$*trace.init* σ, *trace.rest* σ, *None*$⦈$
⟨*proof*⟩

lemmas *bot*[*simp*] = *spec.term.lower-bot*

lemmas *monotone* = *spec.term.monotone-lower*
lemmas *mono* = *monotoneD*[*OF spec.term.none.monotone*]

lemmas *Sup* = *spec.term.lower-Sup*
lemmas *sup* = *spec.term.lower-sup*

lemmas *Inf-le* = *spec.term.lower-Inf-le*

lemma *Inf-not-empty*:
assumes $X \ne \{\}$
shows *spec.term.none* $(\bigsqcap X) = (\bigsqcap x{\in}X.$ *spec.term.none* $x)$
⟨*proof*⟩

lemma *inf*:
shows *spec.term.none* $(P \sqcap Q) =$ *spec.term.none* $P \sqcap$ *spec.term.none* Q
and *spec.term.none* $(Q \sqcap P) =$ *spec.term.none* $Q \sqcap$ *spec.term.none* P
⟨*proof*⟩

lemma *inf-unit*:

fixes *P Q* :: (-, -, *unit*) *spec*
shows *spec.term.none* (*P* ⊓ *Q*) = *spec.term.none P* ⊓ *Q* (**is** *?thesis1 P Q*)
and *spec.term.none* (*P* ⊓ *Q*) = *P* ⊓ *spec.term.none Q* (**is** *?thesis2*)
⟨*proof*⟩

lemma *idempotent*[*simp*]:
shows *spec.term.none* (*spec.term.none P*) = *spec.term.none P*
⟨*proof*⟩

lemma *contractive*[*iff*]:
shows *spec.term.none P* ≤ *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemmas *invmap* = *spec.term.none.invmap-gen*[**where** *vf'*=*id*] — *simp*-friendly

lemma *idle*:
shows *spec.term.none spec.idle* = *spec.idle*
⟨*proof*⟩

lemma *return*:
shows *spec.term.none* (*spec.return v*) = *spec.idle*
⟨*proof*⟩

lemma *guard*:
shows *spec.term.none* (*spec.guard g*) = *spec.idle*
⟨*proof*⟩

⟨*ML*⟩

lemma *none-all-le*:
shows *spec.term.none P* ≤ *spec.term.all P*
⟨*proof*⟩

lemma *none-all*[*simp*]:
shows *spec.term.none* (*spec.term.all P*) = *spec.term.none P*
⟨*proof*⟩

lemma *all-none*[*simp*]:
shows *spec.term.all* (*spec.term.none P*) = *spec.term.all P*
⟨*proof*⟩

⟨*ML*⟩

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 $P \leq$ *spec.term.all P*
⟨*proof*⟩

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*=⦇σ⦈] **for** σ

lemma *monomorphic*:
  shows *spec.term.cl* - = *spec.term.all*
⟨*proof*⟩

lemma *closed-conv*:
  assumes $P \in$ *spec.term.closed* -
  shows *P* = *spec.term.all P*
⟨*proof*⟩

lemma *closed*[*iff*]:
  shows *spec.term.all P* ∈ *spec.term.closed* -
⟨*proof*⟩

lemma *idempotent*[*simp*]:
  shows *spec.term.all* (*spec.term.all P*) = *spec.term.all P*
⟨*proof*⟩

lemma *map*: — *vf* = *id* on the RHS
  fixes *vf* :: ′*v* ⇒ ′*w*
  shows *spec.term.all* (*spec.map af sf vf P*) = *spec.map af sf id* (*spec.term.all P*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *invmap*: — *vf* = *id* on the RHS
  fixes *vf* :: ′*v* ⇒ ′*w*
  shows *spec.term.all* (*spec.invmap af sf vf P*) = *spec.invmap af sf id* (*spec.term.all P*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *vmap-unit-absorb*:
  shows *spec.vmap* ⟨()⟩ (*spec.term.all P*) = *spec.term.all P* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *vmap-unit*:
  shows *spec.vmap* ⟨()⟩ (*spec.term.all P*) = *spec.term.all* (*spec.vmap* ⟨()⟩ *P*)
⟨*proof*⟩

lemma *idle*:
  shows *spec.term.all spec.idle* = (⨆ *v*. *spec.return v*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *action*:

fixes F :: $('v \times 'a \times 's \times 's)$ *set*
shows *spec.term.all* (*spec.action* F) = *spec.action* (*UNIV* × *snd* ‘ F) ⊔ (⨆*v*. *spec.return v*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *return*:
shows *spec.term.all* (*spec.return v*) = (⨆*v*. *spec.return v*)
⟨*proof*⟩

lemma *guard*:
shows *spec.term.all* (*spec.guard g*) = (⨆*v*. *spec.return v*)
⟨*proof*⟩

⟨*ML*⟩

lemma *none-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.term.none P* ⟷ *spec.idle* ≤ *P*
⟨*proof*⟩

lemma *all-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.term.all P* ⟷ *spec.idle* ≤ *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *return-unit*:
shows *spec.return* () ∈ *spec.term.closed* -
⟨*proof*⟩

lemma *none-inf*:
fixes P :: $('a, 's, 'v)$ *spec*
fixes Q :: $('a, 's, 'w)$ *spec*
assumes P ∈ *spec.term.closed* -
shows P ⊓ *spec.term.none Q* = *spec.term.none* (*spec.term.none P* ⊓ Q) (**is** *?lhs* = *?rhs*)
and *spec.term.none Q* ⊓ P = *spec.term.none* (Q ⊓ *spec.term.none P*) (**is** *?thesis1*)
⟨*proof*⟩

lemma *none-inf-monomorphic*:
fixes P :: $('a, 's, 'v)$ *spec*
fixes Q :: $('a, 's, 'v)$ *spec*
assumes P ∈ *spec.term.closed* -
shows P ⊓ *spec.term.none Q* = *spec.term.none* (P ⊓ Q) (**is** *?thesis1*)
and *spec.term.none Q* ⊓ P = *spec.term.none* (Q ⊓ P) (**is** *?thesis2*)
⟨*proof*⟩

lemma *singleton-le-extI*:
assumes Q ∈ *spec.term.closed* -
assumes ⋀*s xs*. ⦇*s*, *xs*, *None*⦈ ≤ P ⟹ ⦇*s*, *xs*, *None*⦈ ≤ Q
shows $P \le Q$
⟨*proof*⟩

⟨*ML*⟩

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., *spec.term.none* $f \le f \gg= g$, 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.

⟨ML⟩

definition *continue* :: $('a, 's, 'v)$ *spec* ⇒ $('v \Rightarrow ('a, 's, 'w)$ *spec*$) \Rightarrow ('a, 's, 'w)$ *spec* **where**
continue f g =
$\bigsqcup\{\langle\!|$trace.init σ_f, trace.rest σ_f @ trace.rest σ_g, trace.term $\sigma_g|\!\rangle$
$|\sigma_f\ \sigma_g\ v.\ \langle\!|\sigma_f|\!\rangle \leq f \wedge$ trace.init σ_g = trace.final $\sigma_f \wedge$ trace.term σ_f = Some $v \wedge \langle\!|\sigma_g|\!\rangle \leq g\ v\}$

definition *bind* :: $('a, 's, 'v)$ *spec* ⇒ $('v \Rightarrow ('a, 's, 'w)$ *spec*$) \Rightarrow ('a, 's, 'w)$ *spec* **where**
bind f g = *spec.term.none f* ⊔ *spec.continue f g*

adhoc-overloading
Monad-Syntax.bind ⇌ *spec.bind*

⟨ML⟩

lemma *continue-le-conv*:
shows $\langle\!|\sigma|\!\rangle \leq$ *spec.continue f g*
⟷ (∃ *xs ys v w*. ⟨|*trace.init* σ, *xs*, *Some v*|⟩ ≤ *f*
∧ ⟨|*trace.final′* (*trace.init* σ) *xs*, *ys*, *w*|⟩ ≤ *g v*
∧ σ ≤ *trace.T* (*trace.init* σ) (*xs* @ *ys*) *w*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨ML⟩

lemma *mono*:
assumes $f \leq f'$
assumes $\bigwedge v.\ g\ v \leq g'\ v$
shows *spec.continue f g* ≤ *spec.continue f′ g′*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord F f f′*
assumes ⋀*x*. *st-ord F* (*g x*) (*g′ x*)
shows *st-ord F* (*spec.continue f g*) (*spec.continue f′ g′*)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* (≤) *f*
assumes ⋀*x*. *monotone orda* (≤) (λ*y*. *g y x*)
shows *monotone orda* (≤) (λ*x*. *spec.continue* (*f x*) (*g x*))
⟨*proof*⟩

definition *resL* :: $('v \Rightarrow ('a, 's, 'w)$ *spec*$) \Rightarrow ('a, 's, 'w)$ *spec* ⇒ $('a, 's, 'v)$ *spec* **where**
resL g P = ⨆{*f*. *spec.continue f g* ≤ *P*}

definition *resR* :: $('a, 's, 'v)$ *spec* ⇒ $('a, 's, 'w)$ *spec* ⇒ $('v \Rightarrow ('a, 's, 'w)$ *spec*$)$ **where**
resR f P = ⨆{*g*. *spec.continue f g* ≤ *P*}

interpretation *L*: *galois.complete-lattice-class* λ*f*. *spec.continue f g spec.continue.resL g* **for** *g*
⟨*proof*⟩

interpretation *R*: *galois.complete-lattice-class* λ*g*. *spec.continue f g spec.continue.resR f*
for *f* :: $('a, 's, 'v)$ *spec*
⟨*proof*⟩

⟨ML⟩

lemma *bind-le-conv*:
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.bind f g* $\longleftrightarrow$ $\langle\!|\sigma|\!\rangle \leq$ *spec.term.none f* $\vee$ $\langle\!|\sigma|\!\rangle \leq$ *spec.continue f g*
⟨*proof*⟩

lemma *bind-le*[*consumes 1*]:
  assumes $\langle\!|\sigma|\!\rangle \leq f \ggg g$
  obtains
    (*incomplete*) $\langle\!|\sigma|\!\rangle \leq$ *spec.term.none f*
  | (*continue*) σ_f σ_g v_f
    where $\langle\!|\sigma_f|\!\rangle \leq f$ **and** *trace.final* σ_f = *trace.init* σ_g **and** *trace.term* σ_f = *Some* v_f
      and $\langle\!|\sigma_g|\!\rangle \leq g\ v_f$ **and** $\natural\sigma_g \neq$ *trace.T* (*trace.init* σ_g) [] *None*
      and σ = *trace.T* (*trace.init* σ_f) (*trace.rest* σ_f @ *trace.rest* σ_g) (*trace.term* σ_g)
⟨*proof*⟩

⟨*ML*⟩

lemma *bind-le*[*case-names incomplete continue*]:
  assumes *spec.term.none f* $\leq P$
  assumes $\bigwedge\sigma_f\ \sigma_g\ v.$ ⟦$\langle\!|\sigma_f|\!\rangle \leq f$; *trace.init* σ_g = *trace.final* σ_f; *trace.term* σ_f = *Some v*; $\langle\!|\sigma_g|\!\rangle \leq g\ v$;
        $\natural\sigma_g \neq$ *trace.T* (*trace.init* σ_g) [] *None*⟧
      $\Longrightarrow$ ⟨|*trace.init* σ_f, *trace.rest* σ_f @ *trace.rest* σ_g, *trace.term* σ_g|⟩ $\leq P$
  shows $f \ggg g \leq P$
⟨*proof*⟩

⟨*ML*⟩

definition *resL* :: ('*v* ⇒ ('*a*, '*s*, '*w*) *spec*) ⇒ ('*a*, '*s*, '*w*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
  resL g P = $\bigsqcup\{f.\ f \ggg g \leq P\}$

lemma *incompleteI*:
  assumes ⟨|*s*, *xs*, *None*|⟩ $\leq f$
  shows ⟨|*s*, *xs*, *None*|⟩ $\leq f \ggg g$
⟨*proof*⟩

lemma *continueI*:
  assumes *f*: ⟨|*s*, *xs*, *Some v*|⟩ $\leq f$
  assumes *g*: ⟨|*trace.final'* *s xs*, *ys*, *w*|⟩ $\leq g\ v$
  shows ⟨|*s*, *xs* @ *ys*, *w*|⟩ $\leq f \ggg g$
⟨*proof*⟩

lemma *singletonL*:
  shows $\langle\!|\sigma|\!\rangle \ggg g$
    = *spec.term.none* $\langle\!|\sigma|\!\rangle$
      $\sqcup \bigsqcup\{$⟨|*trace.init* σ, *trace.rest* σ @ *trace.rest* σ_g, *trace.term* σ_g|⟩ $|\sigma_g.$
        trace.final σ = *trace.init* σ_g $\wedge$ ($\exists\, v.$ *trace.term* σ = *Some v* $\wedge$ $\langle\!|\sigma_g|\!\rangle \leq g\ v$)$\}$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *mono*:
  assumes $f \leq f'$
  assumes $\bigwedge v.\ g\ v \leq g'\ v$
  shows *spec.bind f g* $\leq$ *spec.bind f' g'*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
  assumes *st-ord F f f'*
  assumes $\bigwedge x.$ *st-ord F* (*g x*) (*g' x*)
  shows *st-ord F* (*spec.bind f g*) (*spec.bind f' g'*)

⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *f*
  assumes ⋀*x*. *monotone orda* (≤) (λ*y*. *g y x*)
  shows *monotone orda* (≤) (λ*x*. *spec.bind* (*f x*) (*g x*))
⟨*proof*⟩

interpretation *L*: *galois.complete-lattice-class* λ*f*. *f* ⤜ *g spec.bind.resL g* **for** *g*
⟨*proof*⟩

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 *spec.bind f* (λ*v*. ⨆*x*∈*X*. *g x v*) = (⨆*x*∈*X*. *f* ⤜ *g x*) ⊔ (*f* ⤜ ⊥) (**is** *?thesis1*) — *Sup* over ('*a*, '*s*, '*v*) *spec*
    and *spec.bind f* (⨆*x*∈*X*. *g x*) = (⨆*x*∈*X*. *f* ⤜ *g x*) ⊔ (*f* ⤜ ⊥) (**is** *?thesis2*) — *Sup* over functions
⟨*proof*⟩

lemma *SUPR-not-empty*:
  assumes *X* ≠ {}
  shows *spec.bind f* (λ*v*. ⨆*x*∈*X*. *g x v*) = (⨆*x*∈*X*. *f* ⤜ *g x*)
⟨*proof*⟩

lemmas *supR* = *spec.bind.SUPR-not-empty*[**where** *g*=*id* **and** *X*={g_1, g_2} **for** g_1 g_2, *simplified*]

lemma *InfR-le*:
  shows *spec.bind f* (λ*v*. ⨅*x*∈*X*. *g x v*) ≤ (⨅*x*∈*X*. *f* ⤜ *g x*)
⟨*proof*⟩

lemma *infR-le*:
  shows *spec.bind f* (g_1 ⊓ g_2) ≤ (*f* ⤜ g_1) ⊓ (*f* ⤜ g_2)
    and *spec.bind f* (λ*v*. g_1 *v* ⊓ g_2 *v*) ≤ (*f* ⤜ g_1) ⊓ (*f* ⤜ g_2)
⟨*proof*⟩

lemma *Inf-le*:
  shows *spec.bind* (⨅*x*∈*X*. *f x*) (λ*v*. (⨅*x*∈*X*. *g x v*)) ≤ (⨅*x*∈*X*. *spec.bind* (*f x*) (*g x*))
⟨*proof*⟩

lemma *inf-le*:
  shows *spec.bind* (f_1 ⊓ f_2) (λ*v*. g_1 *v* ⊓ g_2 *v*) ≤ *spec.bind* f_1 g_1 ⊓ *spec.bind* f_2 g_2
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *f*
  assumes ⋀*v*. *mcont luba orda Sup* (≤) (λ*x*. *g x v*)
  shows *mcont luba orda Sup* (≤) (λ*x*. *spec.bind* (*f x*) (*g x*))
⟨*proof*⟩

lemmas *botL*[*simp*] = *spec.bind.L.lower-bot*

lemma *botR*:

shows $f \ggg \bot = spec.term.none\ f$
⟨*proof*⟩

lemma *eq-bot-conv*:
shows $spec.bind\ f\ g = \bot \longleftrightarrow f = \bot$
⟨*proof*⟩

lemma *idleL*[*simp*]:
shows $spec.idle \ggg g = spec.idle$
⟨*proof*⟩

lemma *idleR*:
shows $f \gg spec.idle = f \ggg \bot$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemmas *ifL* = *if-distrib*[**where** $f{=}\lambda f.\ spec.bind\ f\ g$ **for** *g*]

⟨*ML*⟩

lemma *bind-le-conv*[*spec.idle-le*]:
shows $spec.idle \le f \ggg g \longleftrightarrow spec.idle \le f$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bindL-le*[*iff*]:
shows $spec.term.none\ f \le f \ggg g$
⟨*proof*⟩

lemma *bind*:
shows $spec.term.none\ (f \ggg g) = f \ggg (\lambda v.\ spec.term.none\ (g\ v))$
⟨*proof*⟩

⟨*ML*⟩

lemma *bind*:
shows $spec.term.all\ (f \ggg g) = spec.term.all\ f \sqcup (f \ggg (\lambda v.\ spec.term.all\ (g\ v)))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

The monad laws for (≫=). ⟨*ML*⟩

lemma *bind*:
fixes *f* :: (-, -, -) *spec*
shows $f \ggg g \ggg h = f \ggg (\lambda v.\ g\ v \ggg h)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemmas *assoc* = *spec.bind.bind*

lemma *returnL-le*:
shows $g\ v \le spec.return\ v \ggg g$ (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *returnL*:
assumes $spec.idle \le g\ v$
shows $spec.return\ v \ggg g = g\ v$
⟨*proof*⟩

lemma *returnR*[*simp*]:
  shows $f \ggg= spec.return = f$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *return*: — Does not require $spec.idle \le g\ v$
  fixes f :: ('*a*, '*s*, '*v*) *spec*
  fixes g :: '*v* ⇒ '*x* ⇒ ('*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*⟩

⟨*ML*⟩

lemma *noneL*[*simp*]:
  shows $spec.term.none\ f \ggg= g = spec.term.none\ f$
⟨*proof*⟩

⟨*ML*⟩

lemma *bind-le*: — Converse does not hold: it may be that no final states of *f* satisfy *g*
  fixes f :: ('*a*, '*s*, '*v*) *spec*
  fixes g :: '*v* ⇒ ('*a*, '*s*, '*w*) *spec*
  fixes *af* :: '*a* ⇒ '*b*
  fixes *sf* :: '*s* ⇒ '*t*
  fixes *vf* :: '*w* ⇒ '*x*
  shows $spec.map\ af\ sf\ vf\ (f \ggg= g) \le spec.map\ af\ sf\ id\ f \ggg= (\lambda v.\ spec.map\ af\ sf\ vf\ (g\ v))$
⟨*proof*⟩

lemma *bind-inj-sf*:
  fixes f :: ('*a*, '*s*, '*x*) *spec*
  fixes g :: '*x* ⇒ ('*a*, '*s*, '*v*) *spec*
  assumes *inj sf*
  shows $spec.map\ af\ sf\ vf\ (f \ggg= g) = spec.map\ af\ sf\ id\ f \ggg= (\lambda v.\ spec.map\ af\ sf\ vf\ (g\ v))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *eq-return*: — generalizes *spec.bind.returnR*
  shows $spec.vmap\ vf\ P = P \ggg= spec.return \circ vf$ (**is** *?thesis1*)
    and $spec.vmap\ vf\ P = P \ggg= (\lambda v.\ spec.return\ (vf\ v))$ (**is** *?lhs* = *?rhs*) — useful for flip/symmetric
⟨*proof*⟩

lemma *unitL*: — monomorphise ignored return values
  shows $f \gg g = spec.vmap\ \langle()\rangle\ f \gg g$
⟨*proof*⟩

⟨*ML*⟩

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 $spec.invmap\ af\ sf\ vf\ (f \ggg= g) = spec.invmap\ af\ sf\ id\ f \ggg= (\lambda v.\ spec.invmap\ af\ sf\ vf\ (g\ v))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-vinvmap*:
shows *spec.invmap af sf vf P* = *spec.invmap af sf id P* $\gg=$ (λv. $\bigsqcup v' \in vf$ $-$‘ $\{v\}$. *spec.return v'*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *return-const*:
assumes $V \neq \{\}$
assumes $W \neq \{\}$
shows *spec.action* $(V \times F)$ = *spec.action* $(W \times F) \gg (\bigsqcup v \in V$. *spec.return v*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bind-all-return*:
assumes $f \in$ *spec.term.closed* -
shows $f \gg$ ($\bigsqcup$ *range spec.return*) = *spec.term.all f* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

8.9 Kleene star

We instantiate the generic Kleene locale with monomorphic *spec.return* (). The polymorphic ($\bigsqcup v$. *spec.return v*) fails the *comp-unitR* axiom ($\varepsilon \leq x \Longrightarrow x \cdot \varepsilon = x$).

⟨*ML*⟩

interpretation *kleene*: *weak-kleene spec.return* () $\lambda x\ y$. *spec.bind x* ⟨*y*⟩
⟨*proof*⟩

⟨*ML*⟩

lemmas *star-le*[*spec.idle-le*] = *order.trans*[*OF spec.idle.return-le spec.kleene.epsilon-star-le*]

lemmas *rev-star-le*[*spec.idle-le*] = *spec.idle.kleene.star-le*[*unfolded spec.kleene.star-rev-star*]

⟨*ML*⟩

lemmas *star-le* = *spec.kleene.epsilon-star-le*

lemmas *rev-star-le* = *spec.return.kleene.star-le*[*unfolded spec.kleene.star-rev-star*]

⟨*ML*⟩

lemma *star-idle*:
shows *spec.kleene.star spec.idle* = *spec.return* ()
⟨*proof*⟩

lemmas *rev-star-idle* = *spec.kleene.star-idle*[*unfolded spec.kleene.star-rev-star*]

⟨*ML*⟩

lemma *star-closed-le*:
fixes *P* :: (-, -, *unit*) *spec*
assumes $P \in$ *spec.term.closed* -
shows *spec.term.all* (*spec.kleene.star P*) $\leq$ *spec.kleene.star P* (**is** - $\leq$ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *star*:
  assumes $P \in$ *spec.term.closed* -
  shows *spec.kleene.star* $P \in$ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

8.10 Transition relations

Using *spec.kleene.star* we can specify the transitions each agent is allowed to perform. These constraints ((⊓) *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

⟨*ML*⟩

definition *act* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, *unit*) *spec* **where** — lift above *spec.return* to ease some proofs
  act r = *spec.action* ({()} × (r ∪ *UNIV* × *Id*))

abbreviation *monomorphic* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, *unit*) *spec* **where**
  monomorphic r ≡ *spec.kleene.star* (*spec.rel.act* r)

lemma *act-alt-def*:
  shows *spec.rel.act* r = *spec.action* ({()} × r) ⊔ *spec.return* ()
⟨*proof*⟩

⟨*ML*⟩

definition *rel* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
  rel r = *spec.term.all* (*spec.rel.monomorphic* r)

definition *steps* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*) *steps* **where**
  steps $P = \bigcap\{r.\ P \leq$ *spec.rel* $r\}$

⟨*ML*⟩

lemma *monotone*:
  shows *mono spec.rel.act*
⟨*proof*⟩

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* ()
⟨*proof*⟩

lemma *UNIV*:
  shows *spec.rel.act UNIV* = *spec.action* ({()} × *UNIV*)
⟨*proof*⟩

lemma *sup*:
  shows *spec.rel.act* $(r \cup s)$ = *spec.rel.act* $r \sqcup$ *spec.rel.act* s
⟨*proof*⟩

lemma *stutter*:
  shows *spec.rel.act* (*UNIV* × *Id*) = *spec.return* ()
⟨*proof*⟩

⟨*ML*⟩

lemma *act-mono*:
  shows *spec.term.all* (*spec.rel.act* r) = *spec.rel.act* r
⟨*proof*⟩

⟨*ML*⟩

lemma *rel*:
  shows *spec.term.all* (*spec.rel* r) = *spec.rel* r
⟨*proof*⟩

⟨*ML*⟩

lemma *act*:
  shows *spec.rel.act* $r \in$ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

lemma *rel*:
  shows *spec.rel* $r \in$ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-none-rel*: — polymorphic constants
  shows *spec.term.none* (*spec.rel* r :: ('a, 's, 'w) *spec*) ⊓ *spec.term.none* P
      = *spec.rel* r ⊓ (*spec.term.none* P :: ('a, 's, 'v) *spec*) (**is** *?thesis1*)
    and *spec.term.none* P ⊓ *spec.term.none* (*spec.rel* r :: ('a, 's, 'w) *spec*)
      = *spec.term.none* P ⊓ (*spec.rel* r :: ('a, 's, 'v) *spec*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *inf-rel*:
  shows *spec.term.none* P ⊓ *spec.rel* r = *spec.term.none* (P ⊓ *spec.rel* r) (**is** *?thesis1*)
    and *spec.rel* r ⊓ *spec.term.none* P = *spec.term.none* (*spec.rel* r ⊓ P) (**is** *?thesis2*)
⟨*proof*⟩

⟨*ML*⟩

lemma *act-le*:
  shows *spec.return* () $\leq$ *spec.rel.act* r
⟨*proof*⟩

⟨*ML*⟩

lemma *rel-le*:
  shows *spec.return* $v \leq$ *spec.rel* r
⟨*proof*⟩

lemma *Sup-rel-le*:
  shows $\bigsqcup$ *range spec.return* $\leq$ *spec.rel r*
⟨*proof*⟩

⟨*ML*⟩

lemmas *act-le*[*spec.idle-le*] = *order.trans*[*OF spec.idle.return-le spec.return.rel.act-le*]

⟨*ML*⟩

lemmas *rel-le*[*spec.idle-le*] = *order.trans*[*OF spec.idle.return-le spec.return.rel-le*]

⟨*ML*⟩

lemma *le-conv*[*spec.singleton.le-conv*]:
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.rel.act r* $\longleftrightarrow$ *trace.steps* $\sigma = \{\} \lor (\exists x \in r.$ *trace.steps* $\sigma = \{x\})$
⟨*proof*⟩

⟨*ML*⟩

lemma *le-steps*:
  assumes *trace.steps* $\sigma \subseteq r$
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.rel.monomorphic r*
⟨*proof*⟩

⟨*ML*⟩

lemmas *mono-le* = *spec.kleene.expansive-star*

⟨*ML*⟩

lemma *alt-def*:
  shows *spec.rel.monomorphic r* = $\bigsqcup$(*spec.singleton* ‘ $\{\sigma.$ *trace.steps* $\sigma \subseteq r\}$) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *monomorphic-le-conv*[*spec.singleton.le-conv*]:
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.rel.monomorphic r* $\longleftrightarrow$ *trace.steps* $\sigma \subseteq r$
⟨*proof*⟩

⟨*ML*⟩

lemma *rel-le-conv*[*spec.singleton.le-conv*]:
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.rel r* $\longleftrightarrow$ *trace.steps* $\sigma \subseteq r$
⟨*proof*⟩

⟨*ML*⟩

interpretation *rel*: *galois.complete-lattice-class spec.steps spec.rel*
⟨*proof*⟩

lemma *rel-alt-def*:
  shows *spec.rel r* = $\bigsqcup$(*spec.singleton* ‘ $\{\sigma.$ *trace.steps* $\sigma \subseteq r\}$)
⟨*proof*⟩

⟨*ML*⟩

lemma *unit-rel*:
  shows *spec.vmap* ⟨()⟩ (*spec.rel r*) = *spec.rel r*
⟨*proof*⟩

⟨*ML*⟩

lemma *monomorphic-conv*: — if the return type is *unit*
  shows *spec.rel r* = *spec.rel.monomorphic r*
⟨*proof*⟩

lemma *monomorphic-act-le*: — *unit* return type
  shows *spec.rel.act r* ≤ *spec.rel r*
⟨*proof*⟩

lemma *empty*:
  shows *spec.rel* {} = ($\bigsqcup$*v*. *spec.return v*)
⟨*proof*⟩

lemmas *UNIV* = *spec.rel.upper-top*
lemmas *top* = *spec.rel.UNIV*

lemmas *INF* = *spec.rel.upper-INF*
lemmas *Inf* = *spec.rel.upper-Inf*
lemmas *inf* = *spec.rel.upper-inf*

lemmas *Sup-le* = *spec.rel.Sup-upper-le*
lemmas *sup-le* = *spec.rel.sup-upper-le* — Converse does not hold: the RHS allows interleaving of *r* and *s* steps

lemma *reflcl*:
  shows *spec.rel* (*r* ∪ *A* × *Id*) = *spec.rel r*
    and *spec.rel* (*A* × *Id* ∪ *r*) = *spec.rel r*
⟨*proof*⟩

lemma *minus-Id*:
  shows *spec.rel* (*r* − *A* × *Id*) = *spec.rel r*
⟨*proof*⟩

lemma *Id*:
  shows *spec.rel* (*A* × *Id*) = ($\bigsqcup$*v*. *spec.return v*)
⟨*proof*⟩

lemmas *monotone* = *spec.rel.monotone-upper*
lemmas *mono* = *monotoneD*[*OF spec.rel.monotone*, *of r r'* **for** *r r'*]

lemma *mono-reflcl*:
  assumes *r* ⊆ *s* ∪ *UNIV* × *Id*
  shows *spec.rel r* ≤ *spec.rel s*
⟨*proof*⟩

lemma *unfoldL*:
  shows *spec.rel r* = *spec.rel.act r* ≫ *spec.rel r* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *foldR*: — arbitrary interstitial return type
  shows *spec.rel r* ≫ *spec.rel.act r* = *spec.rel r* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *wind-bind*: — arbitrary interstitial return type

shows $spec.rel\ r \gg spec.rel\ r = spec.rel\ r$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *wind-bind-leading*: — arbitrary interstitial return type
assumes $r' \subseteq r$
shows $spec.rel\ r' \gg spec.rel\ r = spec.rel\ r$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *wind-bind-trailing*: — arbitrary interstitial return type
assumes $r' \subseteq r$
shows $spec.rel\ r \gg spec.rel\ r' = spec.rel\ r$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

Interstitial unit, for unfolding

lemmas *unwind-bind* = *spec.rel.wind-bind*[**where** *'d=unit*, *symmetric*]
lemmas *unwind-bind-leading* = *spec.rel.wind-bind-leading*[**where** *'d=unit*, *symmetric*]
lemmas *unwind-bind-trailing* = *spec.rel.wind-bind-trailing*[**where** *'d=unit*, *symmetric*]

⟨*ML*⟩

lemma *rel*:
shows $spec.invmap\ af\ sf\ vf\ (spec.rel\ r) = spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ -`\ (r \cup UNIV \times Id))$
⟨*proof*⟩

lemma *range*:
shows $spec.invmap\ af\ sf\ vf\ P = spec.invmap\ af\ sf\ vf\ (P \sqcap spec.rel\ (range\ af \times range\ sf \times range\ sf))$
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-rel*:
shows $spec.map\ af\ sf\ vf\ P \sqcap spec.rel\ r$
$= spec.map\ af\ sf\ vf\ (P \sqcap spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ -`\ (r \cup UNIV \times Id)))$
and $spec.rel\ r \sqcap spec.map\ af\ sf\ vf\ P$
$= spec.map\ af\ sf\ vf\ (spec.rel\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ -`\ (r \cup UNIV \times Id)) \sqcap P)$
⟨*proof*⟩

⟨*ML*⟩

lemma *rel-le*:
fixes $F :: ('v \times 'a \times 's \times 's)\ set$
fixes $r :: ('a, 's)\ steps$
assumes $\bigwedge v\ a\ s\ s'.\ (v, a, s, s') \in F \Longrightarrow (a, s, s') \in r \vee s = s'$
shows $spec.action\ F \leq spec.rel\ r$
⟨*proof*⟩

⟨*ML*⟩

lemma *star-le*:
assumes $S \leq spec.rel\ r$
shows $spec.kleene.star\ S \leq spec.rel\ r$
⟨*proof*⟩

⟨*ML*⟩

lemma *relL-le*:
shows $g\ x \leq spec.rel\ r \ggg g$
⟨*proof*⟩

lemma *relR-le*:
  shows $f \leq f \gg spec.rel\ r$
⟨*proof*⟩

lemma *inf-rel*:
  shows $(f \gg= g) \sqcap spec.rel\ r = (spec.rel\ r \sqcap f) \gg= (\lambda x.\ spec.rel\ r \sqcap g\ x)$ (**is** *?thesis1*)
    and $spec.rel\ r \sqcap (f \gg= g) = (spec.rel\ r \sqcap f) \gg= (\lambda x.\ spec.rel\ r \sqcap g\ x)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *inf-rel-distr-le*:
  shows $(f \sqcap spec.rel\ r) \gg= (\lambda v.\ g_1\ v \sqcap g_2) \leq (f \gg= g_1) \sqcap (spec.rel\ r \gg= (\lambda\text{-}::unit.\ g_2))$
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-rel*:
  shows $⦇\sigma⦈ \sqcap spec.rel\ r = \bigsqcup(spec.singleton\ `\ \{\sigma'.\ \sigma' \leq \sigma \wedge trace.steps\ \sigma' \subseteq r\})$ (**is** *?lhs = ?rhs*)
    and $spec.rel\ r \sqcap ⦇\sigma⦈ = \bigsqcup(spec.singleton\ `\ \{\sigma'.\ \sigma' \leq \sigma \wedge trace.steps\ \sigma' \subseteq r\})$ (**is** *?thesis2*)
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-rel*:
  fixes $F :: ('v \times 'a \times 's \times 's)\ set$
  fixes $r :: ('a, 's)\ steps$
  assumes $\bigwedge a.\ refl\ (r\ ``\ \{a\})$
  shows $spec.action\ F \sqcap spec.rel\ r = spec.action\ (F \cap UNIV \times r)$ (**is** *?lhs = ?rhs*)
    and $spec.rel\ r \sqcap spec.action\ F = spec.action\ (F \cap UNIV \times r)$ (**is** *?thesis1*)
⟨*proof*⟩

lemma *inf-rel-reflcl*:
  shows $spec.action\ F \sqcap spec.rel\ r = spec.action\ (F \cap UNIV \times (r \cup UNIV \times Id))$
    and $spec.rel\ r \sqcap spec.action\ F = spec.action\ (F \cap UNIV \times (r \cup UNIV \times Id))$
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-rel*:
  shows $spec.rel\ r \sqcap spec.return\ v = spec.return\ v$
    and $spec.return\ v \sqcap spec.rel\ r = spec.return\ v$
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-rel*:
  shows $spec.kleene.star\ P \sqcap spec.rel\ r = spec.kleene.star\ (P \sqcap spec.rel\ r)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*[*simp*]:
  shows $(a, s, s) \notin spec.steps\ P$
⟨*proof*⟩

lemma *member-conv*:
  shows $x \in spec.steps\ P \longleftrightarrow (\exists \sigma.\ ⦇\sigma⦈ \leq P \wedge x \in trace.steps\ \sigma)$
⟨*proof*⟩

⟨ML⟩

lemma *none*:
  shows $spec.steps\ (spec.term.none\ P) = spec.steps\ P$
⟨proof⟩

lemma *all*:
  shows $spec.steps\ (spec.term.all\ P) = spec.steps\ P$
⟨proof⟩

⟨ML⟩

lemmas $bot = spec.rel.lower\text{-}bot$

lemmas $monotone = spec.rel.monotone\text{-}lower$
lemmas $mono = monotoneD[OF\ spec.steps.monotone]$

lemmas $Sup = spec.rel.lower\text{-}Sup$
lemmas $sup = spec.rel.lower\text{-}sup$
lemmas $Inf\text{-}le = spec.rel.lower\text{-}Inf\text{-}le$
lemmas $inf\text{-}le = spec.rel.lower\text{-}inf\text{-}le$

lemma *singleton*:
  shows $spec.steps\ \langle\!|\sigma|\!\rangle = trace.steps\ \sigma$
⟨proof⟩

lemma *idle*:
  shows $spec.steps\ spec.idle = \{\}$
⟨proof⟩

lemma *action*:
  shows $spec.steps\ (spec.action\ F) = snd\ `\ F - UNIV \times Id$
⟨proof⟩

lemma *return*:
  shows $spec.steps\ (spec.return\ v) = \{\}$
⟨proof⟩

lemma *bind-le*: — see *spec.steps.bind*
  shows $spec.steps\ (f \gg\!= g) \subseteq spec.steps\ f \cup (\bigcup v.\ spec.steps\ (g\ v))$
⟨proof⟩

lemma *kleene-star*:
  shows $spec.steps\ (spec.kleene.star\ P) = spec.steps\ P$ (**is** $?lhs = ?rhs$)
⟨proof⟩

lemma *map*:
  shows $spec.steps\ (spec.map\ af\ sf\ vf\ P)$
    $= map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf)\ `\ spec.steps\ P - UNIV \times Id$
⟨proof⟩

lemma *invmap-le*:
  shows $spec.steps\ (spec.invmap\ af\ sf\ vf\ P)$
    $\subseteq map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf) -`\ (spec.steps\ (P \sqcap spec.rel\ (range\ af \times range\ sf \times range\ sf)) \cup UNIV \times Id) - UNIV \times Id$
⟨proof⟩

⟨*ML*⟩

lemma *monomorphic*:
fixes r :: ('*a*, '*s*) *steps*
shows *spec.steps* (*spec.rel.monomorphic* r) = r − *UNIV* × *Id* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *rel*:
fixes r :: ('*a*, '*s*) *steps*
shows *spec.steps* (*spec.rel* r) = r − *UNIV* × *Id*
⟨*proof*⟩

lemma *top*:
shows *spec.steps* ⊤ = *UNIV* × − *Id*
⟨*proof*⟩

⟨*ML*⟩

8.11 Sequential assertions

We specify sequential behavior with preconditions and postconditions.

8.11.1 Preconditions

⟨*ML*⟩

definition *pre* :: '*s pred* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
pre P = ⨆(*spec.singleton* ‘ {σ. P (*trace.init* σ)})

⟨*ML*⟩

lemma *pre-le-conv*[*spec.singleton.le-conv*]:
shows ⦇σ⦈ ≤ *spec.pre* P ⟷ P (*trace.init* σ)
⟨*proof*⟩

lemma *inf-pre*:
shows *spec.pre* P ⊓ ⦇σ⦈ = (*if* P (*trace.init* σ) *then* ⦇σ⦈ *else* ⊥) (**is** *?thesis1*)
and ⦇σ⦈ ⊓ *spec.pre* P = (*if* P (*trace.init* σ) *then* ⦇σ⦈ *else* ⊥) (**is** *?thesis2*)
⟨*proof*⟩

⟨*ML*⟩

lemma *pre-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ (*spec.pre* P :: ('*a*, '*s*, '*v*) *spec*) ⟷ P = ⊤ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *pre*:
shows *spec.term.all* (*spec.pre* P) = *spec.pre* P
⟨*proof*⟩

⟨*ML*⟩

lemma *pre*:
shows *spec.pre* P ∈ *spec.term.closed* -

⟨proof⟩

lemma *none-inf-pre*:
  fixes P :: $'s$ *pred*
  fixes Q :: $('a, 's, 'v)$ *spec*
  shows *spec.term.none* $(Q \sqcap$ *spec.pre* $P) = ($*spec.term.none* $Q \sqcap$ *spec.pre* P :: $('a, 's, 'w)$ *spec*$)$ (**is** *?lhs* = *?rhs*)
    and *spec.term.none* (*spec.pre* $P \sqcap Q) = ($*spec.pre* $P \sqcap$ *spec.term.none* Q :: $('a, 's, 'w)$ *spec*$)$ (**is** *?thesis2*)
⟨proof⟩

⟨ML⟩

lemma *bot*[*iff*]:
  shows *spec.pre* ⟨*False*⟩ = ⊥
    and *spec.pre* ⊥ = ⊥
⟨proof⟩

lemma *top*[*iff*]:
  shows *spec.pre* ⟨*True*⟩ = ⊤
    and *spec.pre* ⊤ = ⊤
⟨proof⟩

lemma *top-conv*:
  shows *spec.pre* $P = (\top$ :: $('a, 's, 'v)$ *spec*$) \longleftrightarrow P = \top$
⟨proof⟩

lemma *K*:
  shows *spec.pre* ⟨P⟩ = (*if* P *then* ⊤ *else* ⊥)
⟨proof⟩

lemma *monotone*:
  shows *mono spec.pre*
⟨proof⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.pre.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.pre.monotone*]

lemma *SUP*:
  shows *spec.pre* $(\bigsqcup x \in X.\ P\ x) = (\bigsqcup x \in X.$ *spec.pre* $(P\ x))$
⟨proof⟩

lemma *Sup*:
  shows *spec.pre* $(\bigsqcup X) = (\bigsqcup x \in X.$ *spec.pre* $x)$
⟨proof⟩

lemma *Bex*:
  shows *spec.pre* $(\lambda s.\ \exists x \in X.\ P\ x\ s) = (\bigsqcup x \in X.$ *spec.pre* $(P\ x))$
⟨proof⟩

lemma *Ex*:
  shows *spec.pre* $(\lambda s.\ \exists x.\ P\ x\ s) = (\bigsqcup x.$ *spec.pre* $(P\ x))$
⟨proof⟩

lemma
  shows *disj*: *spec.pre* $(P \vee Q) =$ *spec.pre* $P \sqcup$ *spec.pre* Q
    and *sup*: *spec.pre* $(P \sqcup Q) =$ *spec.pre* $P \sqcup$ *spec.pre* Q
⟨proof⟩

lemma *INF*:

shows *spec.pre* $(\bigsqcap x{\in}X.\ P\ x) = (\bigsqcap x{\in}X.\ spec.pre\ (P\ x))$
⟨*proof*⟩

lemma *Inf*:
shows *spec.pre* $(\bigsqcap X) = (\bigsqcap x{\in}X.\ spec.pre\ x)$
⟨*proof*⟩

lemma *Ball*:
shows *spec.pre* $(\lambda s.\ \forall x{\in}X.\ P\ x\ s) = (\bigsqcap x{\in}X.\ spec.pre\ (P\ x))$
⟨*proof*⟩

lemma *All*:
shows *spec.pre* $(\lambda s.\ \forall x.\ P\ x\ s) = (\bigsqcap x.\ spec.pre\ (P\ x))$
⟨*proof*⟩

lemma *inf*:
shows *conj*: *spec.pre* $(P \wedge Q) =$ *spec.pre* $P \sqcap$ *spec.pre* Q
and *spec.pre* $(P \sqcap Q) =$ *spec.pre* $P \sqcap$ *spec.pre* Q
⟨*proof*⟩

lemma *inf-action-le*: — Converse does not hold
shows *spec.pre* $P \sqcap$ *spec.action* $F \leq$ *spec.action* $(UNIV \times UNIV \times Collect\ P \times UNIV \cap F)$ (**is** *?lhs* $\leq$ *?rhs*)
and *spec.action* $F \sqcap$ *spec.pre* $P \leq$ *spec.action* $(F \cap UNIV \times UNIV \times Collect\ P \times UNIV)$ (**is** *?thesis2*)
⟨*proof*⟩

⟨*ML*⟩

lemma *pre*:
shows *spec.invmap af sf vf* (*spec.pre* P) = *spec.pre* $(\lambda s.\ P\ (sf\ s))$
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-pre*:
shows *spec.pre* $P \sqcap (f \gg\!\!= g) =$ (*spec.pre* $P \sqcap f) \gg\!\!= g$ (**is** *?lhs* = *?rhs*)
and $(f \gg\!\!= g) \sqcap$ *spec.pre* $P = (f \sqcap$ *spec.pre* $P) \gg\!\!= g$ (**is** *?thesis1*)
⟨*proof*⟩

⟨*ML*⟩

lemma *pre*:
assumes $P\ s_0$
shows *spec.steps* (*spec.pre* P :: $('a, 's, 'v)$ *spec*) = $UNIV \times -\ Id$
⟨*proof*⟩

⟨*ML*⟩

8.11.2 Postconditions

Unlike *spec.pre spec.post* can be expressed in terms of other constants.

⟨*ML*⟩

definition *act* :: $('v \Rightarrow 's\ pred) \Rightarrow ('v \times 'a \times 's \times 's)\ set$ **where**
$act\ Q = \{(v, a, s, s') \mid v\ a\ s\ s'.\ Q\ v\ s'\}$

⟨*ML*⟩

lemma *simps*[*simp*]:

shows *spec.post.act* ⟨⟨*False*⟩⟩ = {}
and *spec.post.act* ⟨⊥⟩ = {}
and *spec.post.act* ⊥ = {}
and *spec.post.act* ⟨⟨*True*⟩⟩ = *UNIV*
and *spec.post.act* ⟨⊤⟩ = *UNIV*
and *spec.post.act* ⊤ = *UNIV*
and *spec.post.act* (*Q* ⊔ *Q′*) = *spec.post.act Q* ∪ *spec.post.act Q′*
and *spec.post.act* (⨆*X*) = (⋃*x*∈*X*. *spec.post.act x*)
and *spec.post.act* (λ*v*. ⨆*x*∈*Y*. *R x v*) = (⋃*x*∈*Y*. *spec.post.act* (*R x*))
⟨*proof*⟩

lemma *monotone*:
shows *mono spec.post.act*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.post.act.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.post.act.monotone*]

⟨*ML*⟩

definition *post* :: (*′v* ⇒ *′s pred*) ⇒ (*′a*, *′s*, *′v*) *spec* **where**
post Q = ⊤ ⋙ (λ-::*unit*. *spec.action* (*spec.post.act Q*))

⟨*ML*⟩

lemma *post-le-conv*[*spec.singleton.le-conv*]:
fixes *Q* :: *′v* ⇒ *′s pred*
shows ⦇σ⦈ ≤ *spec.post Q*
⟷ (*case trace.term* σ *of None* ⇒ *True* | *Some v* ⇒ *Q v* (*trace.final* σ)) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *post-le*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.post Q*
⟨*proof*⟩

⟨*ML*⟩

lemma *post-le*:
shows *spec.term.none P* ≤ *spec.post Q*
⟨*proof*⟩

lemma *post*:
shows *spec.term.none* (*spec.post Q* :: (*′a*, *′s*, *′v*) *spec*)
= *spec.term.none* (⊤ :: (*′a*, *′s*, *unit*) *spec*)
⟨*proof*⟩

⟨*ML*⟩

lemma *post*:
shows *spec.term.all* (*spec.post Q*) = ⊤
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*[*iff*]:
shows *spec.post* ⟨⟨*False*⟩⟩ = *spec.term.none* (⊤ :: (-, -, *unit*) *spec*)

and *spec.post* $\langle\bot\rangle$ = *spec.term.none* ($\top$ :: (-, -, *unit*) *spec*)
and *spec.post* $\bot$ = *spec.term.none* ($\top$ :: (-, -, *unit*) *spec*)
⟨*proof*⟩

lemma *monotone*:
shows *mono spec.post*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.post.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.post.monotone*]

lemma *SUP-not-empty*:
fixes X :: *'a set*
fixes Q :: *'a* $\Rightarrow$ *'v* $\Rightarrow$ *'s pred*
assumes $X \neq \{\}$
shows *spec.post* ($\lambda v.\ \bigsqcup x{\in}X.\ Q\ x\ v$) = ($\bigsqcup x{\in}X.$ *spec.post* ($Q\ x$))
⟨*proof*⟩

lemma *disj*:
shows *spec.post* ($Q \sqcup Q'$) = *spec.post* $Q \sqcup$ *spec.post* Q'
and *spec.post* ($\lambda rv.\ Q\ rv \sqcup Q'\ rv$) = *spec.post* $Q \sqcup$ *spec.post* Q'
and *spec.post* ($\lambda rv.\ Q\ rv \vee Q'\ rv$) = *spec.post* $Q \sqcup$ *spec.post* Q'
⟨*proof*⟩

lemma *INF*:
shows *spec.post* ($\bigsqcap x{\in}X.\ Q\ x$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
and *spec.post* ($\lambda v.\ \bigsqcap x{\in}X.\ Q\ x\ v$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
and *spec.post* ($\lambda v\ s.\ \bigsqcap x{\in}X.\ Q\ x\ v\ s$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
⟨*proof*⟩

lemma *Inf*:
shows *spec.post* ($\bigsqcap X$) = ($\bigsqcap x{\in}X.$ *spec.post* x)
⟨*proof*⟩

lemma *Ball*:
shows *spec.post* ($\lambda v\ s.\ \forall x{\in}X.\ Q\ x\ v\ s$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
⟨*proof*⟩

lemma *All*:
shows *spec.post* ($\lambda v\ s.\ \forall x.\ Q\ x\ v\ s$) = ($\bigsqcap x.$ *spec.post* ($Q\ x$))
⟨*proof*⟩

lemma *inf*:
shows *spec.post* ($Q \sqcap Q'$) = *spec.post* $Q \sqcap$ *spec.post* Q'
and *spec.post* ($\lambda rv.\ Q\ rv \sqcap Q'\ rv$) = *spec.post* $Q \sqcap$ *spec.post* Q'
and *conj*: *spec.post* ($\lambda rv.\ Q\ rv \wedge Q'\ rv$) = *spec.post* $Q \sqcap$ *spec.post* Q'
⟨*proof*⟩

lemma *top*[*iff*]:
shows *spec.post* ⟨⟨*True*⟩⟩ = $\top$
and *spec.post* $\langle\top\rangle$ = $\top$
and *spec.post* $\top$ = $\top$
⟨*proof*⟩

lemma *top-conv*:
shows *spec.post* Q = ($\top$ :: (*'a*, *'s*, *'v*) *spec*) $\longleftrightarrow$ $Q = \top$
⟨*proof*⟩

lemma *K*:
shows *spec.post* (λ- -. *Q*) = (*if Q then* ⊤ *else* ⊤ ⫺ (λ-::*unit*. ⊥))
⟨*proof*⟩

⟨*ML*⟩

lemma *bind-post-pre*:
shows *f* ⊓ *spec.post Q* ⫺ *g* = *f* ⫺ (λ*v*. *g v* ⊓ *spec.pre* (*Q v*)) (**is** *?lhs* = *?rhs*)
and *spec.post Q* ⊓ *f* ⫺ *g* = *f* ⫺ (λ*v*. *spec.pre* (*Q v*) ⊓ *g v*) (**is** *?thesis1*)
⟨*proof*⟩

⟨*ML*⟩

lemma *post*:
shows *spec.invmap af sf vf* (*spec.post Q*) = *spec.post* (λ*v s*. *Q* (*vf v*) (*sf s*))
⟨*proof*⟩

⟨*ML*⟩

lemma *post-le-conv*:
shows *spec.action F* ≤ *spec.post Q* ⟷ (∀ *v a s s*′. (*v*, *a*, *s*, *s*′) ∈ *F* ⟶ *Q v s*′)
⟨*proof*⟩

⟨*ML*⟩

lemma *post-le*:
assumes ⋀*v*. *g v* ≤ *spec.post Q*
shows *f* ⫺ *g* ≤ *spec.post Q*
⟨*proof*⟩

lemma *inf-post*:
shows (*f* ⫺ *g*) ⊓ *spec.post Q* = *f* ⫺ (λ*v*. *g v* ⊓ *spec.post Q*) (**is** *?lhs* = *?rhs*)
and *spec.post Q* ⊓ (*f* ⫺ *g*) = *f* ⫺ (λ*v*. *spec.post Q* ⊓ *g v*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *mono-stronger*:
assumes *f*: *f* ≤ *f*′ ⊓ *spec.post Q*
assumes *g*: ⋀*v*. *g v* ⊓ *spec.pre* (*Q v*) ≤ *g*′ *v*
shows *spec.bind f g* ≤ *spec.bind f*′ *g*′
⟨*proof*⟩

⟨*ML*⟩

8.11.3 Strongest postconditions

⟨*ML*⟩

definition *strongest* :: (′*a*, ′*s*, ′*v*) *spec* ⇒ ′*v* ⇒ ′*s pred* **where**
strongest P = ⨅{*Q*. *P* ≤ *spec.post Q*}

interpretation *strongest*: *galois.complete-lattice-class spec.post.strongest spec.post*
⟨*proof*⟩

lemma *strongest-alt-def*:
shows *spec.post.strongest P* = (λ*v s*. ∃σ. ⦃σ⦄ ≤ *P* ∧ *trace.term* σ = *Some v* ∧ *trace.final* σ = *s*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨ML⟩

lemma *singleton*:
  shows $spec.post.strongest\ \langle\!|\sigma|\!\rangle$
    $= (\lambda v\ s.\ case\ trace.term\ \sigma\ of\ None \Rightarrow False \mid Some\ v' \Rightarrow v' = v \wedge trace.final\ \sigma = s)$
⟨*proof*⟩

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 $spec.post.strongest\ \top = \top$
⟨*proof*⟩

lemma *action*:
  shows $spec.post.strongest\ (spec.action\ F) = (\lambda v\ s'.\ \exists\, a\ s.\ (v,\ a,\ s,\ s') \in F)$
⟨*proof*⟩

lemma *return*:
  shows $spec.post.strongest\ (spec.return\ v) = (\lambda v'\ s.\ v' = v)$
⟨*proof*⟩

⟨ML⟩

lemma *none*:
  shows $spec.post.strongest\ (spec.term.none\ P) = \bot$
⟨*proof*⟩

lemma *all*:
  assumes $spec.idle \leq P$
  shows $spec.post.strongest\ (spec.term.all\ P) = \top$
⟨*proof*⟩

lemma *closed*:
  assumes $spec.idle \leq P$
  assumes $P \in spec.term.closed$ -
  shows $spec.post.strongest\ P = \top$
⟨*proof*⟩

⟨ML⟩

lemma *bind*:
  shows $spec.post.strongest\ (f \gg\!= g)$
    $= spec.post.strongest\ (\bigsqcup v.\ spec.pre\ (spec.post.strongest\ f\ v) \sqcap g\ v)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rel*:
  shows $spec.post.strongest\ (spec.rel\ r) = \top$
⟨*proof*⟩

lemma *pre*:
  shows $spec.post.strongest\ (spec.pre\ P) = (\lambda v\ s'.\ \exists\, s.\ P\ s)$
⟨*proof*⟩

lemma *post*:
  shows $spec.post.strongest\ (spec.post\ Q) = Q$

⟨*proof*⟩

⟨*ML*⟩

8.12 Initial steps

The initial transition of a process.

⟨*ML*⟩

definition *initial-steps* :: $('a, 's, 'v)$ *spec* $\Rightarrow$ $('a, 's)$ *steps* **where**
initial-steps $P = \{(a, s, s'). \langle\!|s, [(a, s')], None|\!\rangle \le P\}$

⟨*ML*⟩

lemma *steps-le*:
shows *spec.initial-steps* $P \subseteq$ *spec.steps* $P \cup UNIV \times Id$
⟨*proof*⟩

lemma *galois*:
shows $r \subseteq$ *spec.initial-steps* $P \wedge$ *spec.idle* $\le P \longleftrightarrow$ *spec.action* $(\{()\} \times r) \gg\!= \bot \le P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *bot*:
shows *spec.initial-steps* $\bot = \{\}$
⟨*proof*⟩

lemma *top*:
shows *spec.initial-steps* $\top = UNIV$
⟨*proof*⟩

lemma *monotone*:
shows *mono spec.initial-steps*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.initial-steps.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.initial-steps.monotone*]

lemma *Sup*:
shows *spec.initial-steps* $(\bigsqcup X) = \bigcup($*spec.initial-steps* ‘ $X)$
⟨*proof*⟩

lemma *Inf*:
shows *spec.initial-steps* $(\bigsqcap X) = \bigcap($*spec.initial-steps* ‘ $X)$
⟨*proof*⟩

lemma *idle*:
shows *spec.initial-steps spec.idle* $= UNIV \times Id$
⟨*proof*⟩

lemma *action*:
shows *spec.initial-steps* (*spec.action* F) = *snd* ‘ $F \cup UNIV \times Id$
⟨*proof*⟩

lemma *return*:
shows *spec.initial-steps* (*spec.return* v) $= UNIV \times Id$
⟨*proof*⟩

lemma *bind*:

shows *spec.initial-steps* $(f \gg g)$
$= spec.initial\text{-}steps\ f$
$\cup\ spec.initial\text{-}steps\ (\bigsqcup v.\ spec.pre\ (spec.post.strongest\ (f \sqcap spec.return\ v)\ v) \sqcap g\ v)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *rel*:
shows $spec.initial\text{-}steps\ (spec.rel\ r) = r \cup UNIV \times Id$
⟨*proof*⟩

lemma *pre*:
shows $spec.initial\text{-}steps\ (spec.pre\ P) = UNIV \times Pre\ P$
⟨*proof*⟩

lemma *post*:
shows $spec.initial\text{-}steps\ (spec.post\ Q) = UNIV$
⟨*proof*⟩

⟨*ML*⟩

lemma *none*:
shows $spec.initial\text{-}steps\ (spec.term.none\ P) = spec.initial\text{-}steps\ P$
⟨*proof*⟩

lemma *all*:
shows $spec.initial\text{-}steps\ (spec.term.all\ P) = spec.initial\text{-}steps\ P$
⟨*proof*⟩

⟨*ML*⟩

8.13 Heyting implication

⟨*ML*⟩

lemma *heyting-le-conv*:
shows $\langle\!|\sigma|\!\rangle \le P \longrightarrow_H Q \longleftrightarrow (\forall \sigma' \le \sigma.\ \langle\!|\sigma'|\!\rangle \le P \longrightarrow \langle\!|\sigma'|\!\rangle \le Q)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

Connect the generic definition of Heyting implication to a concrete one in the model.

lift-definition *heyting* :: $('a, 's, 'v)\ spec \Rightarrow ('a, 's, 'v)\ spec \Rightarrow ('a, 's, 'v)\ spec$ **is**
downwards.imp
⟨*proof*⟩

lemma *heyting-alt-def*:
shows $(\longrightarrow_H) = (spec.heyting :: \text{-}\Rightarrow\text{-}\Rightarrow('a, 's, 'v)\ spec)$
⟨*proof*⟩

declare *spec.heyting.transfer*[*transfer-rule del*]

⟨*ML*⟩

lemma *transfer-alt*[*transfer-rule*]:
shows *rel-fun* (*pcr-spec* (=) (=) (=)) (*rel-fun* (*pcr-spec* (=) (=) (=)) (*pcr-spec* (=) (=) (=))) *downwards.imp*
$(\longrightarrow_H)$
⟨*proof*⟩

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 $((P \longrightarrow_H Q) \longrightarrow_H R) \sqcap ((Q \longrightarrow_H P) \longrightarrow_H R) \leq R$
⟨*proof*⟩

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*⟩

lemma *cont*:
  fixes P :: (-, -, -) *spec*
  shows *cont Sup* (≤) *Sup* (≤) $((\longrightarrow_H)\ P)$
⟨*proof*⟩

lemma *mcont*:
  fixes P :: (-, -, -) *spec*
  shows *mcont Sup* (≤) *Sup* (≤) $((\longrightarrow_H)\ P)$
⟨*proof*⟩

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF spec.heyting.mcont*, *of luba orda Q P*] **for** *luba orda Q P*

lemma *non-triv*:
  shows $P \longrightarrow_H \bot \leq P \longleftrightarrow$ *spec.idle* ≤ P (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *post*:
  shows *spec.post* $Q \longrightarrow_H$ *spec.post* Q' = *spec.post* $(\lambda v\ s.\ Q\ v\ s \longrightarrow Q'\ v\ s)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *heyting*:
  shows *spec.invmap af sf vf* $(P \longrightarrow_H Q)$ = *spec.invmap af sf vf* $P \longrightarrow_H$ *spec.invmap af sf vf* Q (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *heyting-noneL-allR-mono*:
  fixes P :: (-, -, 'v) *spec*
  fixes Q :: (-, -, 'v) *spec*
  shows *spec.term.none* $P \longrightarrow_H Q = P \longrightarrow_H$ *spec.term.all* Q (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *heyting*: — polymorphic *spec.term.all*
  fixes P :: (-, -, 'v) *spec*
  fixes Q :: (-, -, 'v) *spec*
  shows (*spec.term.all* $(P \longrightarrow_H Q)$ :: (-, -, 'w) *spec*)
    = *spec.term.none* $P \longrightarrow_H$ *spec.term.all* Q (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *heyting-le*:
  shows *spec.term.none* $(P \longrightarrow_H Q) \leq$ *spec.term.all* $P \longrightarrow_H$ *spec.term.none* Q

⟨*proof*⟩

⟨*ML*⟩

lemma *heyting*:
  assumes $Q \in$ *spec.term.closed* -
  shows $P \longrightarrow_H Q \in$ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

8.14 Miscellaneous algebra

⟨*ML*⟩

lemma *bind*:
  shows *spec.steps* $(f \gg= g)$
    $=$ *spec.steps* $f \cup (\bigcup v.$ *spec.steps* (*spec.pre* (*spec.post.strongest* f v) $\sqcap$ g v)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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*⟩

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*⟩

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*⟩

lemma *rel-le*:
  shows *spec.map af sf vf* (*spec.rel* r) $\leq$ *spec.rel* (*map-prod af* (*map-prod sf sf*) ‘ r)
⟨*proof*⟩

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-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-relE*[*consumes 1*]:
  fixes *xs* :: (*'b* × *'t*) *list*
  assumes *trace.steps' s xs* ⊆ *map-prod af* (*map-prod sf sf*) ‘ *r*
  obtains (*Idle*) *snd* ‘ *set xs* ⊆ {*s*}
    | (*Step*) *s' xs'*
      where *sf s'* = *s*
      and *trace.natural' s xs* = *map* (*map-prod af sf*) *xs'*
      and *trace.steps' s' xs'* ⊆ *r*
⟨*proof*⟩

lemma *rel*:
  shows *spec.map af sf vf* (*spec.rel r*)
    = *spec.rel* (*map-prod af* (*map-prod sf sf*) ‘ *r*)
    ⊓ *spec.pre* (λ*s*. *s* ∈ *range sf*)
    ⊓ *spec.post* (λ*v s*. *v* ∈ *range vf*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pre*:
  fixes *P* :: *'t pred*
  shows *spec.map af sf vf* (*spec.pre* (λ*s*. *P* (*sf s*)))
    = *spec.pre* (λ*s*. *P s* ∧ *s* ∈ *range sf*) ⊓ *spec.post* (λ*v s*. *s* ∈ *range sf* ⟶ *v* ∈ *range vf*)
      ⊓ *spec.rel* (*range af* × *range sf* × *range sf*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *post*:
  fixes *Q* :: *'w* ⇒ *'t pred*
  shows *spec.map af sf vf* (*spec.post* (λ*v s*. *Q* (*vf v*) (*sf s*)))
    = *spec.pre* (λ*s*. *s* ∈ *range sf*) ⊓ *spec.post* (λ*v s*. *s* ∈ *range sf* ⟶ *Q v s* ∧ *v* ∈ *range vf*)
      ⊓ *spec.rel* (*range af* × *range sf* × *range sf*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

end

⟨*ML*⟩

lemma *idle*:
  shows *spec.invmap af sf vf spec.idle*
    = *spec.term.none* (*spec.rel* (*UNIV* × *map-prod sf sf* −‘ *Id*) :: (*'a*, *'s*, *unit*) *spec*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *inf-rel*:
  shows *spec.rel* (*map-prod af* (*map-prod sf sf*) −‘ (*r* ∪ *UNIV* × *Id*)) ⊓ *spec.invmap af sf vf P* = *spec.invmap af sf vf* (*spec.rel r* ⊓ *P*)
  and *spec.invmap af sf vf P* ⊓ *spec.rel* (*map-prod af* (*map-prod sf sf*) −‘ (*r* ∪ *UNIV* × *Id*)) = *spec.invmap af sf vf* (*P* ⊓ *spec.rel r*)
⟨*proof*⟩

lemma *action*: — (* could restrict the stuttering expansion to *range af* or an arbitrary element of that
  fixes *af* :: *'a* ⇒ *'b*
  fixes *sf* :: *'s* ⇒ *'t*
  fixes *vf* :: *'v* ⇒ *'w*
  fixes *F* :: (*'w* × *'b* × *'t* × *'t*) *set*
  defines *F'* ≡ *map-prod id* (*map-prod af* (*map-prod sf sf*))

$-` (F \cup \{(v, a', s, s) \mid v\ a\ a'\ s.\ (v, a, s, s) \in F \wedge \neg surj\ af\})$
shows *spec.invmap af sf vf* (*spec.action F*)
$= spec.rel\ (UNIV \times map\text{-}prod\ sf\ sf\ -`\ Id)$
$\ggg (\lambda\text{-}::unit.\ spec.action\ F'$
$\ggg (\lambda v.\ spec.rel\ (UNIV \times map\text{-}prod\ sf\ sf\ -`\ Id)$
$\ggg (\lambda\text{-}::unit.\ \bigsqcup v' \in vf\ -`\ \{v\}.\ spec.return\ v')))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

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*)
$= spec.rel\ (UNIV \times map\text{-}prod\ sf\ sf\ -`\ Id) \ggg (\lambda\text{-}::unit.\ \bigsqcup v' \in vf\ -`\ \{v\}.\ spec.return\ v')$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

definition $cl :: ('a, 's)\ steps \Rightarrow ('a, 's, 'v)\ spec \Rightarrow ('a, 's, 'v)\ spec$ **where**
$cl\ r\ P = P \sqcup spec.term.none\ (spec.term.all\ P \ggg (\lambda\text{-}::unit.\ spec.rel\ r :: (\text{-}, \text{-}, unit)\ spec))$

⟨*ML*⟩

lemma *cl*:
shows *spec.term.none* (*spec.cam.cl r P*) = *spec.cam.cl r* (*spec.term.none P*)
⟨*proof*⟩

lemma *cl-rel-wind*:
fixes $P :: ('a, 's, 'v)\ spec$
shows $spec.cam.cl\ r\ P \gg spec.term.none\ (spec.rel\ r :: ('a, 's, 'w)\ spec)$
$= spec.term.none\ (spec.cam.cl\ r\ P)$
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le*: — Converse does not hold
shows $spec.cam.cl\ r\ (spec.term.all\ P) \le spec.term.all\ (spec.cam.cl\ r\ P)$
⟨*proof*⟩

⟨*ML*⟩

interpretation *cam*: *closure-complete-distrib-lattice-distributive-class spec.cam.cl r* **for** r :: ($'a$, $'s$) *steps*
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*[*simp*]:
  shows *spec.cam.cl r* $\bot = \bot$
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *Sup*:
  shows *spec.cam.cl r* ($\bigsqcup X$) = ($\bigsqcup P \in X$. *spec.cam.cl r P*)
⟨*proof*⟩

lemmas *sup* = *spec.cam.cl.Sup*[**where** X={P, Q} **for** P Q, *simplified*]

lemma *rel-empty*:
  shows *spec.cam.cl* {} $P = P$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-minus-Id*:
  shows *spec.cam.cl* ($r - UNIV \times Id$) P = *spec.cam.cl r P*
⟨*proof*⟩

lemma *Inf*:
  shows *spec.cam.cl r* ($\bigsqcap X$) = $\bigsqcap$(*spec.cam.cl r* ‘ X) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bind*:
  shows *spec.cam.cl r* ($f \gg= g$) = *spec.cam.cl r f* $\gg=$ (λv. *spec.cam.cl r* (g v))
⟨*proof*⟩

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* $F \gg$ (*spec.rel* r :: (-, -, *unit*) *spec*))
    $\sqcup$ *spec.term.none* (*spec.rel* r :: (-, -, *unit*) *spec*)
⟨*proof*⟩

lemma *return*:
  shows *spec.cam.cl* r (*spec.return* v) $=$ *spec.return* v $\sqcup$ *spec.term.none* (*spec.rel* r :: (-, -, *unit*) *spec*)
⟨*proof*⟩

lemma *rel-le*:
  assumes $r \subseteq r' \vee r' \subseteq r$
  shows *spec.cam.cl* r (*spec.rel* r') $\leq$ *spec.rel* ($r \cup r'$)
⟨*proof*⟩

lemma *rel*:
  assumes $r \subseteq r'$
  shows *spec.cam.cl* r (*spec.rel* r') $=$ *spec.rel* r'
⟨*proof*⟩

lemma *inf-rel*:
  fixes r :: ($'a$, $'s$) *steps*
  fixes s :: ($'a$, $'s$) *steps*
  fixes P :: ($'a$, $'s$, $'v$) *spec*
  shows *spec.rel* r $\sqcap$ *spec.cam.cl* r' P $=$ *spec.cam.cl* ($r \cap r'$) (*spec.rel* r $\sqcap$ P) (**is** *?thesis1*)
   and *spec.cam.cl* r' P $\sqcap$ *spec.rel* r $=$ *spec.cam.cl* ($r \cap r'$) (*spec.rel* r $\sqcap$ P) (**is** *?thesis2*)
⟨*proof*⟩

lemma *bind-return*:
  shows *spec.cam.cl* r ($f \gg$ *spec.return* v) $=$ *spec.cam.cl* r $f \gg$ *spec.return* v
⟨*proof*⟩

lemma *heyting-le*:
  shows *spec.cam.cl* r ($P \longrightarrow_H Q$) $\leq$ $P \longrightarrow_H$ *spec.cam.cl* r Q
⟨*proof*⟩

lemma *pre*:
  shows *spec.cam.cl* r (*spec.pre* P) $=$ *spec.pre* P
⟨*proof*⟩

lemma *post*:
  shows *spec.cam.cl* r (*spec.post* Q) $=$ *spec.post* Q
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
  shows *spec.cam.closed* {} $=$ *UNIV*
⟨*proof*⟩

lemma *antimonotone*:
  shows *antimono spec.cam.closed*
⟨*proof*⟩

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 *spec.cam.closed* ($r \cup A \times Id$) = *spec.cam.closed r*
⟨*proof*⟩

⟨*ML*⟩

lemma *none*:
  assumes $P \in$ *spec.cam.closed r*
  shows *spec.term.none* $P \in$ *spec.cam.closed r*
⟨*proof*⟩

⟨*ML*⟩

lemma *bind*:
  fixes f :: (′*a*, ′*s*, ′*v*) *spec*
  fixes g :: ′*v* ⇒ (′*a*, ′*s*, ′*w*) *spec*
  assumes $f \in$ *spec.cam.closed r*
  assumes $\bigwedge x.\ g\ x \in$ *spec.cam.closed r*
  shows $f \gg= g \in$ *spec.cam.closed r*
⟨*proof*⟩

lemma *rel*[*intro*]:
  assumes $r \subseteq r'$
  shows *spec.rel* $r' \in$ *spec.cam.closed r*
⟨*proof*⟩

lemma *pre*[*intro*]:
  shows *spec.pre* $P \in$ *spec.cam.closed r*
⟨*proof*⟩

lemma *post*[*intro*]:
  shows *spec.post* $Q \in$ *spec.cam.closed r*
⟨*proof*⟩

lemma *heyting*[*intro*]:
  assumes $Q \in$ *spec.cam.closed r*
  shows $P \longrightarrow_H Q \in$ *spec.cam.closed r*
⟨*proof*⟩

lemma *snoc-conv*:
  fixes P :: (′*a*, ′*s*, ′*v*) *spec*
  assumes $P \in$ *spec.cam.closed r*
  assumes (*fst x*, *trace.final′ s xs*, *snd x*) $\in r \cup$ *UNIV* $\times$ *Id*
  shows ⟪*s*, *xs* @ [*x*], *None*⟫ $\leq P \longleftrightarrow$ ⟪*s*, *xs*, *None*⟫ $\leq P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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*)

$= \mathit{spec.cam.cl}\ (\mathit{map\text{-}prod}\ \mathit{af}\ (\mathit{map\text{-}prod}\ \mathit{sf}\ \mathit{sf})\ -`\ (r \cup \mathit{UNIV} \times \mathit{Id}))\ (\mathit{spec.invmap}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ P)$
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le*:
fixes $\mathit{af} :: 'a \Rightarrow 'b$
fixes $\mathit{sf} :: 's \Rightarrow 't$
fixes $\mathit{vf} :: 'v \Rightarrow 'w$
fixes $r :: ('a, 's)\ \mathit{steps}$
fixes $P :: ('a, 's, 'v)\ \mathit{spec}$
shows $\mathit{spec.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ (\mathit{spec.cam.cl}\ r\ P)$
$\leq \mathit{spec.cam.cl}\ (\mathit{map\text{-}prod}\ \mathit{af}\ (\mathit{map\text{-}prod}\ \mathit{sf}\ \mathit{sf})\ `\ r)\ (\mathit{spec.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ P)$
⟨*proof*⟩

lemma *cl-inj-sf*:
fixes $\mathit{af} :: 'a \Rightarrow 'b$
fixes $\mathit{sf} :: 's \Rightarrow 't$
fixes $\mathit{vf} :: 'v \Rightarrow 'w$
fixes $r :: ('a, 's)\ \mathit{steps}$
fixes $P :: ('a, 's, 'v)\ \mathit{spec}$
assumes *inj sf*
shows $\mathit{spec.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ (\mathit{spec.cam.cl}\ r\ P)$
$= \mathit{spec.cam.cl}\ (\mathit{map\text{-}prod}\ \mathit{af}\ (\mathit{map\text{-}prod}\ \mathit{sf}\ \mathit{sf})\ `\ r)\ (\mathit{spec.map}\ \mathit{af}\ \mathit{sf}\ \mathit{vf}\ P)$
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

abbreviation $\mathit{ap\text{-}cam\text{-}cl} :: 'a\ \mathit{set} \Rightarrow ('a, 's, 'v)\ \mathit{spec} \Rightarrow ('a, 's, 'v)\ \mathit{spec}$ **where**
$\mathit{ap\text{-}cam\text{-}cl}\ \mathit{as} \equiv \mathit{spec.cam.cl}\ ((-\mathit{as}) \times \mathit{UNIV})$

abbreviation (*input*) $\mathit{ap\text{-}cam\text{-}closed} :: 'a\ \mathit{set} \Rightarrow ('a, 's, 'v)\ \mathit{spec}\ \mathit{set}$ **where**
$\mathit{ap\text{-}cam\text{-}closed}\ \mathit{as} \equiv \mathit{spec.cam.closed}\ ((-\mathit{as}) \times \mathit{UNIV})$

lemma *composition-principle-1*:
fixes $P :: ('a, 's, 'v)\ \mathit{spec}$
assumes $P \in \mathit{spec.ap\text{-}cam\text{-}closed}\ \mathit{as}$
assumes $P \in \mathit{spec.term.closed}$ -
assumes $\mathit{spec.idle} \leq P$
shows $\mathit{spec.ap\text{-}cam\text{-}cl}\ (-\mathit{as})\ P \longrightarrow_H P \leq P$ (**is** $?lhs \leq ?rhs$)
⟨*proof*⟩

lemma *composition-principle-half*: — Abadi and Plotkin (1993, §3.1(4)) – cleaner than in Abadi and Plotkin (1991, §3.1)
assumes $M_1 \in \mathit{spec.ap\text{-}cam\text{-}closed}\ a_1$
assumes $M_2 \in \mathit{spec.ap\text{-}cam\text{-}closed}\ a_2$
assumes $M_1 \in \mathit{spec.term.closed}$ -
assumes $\mathit{spec.idle} \leq M_1$
assumes $a_1 \cap a_2 = \{\}$
shows $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1$
⟨*proof*⟩

theorem *composition-principle*: — Abadi and Plotkin (1993, §3.1(3))
  assumes $M_1 \in$ *spec.ap-cam-closed* a_1
  assumes $M_2 \in$ *spec.ap-cam-closed* a_2
  assumes $M_1 \in$ *spec.term.closed* -
  assumes $M_2 \in$ *spec.term.closed* -
  assumes *spec.idle* $\leq M_1$
  assumes *spec.idle* $\leq M_2$
  assumes $a_1 \cap a_2 = \{\}$
  shows $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1 \sqcap M_2$
⟨*proof*⟩

An infinitary variant can be established in essentially the same way as *spec.composition-principle-1*.

lemma *ag-circular*:
  fixes *Ps* :: $'a \Rightarrow ('a, 's, 'v)$ *spec*
  assumes *cam-closed*: $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \in$ *spec.ap-cam-closed* $\{a\}$
  assumes *term-closed*: $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \in$ *spec.term.closed* -
  assumes *idle*: $\bigwedge a.\ a \in as \Longrightarrow$ *spec.idle* $\leq Ps\ a$
  shows $(\bigsqcap a{\in}as.\ (\bigsqcap a'{\in}as{-}\{a\}.\ Ps\ a') \longrightarrow_H Ps\ a) \leq (\bigsqcap a{\in}as.\ Ps\ a)$ (**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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 $('a, 's, 'v)$ *spec* lattice (see §13). Observations:

- if processes are made out of actions then it is not necessary to apply *spec.cam.cl*

⟨*ML*⟩

definition *cl* :: $('a, 's)$ *steps* $\Rightarrow ('a, 's, 'v)$ *spec* $\Rightarrow ('a, 's, 'v)$ *spec* **where**
  cl r P = *spec.rel r* $\gg=$ (λ-::*unit*. *spec.cam.cl r P*) $\gg=$ (λ*v*. *spec.rel r* $\gg=$ (λ-::*unit*. *spec.return v*))

⟨*ML*⟩

interpretation *interference*: *closure-complete-distrib-lattice-distributive-class spec.interference.cl r*
    for *r* :: $('a, 's)$ *steps*
⟨*proof*⟩

⟨*ML*⟩

lemma *cl*:
  shows *spec.term.none* (*spec.interference.cl r P*) = *spec.interference.cl r* (*spec.term.none P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *rel-le*:
  assumes $P \in$ *spec.interference.closed r*

shows *spec.term.none* (*spec.rel r*) $\leq$ *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le*: — Converse does not hold
shows *spec.interference.cl r* (*spec.term.all P*) $\leq$ *spec.term.all* (*spec.interference.cl r P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cl*:
shows *spec.interference.cl r P* $\in$ *spec.cam.closed r*
⟨*proof*⟩

lemma *closed-subseteq*:
shows *spec.interference.closed r* $\subseteq$ *spec.cam.closed r*
⟨*proof*⟩

⟨*ML*⟩

lemma *mono*:
assumes $r \subseteq r'$
assumes $P \leq P'$
shows *spec.interference.cl r P* $\leq$ *spec.interference.cl r' P'*
⟨*proof*⟩

declare *spec.interference.strengthen-cl*[*strg del*]

lemma *strengthen*[*strg*]:
assumes *st-ord F r r'*
assumes *st-ord F P P'*
shows *st-ord F* (*spec.interference.cl r P*) (*spec.interference.cl r' P'*)
⟨*proof*⟩

lemma *bot*:
shows *spec.interference.cl r* $\bot$ = *spec.term.none* (*spec.rel r* :: (-, -, *unit*) *spec*)
⟨*proof*⟩

lemmas *Sup* = *spec.interference.cl-Sup*
lemmas *sup* = *spec.interference.cl-sup*

lemma *idle*:
shows *spec.interference.cl r spec.idle* = *spec.term.none* (*spec.rel r* :: (-, -, *unit*) *spec*)
⟨*proof*⟩

lemma *rel-empty*:
assumes *spec.idle* $\leq$ *P*
shows *spec.interference.cl* {} *P* = *P*
⟨*proof*⟩

lemma *rel-reflcl*:
shows *spec.interference.cl* ($r \cup A \times Id$) *P* = *spec.interference.cl r P*
and *spec.interference.cl* ($A \times Id \cup r$) *P* = *spec.interference.cl r P*
⟨*proof*⟩

lemma *rel-minus-Id*:
shows *spec.interference.cl* ($r - UNIV \times Id$) *P* = *spec.interference.cl r P*

⟨*proof*⟩

lemma *inf-rel*:
shows *spec.interference.cl s P* ⊓ *spec.rel r* = *spec.interference.cl* (*r* ∩ *s*) (*spec.rel r* ⊓ *P*)
and *spec.rel r* ⊓ *spec.interference.cl s P* = *spec.interference.cl* (*r* ∩ *s*) (*spec.rel r* ⊓ *P*)
⟨*proof*⟩

lemma *bindL*:
assumes *f* ∈ *spec.interference.closed r*
shows *spec.interference.cl r* (*f* ⤜ *g*) = *f* ⤜ (λ*v*. *spec.interference.cl r* (*g v*))
⟨*proof*⟩

lemma *bindR*:
assumes ⋀*v*. *g v* ∈ *spec.interference.closed r*
shows *spec.interference.cl r* (*f* ⤜ *g*) = *spec.interference.cl r f* ⤜ *g* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *bind-conv*:
assumes *f* ∈ *spec.interference.closed r*
assumes ∀ *x*. *g x* ∈ *spec.interference.closed r*
shows *spec.interference.cl r* (*f* ⤜ *g*) = *f* ⤜ *g*
⟨*proof*⟩

lemma *action*:
shows *spec.interference.cl r* (*spec.action F*)
= *spec.rel r* ⤜ (λ-::*unit*. *spec.action F* ⤜ (λ*v*. *spec.rel r* ⤜ (λ-::*unit*. *spec.return v*)))
⟨*proof*⟩

lemma *return*:
shows *spec.interference.cl r* (*spec.return v*) = *spec.rel r* ⤜ (λ-::*unit*. *spec.return v*)
⟨*proof*⟩

lemma *bind-return*:
shows *spec.interference.cl r* (*f* ≫ *spec.return v*) = *spec.interference.cl r f* ≫ *spec.return v*
⟨*proof*⟩

lemma *rel*: — complicated by polymorphic *spec.rel*
assumes $r \subseteq r' \vee r' \subseteq r$
shows *spec.interference.cl r* (*spec.rel r′*) = *spec.rel* (*r* ∪ *r′*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.interference.cl r P*
⟨*proof*⟩

lemma *closed-le*[*spec.idle-le*]:
assumes *P* ∈ *spec.interference.closed r*
shows *spec.idle* ≤ *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-sf-id*:
shows *spec.map af id vf* (*spec.interference.cl r P*)

$= spec.interference.cl\ (map\text{-}prod\ af\ id\ `\ r)\ (spec.map\ af\ id\ vf\ P)$
⟨proof⟩

⟨ML⟩

lemma *cl*:
fixes $as :: 'b\ set$
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
fixes $r :: ('b, 't)\ steps$
fixes $P :: ('b, 't, 'w)\ spec$
shows $spec.invmap\ af\ sf\ vf\ (spec.interference.cl\ r\ P)$
$= spec.interference.cl\ (map\text{-}prod\ af\ (map\text{-}prod\ sf\ sf) -`\ (r \cup UNIV \times Id))\ (spec.invmap\ af\ sf\ vf\ P)$
⟨proof⟩

⟨ML⟩

lemma *antimonotone*:
shows *antimono spec.interference.closed*
⟨proof⟩

lemmas *strengthen*[*strg*] = *st-ord-antimono*[*OF spec.interference.closed.antimonotone*]
lemmas *antimono* = *antimonoD*[*OF spec.interference.closed.antimonotone*]

lemma *Sup'*:
assumes $X \subseteq spec.interference.closed\ r$
shows $\bigsqcup X \sqcup spec.term.none\ (spec.rel\ r :: (\text{-}, \text{-}, unit)\ spec) \in spec.interference.closed\ r$
⟨proof⟩

lemma *Sup-not-empty*:
assumes $X \subseteq spec.interference.closed\ r$
assumes $X \neq \{\}$
shows $\bigsqcup X \in spec.interference.closed\ r$
⟨proof⟩

lemma *rel*:
assumes $r' \subseteq r$
shows $spec.rel\ r \in spec.interference.closed\ r'$
⟨proof⟩

lemma *bind-relL*:
fixes $P :: ('a, 's, 'v)\ spec$
assumes $P \in spec.interference.closed\ r$
shows $spec.rel\ r \gg= (\lambda\text{-}::unit.\ P) = P$
⟨proof⟩

lemma *bind-relR*:
assumes $P \in spec.interference.closed\ r$
shows $P \gg= (\lambda v.\ spec.rel\ r \gg= (\lambda\text{-}::unit.\ Q\ v)) = P \gg= Q$
⟨proof⟩

lemma *bind-rel-unitR*:
assumes $P \in spec.interference.closed\ r$
shows $P \gg (spec.rel\ r :: (\text{-}, \text{-}, unit)\ spec) = P$
⟨proof⟩

lemma *bind-rel-botR*:

assumes $P \in$ *spec.interference.closed r*
shows $P \gg= (\lambda v.\ spec.rel\ r \gg= (\lambda\text{-}::unit.\ \bot)) = P \gg= \bot$
⟨*proof*⟩

lemma *bind*[*intro*]:
fixes f :: ('*a*, '*s*, '*v*) *spec*
fixes g :: '*v* ⇒ ('*a*, '*s*, '*w*) *spec*
assumes $f \in$ *spec.interference.closed r*
assumes $\bigwedge x.\ g\ x \in$ *spec.interference.closed r*
shows $(f \gg= g) \in$ *spec.interference.closed r*
⟨*proof*⟩

lemma *kleene-star*:
assumes $P \in$ *spec.interference.closed r*
assumes *spec.return* () $\leq P$
shows *spec.kleene.star P* $\in$ *spec.interference.closed r*
⟨*proof*⟩

lemma *map-sf-id*:
fixes *af* :: '*a* ⇒ '*b*
fixes *vf* :: '*v* ⇒ '*w*
assumes $P \in$ *spec.interference.closed r*
shows *spec.map af id vf P* $\in$ *spec.interference.closed* (*map-prod af id* ‘ *r*)
⟨*proof*⟩

lemma *invmap*:
fixes *af* :: '*a* ⇒ '*b*
fixes *sf* :: '*s* ⇒ '*t*
fixes *vf* :: '*v* ⇒ '*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*)
⟨*proof*⟩

⟨*ML*⟩

lemma *none*:
assumes $P \in$ *spec.interference.closed r*
shows *spec.term.none P* $\in$ *spec.interference.closed r*
⟨*proof*⟩

⟨*ML*⟩

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* ≡ *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* ∉ *range proc*
⟨*proof*⟩

lemma *range-proc-conv*[*simp*]:
  shows $x \in range\ proc \longleftrightarrow x \neq env$
⟨*proof*⟩

lemma *inj-proc*[*iff*]:
  shows *inj proc*
⟨*proof*⟩

lemma *surj-the-inv-proc*[*iff*]:
  shows *surj* (*the-inv proc*)
⟨*proof*⟩

lemma *the-inv-proc*[*simp*]:
  shows $the\text{-}inv\ proc\ (proc\ a) = a$
⟨*proof*⟩

lemma *uminus-env-range-proc*[*simp*]:
  shows $-\{env\} = range\ proc$
⟨*proof*⟩

lemma *env-range-proc-UNIV*[*simp*]:
  shows $insert\ env\ (range\ proc) = UNIV$
⟨*proof*⟩

⟨*ML*⟩

lemma *not-conv*[*simp*]:
  shows $a \neq env \longleftrightarrow a = self$
    and $a \neq self \longleftrightarrow a = env$
⟨*proof*⟩

lemma *range-proc-self*[*simp*]:
  shows $range\ proc = \{self\}$
⟨*proof*⟩

lemma *UNIV*:
  shows $UNIV = \{env, self\}$
⟨*proof*⟩

lemma *rev-UNIV*[*simp*]:
  shows $\{env, self\} = UNIV$
    and $\{self, env\} = UNIV$
⟨*proof*⟩

lemma *uminus-self-env*[*simp*]:
  shows $-\{self\} = \{env\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *eq-conv*:
  shows $map\text{-}agent\ f\ x = env \longleftrightarrow x = env$
    and $env = map\text{-}agent\ f\ x \longleftrightarrow x = env$
    and $map\text{-}agent\ f\ x = proc\ a \longleftrightarrow (\exists\, a'.\ x = proc\ a' \land a = f\ a')$
    and $proc\ a = map\text{-}agent\ f\ x \longleftrightarrow (\exists\, a'.\ x = proc\ a' \land a = f\ a')$
⟨*proof*⟩

lemma *surj*:

fixes π :: $'a \Rightarrow 'b$
assumes *surj* π
shows *surj* (*map-agent* π)
⟨*proof*⟩

lemma *bij*:
fixes π :: $'a \Rightarrow 'b$
assumes *bij* π
shows *bij* (*map-agent* π)
⟨*proof*⟩

⟨*ML*⟩

definition *swap-env-self-fn* :: *sequential* ⇒ *sequential* **where**
swap-env-self-fn a = (*case a of proc* () ⇒ *env* | *env* ⇒ *self*)

lemma *swap-env-self-fn-simps*:
shows *swap-env-self-fn self* = *env*
swap-env-self-fn env = *self*
⟨*proof*⟩

lemma *bij-swap-env-self-fn*:
shows *bij swap-env-self-fn*
⟨*proof*⟩

lemma *swap-env-self-fn-vimage-singleton*:
shows *swap-env-self-fn* −' {*env*} = {*self*}
and *swap-env-self-fn* −' {*self*} = {*env*}
⟨*proof*⟩

⟨*ML*⟩

abbreviation *swap-env-self* :: (*sequential*, *'s*, *'v*) *spec* ⇒ (*sequential*, *'s*, *'v*) *spec* **where**
swap-env-self ≡ *spec.amap swap-env-self-fn*

⟨*ML*⟩

9.5 Parallel composition

We compose a collection of programs (*sequential*, *'s*, *'v*) *spec* in parallel by mapping these into the (*'a agent*, *'s*, *'v*) *spec* lattice, taking the infimum, and mapping back.

definition *toConcurrent-fn* :: *'a* ⇒ *'a* ⇒ *sequential* **where**
toConcurrent-fn = (λ*a a'*. *if a'* = *a then self else env*)

definition *toSequential-fn* :: *'a agent* ⇒ *sequential* **where**
toSequential-fn = *map-agent* ⟨()⟩

lemma *toSequential-fn-alt-def*:
shows *toSequential-fn* = (λ*x*. *case x of proc x* ⇒ *self* | *env* ⇒ *env*)
⟨*proof*⟩

⟨*ML*⟩

abbreviation *toConcurrent* :: *'a* ⇒ (*sequential*, *'s*, *'v*) *spec* ⇒ (*'a agent*, *'s*, *'v*) *spec* **where**
toConcurrent a ≡ *spec.ainvmap* (*toConcurrent-fn* (*proc a*))

abbreviation *toSequential* :: (*'a agent*, *'s*, *'v*) *spec* ⇒ (*sequential*, *'s*, *'v*) *spec* **where**
toSequential ≡ *spec.amap toSequential-fn*

definition *Parallel* :: $'a$ *set* $\Rightarrow$ ($'a \Rightarrow$ (*sequential*, $'s$, *unit*) *spec*) $\Rightarrow$ (*sequential*, $'s$, *unit*) *spec* **where**
Parallel as Ps = *spec.toSequential* (*spec.rel* (*insert env* (*proc* ‘ *as*) × *UNIV*) ⊓ (⨅*a*∈*as*. *spec.toConcurrent a* (*Ps a*)))

definition *parallel* :: (*sequential*, $'s$, *unit*) *spec* $\Rightarrow$ (*sequential*, $'s$, *unit*) *spec* $\Rightarrow$ (*sequential*, $'s$, *unit*) *spec* **where**
parallel P Q = *spec.Parallel UNIV* (λ*a*::*bool*. *if a then P else Q*)

adhoc-overloading
Parallel ⇌ *spec.Parallel*
adhoc-overloading
parallel ⇌ *spec.parallel*

lemma *parallel-alt-def*:
shows *spec.parallel P Q* = *spec.toSequential* (*spec.toConcurrent True P* ⊓ *spec.toConcurrent False Q*)
⟨*proof*⟩

⟨*ML*⟩

lemma *simps*[*simp*]:
shows *toConcurrent-fn* (*proc a*) (*proc a*) = *self*
and *toConcurrent-fn* (*proc a*) *env* = *env*
and *toConcurrent-fn* a' a'' = *self* ⟷ a'' = a'
and *self* = *toConcurrent-fn* a' a'' ⟷ a'' = a'
and *toConcurrent-fn* a' a'' = *env* ⟷ $a'' \neq a'$
and *env* = *toConcurrent-fn* a' a'' ⟷ $a'' \neq a'$
and *toConcurrent-fn* (*proc a*) (*map-agent* ⟨*a*⟩ *x*) = *map-agent* ⟨()⟩ *x*
⟨*proof*⟩

lemma *inj-map-agent*:
assumes *inj-on f* (*insert x* (*set-agent a*))
shows *toConcurrent-fn* (*proc* (*f x*)) (*map-agent f a*) = *toConcurrent-fn* (*proc x*) *a*
⟨*proof*⟩

lemma *inv-into-map-agent*:
fixes *f* :: $'a \Rightarrow 'b$
fixes *a* :: $'b$ *agent*
fixes *x* :: $'a$
assumes *inj-on f as*
assumes *x* ∈ *as*
assumes *a* ∈ *insert env* ((λ*x*. *proc* (*f x*)) ‘ *as*)
shows *toConcurrent-fn* (*proc x*) (*map-agent* (*inv-into as f*) *a*) = *toConcurrent-fn* (*proc* (*f x*)) *a*
⟨*proof*⟩

lemma *vimage-sequential*[*simp*]:
shows *toConcurrent-fn* (*proc a*) −‘ {*self*} = {*proc a*}
and *toConcurrent-fn* (*proc a*) −‘ {*env*} = −{*proc a*}
⟨*proof*⟩

⟨*ML*⟩

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* σ = σ
and *trace.map toSequential-fn* (λ*x*. *x*) (λ*x*. *x*) σ = σ
and (λ*x*. *if x* = *self then self else env*) = *id*

⟨*proof*⟩

lemma *eq-conv*:
  shows *toSequential-fn* x = *env* ⟷ x = *env*
    and *toSequential-fn* x = *self* ⟷ (∃ a. x = *proc* a)
⟨*proof*⟩

lemma *surj*:
  shows *surj toSequential-fn*
⟨*proof*⟩

lemma *image*[*simp*]:
  assumes *as* ≠ {}
  shows *toSequential-fn* ‘ *proc* ‘ *as* = {*self*}
⟨*proof*⟩

lemma *vimage-sequential*[*simp*]:
  shows *toSequential-fn* −‘ {*env*} = {*env*}
    and *toSequential-fn* −‘ {*self*} = *range proc*
⟨*proof*⟩

⟨*ML*⟩

lemma *toSequential-fn-eq-toConcurrent-fn-conv*:
  shows *toSequential-fn* a = *toConcurrent-fn* a' a'' ⟷ (*case* a *of env* ⇒ $a'' \neq a'$ | *proc* - ⇒ $a'' = a'$)
    and *toConcurrent-fn* a' a'' = *toSequential-fn* a ⟷ (*case* a *of env* ⇒ $a'' \neq a'$ | *proc* - ⇒ $a'' = a'$)
⟨*proof*⟩

⟨*ML*⟩

lemma *interference*:
  shows *spec.toSequential* (*spec.rel* ({*env*} × r)) = *spec.rel* ({*env*} × r)
⟨*proof*⟩

lemma *interference-inf-toConcurrent*:
  fixes a :: $'a$
  fixes P :: (*sequential*, $'s$, $'v$) *spec*
  shows *spec.toSequential* (*spec.rel* ({*env*, *proc* a} × *UNIV*) ⊓ *spec.toConcurrent* a P) = P (**is** *?lhs* = *?rhs*)
    and *spec.toSequential* (*spec.toConcurrent* a P ⊓ *spec.rel* ({*env*, *proc* a} × *UNIV*)) = P (**is** *?thesis1*)
⟨*proof*⟩

⟨*ML*⟩

lemma *interference*:
  shows *spec.toConcurrent* a (*spec.rel* ({*env*} × *UNIV*)) = *spec.rel* ((− {*proc* a}) × *UNIV*)
⟨*proof*⟩

⟨*ML*⟩

lemma *Parallel-le*[*spec.idle-le*]:
  assumes $\bigwedge a.\ a \in as \Longrightarrow$ *spec.idle* ≤ *Ps* a
  shows *spec.idle* ≤ *spec.Parallel as Ps*
⟨*proof*⟩

⟨*ML*⟩

lemma *cong*:
  assumes *as* = *as'*

assumes $\bigwedge a.\ a \in as' \Longrightarrow Ps\ a = Ps'\ a$
shows *spec.Parallel as Ps* = *spec.Parallel as' Ps'*
⟨*proof*⟩

lemma *no-agents*:
shows *spec.Parallel* {} *Ps* = *spec.rel* ({*env*} × *UNIV*)
⟨*proof*⟩

lemma *singleton-agents*:
shows *spec.Parallel* {*a*} *Ps* = *Ps a*
⟨*proof*⟩

lemma *bot*:
assumes $Ps\ a = \bot$
assumes $a \in as$
shows *spec.Parallel as Ps* = ⊥
⟨*proof*⟩

lemma *top*:
shows *spec.Parallel as* ⊤ = (*if as* = {} *then spec.rel* ({*env*} × *UNIV*) *else* ⊤)
⟨*proof*⟩

lemma *mono*:
assumes $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \le Ps'\ a$
shows *spec.Parallel as Ps* ≤ *spec.Parallel as Ps'*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes $\bigwedge a.\ a \in as \Longrightarrow$ *st-ord F* (*Ps a*) (*Ps' a*)
shows *st-ord F* (*spec.Parallel as Ps*) (*spec.Parallel as Ps'*)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
fixes *Ps* :: *'a* ⇒ *'b* ⇒ (*sequential*, *'s*, *unit*) *spec*
assumes $\bigwedge a.\ a \in as \Longrightarrow$ *monotone orda* (≤) (*Ps a*)
shows *monotone orda* (≤) (λ*x*::*'b*. *spec.Parallel as* (λ*a*. *Ps a x*))
⟨*proof*⟩

lemma *invmap*: — *af* = *id* in *spec.invmap*
shows *spec.invmap id sf vf* (*spec.Parallel UNIV Ps*) = *spec.Parallel UNIV* (*spec.invmap id sf vf* ∘ *Ps*)
⟨*proof*⟩

lemma *discard-interference*:
assumes $\bigwedge a.\ a \in bs \Longrightarrow$ *Ps a* = *spec.rel* ({*env*} × *UNIV*)
shows *spec.Parallel as Ps* = *spec.Parallel* (*as* − *bs*) *Ps*
⟨*proof*⟩

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*⟩

lemma *rename-UNIV*: — expand the set of agents to *UNIV*
  fixes $f :: 'a \Rightarrow 'b$
  assumes *inj-on f as*
  shows *spec.Parallel as Ps*
    $=$ *spec.Parallel* (*UNIV* :: *'b set*)
      (λb. *if* $b \in f$ ‘ *as then Ps* (*inv-into as f b*) *else spec.rel* ($\{env\} \times UNIV$))
(**is** *?lhs = spec.Parallel - ?f*)
⟨*proof*⟩

lemma *rename*:
  fixes $\pi :: 'a \Rightarrow 'b$
  fixes *Ps* :: $'b \Rightarrow$ (*sequential*, *'s*, *unit*) *spec*
  assumes *bij-betw* π *as bs*
  shows *spec.Parallel as* ($Ps \circ \pi$) $=$ *spec.Parallel bs Ps*
⟨*proof*⟩

lemma *rename-cong*:
  fixes $\pi :: 'a \Rightarrow 'b$
  fixes *Ps* :: $'a \Rightarrow$ (-, -, -) *spec*
  fixes Ps' :: $'b \Rightarrow$ (-, -, -) *spec*
  assumes *bij-betw* π *as bs*
  assumes $\bigwedge a.\ a \in as \Longrightarrow Ps\ a = Ps'\ (\pi\ a)$
  shows *spec.Parallel as Ps* $=$ *spec.Parallel bs* Ps'
⟨*proof*⟩

lemma *inf-pre*:
  assumes $as \neq \{\}$
  shows *spec.Parallel as Ps* $\sqcap$ *spec.pre P* $=$ ($||i{\in}as.$ *Ps i* $\sqcap$ *spec.pre P*) (**is** *?thesis1*)
    and *spec.pre P* $\sqcap$ *spec.Parallel as Ps* $=$ ($||i{\in}as.$ *spec.pre P* $\sqcap$ *Ps i*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *inf-post*:
  assumes $as \neq \{\}$
  shows *spec.Parallel as Ps* $\sqcap$ *spec.post Q* $=$ *spec.Parallel as* (λi. *Ps i* $\sqcap$ *spec.post Q*) (**is** *?thesis1*)
    and *spec.post Q* $\sqcap$ *spec.Parallel as Ps* $=$ *spec.Parallel as* (λi. *spec.post Q* $\sqcap$ *Ps i*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *unwind*:
    — All other processes begin with interference
  assumes *b*: $\bigwedge b.\ b \in as - \{a\} \Longrightarrow$ *spec.rel* ($\{env\} \times UNIV$) $\gg=$ (λ-::*unit*. *Ps b*) $\leq$ *Ps b*
  assumes *a*: $f \gg= g \leq Ps\ a$ — The selected process starts with *f*
  assumes $a \in as$
  shows $f \gg=$ (λv. *spec.Parallel as* ($Ps(a := g\ v)$)) $\leq$ *spec.Parallel as Ps*
⟨*proof*⟩

lemma *inf-rel*:
  fixes *as* :: *'a set*
  fixes *r* :: *'s rel*
  shows *spec.rel* ($\{env\} \times UNIV \cup \{self\} \times r$) $\sqcap$ *spec.Parallel as Ps*
    $=$ *spec.Parallel as* (λa. *spec.rel* ($\{env\} \times UNIV \cup \{self\} \times r$) $\sqcap$ *Ps a*) (**is** *?lhs = ?rhs*)
    and *spec.Parallel as Ps* $\sqcap$ *spec.rel* ($\{env\} \times UNIV \cup \{self\} \times r$)
    $=$ *spec.Parallel as* (λa. *Ps a* $\sqcap$ *spec.rel* ($\{env\} \times UNIV \cup \{self\} \times r$)) (**is** *?thesis1*)
⟨*proof*⟩

lemma *flatten*:
  fixes *as* :: *'a set*
  fixes *a* :: *'a*
  fixes *bs* :: *'b set*

fixes *Ps* :: $'a \Rightarrow$ (*sequential*, $'s$, *unit*) *spec*
fixes *Ps'* :: $'b \Rightarrow$ (*sequential*, $'s$, *unit*) *spec*
assumes *Ps a* = *spec.Parallel bs Ps'*
assumes $a \in as$
shows *spec.Parallel as Ps* = *spec.Parallel* ((*as* − {*a*}) <+> *bs*) (*case-sum Ps Ps'*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *Parallel-some-agents*:
assumes $\bigwedge a.\ a \in bs \Longrightarrow Ps\ a = spec.term.none\ (Ps'\ a)$
assumes $as \cap bs \neq \{\}$
shows *spec.Parallel as Ps* = *spec.term.none* (‖*a*∈*as*. *if a* ∈ *as* ∩ *bs then Ps' a else Ps a*)
⟨*proof*⟩

lemma *Parallel-not-empty*:
assumes $as \neq \{\}$
shows *spec.term.none* (*Parallel as Ps*) = *Parallel as* (*spec.term.none* ∘ *Ps*)
⟨*proof*⟩

lemma *parallel*:
shows *spec.term.none* (*P* ∥ *Q*) = *spec.term.none P* ∥ *spec.term.none Q*
⟨*proof*⟩

lemma
shows *parallelL*: *spec.term.none P* ∥ *Q* = *spec.term.none* (*P* ∥ *Q*)
and *parallelR*: *P* ∥ *spec.term.none Q* = *spec.term.none* (*P* ∥ *Q*)
⟨*proof*⟩

⟨*ML*⟩

lemma *Parallel*:
shows *spec.term.all* (*spec.Parallel as Ps*) = *spec.Parallel as* (*spec.term.all* ∘ *Ps*)
⟨*proof*⟩

⟨*ML*⟩

lemma *parallel-le*:
assumes *spec.idle* ≤ *P*
assumes *spec.idle* ≤ *Q*
shows *spec.idle* ≤ *P* ∥ *Q*
⟨*proof*⟩

⟨*ML*⟩

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*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
shows *botL*: *spec.parallel* ⊥ *P* = ⊥
and *botR*: *spec.parallel P* ⊥ = ⊥
⟨*proof*⟩

lemma *commute*:

shows *spec.parallel* $P\ Q$ = *spec.parallel* $Q\ P$
⟨*proof*⟩

lemma *mono*:
assumes $P \leq P'$
assumes $Q \leq Q'$
shows *spec.parallel* $P\ Q \leq$ *spec.parallel* $P'\ Q'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* $F\ P\ P'$
assumes *st-ord* $F\ Q\ Q'$
shows *st-ord* F (*spec.parallel* $P\ Q$) (*spec.parallel* $P'\ Q'$)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* $(\leq)$ F
assumes *monotone orda* $(\leq)$ G
shows *monotone orda* $(\leq)$ $(\lambda f.$ *spec.parallel* $(F\ f)\ (G\ f))$
⟨*proof*⟩

lemma *Sup*:
fixes *Ps* :: (*sequential*, *'s*, *unit*) *spec set*
shows *SupL*: $\bigsqcup Ps \parallel Q = (\bigsqcup P{\in}Ps.\ P \parallel Q)$
and *SupR*: $Q \parallel \bigsqcup Ps = (\bigsqcup P{\in}Ps.\ Q \parallel P)$
⟨*proof*⟩

lemma *sup*:
fixes *P* :: (*sequential*, *'s*, *unit*) *spec*
shows *supL*: $(P \sqcup Q) \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel (Q \sqcup R) = (P \parallel Q) \sqcup (P \parallel R)$
⟨*proof*⟩

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* :: (*sequential*, *'s*, *unit*) *spec* ⇒ (*sequential*, *'s*, *unit*) *spec* ⇒ (*sequential*, *'s*, *unit*) *spec* **where**
res $S\ i = \bigsqcup\{P.\ P \parallel i \leq S\}$

interpretation *res*: *galois.complete-lattice-class* $\lambda S.$ *spec.parallel* $S\ i$ $\lambda S.$ *spec.parallel.res* $S\ i$ **for** i — Hayes (2016, Law 23 (rely refinement))
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* $(\leq)$ P
assumes *mcont luba orda Sup* $(\leq)$ Q
shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.$ *spec.parallel* $(P\ x)\ (Q\ x))$
⟨*proof*⟩

lemma *inf-rel*:
shows *spec.rel* $(\{env\} \times UNIV \cup \{self\} \times r) \sqcap (P \parallel Q)$
$= ($*spec.rel* $(\{env\} \times UNIV \cup \{self\} \times r) \sqcap P) \parallel ($*spec.rel* $(\{env\} \times UNIV \cup \{self\} \times r) \sqcap Q)$

and $(P \parallel Q) \sqcap spec.rel\ (\{env\} \times UNIV \cup \{self\} \times r)$
$= (spec.rel\ (\{env\} \times UNIV \cup \{self\} \times r) \sqcap P) \parallel (spec.rel\ (\{env\} \times UNIV \cup \{self\} \times r) \sqcap Q)$
⟨*proof*⟩

lemma *assoc*:
shows $spec.parallel\ P\ (spec.parallel\ Q\ R) = spec.parallel\ (spec.parallel\ P\ Q)\ R$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *bind-botR*:
shows $spec.parallel\ (P \gg\!= \bot)\ Q\ = spec.parallel\ P\ Q \gg\!= \bot$
and $spec.parallel\ P\ (Q \gg\!= \bot)\ = spec.parallel\ P\ Q \gg\!= \bot$
⟨*proof*⟩

lemma *interference*:
shows *interferenceL*: $spec.rel\ (\{env\} \times UNIV) \parallel c = c$
and *interferenceR*: $c \parallel spec.rel\ (\{env\} \times UNIV) = c$
⟨*proof*⟩

lemma *unwindL*:
assumes $spec.rel\ (\{env\} \times UNIV) \gg\!= (\lambda\text{-}::unit.\ Q) \leq Q$ — All other processes begin with interference
assumes $f \gg\!= g \leq P$ — The selected process starts with action f
shows $f \gg\!= (\lambda v.\ g\ v \parallel Q) \leq P \parallel Q$
⟨*proof*⟩

lemma *unwindR*:
assumes $spec.rel\ (\{env\} \times UNIV) \gg\!= (\lambda\text{-}::unit.\ P) \leq P$ — All other processes begin with interference
assumes $f \gg\!= g \leq Q$ — The selected process starts with action f
shows $f \gg\!= (\lambda v.\ P \parallel g\ v) \leq P \parallel Q$
⟨*proof*⟩

⟨*ML*⟩

lemma *toConcurrent-gen*:
fixes $P :: (sequential, 's, 'v)\ spec$
fixes $a :: 'a$
assumes P: $P \in spec.interference.closed\ (\{env\} \times r)$
shows $spec.toConcurrent\ a\ P \in spec.interference.closed\ ((-\{proc\ a\}) \times r)$
⟨*proof*⟩

lemma *toConcurrent*:
fixes $P :: (sequential, 's, 'v)\ spec$
fixes $a :: 'a$
assumes P: $P \in spec.interference.closed\ (\{env\} \times r)$
shows $spec.toConcurrent\ a\ P \in spec.interference.closed\ (\{env\} \times r)$
⟨*proof*⟩

lemma *toSequential*:
fixes $P :: ('a\ agent, 's, 'v)\ spec$
assumes $P \in spec.interference.closed\ (\{env\} \times r)$
shows $spec.toSequential\ P \in spec.interference.closed\ (\{env\} \times r)$
⟨*proof*⟩

lemma *Parallel*:
assumes $\bigwedge a.\ Ps\ a \in spec.interference.closed\ (\{env\} \times UNIV)$
shows $spec.Parallel\ as\ Ps \in spec.interference.closed\ (\{env\} \times UNIV)$
⟨*proof*⟩

lemma *parallel*:

assumes $P \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
assumes $Q \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
shows $P \parallel Q \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
$\langle proof \rangle$

$\langle ML \rangle$

9.6 Specification Inhabitation

Given that $\bot$ 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* $\Rightarrow$ $'s \Rightarrow ('a \times 's)$ *list* $\Rightarrow$ $('a, 's, 'w)$ *spec* $\Rightarrow$ *bool* (‹-/ −-, -→/ -› [*50*, *0*, *0*, *50*] *50*) **where**
S −s, xs→ T $\longleftrightarrow$ ⟪s, xs, *Some* ()⟫ $\gg$ $T \le S$

$\langle ML \rangle$

lemma *incomplete*:
assumes S −s, xs→ S'
shows ⟪s, xs, *None*⟫ $\le S$
$\langle proof \rangle$

lemma *complete*:
assumes S −s, xs→ *spec.return* v
shows ⟪s, xs, *Some* v⟫ $\le S$
$\langle proof \rangle$

lemmas I = *inhabits.complete inhabits.incomplete*

lemma *mono*:
assumes $S \le S'$
assumes $T' \le T$
assumes S −s, xs→ T
shows S' −s, xs→ T'
$\langle proof \rangle$

lemma *strengthen*[*strg*]:
assumes *st-ord* F S S'
assumes *st-ord* $(\neg F)$ T T'
shows *st* F $(\longrightarrow)$ (S −s, xs→ T) (S' −s, xs→ T')
$\langle proof \rangle$

lemma *pre*:
  assumes $S\ -s,\ xs'\!\rightarrow\ T$
  assumes $T' \leq T$
  assumes $xs = xs'$
  shows $S\ -s,\ xs\!\rightarrow\ T'$
⟨*proof*⟩

lemma *tau*:
  assumes $spec.idle \leq S$
  shows $S\ -s,\ []\!\rightarrow\ S$
⟨*proof*⟩

lemma *trans*:
  assumes $R\ -s,\ xs\!\rightarrow\ S$
  assumes $S\ -trace.final'\ s\ xs,\ ys\!\rightarrow\ T$
  shows $R\ -s,\ xs\ @\ ys\!\rightarrow\ T$
⟨*proof*⟩

lemma *Sup*:
  assumes $P\ -s,\ xs\!\rightarrow\ P'$
  assumes $P \in X$
  shows $\bigsqcup X\ -s,\ xs\!\rightarrow\ P'$
⟨*proof*⟩

lemma *supL*:
  assumes $P\ -s,\ xs\!\rightarrow\ P'$
  shows $P \sqcup Q\ -s,\ xs\!\rightarrow\ P'$
⟨*proof*⟩

lemma *supR*:
  assumes $Q\ -s,\ xs\!\rightarrow\ Q'$
  shows $P \sqcup Q\ -s,\ xs\!\rightarrow\ Q'$
⟨*proof*⟩

lemma *inf*:
  assumes $P\ -s,\ xs\!\rightarrow\ P'$
  assumes $Q\ -s,\ xs\!\rightarrow\ Q'$
  shows $P \sqcap Q\ -s,\ xs\!\rightarrow\ P' \sqcap Q'$
⟨*proof*⟩

lemma *infL*:
  assumes $P\ -s,\ xs\!\rightarrow\ R$
  assumes $Q\ -s,\ xs\!\rightarrow\ R$
  shows $P \sqcap Q\ -s,\ xs\!\rightarrow\ R$
⟨*proof*⟩

⟨*ML*⟩

lemma *bind*:
  assumes $f\ -s,\ xs\!\rightarrow\ f'$
  shows $f \ggg g\ -s,\ xs\!\rightarrow\ f' \ggg g$
⟨*proof*⟩

lemmas $bind' = inhabits.trans[OF\ inhabits.spec.bind]$

lemma *parallelL*:
  assumes $P\ -s,\ xs\!\rightarrow\ P'$
  assumes $spec.rel\ (\{env\} \times UNIV) \ggg (\lambda\text{-}::unit.\ Q) \leq Q$

shows $P \parallel Q\ -s,\ xs\rightarrow\ P' \parallel Q$
⟨*proof*⟩

lemma *parallelR*:
assumes $Q\ -s,\ xs\rightarrow\ Q'$
assumes $spec.rel\ (\{env\} \times UNIV) \gg= (\lambda\text{-}::unit.\ P) \leq P$
shows $P \parallel Q\ -s,\ xs\rightarrow\ P \parallel Q'$
⟨*proof*⟩

lemmas $parallelL' = inhabits.trans[OF\ inhabits.spec.parallelL]$
lemmas $parallelR' = inhabits.trans[OF\ inhabits.spec.parallelR]$

⟨*ML*⟩

lemma *step*:
assumes $(v,\ a,\ s,\ s') \in F$
shows $spec.action\ F\ -s,\ [(a,\ s')]\rightarrow\ spec.return\ v$
⟨*proof*⟩

lemma *stutter*:
assumes $(v,\ a,\ s,\ s) \in F$
shows $spec.action\ F\ -s,\ []\rightarrow\ spec.return\ v$
⟨*proof*⟩

⟨*ML*⟩

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 $spec.map\ af\ sf\ vf\ P\ -sf\ s,\ map\ (map\text{-}prod\ af\ sf)\ xs\rightarrow\ spec.return\ (vf\ v)$
⟨*proof*⟩

lemma *invmap*:
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P\ -sf\ s,\ map\ (map\text{-}prod\ af\ sf)\ xs\ \rightarrow\ P'$
shows $spec.invmap\ af\ sf\ vf\ P\ -s,\ xs\rightarrow\ spec.invmap\ af\ sf\ vf\ P'$
⟨*proof*⟩

⟨*ML*⟩

lemma *step*:
assumes $P\ -s,\ xs\rightarrow\ P'$
shows $spec.term.none\ P\ -s,\ xs\rightarrow\ spec.term.none\ P'$
⟨*proof*⟩

⟨*ML*⟩

lemma *step*:
assumes $P\ -s,\ xs\rightarrow\ P'$
shows $spec.term.all\ P\ -s,\ xs\rightarrow\ spec.term.all\ P'$
⟨*proof*⟩

lemma *term*:
assumes $spec.idle \leq P$

shows *spec.term.all* P $-s$, $[]\rightarrow$ *spec.return* v
⟨*proof*⟩

⟨*ML*⟩

lemma *step*:
assumes P $-s$, $xs\rightarrow$ P'
shows *spec.kleene.star* P $-s$, $xs\rightarrow$ $P' \gg$ *spec.kleene.star* P
⟨*proof*⟩

lemma *term*:
shows *spec.kleene.star* P $-s$, $[]\rightarrow$ *spec.return* ()
⟨*proof*⟩

⟨*ML*⟩

lemma *rel*:
assumes *trace.steps'* s $xs \subseteq r$
shows *spec.rel* r $-s$, $xs\rightarrow$ *spec.rel* r
⟨*proof*⟩

lemma *rel-term*:
assumes *trace.steps'* s $xs \subseteq r$
shows *spec.rel* r $-s$, $xs\rightarrow$ *spec.return* v
⟨*proof*⟩

lemma *step*:
assumes $(a, s, s') \in r$
shows *spec.rel* r $-s$, $[(a, s')]\rightarrow$ *spec.rel* r
⟨*proof*⟩

lemma *term*:
shows *spec.rel* r $-s$, $[]\rightarrow$ *spec.return* v
⟨*proof*⟩

⟨*ML*⟩

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* ⇒ *'a set* ⇒ *'a set* **where** — Abadi and Merz (1995, §2)
  next-imp P Q = {σ. ∀σ'≤σ. (∀σ''<σ'. σ'' ∈ P) ⟶ σ' ∈ Q}

⟨*ML*⟩

lemma *downwards-closed*:
  assumes *P* ∈ *downwards.closed*
  shows *next-imp P Q* ∈ *downwards.closed*
⟨*proof*⟩

lemma *mono*:
  assumes *x'* ≤ *x*
  assumes *y* ≤ *y'*
  shows *next-imp x y* ≤ *next-imp x' y'*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
  assumes *st-ord* (¬ *F*) *X X'*
  assumes *st-ord F Y Y'*
  shows *st-ord F* (*next-imp X Y*) (*next-imp X' Y'*)
⟨*proof*⟩

lemma *minimal*:
  assumes *trace.T s xs v* ∈ *next-imp P Q*
  shows *trace.T s* [] *None* ∈ *Q*
⟨*proof*⟩

lemma *alt-def*: — This definition coincides with Cau and Collette (1996), Abadi and Lamport (1995, §3.5.3)
  assumes *P* ∈ *downwards.closed*
  shows *next-imp P Q*
    = {σ. *trace.T* (*trace.init* σ) [] *None* ∈ *Q*
      ∧ (∀*i*. *trace.take i* σ ∈ *P* ⟶ *trace.take* (*Suc i*) σ ∈ *Q*)} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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* ∈ *downwards.closed*
  assumes *wf*: *wfP* ((<) :: *'a relp*)
  shows *next-imp P Q* ⊆ *downwards.imp* (*downwards.imp Q P*) *Q* (**is** *?lhs* ⊆ *?rhs*)
⟨*proof*⟩

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 set*
  assumes *P* ∈ *downwards.closed*
  assumes *Q* ∈ *downwards.closed*
  assumes *wf*: *wfP* ((<) :: *'a relp*)
  shows *next-imp P Q* = *downwards.imp* (*downwards.imp Q P*) *Q* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Abadi-Lamport-Lemma6*: — Abadi and Lamport (1995, Lemma 6) (no proof given there)
  fixes *P* :: *'a::order set*

assumes $P \in$ *downwards.closed*
assumes *wf*: *wfP* $((<) :: 'a\ relp)$
shows *next-imp* $P\ Q =$ *downwards.imp* (*downwards.imp* $Q\ P$) Q (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemmas *downwards-imp* = *next-imp.Abadi-Lamport-Lemma6*[*OF* - *trace.wfP-less*]

lemma *boolean-implication-le*:
assumes $P \in$ *downwards.closed*
shows *next-imp* $P\ Q \subseteq P \longrightarrow_B Q$
⟨*proof*⟩

⟨*ML*⟩

lift-definition *next-imp* :: $('a, 's, 'v)$ *spec* $\Rightarrow$ $('a, 's, 'v)$ *spec* $\Rightarrow$ $('a, 's, 'v)$ *spec* (**infixr** ‹$\longrightarrow_+$› *61*) **is**
Next-Imp.next-imp
⟨*proof*⟩

⟨*ML*⟩

lemma *heyting*: — fundamental
shows $P \longrightarrow_+ Q = (Q \longrightarrow_H P) \longrightarrow_H Q$
⟨*proof*⟩

⟨*ML*⟩

lemma *next-imp-le-conv*:
fixes P :: $('a, 's, 'v)$ *spec*
shows $⦃\sigma⦄ \le P \longrightarrow_+ Q \longleftrightarrow (\forall \sigma' \le \sigma.\ (\forall \sigma'' < \sigma'.\ ⦃\sigma''⦄ \le P) \longrightarrow ⦃\sigma'⦄ \le Q)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *mono*:
assumes $x' \le x$
assumes $y \le y'$
shows $x \longrightarrow_+ y \le x' \longrightarrow_+ y'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F)\ X\ X'$
assumes *st-ord* $F\ Y\ Y'$
shows *st-ord* $F\ (X \longrightarrow_+ Y)\ (X' \longrightarrow_+ Y')$
⟨*proof*⟩

lemma *idempotentR*:
shows $P \longrightarrow_+ (P \longrightarrow_+ Q) = P \longrightarrow_+ Q$
⟨*proof*⟩

lemma *contains*:
assumes $X \le Q$
shows $X \le P \longrightarrow_+ Q$
⟨*proof*⟩

interpretation *closure*: *closure-complete-lattice-class* $(\longrightarrow_+)$ P **for** P
⟨*proof*⟩

lemma *InfR*:

shows $x \longrightarrow_+ \bigsqcap X = \bigsqcap ((\longrightarrow_+)\ x\ `\ X)$
⟨*proof*⟩

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*⟩

lemma *cont*:
shows *cont Sup* (≤) *Sup* (≤) ((⟶₊) *P*)
⟨*proof*⟩

lemma *mcont*:
shows *mcont Sup* (≤) *Sup* (≤) ((⟶₊) *P*)
⟨*proof*⟩

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* ≤ *P*
shows $\bot \longrightarrow_+ P = \top$
⟨*proof*⟩

lemma *topL*[*simp*]:
shows $\top \longrightarrow_+ P = P$
⟨*proof*⟩

lemmas *topR*[*simp*] = *spec.next-imp.closure.cl-top*

lemma *refl*:
shows $P \longrightarrow_+ P \leq P$
⟨*proof*⟩

lemma *heyting-le*:
shows $P \longrightarrow_+ Q \leq P \longrightarrow_H Q$
⟨*proof*⟩

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*⟩

lemma *detachment*:
shows $x \sqcap (x \longrightarrow_+ y) \leq y$
and $(x \longrightarrow_+ y) \sqcap x \leq y$
⟨*proof*⟩

lemma *infR*:
shows $P \longrightarrow_+ Q \sqcap R = (P \longrightarrow_+ Q) \sqcap (P \longrightarrow_+ R)$
⟨*proof*⟩

lemma *supL-le*:
shows $x \sqcup y \longrightarrow_+ z \leq (x \longrightarrow_+ z) \sqcup (y \longrightarrow_+ z)$
⟨*proof*⟩

lemma *heytingL*:
  shows $(P \longrightarrow_H Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
⟨*proof*⟩

lemma *heytingR*:
  shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_H R) \leq P \longrightarrow_+ R$
⟨*proof*⟩

lemma *heytingL-distrib*:
  shows $P \longrightarrow_H (Q \longrightarrow_+ R) = (P \sqcap Q) \longrightarrow_+ (P \longrightarrow_H R)$
⟨*proof*⟩

lemma *trans*:
  shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
⟨*proof*⟩

lemma *rev-trans*:
  shows $(Q \longrightarrow_+ R) \sqcap (P \longrightarrow_+ Q) \leq P \longrightarrow_+ R$
⟨*proof*⟩

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*⟩

lemma *invmap*:
  shows *spec.invmap af sf vf* $(P \longrightarrow_+ Q) =$ *spec.invmap af sf vf* $P \longrightarrow_+$ *spec.invmap af sf vf* Q
⟨*proof*⟩

lemma *Abadi-Lamport-Lemma7*:
  assumes $Q \sqcap R \leq P$
  shows $P \longrightarrow_+ Q \leq R \longrightarrow_+ Q$
⟨*proof*⟩

⟨*ML*⟩

lemma *next-imp*:
  shows *spec.term.none* $(P \longrightarrow_+ Q) \leq$ *spec.term.all* $P \longrightarrow_+$ *spec.term.none* Q
⟨*proof*⟩

⟨*ML*⟩

lemma *next-imp*:
  shows *spec.term.all* $(P \longrightarrow_+ Q) =$ *spec.term.all* $P \longrightarrow_+$ *spec.term.all* Q
⟨*proof*⟩

⟨*ML*⟩

lemma *next-imp*:
  assumes $Q \in$ *spec.term.closed* -
  shows $P \longrightarrow_+ Q \in$ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

lemma *next-imp-eq-heyting*:
  assumes *spec.idle* $\leq R$

shows $Q \sqcap spec.pre\ P \longrightarrow_{+} R = spec.pre\ P \longrightarrow_{H} (Q \longrightarrow_{+} R)$ (**is** *?lhs = ?rhs*)
and $spec.pre\ P \sqcap Q \longrightarrow_{+} R = spec.pre\ P \longrightarrow_{H} (Q \longrightarrow_{+} R)$ (**is** *?thesis1*)
⟨*proof*⟩

⟨*ML*⟩

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).

⟨*ML*⟩

lemma *Abadi-Lamport-Lemma5*:
shows $(\bigsqcap i{\in}I.\ P\ i \longrightarrow_{+} Q\ i) \le (\bigsqcap i{\in}I.\ P\ i) \longrightarrow_{+} (\bigsqcap i{\in}I.\ Q\ i)$
⟨*proof*⟩

lemma *Abadi-Merz-Prop2-1*:
shows $(P \longrightarrow_{+} Q) \sqcap (P \longrightarrow_{+} (Q \longrightarrow_{H} R)) \le P \longrightarrow_{+} R$
⟨*proof*⟩

lemma *Abadi-Merz-Theorem3-5*:
shows $P \longrightarrow_{H} (Q \longrightarrow_{H} R) \le (R \longrightarrow_{+} Q) \longrightarrow_{H} (P \longrightarrow_{+} Q)$
⟨*proof*⟩

theorem *Abadi-Merz-Theorem4*:
shows $(A \sqcap (\bigsqcap i{\in}I.\ Cs\ i) \longrightarrow_{H} (\bigsqcap i{\in}I.\ As\ i))$
$\sqcap (A \longrightarrow_{+} ((\bigsqcap i{\in}I.\ Cs\ i) \longrightarrow_{H} C))$
$\sqcap (\bigsqcap i{\in}I.\ As\ i \longrightarrow_{+} Cs\ i)$
$\le A \longrightarrow_{+} C$ (**is** *?lhs ≤ ?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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 $(\lambda x\ y.\ (x, y) \in r)\ (\le)\ P$

⟨*ML*⟩

named-theorems *intro stability intro rules*

lemma *singleton-conv*:
shows $stable\ \{(s, s')\}\ P \longleftrightarrow (P\ s \longrightarrow P\ s')$
⟨*proof*⟩

lemma *closed*:
shows $stable\ r\ P \longleftrightarrow r\ ``\ Collect\ P \subseteq Collect\ P$
⟨*proof*⟩

lemma *rtrancl-conv*:
  shows *stable* (r^*) = *stable r*
⟨*proof*⟩

lemma *reflcl-conv*:
  shows *stable* $(r^=)$ = *stable r*
⟨*proof*⟩

lemma *empty*[*stable.intro*]:
  shows *stable* {} *P*
⟨*proof*⟩

lemma [*stable.intro*]:
  shows *Id*: *stable Id P*
    and *Id-fst*: $\bigwedge P$. *stable* (*Id* $\times_R$ *A*) (λs. *P* (*fst s*))
    and *Id-fst-fst-snd*: $\bigwedge P$. *stable* (*Id* $\times_R$ *Id* $\times_R$ *A*) (λs. *P* (*fst s*) (*fst* (*snd s*)))
⟨*proof*⟩

lemma *UNIV*:
  shows *stable UNIV P* ⟷ (∃ *c*. *P* = ⟨*c*⟩)
⟨*proof*⟩

lemma *antimono-rel*:
  shows *antimono* (λr. *stable r P*)
⟨*proof*⟩

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* ∩ *s*) *P*
    and *infI2*: *stable* (*s* ∩ *r*) *P*
⟨*proof*⟩

lemma *UNION-conv*:
  shows *stable* ($\bigcup x{\in}X$. *r x*) *P* ⟷ (∀ *x*∈*X*. *stable* (*r x*) *P*)
⟨*proof*⟩

lemmas *UNIONI*[*stable.intro*] = *iffD2*[*OF stable.UNION-conv*, *rule-format*]

lemma *Union-conv*:
  shows *stable* ($\bigcup X$) *P* ⟷ (∀ *x*∈*X*. *stable x P*)
⟨*proof*⟩

lemma *union-conv*:
  shows *stable* (*r* ∪ *s*) *P* ⟷ *stable r P* ∧ *stable s P*
⟨*proof*⟩

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* ⟨*c*⟩
    and *stable r* ⊥
    and *stable r* ⊤
⟨*proof*⟩

lemma *conjI*[*stable.intro*]:
  assumes *stable r P*
  assumes *stable r Q*
  shows *stable r* $(P \wedge Q)$
⟨*proof*⟩

lemma *disjI*[*stable.intro*]:
  assumes *stable r P*
  assumes *stable r Q*
  shows *stable r* $(P \vee Q)$
⟨*proof*⟩

lemma *implies-constI*[*stable.intro*]:
  assumes $P \Longrightarrow$ *stable r Q*
  shows *stable r* $(\lambda s.\ P \longrightarrow Q\ s)$
⟨*proof*⟩

lemma *allI*[*stable.intro*]:
  assumes $\bigwedge x.$ *stable r* $(P\ x)$
  shows *stable r* $(\forall x.\ P\ x)$
⟨*proof*⟩

lemma *ballI*[*stable.intro*]:
  assumes $\bigwedge x.\ x \in X \Longrightarrow$ *stable r* $(P\ x)$
  shows *stable r* $(\lambda s.\ \forall x {\in} X.\ P\ x\ s)$
⟨*proof*⟩

lemma *stable-relprod-fstI*[*stable.intro*]:
  assumes *stable r P*
  shows *stable* $(r \times_R s)$ $(\lambda s.\ P\ (\mathit{fst}\ s))$
⟨*proof*⟩

lemma *stable-relprod-sndI*[*stable.intro*]:
  assumes *stable s P*
  shows *stable* $(r \times_R s)$ $(\lambda s.\ P\ (\mathit{snd}\ s))$
⟨*proof*⟩

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 *stable* $(\mathit{Id} \times_R \mathit{UNIV})$ P
⟨*proof*⟩

lemma *Id-on-proj*:
  assumes $\bigwedge v.$ *stable* Id_f $(\lambda s.\ P\ v\ s)$
  shows *stable* Id_f $(\lambda s.\ P\ (f\ s)\ s)$
⟨*proof*⟩

lemma *if-const-conv*:
  shows *stable r* (*if c then P else Q*) $\longleftrightarrow$ *stable r* $(\lambda s.\ c \longrightarrow P\ s) \wedge$ *stable r* $(\lambda s.\ \neg c \longrightarrow Q\ s)$
⟨*proof*⟩

lemma *ifI*[*stable.intro*]:
  assumes *stable r* $(\lambda s.\ c\ s \longrightarrow P\ s)$
  assumes *stable r* $(\lambda s.\ \neg c\ s \longrightarrow Q\ s)$
  shows *stable r* (λs. *if c s then P s else Q s*)
⟨*proof*⟩

lemma *ifI2*[*stable.intro*]:

assumes *stable r* ($\lambda s.$ *c s* $\longrightarrow$ *P s s*)
assumes *stable r* ($\lambda s.$ $\neg$*c s* $\longrightarrow$ *Q s s*)
shows *stable r* ($\lambda s.$ (*if c s then P s else Q s*) *s*)
⟨*proof*⟩

lemma *case-optionI*[*stable.intro*]:
assumes *stable r* ($\lambda s.$ *opt s* = *None* $\longrightarrow$ *none s*)
assumes $\bigwedge v.$ *stable r* ($\lambda s.$ *opt s* = *Some v* $\longrightarrow$ *some v s*)
shows *stable r* ($\lambda s.$ *case opt s of None* $\Rightarrow$ *none s* | *Some v* $\Rightarrow$ *some v s*)
⟨*proof*⟩

lemma *case-optionI2*[*stable.intro*]:
assumes *opt* = *None* $\Longrightarrow$ *stable r none*
assumes $\bigwedge v.$ *opt* = *Some v* $\Longrightarrow$ *stable r* (*some v*)
shows *stable r* (*case opt of None* $\Rightarrow$ *none* | *Some v* $\Rightarrow$ *some v*)
⟨*proof*⟩

In practice the following rules are often too weak

lemma *impliesI*:
assumes *stable r* ($\neg P$)
assumes *stable r Q*
shows *stable r* ($P \longrightarrow Q$)
⟨*proof*⟩

lemma *exI*:
assumes $\bigwedge x.$ *stable r* (*P x*)
shows *stable r* ($\exists x.$ *P x*)
⟨*proof*⟩

lemma *bexI*:
assumes $\bigwedge x.$ $x \in X \Longrightarrow$ *stable r* (*P x*)
shows *stable r* ($\lambda s.$ $\exists x \in X.$ *P x s*)
⟨*proof*⟩

⟨*ML*⟩

12 Refinement

We develop a refinement story for the (*'a, 's, '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* :: *'s pred* $\Rightarrow$ (*'a, 's, 'v*) *spec* $\Rightarrow$ (*'a, 's, 'v*) *spec* $\Rightarrow$ (*'v* $\Rightarrow$ *'s pred*) $\Rightarrow$ (*'a, 's, 'v*) *spec* (‹⦃-⦄, -
⊩ -, ⦃-⦄› [*0,0,0,0*] *100*) **where**
⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄ = *spec.pre P* $\sqcap$ *A* $\longrightarrow_+$ *G* $\sqcap$ *spec.post Q*

An intuitive gloss on the proposition $c \leq$ ⦃*P*⦄, *A* ⊩ *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 $idle \le Q$, in practice $idle \le G$
- see §13.4.1 for some introduction rules

⟨*ML*⟩

lemma *E*:
  assumes $c \le \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
  obtains $c \le spec.pre\ P \sqcap A \longrightarrow_+ G$
    and $c \le spec.pre\ P \sqcap A \longrightarrow_+ spec.post\ Q$
⟨*proof*⟩

lemma *pre-post-cong*:
  assumes $P = P'$
  assumes $Q = Q'$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|P'|\!\}, A \Vdash G, \{\!|Q'|\!\}$
⟨*proof*⟩

lemma *top*:
  shows $\{\!|P|\!\}, A \Vdash \top, \{\!|\top|\!\} = \top$
   and $\{\!|P|\!\}, A \Vdash \top, \{\!|\langle\top\rangle|\!\} = \top$
   and $\{\!|P|\!\}, A \Vdash \top, \{\!|\lambda\text{- -}.\ True|\!\} = \top$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* $(\le)$ G
  shows *mcont luba orda Sup* $(\le)$ $(\lambda x.\ \{\!|P|\!\}, A \Vdash G\ x, \{\!|Q|\!\})$
⟨*proof*⟩

lemma *mono*:
  assumes $P' \le P$
  assumes $A' \le A$
  assumes $G \le G'$
  assumes $Q \le Q'$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \le \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$
⟨*proof*⟩

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* $F\ (\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\})\ (\{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\})$
⟨*proof*⟩

lemma *mono-stronger*:
  assumes $P' \le P$
  assumes $spec.pre\ P' \sqcap A' \le A$
  assumes $spec.pre\ P' \sqcap G \le A' \longrightarrow_+ G'$
  assumes $Q \le Q'$
  assumes $spec.idle \le G'$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \le \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$
⟨*proof*⟩

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 $c \le \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
  assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
  assumes $A' \le A$
  assumes $G \le G'$
  assumes $\bigwedge s\ v.\ Q\ s\ v \Longrightarrow Q'\ s\ v$
  shows $c \le \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$
⟨*proof*⟩

lemmas *pre-pre-post* = *refinement.pre*[*OF - - order.refl order.refl*, *of c*] **for** *c*

lemma *pre-imp*:
  assumes $\bigwedge s.\ P\ s \Longrightarrow P'\ s$
  assumes $c \le \{\!|P'|\!\}, A \Vdash G, \{\!|Q|\!\}$
  shows $c \le \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemmas *pre-pre* = *refinement.pre-imp*[*rotated*]

lemma *post-imp*:
  assumes $\bigwedge v\ s.\ Q\ v\ s \Longrightarrow R\ v\ s$
  assumes $c \le \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
  shows $c \le \{\!|P|\!\}, A \Vdash G, \{\!|R|\!\}$
⟨*proof*⟩

lemmas *pre-post* = *refinement.post-imp*[*rotated*]
lemmas *strengthen-post* = *refinement.pre-post*

lemma *pre-inf-assume*:
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, A \sqcap \mathit{spec.pre}\ P \Vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *pre-assume-absorb*:
  assumes $A \le \mathit{spec.pre}\ P$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|\top|\!\}, A \Vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemmas *sup* = *sup-least*[**where** $x=\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$] **for** *A G P Q*

lemma
  shows *supRL*: $c \le \{\!|P|\!\}, A \Vdash G_1, \{\!|Q|\!\} \Longrightarrow c \le \{\!|P|\!\}, A \Vdash G_1 \sqcup G_2, \{\!|Q|\!\}$
    and *supRR*: $c \le \{\!|P|\!\}, A \Vdash G_2, \{\!|Q|\!\} \Longrightarrow c \le \{\!|P|\!\}, A \Vdash G_1 \sqcup G_2, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *infR-conv*:
  shows $\{\!|P|\!\}, A \Vdash G_1 \sqcap G_2, \{\!|Q_1 \sqcap Q_2|\!\} = \{\!|P|\!\}, A \Vdash G_1, \{\!|Q_1|\!\} \sqcap \{\!|P|\!\}, A \Vdash G_2, \{\!|Q_2|\!\}$
⟨*proof*⟩

lemma *inf-le*:
  shows $X \sqcap \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \le \{\!|P|\!\}, X \sqcap A \Vdash X \sqcap G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *heyting-le*:
  shows $\{\!|P|\!\}, A \sqcap B \Vdash B \longrightarrow_H G, \{\!|Q|\!\} \le B \longrightarrow_H \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *heyting-pre*:
  assumes *spec.idle* $\leq$ *G*
  shows *spec.pre* $P \longrightarrow_H \{\!\!\{P'\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\} = \{\!\!\{P \wedge P'\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *sort-of-refl*:
  assumes $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
  shows $c \leq \{\!\!\{P\}\!\!\}, A \Vdash c, \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *gen-asm-base*:
  assumes $P \Longrightarrow c \leq \{\!\!\{P' \wedge P''\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
  assumes *spec.idle* $\leq$ *G*
  shows $c \leq \{\!\!\{P' \wedge \langle P\rangle \wedge P''\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *gen-asm* =
  refinement.gen-asm-base[**where** P'=⟨*True*⟩ **and** P''=⟨*True*⟩, *simplified*]
  refinement.gen-asm-base[**where** P'=⟨*True*⟩, *simplified*]
  refinement.gen-asm-base[**where** P''=⟨*True*⟩, *simplified*]
  refinement.gen-asm-base

lemma *post-conj*:
  assumes $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
  assumes $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q'\}\!\!\}$
  shows $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{\lambda rv.\ Q\ rv \wedge Q'\ rv\}\!\!\}$
⟨*proof*⟩

lemma *conj-lift*:
  assumes $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
  assumes $c \leq \{\!\!\{P'\}\!\!\}, A \Vdash G, \{\!\!\{Q'\}\!\!\}$
  shows $c \leq \{\!\!\{P \wedge P'\}\!\!\}, A \Vdash G, \{\!\!\{\lambda rv.\ Q\ rv \wedge Q'\ rv\}\!\!\}$
⟨*proof*⟩

lemma *drop-imp*:
  assumes $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
  shows $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{\lambda rv.\ Q'\ rv \longrightarrow Q\ rv\}\!\!\}$
⟨*proof*⟩

lemma *prop*:
  shows $c \leq \{\!\!\{\langle P\rangle\}\!\!\}, A \Vdash c, \{\!\!\{\lambda v.\ \langle P\rangle\}\!\!\}$
⟨*proof*⟩

lemma *name-pre-state*:
  assumes $\bigwedge s.\ P\ s \Longrightarrow c \leq \{\!\!\{(=)\ s\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$
  assumes *spec.idle* $\leq$ *G*
  shows $c \leq \{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}$ (**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

12.1 Geenral rules for the *('a, 's, 'v) spec* lattice

⟨*ML*⟩

lemma *refinement*:
  shows *spec.term.all* $(\{\!\!\{P\}\!\!\}, A \Vdash G, \{\!\!\{Q\}\!\!\}) = \{\!\!\{P\}\!\!\}$, *spec.term.all* $A \Vdash$ *spec.term.all* $G, \{\!\!\{\top\}\!\!\}$

⟨*proof*⟩

⟨*ML*⟩

lemma *refinement-le*:
  shows $spec.term.none\ (\{\!|P|\!\},\ A \Vdash\ G,\ \{\!|Q|\!\}) \le \{\!|P|\!\},\ spec.term.all\ A \Vdash spec.term.all\ G,\ \{\!|\bot|\!\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *refinement*:
  fixes $af :: 'a \Rightarrow 'b$
  fixes $sf :: 's \Rightarrow 't$
  fixes $vf :: 'v \Rightarrow 'w$
  fixes $A :: ('b,\ 't,\ 'w)\ spec$
  fixes $G :: ('b,\ 't,\ 'w)\ spec$
  fixes $P :: 't\ pred$
  fixes $Q :: 'w \Rightarrow 't\ pred$
  shows $spec.invmap\ af\ sf\ vf\ (\{\!|P|\!\},\ A \Vdash\ G,\ \{\!|Q|\!\})$
    $= (\{\!|\lambda s.\ P\ (sf\ s)|\!\},\ spec.invmap\ af\ sf\ vf\ A \Vdash spec.invmap\ af\ sf\ vf\ G,\ \{\!|\lambda v\ s.\ Q\ (vf\ v)\ (sf\ s)|\!\})$
⟨*proof*⟩

⟨*ML*⟩

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 $idle \le A$.

⟨*ML*⟩

lemma *action*:
  fixes $F :: ('v \times 'a \times 's \times 's)\ set$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s') \in spec.initial\text{-}steps\ A]\!] \Longrightarrow Q\ v\ s'$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s) \in spec.initial\text{-}steps\ A]\!] \Longrightarrow (v,\ a,\ s,\ s') \in F'$
  shows $spec.action\ F \le \{\!|P|\!\},\ A \Vdash spec.action\ F',\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *return*:
  shows $spec.return\ v \le \{\!|Q\ v|\!\},\ A \Vdash spec.return\ v,\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *action-rel*:
  fixes $F :: ('v \times 'a \times 's \times 's)\ set$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s') \in spec.initial\text{-}steps\ A]\!] \Longrightarrow Q\ v\ s'$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s) \in spec.initial\text{-}steps\ A;\ s \ne s']\!] \Longrightarrow (a,\ s,\ s') \in r$
  shows $spec.action\ F \le \{\!|P|\!\},\ A \Vdash spec.rel\ r,\ \{\!|Q|\!\}$
⟨*proof*⟩

⟨*ML*⟩

12.1.2 Bind

Consider showing $f \gg= g \le f' \gg= g'$ 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 ($\gg=$) should do the job
  - however unlike Hoare and He (1987) we have no chance of a right residual for ($\gg=$) as we use traces (they use relations)
    * i.e., if it is not the case that $f \gg= \bot \le A$ then there is no continuation h such that $f \gg= h \le A$.
    * also such a residual h has arbitrary behavior starting from states that f cannot reach
      · i.e., for traces $\neg\sigma \le f$ $\langle\!|\sigma|\!\rangle \gg h \le A$ 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.

⟨*ML*⟩

definition *res* :: $('a, 's, 'v)$ *spec* ⇒ $('a, 's, 'w)$ *spec* ⇒ $'v$ ⇒ $('a, 's, 'w)$ *spec* **where**
  res f A v = $\bigsqcup$ {⦇*trace.final′ s xs*, *ys*, *w*⦈ |*s xs ys w*. ⦇*s*, *xs*, *Some v*⦈ ≤ *f* ∧ ⦇*s*, *xs* @ *ys*, *None*⦈ ≤ *A*}

⟨*ML*⟩

lemma *res-le-conv*[*spec.singleton.le-conv*]:
  shows ⦇σ⦈ ≤ *refinement.spec.bind.res f A v*
    ⟷ (∃ *s xs*. ⦇*s*, *xs*, *Some v*⦈ ≤ *f*
        ∧ *trace.init* σ = *trace.final′ s xs*
        ∧ ⦇*s*, *xs* @ *trace.rest* σ, *None*⦈ ≤ *A*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *resL*:
  shows *refinement.spec.bind.res* (*spec.term.none f*) *A v* = ⊥
⟨*proof*⟩

lemma *resR*:
  shows *refinement.spec.bind.res f* (*spec.term.none A*) *v* = *refinement.spec.bind.res f A v*
⟨*proof*⟩

⟨*ML*⟩

lemma *resR-mono*:
  shows *refinement.spec.bind.res f* (*spec.term.all A*) *v* = *refinement.spec.bind.res f A v*
⟨*proof*⟩

lemma *res*:
  shows *spec.term.all* (*refinement.spec.bind.res f A v*) = *refinement.spec.bind.res f A v*
⟨*proof*⟩

⟨*ML*⟩

lemma *res*:
  shows *refinement.spec.bind.res f A v* ∈ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
  shows *botL*: *refinement.spec.bind.res* $\bot = \bot$
    and *botR*: *refinement.spec.bind.res f* $\bot = \bot$
⟨*proof*⟩

lemma *mono*:
  assumes $f \le f'$
  assumes $A \le A'$
  shows *refinement.spec.bind.res f A v* $\le$ *refinement.spec.bind.res f′ A′ v*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
  assumes *st-ord F f f′*
  assumes *st-ord F A A′*
  shows *st-ord F* (*refinement.spec.bind.res f A v*) (*refinement.spec.bind.res f′ A′ v*)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *f*
  assumes *monotone orda* (≤) *A*
  shows *monotone orda* (≤) (λ*x*. *refinement.spec.bind.res* (*f x*) (*A x*) *v*)
⟨*proof*⟩

lemma *SupL*:
  shows *refinement.spec.bind.res* ($\bigsqcup X$) *A v* = ($\bigsqcup x \in X$. *refinement.spec.bind.res x A v*)
⟨*proof*⟩

lemma *SupR*:
  shows *refinement.spec.bind.res f* ($\bigsqcup X$) *v* = ($\bigsqcup x \in X$. *refinement.spec.bind.res f x v*)
⟨*proof*⟩

lemma *InfL-le*:
  shows *refinement.spec.bind.res* ($\bigsqcap X$) *A v* ≤ ($\bigsqcap x \in X$. *refinement.spec.bind.res x A v*)
⟨*proof*⟩

lemma *InfR-le*:
  shows *refinement.spec.bind.res f* ($\bigsqcap X$) *v* ≤ ($\bigsqcap x \in X$. *refinement.spec.bind.res f x v*)
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *f*
  assumes *mcont luba orda Sup* (≤) *A*
  shows *mcont luba orda Sup* (≤) (λ*x*. *refinement.spec.bind.res* (*f x*) (*A x*) *v*)
⟨*proof*⟩

lemma *returnL*:
  assumes *spec.idle* ≤ *A*
  shows *refinement.spec.bind.res* (*spec.return v*) *A v* = *spec.term.all A* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rel-le*:
  assumes $r \subseteq r'$
  shows *refinement.spec.bind.res f* (*spec.rel r*) *v* ≤ *spec.rel r′*
⟨*proof*⟩

⟨*ML*⟩

lemma *res-le*: — we can always discard the extra structure

shows *spec.steps* (*refinement.spec.bind.res f A v*) ⊆ *spec.steps A*
⟨*proof*⟩

⟨*ML*⟩

A refinement rule for (≫=). The function *vf* abstracts interstitial monadic return values.

⟨*ML*⟩

lemma *bind-abstract*:
fixes *f* :: ('*a*, '*s*, '*v*) *spec*
fixes *f'* :: ('*a*, '*s*, '*v'*) *spec*
fixes *g* :: '*v* ⇒ ('*a*, '*s*, '*w*) *spec*
fixes *g'* :: '*v'* ⇒ ('*a*, '*s*, '*w*) *spec*
fixes *vf* :: '*v* ⇒ '*v'*
assumes *g*: ⋀*v*. *g v* ≤ ⦃*Q'* (*vf v*)⦄, *refinement.spec.bind.res* (*spec.pre P* ⊓ *spec.term.all A* ⊓ *f'*) *A* (*vf v*) ⊩ *g'* (*vf v*), ⦃*Q*⦄
assumes *f*: *f* ≤ ⦃*P*⦄, *spec.term.all A* ⊩ *spec.vinvmap vf f'*, ⦃λ*v*. *Q'* (*vf v*)⦄
shows *f* ≫= *g* ≤ ⦃*P*⦄, *A* ⊩ *f'* ≫= *g'*, ⦃*Q*⦄
⟨*proof*⟩

lemmas *bind* = *refinement.spec.bind-abstract*[**where** *vf=id*, *simplified spec.invmap.id*, *simplified*]

12.1.3 Interference

lemma *rel-mono*:
assumes *r* ⊆ *r'*
assumes *stable* (*snd* ‘ (*spec.steps A* ∩ *r*)) *P*
shows *spec.rel r* ≤ ⦃*P*⦄, *A* ⊩ *spec.rel r'*, ⦃λ-::*unit*. *P*⦄
⟨*proof*⟩

⟨*ML*⟩

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).

⟨*ML*⟩

definition — roughly the *Parallel* construction with roles reversed
env-hyp :: ('*a* ⇒ '*s pred*) ⇒ (*sequential*, '*s*, *unit*) *spec* ⇒ '*a set* ⇒ ('*a* ⇒ (*sequential*, '*s*, *unit*) *spec*) ⇒ '*a* ⇒ (*sequential*, '*s*, *unit*) *spec*
where
env-hyp P A as Ps a =
spec.pre (⨅ (*P* ‘ *as*))
⊓ *spec.amap* (*toConcurrent-fn* (*proc a*))
(*spec.rel* (({*env*} ∪ *proc* ‘ *as*) × *UNIV*)
⊓ (⨅*i*∈*as*. *spec.toConcurrent i* (*Ps i*))
⊓ *spec.ainvmap toSequential-fn A*)

⟨*ML*⟩

lemma *mono*:
assumes ⋀*a*. *a* ∈ *as* ⟹ *P a* ≤ *P' a*
assumes *A* ≤ *A'*
assumes ⋀*a*. *a* ∈ *as* ⟹ *Ps a* ≤ *Ps' a*
shows *refinement.spec.env-hyp P A as Ps a* ≤ *refinement.spec.env-hyp P' A' as Ps' a*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
  assumes $\bigwedge a.\ a \in as \Longrightarrow st\text{-}ord\ F\ (P\ a)\ (P'\ a)$
  assumes $st\text{-}ord\ F\ A\ A'$
  assumes $\bigwedge a.\ a \in as \Longrightarrow st\text{-}ord\ F\ (Ps\ a)\ (Ps'\ a)$
  shows $st\text{-}ord\ F\ (refinement.spec.env\text{-}hyp\ P\ A\ as\ Ps\ a)\ (refinement.spec.env\text{-}hyp\ P'\ A'\ as\ Ps'\ a)$
⟨*proof*⟩

⟨*ML*⟩

lemma *Parallel*:
  fixes $A :: (sequential,\ 's,\ unit)\ spec$
  fixes $Q :: 'a \Rightarrow 's \Rightarrow bool$
  fixes $Ps :: 'a \Rightarrow (sequential,\ 's,\ unit)\ spec$
  fixes $Ps' :: 'a \Rightarrow (sequential,\ 's,\ unit)\ spec$
  assumes $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \le \{\!\!\{P\ a\}\!\!\},\ refinement.spec.env\text{-}hyp\ P\ A\ as\ Ps'\ a \Vdash Ps'\ a,\ \{\!\!\{\lambda v.\ Q\ a\}\!\!\}$
  shows $spec.Parallel\ as\ Ps \le \{\!\!\{\bigsqcap a{\in}as.\ P\ a\}\!\!\},\ A \Vdash spec.Parallel\ as\ Ps',\ \{\!\!\{\lambda v.\ \bigsqcap a{\in}as.\ Q\ a\}\!\!\}$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

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**
  $guar\ G \equiv spec.rel\ (\{env\} \times UNIV \cup \{self\} \times G)$

⟨*ML*⟩

definition $ag :: 's\ pred \Rightarrow 's\ rel \Rightarrow 's\ rel \Rightarrow ('v \Rightarrow 's\ pred) \Rightarrow (sequential,\ 's,\ 'v)\ spec$ (‹$\{\!\!\{\text{-}\}\!\!\}$, -/ ⊢ -, $\{\!\!\{\text{-}\}\!\!\}$› [*0,0,0,0*] *100*) **where**
  $\{\!\!\{P\}\!\!\},\ A \vdash G,\ \{\!\!\{Q\}\!\!\} = spec.pre\ P \sqcap ag.assm\ A \longrightarrow_H ag.guar\ G \sqcap spec.post\ Q$

⟨*ML*⟩

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*⟩

⟨*ML*⟩

lemma *refinement*:
shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P|\!\}$, *ag.assm* $A \Vdash$ *ag.guar* G, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *E*:
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
obtains $c \le$ *spec.pre* $P \sqcap$ *ag.assm* $A \longrightarrow_H$ *ag.guar* G
and $c \le$ *spec.pre* $P \sqcap$ *ag.assm* $A \longrightarrow_H$ *spec.post* Q
⟨*proof*⟩

lemma *pre-post-cong*:
assumes $P = P'$
assumes $Q = Q'$
shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$
⟨*proof*⟩

lemma *pre-bot*:
shows $\{\!|\bot|\!\}, A \vdash G, \{\!|Q|\!\} = \top$
and $\{\!|\langle\bot\rangle|\!\}, A \vdash G, \{\!|Q|\!\} = \top$
and $\{\!|\langle False\rangle|\!\}, A \vdash G, \{\!|Q|\!\} = \top$
⟨*proof*⟩

lemma *post-top*:
shows $\{\!|P|\!\}, A \vdash UNIV, \{\!|\top|\!\} = \top$
and $\{\!|P|\!\}, A \vdash UNIV, \{\!|\langle\top\rangle|\!\} = \top$
and $\{\!|P|\!\}, A \vdash UNIV, \{\!|\lambda\text{- -}.\ True|\!\} = \top$
⟨*proof*⟩

lemma *mono*:
assumes $P' \le P$
assumes $A' \le A$
assumes $G \le G'$
assumes $Q \le Q'$
shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} \le \{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\}$
⟨*proof*⟩

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 $(\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\})$ $(\{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\})$
⟨*proof*⟩

lemma *strengthen-pre*:
assumes *st-ord* $(\neg\ F)\ P\ P'$
shows *st-ord* F $(\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\})$ $(\{\!|P'|\!\}, A \vdash G, \{\!|Q|\!\})$
⟨*proof*⟩

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 $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
  assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
  assumes $A' \subseteq A$
  assumes $G \subseteq G'$
  assumes $\bigwedge v\ s.\ Q\ v\ s \Longrightarrow Q'\ v\ s$
  shows $c \le \{\!\!\{P'\}\!\!\}, A' \vdash G', \{\!\!\{Q'\}\!\!\}$
⟨*proof*⟩

lemmas *pre-pre-post* = *ag.pre*[*OF - - order.refl order.refl*, *of c*] **for** *c*

lemma *pre-imp*:
  assumes $\bigwedge s.\ P\ s \Longrightarrow P'\ s$
  assumes $c \le \{\!\!\{P'\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
  shows $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *pre-pre* = *ag.pre-imp*[*rotated*]

lemma *post-imp*:
  assumes $\bigwedge v\ s\ .\ Q\ v\ s \Longrightarrow Q'\ v\ s$
  assumes $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
  shows $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q'\}\!\!\}$
⟨*proof*⟩

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*: $\{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\} = \{\!\!\{P\}\!\!\}, A^{=} \vdash G, \{\!\!\{Q\}\!\!\}$
    and *reflcl-g*: $\{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\} = \{\!\!\{P\}\!\!\}, A \vdash G^{=}, \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *gen-asm-base*:
  assumes $P \Longrightarrow c \le \{\!\!\{P' \wedge P''\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
  shows $c \le \{\!\!\{P' \wedge \langle P\rangle \wedge P''\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *gen-asm* =
  ag.gen-asm-base[**where** *P'*=⟨*True*⟩ **and** *P''*=⟨*True*⟩, *simplified*]
  ag.gen-asm-base[**where** *P'*=⟨*True*⟩, *simplified*]
  ag.gen-asm-base[**where** *P''*=⟨*True*⟩, *simplified*]
  ag.gen-asm-base

lemma *post-conj*:
  assumes $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
  assumes $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q'\}\!\!\}$
  shows $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{\lambda v.\ Q\ v \wedge Q'\ v\}\!\!\}$
⟨*proof*⟩

lemma *pre-disj*:
  assumes $c \le \{\!\!\{P\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$
  assumes $c \le \{\!\!\{P'\}\!\!\}, A \vdash G, \{\!\!\{Q\}\!\!\}$

shows $c \le \{\!|P \vee P'|\!\}, A \vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *drop-imp*:
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
shows $c \le \{\!|P|\!\}, A \vdash G, \{\!|\lambda v.\ Q'\ v \longrightarrow Q\ v|\!\}$
⟨*proof*⟩

lemma *prop*:
shows $c \le \{\!|\langle P\rangle|\!\}, A \vdash UNIV, \{\!|\lambda v.\ \langle P\rangle|\!\}$
⟨*proof*⟩

lemma *name-pre-state*:
assumes $\bigwedge s.\ P\ s \Longrightarrow c \le \{\!|(=)\ s|\!\}, A \vdash G, \{\!|Q|\!\}$
shows $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *conj-lift*:
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $c \le \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$
shows $c \le \{\!|P \wedge P'|\!\}, A \vdash G, \{\!|\lambda v.\ Q\ v \wedge Q'\ v|\!\}$
⟨*proof*⟩

lemma *disj-lift*:
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $c \le \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$
shows $c \le \{\!|P \vee P'|\!\}, A \vdash G, \{\!|\lambda v.\ Q\ v \vee Q'\ v|\!\}$
⟨*proof*⟩

lemma *all-lift*:
assumes $\bigwedge x.\ c \le \{\!|P\ x|\!\}, A \vdash G, \{\!|Q\ x|\!\}$
shows $c \le \{\!|\forall x.\ P\ x|\!\}, A \vdash G, \{\!|\lambda v.\ \forall x.\ Q\ x\ v|\!\}$
⟨*proof*⟩

lemma *interference-le*:
shows $spec.rel\ (\{env\} \times UNIV) \le \{\!|P|\!\}, A \vdash G, \{\!|\top|\!\}$
and $spec.rel\ (\{env\} \times UNIV) \le \{\!|P|\!\}, A \vdash G, \{\!|\lambda\text{-}.\ \top|\!\}$
and $spec.rel\ (\{env\} \times UNIV) \le \{\!|P|\!\}, A \vdash G, \{\!|\lambda\text{- -}.\ True|\!\}$
⟨*proof*⟩

lemma *assm-heyting*:
fixes $Q :: 'v \Rightarrow 's\ pred$
shows $ag.assm\ r \longrightarrow_H \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, A \cap r \vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *augment-a*: — instantiate A'
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
shows $c \le \{\!|P|\!\}, A \cap A' \vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *augment-post*: — instantiate Q
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|\lambda v.\ Q'\ v \wedge Q\ v|\!\}$
shows $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q'|\!\}$
⟨*proof*⟩

lemma *augment-post-imp*: — instantiate Q
assumes $c \le \{\!|P|\!\}, A \vdash G, \{\!|\lambda v.\ (Q\ v \longrightarrow Q'\ v) \wedge Q\ v|\!\}$
shows $c \le \{\!|P|\!\}, A \vdash G, \{\!|Q'|\!\}$

⟨*proof*⟩

⟨*ML*⟩

lemma *ag-le*:
  shows $spec.term.none$ ({P}, $A \vdash G$, {Q}) $\leq$ {P}, $A \vdash G$, {$\bot$}
⟨*proof*⟩

⟨*ML*⟩

lemmas *none-inteference* =
  order.trans[*OF spec.term.none.mono*,
      OF ag.interference-le(*1*) *ag.pre-post*[**where** Q'=Q **for** Q, *OF spec.term.none.ag-le*, *simplified*]]

⟨*ML*⟩

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 $f \gg= g \leq$ {P}, $A \vdash G$, {Q}
⟨*proof*⟩

lemma *action*:
  fixes F :: $('v \times sequential \times 's \times 's)\ set$
  assumes Q: $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!] \Longrightarrow Q\ v\ s'$
  assumes G: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ self,\ s,\ s') \in F;\ s \neq s']\!] \Longrightarrow (s,\ s') \in G$
  shows $spec.action\ F \leq$ {P}, $A \vdash G$, {Q}
⟨*proof*⟩

lemma *return*:
  shows $spec.return\ v \leq$ {$Q\ v$}, $A \vdash G$, {Q}
⟨*proof*⟩

lemma *Parallel-assm*:
  shows $refinement.spec.env\text{-}hyp\ P\ (ag.assm\ A)\ as\ (ag.guar \circ G)\ a \leq ag.assm\ (A \cup \bigcup\ (G\ `\ (as - \{a\})))$
⟨*proof*⟩

lemma *Parallel-guar*:
  shows $spec.Parallel\ as\ (ag.guar \circ G) = ag.guar\ (\bigcup a{\in}as.\ G\ a)$
⟨*proof*⟩

lemma *Parallel*:
  fixes A :: $'s\ rel$
  fixes G :: $'a \Rightarrow 's\ rel$
  fixes Q :: $'a \Rightarrow 's \Rightarrow bool$
  fixes Ps :: $'a \Rightarrow (sequential,\ 's,\ unit)\ spec$
  assumes *proc-ag*: $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \leq$ {$P\ a$}, $A \cup (\bigcup a'{\in}as{-}\{a\}.\ G\ a') \vdash G\ a$, {$\lambda v.\ Q\ a$}
  shows $spec.Parallel\ as\ Ps \leq$ {$\prod a{\in}as.\ P\ a$}, $A \vdash \bigcup a{\in}as.\ G\ a$, {$\lambda rv.\ \prod a{\in}as.\ Q\ a$}
⟨*proof*⟩

⟨*ML*⟩

12.2.1 Stability rules

⟨*ML*⟩

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*⟩

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* $\langle P \rangle$
⟨*proof*⟩

⟨*ML*⟩

lemma *stable-lift*:
assumes *stable* $(A \cup G)$ P' — anything stable over $A \cup G$ is invariant
shows $\{\!|P \wedge P'|\!\}$, $A \vdash G$, $\{\!|\lambda v.\ P' \longrightarrow Q\ v|\!\} \leq \{\!|P \wedge P'|\!\}$, $A \vdash G$, $\{\!|\lambda v.\ Q\ v \wedge P'|\!\}$
⟨*proof*⟩

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 $c \leq \{\!|P \wedge P'|\!\}$, $A \vdash G$, $\{\!|\lambda v.\ Q\ v \wedge P'|\!\}$
⟨*proof*⟩

lemma *stable-augment*:
assumes $c \leq \{\!|P'|\!\}$, $A \vdash G$, $\{\!|Q'|\!\}$
assumes $\bigwedge v\ s.\ [\![P\ s;\ Q'\ v\ s]\!] \Longrightarrow Q\ v\ s$
assumes *stable* $(A \cup G)$ P
shows $c \leq \{\!|P' \wedge P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *stable-augment-post*:
assumes $c \leq \{\!|P'|\!\}$, $A \vdash G$, $\{\!|Q'|\!\}$ — resolve before application
assumes $\bigwedge v.$ *stable* $(A \cup G)$ $(Q'\ v \longrightarrow Q\ v)$
shows $c \leq \{\!|(\forall v.\ Q'\ v \longrightarrow Q\ v) \wedge P'|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

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 $c \leq \{\!|P \wedge P'|\!\}$, $A \vdash G$, $\{\!|\lambda v.\ Q\ v \wedge P'|\!\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *stable-interference*:
assumes *stable* $(A \cap r)$ P
shows *spec.rel* $(\{env\} \times r) \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|\langle P \rangle|\!\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *closed-ag*:
shows $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\} \in$ *spec.cam.closed* $(\{env\} \times r)$
⟨*proof*⟩

⟨*ML*⟩

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* $(\{env\} \times r)$ $(\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}) \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
⟨*proof*⟩

lemma *closed-ag*:
  assumes P: *stable* $(A \cap r)$ P
  assumes Q: $\bigwedge v$. *stable* $(A \cap r)$ $(Q\ v)$
  shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} \in$ *spec.interference.closed* $(\{env\} \times r)$
⟨*proof*⟩

⟨*ML*⟩

13 A programming language

The $('a, 's, 'v)$ *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 $('v, 's)$ *prog* lattice (§13.1) therefore handles the common case of the familiar constructs for sequential programming, and we lean on our $('a, 's, 'v)$ *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 *('s, 'v) prog* lattice

According to Müller-Olm (1997, §2.1), $('s, 'v)$ *prog* is a *sub-lattice* of $('a, 's, 'v)$ *spec* as the corresponding $(\sqcap)$ and $(\sqcup)$ operations coincide. However it is not a *complete* sub-lattice as *Sup* in $('s, 'v)$ *prog* needs to account for the higher bottom of that lattice.

typedef $('s, 'v)$ *prog* = *spec.interference.closed* $(\{env\} \times UNIV)$ :: $(sequential, 's, 'v)$ *spec set*
morphisms *p2s Abs-t*
⟨*proof*⟩

hide-const (**open**) *p2s*

setup-lifting *type-definition-prog*

instantiation *prog* :: $(type, type)$ *complete-distrib-lattice*
begin

lift-definition *bot-prog* :: $('s, 'v)$ *prog* **is** *spec.interference.cl* $(\{env\} \times UNIV)$ $\bot$ ⟨*proof*⟩
lift-definition *top-prog* :: $('s, 'v)$ *prog* **is** $\top$ ⟨*proof*⟩
lift-definition *sup-prog* :: $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* **is** *sup* ⟨*proof*⟩
lift-definition *inf-prog* :: $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* **is** *inf* ⟨*proof*⟩
lift-definition *less-eq-prog* :: $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* $\Rightarrow$ *bool* **is** *less-eq* ⟨*proof*⟩
lift-definition *less-prog* :: $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* $\Rightarrow$ *bool* **is** *less* ⟨*proof*⟩
lift-definition *Inf-prog* :: $('s, 'v)$ *prog set* $\Rightarrow$ $('s, 'v)$ *prog* **is** *Inf* ⟨*proof*⟩
lift-definition *Sup-prog* :: $('s, 'v)$ *prog set* $\Rightarrow$ $('s, 'v)$ *prog* **is** $\lambda X.\ Sup\ X \sqcup$ *spec.interference.cl* $(\{env\} \times UNIV)$ $\bot$ ⟨*proof*⟩

instance
⟨*proof*⟩

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.

⟨*ML*⟩

lemmas *p2s*[*iff*] = *prog.p2s*

⟨*ML*⟩

lemmas *p2s* = *spec.interference.closed-conv*[*OF spec.interference.closed.p2s*, *symmetric*, *of P* **for** *P*]

⟨*ML*⟩

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*]

⟨*ML*⟩

lemma *p2s-leI*:
  assumes *prog.p2s c* ≤ *prog.p2s d*
  shows *c* ≤ *d*
⟨*proof*⟩

⟨*ML*⟩

named-theorems *simps* ‹*simp rules for* ***const***‹*p2s*››

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* $(\bigsqcup X) = \bigsqcup(prog.p2s\ `\ X)$
⟨*proof*⟩

lemma *SUP-not-empty*:
  assumes $X \neq \{\}$
  shows *prog.p2s* $(\bigsqcup x{\in}X.\ f\ x) = (\bigsqcup x{\in}X.\ prog.p2s\ (f\ x))$
⟨*proof*⟩

lemma *monotone*:
  shows *mono prog.p2s*
⟨*proof*⟩

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* (≤) *Sup* (≤) *prog.p2s*
⟨*proof*⟩

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.p2s.mcont*, *of luba orda P* **for** *luba orda P*]

lemmas *Let-distrib* = *Let-distrib*[**where** *f*=*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* ({*env*} × *UNIV*) ⋙ (λ-::*unit*. *prog.p2s P*) = *prog.p2s P*
⟨*proof*⟩

⟨*ML*⟩

definition *s2p* :: (*sequential*, ′*s*, ′*v*) *spec* ⇒ (′*s*, ′*v*) *prog* **where** — Morally the upper of a Galois connection
  s2p P = ⨆{*c*. *prog.p2s c* ≤ *P*}

⟨*ML*⟩

lemma *bottom*:
  shows *prog.s2p* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
  shows *prog.s2p* ⊤ = ⊤
⟨*proof*⟩

lemma *monotone*:
  shows *mono prog.s2p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Sup-le*:
  shows ⨆(*prog.s2p* ‘ *X*) ≤ *prog.s2p* (⨆*X*)
⟨*proof*⟩

lemma *sup-le*:

shows $prog.s2p\ x \sqcup prog.s2p\ y \le prog.s2p\ (x \sqcup y)$
⟨*proof*⟩

lemma *Inf*:
shows $prog.s2p\ (\bigsqcap X) = \bigsqcap (prog.s2p\ `\ X)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *inf*:
shows $prog.s2p\ (x \sqcap y) = prog.s2p\ x \sqcap prog.s2p\ y$
⟨*proof*⟩

⟨*ML*⟩

lemma *galois*: — the Galois connection
shows $prog.p2s\ c \le S$
$\longleftrightarrow c \le prog.s2p\ S \wedge spec.term.none\ (spec.rel\ (\{env\} \times UNIV) :: (\text{-}, \text{-}, unit)\ spec) \le S$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *le*:
shows $prog.p2s\ (prog.s2p\ S) \le spec.interference.cl\ (\{env\} \times UNIV)\ S$
⟨*proof*⟩

lemma *insertion*:
fixes $S :: (sequential, 's, 'v)\ spec$
assumes $S \in spec.interference.closed\ (\{env\} \times UNIV)$
shows $prog.p2s\ (prog.s2p\ S) = S$
⟨*proof*⟩

⟨*ML*⟩

13.3 Programming language constructs

We lift the combinators directly from the $('a, 's, 'v)$ *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*.

⟨*ML*⟩

lift-definition $bind :: ('s, 'v)\ prog \Rightarrow ('v \Rightarrow ('s, 'w)\ prog) \Rightarrow ('s, 'w)\ prog$ **is**
spec.bind ⟨*proof*⟩

adhoc-overloading
Monad-Syntax.bind ⇌ *prog.bind*

lift-definition $action :: ('v \times 's \times 's)\ set \Rightarrow ('s, 'v)\ prog$ **is**
$\lambda F.\ spec.interference.cl\ (\{env\} \times UNIV)\ (spec.action\ (map\text{-}prod\ id\ (Pair\ self)\ `\ F))$ ⟨*proof*⟩

abbreviation (*input*) $det\text{-}action :: ('s \Rightarrow ('v \times 's)) \Rightarrow ('s, 'v)\ prog$ **where**
$det\text{-}action\ f \equiv prog.action\ \{(v, s, s').\ (v, s') = f\ s\}$

definition $return :: 'v \Rightarrow ('s, 'v)\ prog$ **where**
$return\ v = prog.action\ (\{v\} \times Id)$

definition $guard :: 's\ pred \Rightarrow ('s, unit)\ prog$ **where**
$guard\ g \equiv prog.action\ (\{()\} \times Diag\ g)$

abbreviation (*input*) $read :: ('s \Rightarrow 'v) \Rightarrow ('s, 'v)\ prog$ **where**
$read\ F \equiv prog.action\ \{(F\ s, s, s)\ |s.\ True\}$

abbreviation (*input*) *write* :: $('s \Rightarrow 's) \Rightarrow ('s,\ unit)\ prog$ **where**
write $F \equiv prog.action\ \{((),\ s,\ F\ s)\ |s.\ True\}$

lift-definition *Parallel* :: $'a\ set \Rightarrow ('a \Rightarrow ('s,\ unit)\ prog) \Rightarrow ('s,\ unit)\ prog$ **is** *spec.Parallel*
⟨*proof*⟩

lift-definition *parallel* :: $('s,\ unit)\ prog \Rightarrow ('s,\ unit)\ prog \Rightarrow ('s,\ unit)\ prog$ **is** *spec.parallel*
⟨*proof*⟩

lift-definition *vmap* :: $('v \Rightarrow 'w) \Rightarrow ('s,\ 'v)\ prog \Rightarrow ('s,\ 'w)\ prog$ **is** *spec.vmap*
⟨*proof*⟩

adhoc-overloading
Parallel ⇌ *prog.Parallel*
adhoc-overloading
parallel ⇌ *prog.parallel*

lemma *return-alt-def*:
shows $prog.return\ v = prog.read\ \langle v\rangle$
⟨*proof*⟩

lemma *parallel-alt-def*:
shows $prog.parallel\ P\ Q = prog.Parallel\ UNIV\ (\lambda a{::}bool.\ if\ a\ then\ P\ else\ Q)$
⟨*proof*⟩

lift-definition *rel* :: $'s\ rel \Rightarrow ('s,\ 'v)\ prog$ **is** $\lambda r.\ spec.rel\ (\{env\} \times UNIV \cup \{self\} \times r)$
⟨*proof*⟩

lift-definition *steps* :: $('s,\ 'v)\ prog \Rightarrow 's\ rel$ **is** $\lambda P.\ spec.steps\ P\ ``\ \{self\}$ ⟨*proof*⟩

lift-definition *invmap* :: $('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('t,\ 'w)\ prog \Rightarrow ('s,\ 'v)\ prog$ **is**
spec.invmap id
⟨*proof*⟩

abbreviation *sinvmap* ::$('s \Rightarrow 't) \Rightarrow ('t,\ 'v)\ prog \Rightarrow ('s,\ 'v)\ prog$ **where**
sinvmap sf ≡ *prog.invmap sf id*
abbreviation *vinvmap* ::$('v \Rightarrow 'w) \Rightarrow ('s,\ 'w)\ prog \Rightarrow ('s,\ 'v)\ prog$ **where**
vinvmap vf ≡ *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

⟨*ML*⟩

lemma *bind*[*prog.p2s.simps*]:
shows $prog.p2s\ (f \gg= g) = prog.p2s\ f \gg= (\lambda x.\ prog.p2s\ (g\ x))$
⟨*proof*⟩

lemmas *action* = *prog.action.rep-eq*

lemma *return*:
shows *prog.p2s* (*prog.return v*) = *spec.interference.cl* ({*env*} × *UNIV*) (*spec.return v*)
⟨*proof*⟩

lemma *guard*:
shows *prog.p2s* (*prog.guard g*) = *spec.interference.cl* ({*env*} × *UNIV*) (*spec.guard g*)
⟨*proof*⟩

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*

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (=) *cr-prog* (λ*v*. *spec.interference.cl* ({*env*} × *UNIV*) (*spec.return v*)) *prog.return*
⟨*proof*⟩

lemma *cong*:
fixes F :: ($'v \times 's \times 's$) *set*
assumes $\bigwedge v\ s\ s'.\ (v, s, s') \in F \Longrightarrow s' = s$
assumes $\bigwedge v\ s\ s'\ s''.\ v \in fst\ `\ F \Longrightarrow (v, s, s) \in F$
shows *prog.action* F = ($\bigsqcup$(v, s, s')∈F. *prog.return v*)
⟨*proof*⟩

lemma *rel-le*:
shows *prog.return v* ≤ *prog.rel r*
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
shows *prog.action* {} = ⊥
⟨*proof*⟩

lemma *monotone*:
shows *mono* (*prog.action* :: - ⇒ ($'s$, $'v$) *prog*)
⟨*proof*⟩

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*∈*Fs*. *prog.action F*)
⟨*proof*⟩

lemmas *sup* = *prog.action.Sup*[**where** *Fs*={*F*, *G*} **for** *F G*, *simplified*]

lemma *Inf-le*:
shows *prog.action* ($\bigsqcap$ *Fs*) ≤ ($\bigsqcap$ *F*∈*Fs*. *prog.action F*)
⟨*proof*⟩

lemma *inf-le*:

shows *prog.action* $(F \sqcap G) \le$ *prog.action* $F \sqcap$ *prog.action* G
⟨*proof*⟩

lemma *sinvmap-le*: — a strict refinement
shows *prog.p2s* (*prog.action* (*map-prod id* (*map-prod sf sf*) $-$‘ F))
$\le$ *spec.sinvmap sf* (*prog.p2s* (*prog.action* F))
⟨*proof*⟩

lemma *return-const*:
fixes F :: *'s rel*
fixes V :: *'v set*
fixes W :: *'w set*
assumes $V \neq \{\}$
assumes $W \neq \{\}$
shows *prog.action* $(V \times F) =$ *prog.action* $(W \times F) \gg (\bigsqcup v \in V.$ *prog.return* $v)$
⟨*proof*⟩

lemma *rel-le*:
assumes $\bigwedge v\ s\ s'.\ (v,\ s,\ s') \in F \Longrightarrow (s,\ s') \in r \vee s = s'$
shows *prog.action* $F \le$ *prog.rel* r
⟨*proof*⟩

lemma *invmap-le*:
shows *prog.action* (*map-prod vf* (*map-prod sf sf*) $-$‘ F) $\le$ *prog.invmap sf vf* (*prog.action* F)
⟨*proof*⟩

lemma *action-le*:
shows *spec.action* (*map-prod id* (*Pair self*) ‘ F) $\le$ *prog.p2s* (*prog.action* F)
⟨*proof*⟩

⟨*ML*⟩

lemmas *if-distrL* = *if-distrib*[**where** $f=\lambda x.\ x \ggg g$ **for** g :: - ⇒ (-, -) *prog*]

lemma *mono*:
assumes $f \le f'$
assumes $\bigwedge x.\ g\ x \le g'\ x$
shows *prog.bind* $f\ g \le$ *prog.bind* $f'\ g'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* $F\ f\ f'$
assumes $\bigwedge x.$ *st-ord* $F\ (g\ x)\ (g'\ x)$
shows *st-ord* F (*prog.bind* $f\ g$) (*prog.bind* $f'\ g'$)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* $(\le)$ f
assumes $\bigwedge x.$ *monotone orda* $(\le)$ $(\lambda y.\ g\ y\ x)$
shows *monotone orda* $(\le)$ $(\lambda x.$ *prog.bind* $(f\ x)\ (g\ x))$
⟨*proof*⟩

The monad laws hold unconditionally in the (*'s*, *'v*) *prog* lattice.

lemma *bind*:
shows $f \ggg g \ggg h =$ *prog.bind* f $(\lambda x.\ g\ x \ggg h)$
⟨*proof*⟩

lemma *return*:

shows *returnL*: $(\gg\!=)$ $(prog.return\ v) = (\lambda g :: 'v \Rightarrow ('s,\ 'w)\ prog.\ g\ v)$ (**is** *?thesis1*)
and *returnR*: $f \gg\!= prog.return = f$ (**is** *?thesis2*)
⟨*proof*⟩

lemma *botL*:
shows $prog.bind\ \bot = \bot$
⟨*proof*⟩

lemma *botR-le*:
shows $prog.bind\ f\ \langle\bot\rangle \le f$ (**is** *?thesis1*)
and $prog.bind\ f\ \bot \le f$ (**is** *?thesis2*)
⟨*proof*⟩

lemma
fixes f :: (-, -) *prog*
fixes f_1 :: (-, -) *prog*
shows *supL*: $(f_1 \sqcup f_2) \gg\!= g = (f_1 \gg\!= g) \sqcup (f_2 \gg\!= g)$
and *supR*: $f \gg\!= (\lambda x.\ g_1\ x \sqcup g_2\ x) = (f \gg\!= g_1) \sqcup (f \gg\!= g_2)$
⟨*proof*⟩

lemma *SUPL*:
fixes X :: - *set*
fixes f :: - $\Rightarrow$ (-, -) *prog*
shows $(\bigsqcup x{\in}X.\ f\ x) \gg\!= g = (\bigsqcup x{\in}X.\ f\ x \gg\!= g)$
⟨*proof*⟩

lemma *SUPR*:
fixes X :: - *set*
fixes f :: (-, -) *prog*
shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x) \sqcup (f \gg\!= \bot)$
⟨*proof*⟩

lemma *SupR*:
fixes X :: - *set*
fixes f :: (-, -) *prog*
shows $f \gg (\bigsqcup X) = (\bigsqcup x{\in}X.\ f \gg x) \sqcup (f \gg\!= \bot)$
⟨*proof*⟩

lemma *SUPR-not-empty*:
fixes f :: (-, -) *prog*
assumes $X \neq \{\}$
shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x)$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* $(\le)$ f
assumes $\bigwedge v.$ *mcont luba orda Sup* $(\le)$ $(\lambda x.\ g\ x\ v)$
shows *mcont luba orda Sup* $(\le)$ $(\lambda x.\ prog.bind\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *inf-rel*:
shows $prog.rel\ r \sqcap (f \gg\!= g) = prog.rel\ r \sqcap f \gg\!= (\lambda x.\ prog.rel\ r \sqcap g\ x)$
and $(f \gg\!= g) \sqcap prog.rel\ r = prog.rel\ r \sqcap f \gg\!= (\lambda x.\ prog.rel\ r \sqcap g\ x)$
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:

shows *prog.guard* $\bot = \bot$
and *prog.guard* ⟨*False*⟩ $= \bot$
⟨*proof*⟩

lemma *top*:
shows *prog.guard* ($\top$::'*a pred*) = *prog.return* () (**is** *?thesis1*)
and *prog.guard* (⟨*True*⟩::'*a pred*) = *prog.return* () (**is** *?thesis2*)
⟨*proof*⟩

lemma *return-le*:
shows *prog.guard g* $\le$ *prog.return* ()
⟨*proof*⟩

lemma *monotone*:
shows *mono* (*prog.guard* :: '*s pred* $\Rightarrow$ -)
⟨*proof*⟩

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*⟩

lemma *if*:
shows (*if b then t else e*) = (*prog.guard* ⟨*b*⟩ $\gg$ *t*) $\sqcup$ (*prog.guard* ⟨$\neg$*b*⟩ $\gg$ *e*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
assumes $\bigwedge a.\ a \in bs \Longrightarrow Ps\ a = \bot$
assumes $as \cap bs \neq \{\}$
shows *prog.Parallel as Ps* = *prog.Parallel* (*as* − *bs*) *Ps* $\gg= \bot$
⟨*proof*⟩

lemma *mono*:
assumes $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \le Ps'\ a$
shows *prog.Parallel as Ps* $\le$ *prog.Parallel as Ps'*
⟨*proof*⟩

lemma *strengthen-Parallel*[*strg*]:
assumes $\bigwedge a.\ a \in as \Longrightarrow$ *st-ord F* (*Ps a*) (*Ps' a*)
shows *st-ord F* (*prog.Parallel as Ps*) (*prog.Parallel as Ps'*)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes $\bigwedge a.\ a \in as \Longrightarrow$ *monotone orda* ($\le$) (*F a*)
shows *monotone orda* ($\le$) ($\lambda f.$ *prog.Parallel as* ($\lambda a.$ *F a f*))
⟨*proof*⟩

lemma *cong*:
assumes *as* = *as'*
assumes $\bigwedge a.\ a \in as' \Longrightarrow Ps\ a = Ps'\ a$
shows *prog.Parallel as Ps* = *prog.Parallel as' Ps'*

⟨*proof*⟩

lemma *no-agents*:
  shows $prog.Parallel\ \{\}\ Ps = prog.return\ ()$
⟨*proof*⟩

lemma *singleton-agents*:
  shows $prog.Parallel\ \{a\}\ Ps = Ps\ a$
⟨*proof*⟩

lemma *rename-UNIV*:
  assumes *inj-on f as*
  shows *prog.Parallel as Ps*
    $= prog.Parallel\ UNIV\ (\lambda b.\ if\ b \in f\ `\ as\ then\ Ps\ (inv\text{-}into\ as\ f\ b)\ else\ prog.return\ ())$
⟨*proof*⟩

lemmas *rename* = *spec.Parallel.rename*[*transferred*]

lemma *return*:
  assumes $\bigwedge a.\ a \in bs \Longrightarrow Ps\ a = prog.return\ ()$
  shows $prog.Parallel\ as\ Ps = prog.Parallel\ (as - bs)\ Ps$
⟨*proof*⟩

lemma *unwind*:
  assumes a: $f \gg= g \le Ps\ a$ — The selected process starts with action f
  assumes $a \in as$
  shows $f \gg= (\lambda v.\ prog.Parallel\ as\ (Ps(a:=g\ v))) \le prog.Parallel\ as\ Ps$
⟨*proof*⟩

⟨*ML*⟩

lemmas *commute* = *spec.parallel.commute*[*transferred*]
lemmas *assoc* = *spec.parallel.assoc*[*transferred*]
lemmas *mono* = *spec.parallel.mono*[*transferred*]

lemma *strengthen*[*strg*]:
  assumes $st\text{-}ord\ F\ P\ P'$
  assumes $st\text{-}ord\ F\ Q\ Q'$
  shows $st\text{-}ord\ F\ (prog.parallel\ P\ Q)\ (prog.parallel\ P'\ Q')$
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes $monotone\ orda\ (\le)\ F$
  assumes $monotone\ orda\ (\le)\ G$
  shows $monotone\ orda\ (\le)\ (\lambda f.\ prog.parallel\ (F\ f)\ (G\ f))$
⟨*proof*⟩

lemma *bot*:
  shows *botL*: $prog.parallel\ \bot\ P = P \gg= \bot$ (**is** *?thesis1*)
    and *botR*: $prog.parallel\ P\ \bot = P \gg= \bot$ (**is** *?thesis2*)
⟨*proof*⟩

lemma *return*:
  shows *returnL*: $prog.return\ ()\ \|\ P = P$ (**is** *?thesis1*)
    and *returnR*: $P\ \|\ prog.return\ () = P$ (**is** *?thesis2*)
⟨*proof*⟩

lemma *Sup-not-empty*:

fixes X :: (-, *unit*) *prog 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*⟩

lemma *sup*:
fixes P :: (-, *unit*) *prog*
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *P*
assumes *mcont luba orda Sup* (≤) *Q*
shows *mcont luba orda Sup* (≤) ($\lambda x.$ *prog.parallel* ($P\ x$) ($Q\ x$))
⟨*proof*⟩

lemma *unwindL*:
fixes f :: ($'s$, $'v$) *prog*
assumes *a*: $f \ggg g \leq P$ — The selected process starts with action f
shows $f \ggg (\lambda v.\ g\ v \parallel Q) \leq P \parallel Q$
⟨*proof*⟩

lemma *unwindR*:
fixes f :: ($'s$, $'v$) *prog*
assumes *a*: $f \ggg g \leq Q$ — The selected process starts with action f
shows $f \ggg (\lambda v.\ P \parallel g\ v) \leq P \parallel Q$
⟨*proof*⟩

⟨*ML*⟩

lemma *parallel-le*:
fixes P :: (-, -) *prog*
shows $P \gg Q \leq P \parallel Q$
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
shows *prog.invmap sf vf* $\bot$ = (*prog.rel* (*map-prod sf sf* −' *Id*) :: (-, *unit*) *prog*) $\ggg \bot$
⟨*proof*⟩

lemma *id*:
shows *prog.invmap id id* $P = P$
and *prog.invmap* ($\lambda x.\ x$) ($\lambda x.\ x$) $P = P$
⟨*proof*⟩

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* ∘ *prog.invmap sg vg* = *prog.invmap* (*sg* ∘ *sf*) (*vg* ∘ *vf*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *monotone*:
shows *mono* (*prog.invmap sf vf*)
⟨*proof*⟩

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* (≤) *t*
  shows *monotone orda* (≤) (λ*x*. *prog.invmap sf vf* (*t x*))
⟨*proof*⟩

lemma *Sup*:
  fixes *sf* :: ′*s* ⇒ ′*t*
  fixes *vf* :: ′*v* ⇒ ′*w*
  shows *prog.invmap sf vf* (⨆*X*) = ⨆(*prog.invmap sf vf* ‘ *X*) ⊔ *prog.invmap sf vf* ⊥
⟨*proof*⟩

lemma *Sup-not-empty*:
  assumes *X* ≠ {}
  shows *prog.invmap sf vf* (⨆*X*) = ⨆(*prog.invmap sf vf* ‘ *X*)
⟨*proof*⟩

lemma *mcont*:
  shows *mcont Sup* (≤) *Sup* (≤) (*prog.invmap sf vf*)
⟨*proof*⟩

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* ⤜ *g*) = *prog.sinvmap sf f* ⤜ (λ*v*. *prog.invmap sf vf* (*g v*))
⟨*proof*⟩

lemma *parallel*:
  shows *prog.invmap sf vf* (*P* ∥ *Q*) = *prog.invmap sf vf P* ∥ *prog.invmap sf vf Q*
⟨*proof*⟩

lemma *invmap-image-vimage-commute*:
  shows *map-prod id* (*map-prod id sf*) −‘ *map-prod id* (*Pair self*) ‘ *F*
      = *map-prod id* (*Pair self*) ‘ *map-prod id sf* −‘ *F*
⟨*proof*⟩

lemma *action*:
  shows *prog.invmap sf vf* (*prog.action F*)
      = *prog.rel* (*map-prod sf sf* −‘ *Id*)
        ⤜ (λ-::*unit*. *prog.action* (*map-prod id* (*map-prod sf sf*) −‘ *F*)
          ⤜ (λ*v*. *prog.rel* (*map-prod sf sf* −‘ *Id*)
            ⤜ (λ-::*unit*. ⨆*v*′∈*vf* −‘ {*v*}. *prog.return v*′)))
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
  shows *prog.vmap vf* ⊥ = ⊥
⟨*proof*⟩

lemma *unitL*:
  shows *f* ≫ *g* = *prog.vmap* ⟨()⟩ *f* ≫ *g*
⟨*proof*⟩

lemma *eq-return*:
  shows *prog.vmap vf P* = *P* ⤜ *prog.return* ∘ *vf* (**is** *?thesis1*)
    and *prog.vmap vf P* = *P* ⤜ (λ*v*. *prog.return* (*vf v*)) (**is** *?thesis2*)

⟨*proof*⟩

lemma *action*:
  shows *prog.vmap vf* (*prog.action F*) = *prog.action* (*map-prod vf id* ‘ *F*)
⟨*proof*⟩

lemma *return*:
  shows *prog.vmap vf* (*prog.return v*) = *prog.return* (*vf v*)
⟨*proof*⟩

⟨*ML*⟩

interpretation *kleene*: *kleene prog.return* () *λx y. prog.bind x* ⟨*y*⟩
⟨*proof*⟩

interpretation *rel*: *galois.complete-lattice-class prog.steps prog.rel*
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
  shows *prog.rel* {} = ⨆*range prog.return*
⟨*proof*⟩

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 *prog.rel* (*r* ∪ *Id*) = (*prog.rel r* :: (′*s*, ′*v*) *prog*) (**is** *?thesis1*)
    and *prog.rel* (*Id* ∪ *r*) = (*prog.rel r* :: (′*s*, ′*v*) *prog*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *minus-Id*:
  shows *prog.rel* (*r* − *Id*) = *prog.rel r*
⟨*proof*⟩

lemma *Id*:
  shows *prog.rel Id* = ⨆*range prog.return*
⟨*proof*⟩

lemma *unfoldL*:
  fixes *r* :: ′*s rel*
  assumes *Id* ⊆ *r*
  shows *prog.rel r* = *prog.action* ({()} × *r*) ≫ *prog.rel r*
⟨*proof*⟩

lemma *wind-bind*: — arbitrary interstitial return type
  shows *prog.rel r* ≫ *prog.rel r* = *prog.rel r*
⟨*proof*⟩

lemma *wind-bind-leading*: — arbitrary interstitial return type
  assumes *r*′ ⊆ *r*
  shows *prog.rel r*′ ≫ *prog.rel r* = *prog.rel r*
⟨*proof*⟩

lemma *wind-bind-trailing*: — arbitrary interstitial return type
  assumes $r' \subseteq r$
  shows *prog.rel r* ≫ *prog.rel r'* = *prog.rel r* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Interstitial unit, for unfolding

lemmas *unwind-bind* = *prog.rel.wind-bind*[**where** *'c=unit*, *symmetric*]
lemmas *unwind-bind-leading* = *prog.rel.wind-bind-leading*[**where** *'c=unit*, *symmetric*]
lemmas *unwind-bind-trailing* = *prog.rel.wind-bind-trailing*[**where** *'c=unit*, *symmetric*]

lemma *mono-conv*:
  shows *prog.rel r* = *prog.kleene.star* (*prog.action* ($\{()\} \times r^{=}$)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-rel*:
  assumes *refl r*
  shows *prog.action F* ⊓ *prog.rel r* = *prog.action* ($F \cap UNIV \times r$) (**is** *?thesis1*)
    and *prog.rel r* ⊓ *prog.action F* = *prog.action* ($F \cap UNIV \times r$) (**is** *?thesis2*)
⟨*proof*⟩

lemma *inf-rel-reflcl*:
  shows *prog.action F* ⊓ *prog.rel r* = *prog.action* ($F \cap UNIV \times r^{=}$)
    and *prog.rel r* ⊓ *prog.action F* = *prog.action* ($F \cap UNIV \times r^{=}$)
⟨*proof*⟩

⟨*ML*⟩

lemma *not-bot*:
  shows *prog.return v* ≠ (⊥ :: (*'s*, *'v*) *prog*)
⟨*proof*⟩

⟨*ML*⟩

lemma *return*:
  shows *prog.invmap sf vf* (*prog.return v*)
      = *prog.rel* (*map-prod sf sf* −‘ *Id*) ⤜ (λ-::*unit*. ⨆*v'*∈*vf* −‘ {*v*}. *prog.return v'*)
⟨*proof*⟩

lemma *split-vinvmap*:
  fixes *P* :: (*'s*, *'v*) *prog*
  shows *prog.invmap sf vf P* = *prog.sinvmap sf P* ⤜ (λ*v*. ⨆*v'*∈*vf* −‘ {*v*}. *prog.return v'*)
⟨*proof*⟩

⟨*ML*⟩

13.4 Refinement for *('s, 'v) prog*

We specialize the rules of §12.1 to the (*'s*, *'v*) *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.

⟨*ML*⟩

lemma *leI*:
assumes *prog.p2s c* $\leq$ ⦃⟨*True*⟩⦄, $\top \Vdash$ *prog.p2s d*, ⦃λ-. ⟨*True*⟩⦄
shows $c \leq d$
⟨*proof*⟩

lemma *eqI*:
assumes *prog.p2s c* $\leq$ ⦃⟨*True*⟩⦄, $\top \Vdash$ *prog.p2s d*, ⦃λ-. ⟨*True*⟩⦄
assumes *prog.p2s d* $\leq$ ⦃⟨*True*⟩⦄, $\top \Vdash$ *prog.p2s c*, ⦃λ-. ⟨*True*⟩⦄
shows $c = d$
⟨*proof*⟩

⟨*ML*⟩

13.4.2 Galois considerations

Refinement quadruples ⦃P⦄, $A \Vdash G$, ⦃Q⦄ denote points in the ($'s$, $'v$) *prog* lattice provided G is suitably interference closed.

⟨*ML*⟩

lemma *galois*:
assumes *spec.term.none* (*spec.rel* ({*env*} × *UNIV*) :: (-, -, *unit*) *spec*) $\leq G$
shows *prog.p2s c* $\leq$ ⦃P⦄, $A \Vdash G$, ⦃Q⦄ $\longleftrightarrow$ $c \leq$ *prog.s2p* (⦃P⦄, $A \Vdash G$, ⦃Q⦄)
⟨*proof*⟩

lemmas *s2p-refinement* = *iffD1*[*OF refinement.prog.galois*, *rotated*]

lemma *p2s-s2p*:
assumes *spec.term.none* (*spec.rel* ({*env*} × *UNIV*) :: (-, -, *unit*) *spec*) $\leq G$
shows *prog.p2s* (*prog.s2p* (⦃P⦄, $A \Vdash G$, ⦃Q⦄)) $\leq$ ⦃P⦄, $A \Vdash G$, ⦃Q⦄
⟨*proof*⟩

⟨*ML*⟩

13.4.3 Rules

⟨*ML*⟩

lemma *bot*[*iff*]:
shows *prog.p2s* $\bot \leq$ ⦃P⦄, $A \Vdash$ *prog.p2s* c', ⦃Q⦄
⟨*proof*⟩

lemma *sup-conv*:
shows *prog.p2s* ($c_1 \sqcup c_2$) $\leq$ ⦃P⦄, $A \Vdash G$, ⦃Q⦄
$\longleftrightarrow$ *prog.p2s* $c_1 \leq$ ⦃P⦄, $A \Vdash G$, ⦃Q⦄ $\wedge$ *prog.p2s* $c_2 \leq$ ⦃P⦄, $A \Vdash G$, ⦃Q⦄
⟨*proof*⟩

lemmas *sup* = *iffD2*[*OF refinement.prog.sup-conv*, *unfolded conj-explode*]

lemma *if*:
assumes $i \Longrightarrow$ *prog.p2s t* $\leq$ ⦃P⦄, $A \Vdash$ *prog.p2s* t', ⦃Q⦄
assumes $\neg i \Longrightarrow$ *prog.p2s e* $\leq$ ⦃P'⦄, $A \Vdash$ *prog.p2s* e', ⦃Q⦄
shows *prog.p2s* (*if i then t else e*) $\leq$ ⦃*if i then P else P'*⦄, $A \Vdash$ *prog.p2s* (*if i then t' else e'*), ⦃Q⦄
⟨*proof*⟩

lemmas *if'* = *refinement.prog.if*[**where** P=P **and** P'=P, *simplified*] **for** P

lemma *case-option*:

assumes $opt = None \Longrightarrow prog.p2s\ none \le \{\!|P_n|\!\}, A \Vdash prog.p2s\ none', \{\!|Q|\!\}$
assumes $\bigwedge v.\ opt = Some\ v \Longrightarrow prog.p2s\ (some\ v) \le \{\!|P_s\ v|\!\}, A \Vdash prog.p2s\ (some'\ v), \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}option\ none\ some\ opt) \le \{\!|case\ opt\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v|\!\}, A \Vdash prog.p2s\ (case\text{-}option\ none'\ some'\ opt), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *case-sum*:
assumes $\bigwedge v.\ x = Inl\ v \Longrightarrow prog.p2s\ (left\ v) \le \{\!|P_l\ v|\!\}, A \Vdash prog.p2s\ (left'\ v), \{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Inr\ v \Longrightarrow prog.p2s\ (right\ v) \le \{\!|P_r\ v|\!\}, A \Vdash prog.p2s\ (right'\ v), \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}sum\ left\ right\ x) \le \{\!|case\text{-}sum\ P_l\ P_r\ x|\!\}, A \Vdash prog.p2s\ (case\text{-}sum\ left'\ right'\ x), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *case-list*:
assumes $x = [] \Longrightarrow prog.p2s\ nil \le \{\!|P_n|\!\}, A \Vdash prog.p2s\ nil', \{\!|Q|\!\}$
assumes $\bigwedge v\ vs.\ x = v\ \#\ vs \Longrightarrow prog.p2s\ (cons\ v\ vs) \le \{\!|P_c\ v\ vs|\!\}, A \Vdash prog.p2s\ (cons'\ v\ vs), \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}list\ nil\ cons\ x) \le \{\!|case\text{-}list\ P_n\ P_c\ x|\!\}, A \Vdash prog.p2s\ (case\text{-}list\ nil'\ cons'\ x), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *action*:
fixes $F :: ('v \times 's \times 's)\ set$
assumes $\bigwedge v\ s\ s'.\ [\![P\ s; (v, s, s') \in F; (self, s, s') \in spec.steps\ A \lor s = s']\!] \Longrightarrow Q\ v\ s'$
assumes $\bigwedge v\ s\ s'.\ [\![P\ s; (v, s, s') \in F]\!] \Longrightarrow (v, s, s') \in F'$
assumes *sP*: $stable\ (spec.steps\ A\ ``\ \{env\})\ P$
assumes $\bigwedge v\ s\ s'.\ [\![P\ s; (v, s, s') \in F]\!] \Longrightarrow stable\ (spec.steps\ A\ ``\ \{env\})\ (Q\ v)$
shows $prog.p2s\ (prog.action\ F) \le \{\!|P|\!\}, A \Vdash prog.p2s\ (prog.action\ F'), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *return*:
assumes *sQ*: $stable\ (spec.steps\ A\ ``\ \{env\})\ (Q\ v)$
shows $prog.p2s\ (prog.return\ v) \le \{\!|Q\ v|\!\}, A \Vdash prog.p2s\ (prog.return\ v), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *invmap-return*:
assumes *sQ*: $stable\ (spec.steps\ A\ ``\ \{env\})\ (Q\ v)$
assumes $vf\ v = v'$
shows $prog.p2s\ (prog.return\ v) \le \{\!|Q\ v|\!\}, A \Vdash prog.p2s\ (prog.invmap\ sf\ vf\ (prog.return\ v')), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *bind-abstract*:
fixes $f :: ('s, 'v)\ prog$
fixes $f' :: ('s, 'v')\ prog$
fixes $g :: 'v \Rightarrow ('s, 'w)\ prog$
fixes $g' :: 'v' \Rightarrow ('s, 'w)\ prog$
fixes $vf :: 'v \Rightarrow 'v'$
assumes $\bigwedge v.\ prog.p2s\ (g\ v) \le \{\!|Q'\ (vf\ v)|\!\}, refinement.spec.bind.res\ (spec.pre\ P \sqcap spec.term.all\ A \sqcap prog.p2s\ f')\ A\ (vf\ v) \Vdash prog.p2s\ (g'\ (vf\ v)), \{\!|Q|\!\}$
assumes $prog.p2s\ f \le \{\!|P|\!\}, spec.term.all\ A \Vdash spec.vinvmap\ vf\ (prog.p2s\ f'), \{\!|\lambda v.\ Q'\ (vf\ v)|\!\}$
shows $prog.p2s\ (f \gg\!= g) \le \{\!|P|\!\}, A \Vdash prog.p2s\ (f' \gg\!= g'), \{\!|Q|\!\}$
⟨*proof*⟩

lemma *bind*:
assumes $\bigwedge v.\ prog.p2s\ (g\ v) \le \{\!|Q'\ v|\!\}, refinement.spec.bind.res\ (spec.pre\ P \sqcap spec.term.all\ A \sqcap prog.p2s\ f')\ A\ v \Vdash prog.p2s\ (g'\ v), \{\!|Q|\!\}$
assumes $prog.p2s\ f \le \{\!|P|\!\}, spec.term.all\ A \Vdash prog.p2s\ f', \{\!|Q'|\!\}$
shows $prog.p2s\ (f \gg\!= g) \le \{\!|P|\!\}, A \Vdash prog.p2s\ (f' \gg\!= g'), \{\!|Q|\!\}$
⟨*proof*⟩

lemmas *rev-bind* = *refinement.prog.bind*[*rotated*]

lemma *Parallel*:
  fixes A :: (*sequential*, *'s*, *unit*) *spec*
  fixes Q :: *'a* ⇒ *'s pred*
  fixes Ps :: *'a* ⇒ (*'s*, *unit*) *prog*
  fixes Ps' :: *'a* ⇒ (*'s*, *unit*) *prog*
  assumes $\bigwedge a.\ a \in as \Longrightarrow$ *prog.p2s* ($Ps\ a$) $\leq$ ⦃$P\ a$⦄, *refinement.spec.env-hyp* $P\ A\ as$ (*prog.p2s* ∘ Ps') $a \Vdash$ *prog.p2s* ($Ps'\ a$), ⦃$\lambda rv.\ Q\ a$⦄
  shows *prog.p2s* (*prog.Parallel as Ps*) $\leq$ ⦃$\bigsqcap a{\in}as.\ P\ a$⦄, $A \Vdash$ *prog.p2s* (*prog.Parallel as Ps'*), ⦃$\lambda rv.\ \bigsqcap a{\in}as.\ Q\ a$⦄
⟨*proof*⟩

lemma *parallel*:
  assumes *prog.p2s* $c_1 \leq$ ⦃P_1⦄, *refinement.spec.env-hyp* ($\lambda a.$ *if* a *then* P_1 *else* P_2) A *UNIV* ($\lambda a.$ *if* a *then prog.p2s* c_1' *else prog.p2s* c_2') *True* $\Vdash$ *prog.p2s* c_1', ⦃Q_1⦄
  assumes *prog.p2s* $c_2 \leq$ ⦃P_2⦄, *refinement.spec.env-hyp* ($\lambda a.$ *if* a *then* P_1 *else* P_2) A *UNIV* ($\lambda a.$ *if* a *then prog.p2s* c_1' *else prog.p2s* c_2') *False* $\Vdash$ *prog.p2s* c_2', ⦃Q_2⦄
  shows *prog.p2s* (*prog.parallel* $c_1\ c_2$) $\leq$ ⦃$P_1 \wedge P_2$⦄, $A \Vdash$ *prog.p2s* (*prog.parallel* $c_1'\ c_2'$), ⦃$\lambda v.\ Q_1\ v \wedge Q_2\ v$⦄
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

lemma *galois*:
  shows *prog.p2s* $c \leq$ ⦃P⦄, $A \vdash G$, ⦃Q⦄ $\longleftrightarrow c \leq$ *prog.s2p* (⦃P⦄, $A \vdash G$, ⦃Q⦄)
⟨*proof*⟩

lemmas *s2p-ag* = *iffD1*[*OF ag.prog.galois*]

lemma *p2s-s2p-ag*:
  shows *prog.p2s* (*prog.s2p* (⦃P⦄, $A \vdash G$, ⦃Q⦄)) $\leq$ ⦃P⦄, $A \vdash G$, ⦃Q⦄
⟨*proof*⟩

lemma *p2s-s2p-ag-stable*:
  assumes *stable* $A\ P$
  assumes $\bigwedge v.$ *stable* A ($Q\ v$)
  shows *prog.p2s* (*prog.s2p* (⦃P⦄, $A \vdash G$, ⦃Q⦄)) = ⦃P⦄, $A \vdash G$, ⦃Q⦄
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*[*iff*]:
  shows *prog.p2s* $\bot \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *sup-conv*:
  shows *prog.p2s* $(c_1 \sqcup c_2) \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\} \longleftrightarrow$ *prog.p2s* $c_1 \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\} \wedge$ *prog.p2s* $c_2 \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemmas *sup* = *iffD2*[*OF ag.prog.sup-conv*, *unfolded conj-explode*]

lemma *bind*: — Assumptions in weakest-pre order
  assumes $\bigwedge v.$ *prog.p2s* $(g\ v) \leq \{\!|Q'\ v|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
  assumes *prog.p2s* $f \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q'|\!\}$
  shows *prog.p2s* $(f \ggg g) \leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *action*: — Conclusion is insufficiently instantiated for use
  fixes $F :: ('v \times 's \times 's)$ *set*
  assumes Q: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow Q\ v\ s'$
  assumes G: $\bigwedge v\ s\ s'.\ [\![P\ s;\ s \neq s';\ (v,\ s,\ s') \in F]\!] \Longrightarrow (s,\ s') \in G$
  assumes *sP*: *stable A P*
  assumes *sQ*: $\bigwedge s\ s'\ v.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow$ *stable* $A\ (Q\ v)$
  shows *prog.p2s* (*prog.action* F) $\leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *guard*:
  assumes $\bigwedge s.\ [\![P\ s;\ g\ s]\!] \Longrightarrow Q\ ()\ s$
  assumes *stable A P*
  assumes *stable* $A\ (Q\ ())$
  shows *prog.p2s* (*prog.guard* g) $\leq \{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *Parallel*:
  assumes $\bigwedge a.\ a \in as \Longrightarrow$ *prog.p2s* $(Ps\ a) \leq \{\!|P\ a|\!\}$, $A \cup (\bigcup a' \in as - \{a\}.\ G\ a') \vdash G\ a$, $\{\!|\lambda v.\ Q\ a|\!\}$
  shows *prog.p2s* (*prog.Parallel as Ps*) $\leq \{\!|\bigsqcap a \in as.\ P\ a|\!\}$, $A \vdash \bigcup a \in as.\ G\ a$, $\{\!|\lambda v.\ \bigsqcap a \in as.\ Q\ a|\!\}$
⟨*proof*⟩

lemma *parallel*:
  assumes *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, $A \cup G_2 \vdash G_1$, $\{\!|Q_1|\!\}$
  assumes *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, $A \cup G_1 \vdash G_2$, $\{\!|Q_2|\!\}$
  shows *prog.p2s* (*prog.parallel* $c_1\ c_2$) $\leq \{\!|P_1 \wedge P_2|\!\}$, $A \vdash G_1 \cup G_2$, $\{\!|\lambda v.\ Q_1\ v \wedge Q_2\ v|\!\}$
⟨*proof*⟩

lemma *return*:
  assumes *sQ*: *stable* $A\ (Q\ v)$
  shows *prog.p2s* (*prog.return* v) $\leq \{\!|Q\ v|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *if*:
  assumes $b \Longrightarrow$ *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
  assumes $\neg b \Longrightarrow$ *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
  shows *prog.p2s* (*if b then* c_1 *else* c_2) $\leq \{\!|$*if b then* P_1 *else* $P_2|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *case-option*:

assumes $x = None \Longrightarrow prog.p2s\ none \leq \{\!|P_n|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Some\ v \Longrightarrow prog.p2s\ (some\ v) \leq \{\!|P_s\ v|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}option\ none\ some\ x) \leq \{\!|case\ x\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *case-sum*:
assumes $\bigwedge v.\ x = Inl\ v \Longrightarrow prog.p2s\ (left\ v) \leq \{\!|P_l\ v|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Inr\ v \Longrightarrow prog.p2s\ (right\ v) \leq \{\!|P_r\ v|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}sum\ left\ right\ x) \leq \{\!|case\text{-}sum\ P_l\ P_r\ x|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *case-list*:
assumes $x = [] \Longrightarrow prog.p2s\ nil \leq \{\!|P_n|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
assumes $\bigwedge v\ vs.\ x = v\ \#\ vs \Longrightarrow prog.p2s\ (cons\ v\ vs) \leq \{\!|P_c\ v\ vs|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}list\ nil\ cons\ x) \leq \{\!|case\text{-}list\ P_n\ P_c\ x|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*proof*⟩

end

13.6 Specification inhabitation

⟨*ML*⟩

lemma *Sup*:
assumes $prog.p2s\ P\ -s,\ xs\rightarrow\ P'$
assumes $P \in X$
shows $prog.p2s\ (\bigsqcup X)\ -s,\ xs\rightarrow\ P'$
⟨*proof*⟩

lemma *supL*:
assumes $prog.p2s\ P\ -s,\ xs\rightarrow\ P'$
shows $prog.p2s\ (P \sqcup Q)\ -s,\ xs\rightarrow\ P'$
⟨*proof*⟩

lemma *supR*:
assumes $prog.p2s\ Q\ -s,\ xs\rightarrow\ Q'$
shows $prog.p2s\ (P \sqcup Q)\ -s,\ xs\rightarrow\ Q'$
⟨*proof*⟩

lemma *bind*:
assumes $prog.p2s\ f\ -s,\ xs\rightarrow\ prog.p2s\ f'$
shows $prog.p2s\ (f \ggg g)\ -s,\ xs\rightarrow\ prog.p2s\ (f' \ggg g)$
⟨*proof*⟩

lemma *return*:
shows $prog.p2s\ (prog.return\ v)\ -s,\ []\rightarrow\ spec.return\ v$

⟨*proof*⟩

lemma *action-step*:
  fixes F :: $('v \times 's \times 's)$ *set*
  assumes $(v, s, s') \in F$
  shows *prog.p2s* (*prog.action* F) $-s$, $[(self, s')]\rightarrow$ *prog.p2s* (*prog.return* v)
⟨*proof*⟩

lemma *action-stutter*:
  fixes F :: $('v \times 's \times 's)$ *set*
  assumes $(v, s, s) \in F$
  shows *prog.p2s* (*prog.action* F) $-s$, $[]\rightarrow$ *prog.p2s* (*prog.return* v)
⟨*proof*⟩

lemma *parallelL*:
  assumes *prog.p2s* P $-s$, $xs\rightarrow$ *prog.p2s* P'
  shows *prog.p2s* $(P \parallel Q)$ $-s$, $xs\rightarrow$ *prog.p2s* $(P' \parallel Q)$
⟨*proof*⟩

lemma *parallelR*:
  assumes *prog.p2s* Q $-s$, $xs\rightarrow$ *prog.p2s* Q'
  shows *prog.p2s* $(P \parallel Q)$ $-s$, $xs\rightarrow$ *prog.p2s* $(P \parallel Q')$
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

definition *select* :: $'v$ *set* $\Rightarrow$ $('s, 'v)$ *prog* **where**
  select $X = (\bigsqcup x\in X.$ *prog.return* $x)$

context
  notes [[*function-internals*]]
begin

partial-function (*lfp*) *while* :: $('k \Rightarrow ('s, 'k + 'v)$ *prog*$) \Rightarrow 'k \Rightarrow ('s, 'v)$ *prog* **where**
  while c k = c k $\ggg$ ($\lambda rv.$ *case* rv *of* *Inl* $k' \Rightarrow$ *while* c k' | *Inr* $v \Rightarrow$ *prog.return* v)

end

abbreviation *loop* :: $('s,$ *unit*$)$ *prog* $\Rightarrow ('s, 'w)$ *prog* **where**
  loop $P \equiv$ *prog.while* ($\lambda()$. $P \gg$ *prog.return* (*Inl* ())) ()

abbreviation *guardM* :: *bool* $\Rightarrow ('s,$ *unit*$)$ *prog* **where**
  guardM $b \equiv$ *if* b *then* $\bot$ *else* *prog.return* ()

abbreviation *unlessM* :: *bool* $\Rightarrow ('s,$ *unit*$)$ *prog* $\Rightarrow ('s,$ *unit*$)$ *prog* **where**
  unlessM b $c \equiv$ *if* b *then* *prog.return* () *else* c

abbreviation *whenM* :: *bool* $\Rightarrow ('s,$ *unit*$)$ *prog* $\Rightarrow ('s,$ *unit*$)$ *prog* **where**

whenM b c ≡ *if b then c else prog.return* ()

definition *app* :: ($'a$ ⇒ ($'s$, *unit*) *prog*) ⇒ $'a$ *list* ⇒ ($'s$, *unit*) *prog* **where** — Haskell's *mapM-*
app f xs = *foldr* (λ*x m*. *f x* ≫ *m*) *xs* (*prog.return* ())

definition *set-app* :: ($'a$ ⇒ ($'s$, *unit*) *prog*) ⇒ $'a$ *set* ⇒ ($'s$, *unit*) *prog* **where**
set-app f =
prog.while (λ*X*. *if X* = {} *then prog.return* (*Inr* ())
else prog.select X ⋙= (λ*x*. *f x* ≫ *prog.return* (*Inl* (*X* − {*x*}))))

primrec *foldM* :: ($'b$ ⇒ $'a$ ⇒ ($'s$, $'b$) *prog*) ⇒ $'b$ ⇒ $'a$ *list* ⇒ ($'s$, $'b$) *prog* **where**
foldM f b [] = *prog.return b*
| *foldM f b* (*x* # *xs*) = *do* {
b' ← *f b x*;
foldM f b' *xs*
}

primrec *fold-mapM* :: ($'a$ ⇒ ($'s$, $'b$) *prog*) ⇒ $'a$ *list* ⇒ ($'s$, $'b$ *list*) *prog* **where**
fold-mapM f [] = *prog.return* []
| *fold-mapM f* (*x* # *xs*) = *do* {
y ← *f x*;
ys ← *fold-mapM f xs*;
prog.return (*y* # *ys*)
}

⟨*ML*⟩

lemma *empty*:
shows *prog.select* {} = ⊥
⟨*proof*⟩

lemma *singleton*:
shows *prog.select* {*x*} = *prog.return x*
⟨*proof*⟩

lemma *monotone*:
shows *mono prog.select*
⟨*proof*⟩

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* (⋃ *X*) = (⨆ *x*∈*X*. *prog.select x*)
⟨*proof*⟩

lemma *mcont*:
shows *mcont* ⋃ (⊆) *Sup* (≤) *prog.select*
⟨*proof*⟩

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.select.mcont*, *of supa orda P* **for** *supa orda P*]

⟨*ML*⟩

lemma *select-le*:
assumes *x* ∈ *X*

shows *prog.return x* $\leq$ *prog.select X*
⟨*proof*⟩

⟨*ML*⟩

lemma *selectL*:
shows *prog.select X* $\gg=$ *g* = ($\bigsqcup x{\in}X.$ *g x*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
shows *prog.while* $\bot = \bot$
⟨*proof*⟩

lemma *monotone*: — could hope to prove this with a *strengthen* rule for *lfp.fixp-fun*
shows *mono* ($\lambda P.$ *prog.while P s*)
⟨*proof*⟩

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 ($\bigsqcup P{\in}X.$ *prog.while P s*) $\leq$ *prog.while* ($\bigsqcup X$) *s*
⟨*proof*⟩

lemma *Inf-le*:
shows *prog.while* ($\bigsqcap X$) *s* $\leq$ ($\bigsqcap P{\in}X.$ *prog.while P s*)
⟨*proof*⟩

lemma *True-skip-eq-bot*:
shows *prog.while* ⟨*prog.return* (*Inl x*)⟩ *s* = $\bot$
⟨*proof*⟩

lemma *Inr-eq-return*:
shows *prog.while* ⟨*prog.return* (*Inr v*)⟩ *s* = *prog.return v*
⟨*proof*⟩

lemma *kleene-star*:
shows *prog.kleene.star P*
= *prog.while* (λ-. (*P* $\gg$ *prog.return* (*Inl* ())) $\sqcup$ *prog.return* (*Inr* ())) () (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *invmap-le*:
fixes *sf* :: *'s* $\Rightarrow$ *'t*
fixes *vf* :: *'v* $\Rightarrow$ *'w*
shows *prog.while* ($\lambda k.$ *prog.invmap sf* (*map-sum id vf*) (*c k*)) *k*
$\leq$ *prog.invmap sf vf* (*prog.while c k*) (**is** *?lhs prog.while k* $\leq$ *?rhs k*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bindL*:
fixes *P* :: (*'s*, *unit*) *prog*
fixes *Q* :: (*'s*, *'w*) *prog*
shows *prog.loop P* $\gg$ *Q* = *prog.loop P* (**is** *?lhs* = *?rhs*)

⟨proof⟩

lemma *parallel-le*:
  shows $prog.loop\ P \le lfp\ (\lambda R.\ P \parallel R)$
⟨proof⟩

⟨ML⟩

lemma *append*:
  shows $prog.foldM\ f\ b\ (xs\ @\ ys) = prog.foldM\ f\ b\ xs \ggg= (\lambda b'.\ prog.foldM\ f\ b'\ ys)$
⟨proof⟩

⟨ML⟩

lemma *foldM-alt-def*:
  shows $prog.foldM\ f\ b\ xs = foldr\ (\lambda x\ m.\ prog.bind\ m\ (\lambda b.\ f\ b\ x))\ (rev\ xs)\ (prog.return\ b)$
⟨proof⟩

⟨ML⟩

lemma *bot*:
  shows $prog.fold\text{-}mapM\ \bot = (\lambda xs.\ case\ xs\ of\ [] \Rightarrow prog.return\ []\ |\ \text{-} \Rightarrow \bot)$
⟨proof⟩

lemma *append*:
  shows $prog.fold\text{-}mapM\ f\ (xs\ @\ ys)$
    $= prog.fold\text{-}mapM\ f\ xs \ggg= (\lambda xs.\ prog.fold\text{-}mapM\ f\ ys \ggg= (\lambda ys.\ prog.return\ (xs\ @\ ys)))$
⟨proof⟩

⟨ML⟩

lemma *bot*:
  shows $prog.app\ \bot = (\lambda xs.\ case\ xs\ of\ [] \Rightarrow prog.return\ ()\ |\ \text{-} \Rightarrow \bot)$
   and $prog.app\ (\lambda\text{-}.\ \bot) = (\lambda xs.\ case\ xs\ of\ [] \Rightarrow prog.return\ ()\ |\ \text{-} \Rightarrow \bot)$
⟨proof⟩

lemma *Nil*:
  shows $prog.app\ f\ [] = prog.return\ ()$
⟨proof⟩

lemma *Cons*:
  shows $prog.app\ f\ (x\ \#\ xs) = f\ x \gg prog.app\ f\ xs$
⟨proof⟩

lemmas *simps* =
  prog.app.bot
  prog.app.Nil
  prog.app.Cons

lemma *append*:
  shows $prog.app\ f\ (xs\ @\ ys) = prog.app\ f\ xs \gg prog.app\ f\ ys$
⟨proof⟩

lemma *monotone*:
  shows $mono\ (\lambda f.\ prog.app\ f\ xs)$
⟨proof⟩

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 $(\bigsqcup f\in X.\ prog.app\ f\ xs) \leq prog.app\ (\bigsqcup X)\ xs$
⟨*proof*⟩

⟨*ML*⟩

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* ())
⟨*proof*⟩

⟨*ML*⟩

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*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
  shows $X \neq \{\} \Longrightarrow$ *prog.set-app* $\bot$ $X = \bot$
    and $X \neq \{\} \Longrightarrow$ *prog.set-app* (λ-. $\bot$) $X = \bot$
⟨*proof*⟩

lemma *empty*:
  shows *prog.set-app f* {} = *prog.return* ()
⟨*proof*⟩

lemma *not-empty*:
  assumes $X \neq \{\}$
  shows *prog.set-app f X* = *prog.select X* $\gg=$ ($\lambda x.$ *f x* $\gg$ *prog.set-app f* ($X - \{x\}$))
⟨*proof*⟩

lemmas *simps* =
  prog.set-app.bot
  prog.set-app.empty
  prog.set-app.not-empty

⟨*ML*⟩

lemma *set-app-le*:
  assumes *X* = *set xs*
  assumes *distinct xs*
  shows *prog.app f xs* $\leq$ *prog.set-app f X*
⟨*proof*⟩

⟨*ML*⟩

lemma *set-app-alt-def*:
  assumes *finite X*

shows *prog.set-app f X* = ($\bigsqcup$*xs*∈{*ys. set ys* = *X* ∧ *distinct ys*}. *prog.app f xs*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *select-sp*:
assumes $\bigwedge s\ x.\ [\![P\ s;\ x \in X]\!] \Longrightarrow Q\ x\ s$
assumes $\bigwedge v.\ stable\ A\ (P \wedge Q\ v)$
shows $prog.p2s\ (prog.select\ X) \le \{\!|P|\!\},\ A \vdash G,\ \{\!|\lambda v.\ P \wedge Q\ v|\!\}$
⟨*proof*⟩

lemma *while*:
fixes $c :: 'k \Rightarrow ('s,\ 'k + 'v)\ prog$
assumes *c*: $\bigwedge k.\ prog.p2s\ (c\ k) \le \{\!|P\ k|\!\},\ A \vdash G,\ \{\!|case\text{-}sum\ I\ Q|\!\}$
assumes *IP*: $\bigwedge s\ v.\ I\ v\ s \Longrightarrow P\ v\ s$
assumes *sQ*: $\bigwedge v.\ stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.while\ c\ k) \le \{\!|I\ k|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *app*:
fixes $xs :: 'a\ list$
fixes $f :: 'a \Rightarrow ('s,\ unit)\ prog$
fixes $P :: 'a\ list \Rightarrow 's\ pred$
assumes $\bigwedge x\ ys\ zs.\ xs = ys\ @\ x\ \#\ zs \Longrightarrow prog.p2s\ (f\ x) \le \{\!|P\ ys|\!\},\ A \vdash G,\ \{\!|\lambda\text{-}.\ P\ (ys\ @\ [x])|\!\}$
assumes $\bigwedge ys.\ prefix\ ys\ xs \Longrightarrow stable\ A\ (P\ ys)$
shows $prog.p2s\ (prog.app\ f\ xs) \le \{\!|P\ []|\!\},\ A \vdash G,\ \{\!|\lambda\text{-}.\ P\ xs|\!\}$
⟨*proof*⟩

lemma *app-set*:
fixes $X :: 'a\ set$
fixes $f :: 'a \Rightarrow ('s,\ unit)\ prog$
fixes $P :: 'a\ set \Rightarrow 's\ pred$
assumes $\bigwedge Y\ x.\ [\![Y \subseteq X;\ x \in X - Y]\!] \Longrightarrow prog.p2s\ (f\ x) \le \{\!|P\ Y|\!\},\ A \vdash G,\ \{\!|\lambda\text{-}.\ P\ (insert\ x\ Y)|\!\}$
assumes $\bigwedge Y.\ Y \subseteq X \Longrightarrow Stability.stable\ A\ (P\ Y)$
shows $prog.p2s\ (prog.set\text{-}app\ f\ X) \le \{\!|P\ \{\}|\!\},\ A \vdash G,\ \{\!|\lambda\text{-}.\ P\ X|\!\}$
⟨*proof*⟩

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*: $\bigwedge b\ x.\ x \in set\ xs \Longrightarrow prog.p2s\ (f\ b\ x) \le \{\!|I\ b\ x|\!\},\ A \vdash G,\ \{\!|P|\!\}$
assumes *P*: $\bigwedge b\ x\ s.\ [\![P\ b\ s;\ x \in set\ xs]\!] \Longrightarrow I\ b\ x\ s$
assumes *sP*: $\bigwedge b.\ stable\ A\ (P\ b)$
shows $prog.p2s\ (prog.foldM\ f\ b\ xs) \le \{\!|P\ b|\!\},\ A \vdash G,\ \{\!|P|\!\}$
⟨*proof*⟩

⟨*ML*⟩
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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 agent*, *'ls* × *'s*, *'v*) *spec*. Applying *spec.smap snd* yields a process of type (*'a 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* (*snd s*)

⟨*ML*⟩

definition *qrm* :: (*'a agent*, *'ls* $\times$ *'s*) *steps* **where** — cf *ag.assm*
qrm = *range proc* $\times$ *UNIV* $\cup$ {*env*} $\times$ (*Id* $\times_R$ *UNIV*)

abbreviation (*input*) *interference* $\equiv$ *spec.rel spec.local.qrm*

⟨*ML*⟩

definition *local* :: (*'a agent*, *'ls* $\times$ *'s*, *'v*) *spec* $\Rightarrow$ (*'a agent*, *'s*, *'v*) *spec* **where**
local P = *spec.smap snd* (*spec.local.interference* $\sqcap$ *P*)

⟨*ML*⟩

lemma *local-le-conv*:
shows ⦇σ⦈ $\leq$ *spec.local P*
$\longleftrightarrow$ ($\exists\sigma'$. ⦇σ'⦈ $\leq$ *P*
$\wedge$ *trace.steps* σ' $\subseteq$ *spec.local.qrm*
$\wedge$ ⦇σ⦈ $\leq$ ⦇*trace.map id snd id* σ'⦈)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-le*[*spec.idle-le*]: — Converse does not hold
assumes *spec.idle* $\leq$ *P*
shows *spec.idle* $\leq$ *spec.local P*
⟨*proof*⟩

⟨*ML*⟩

lemma *refl*:
shows *refl* (*spec.local.qrm* ‘‘ {*a*})
⟨*proof*⟩

lemma *member*:
shows (*proc a*, *s*, *s'*) $\in$ *spec.local.qrm*
and (*env*, *s*, *s'*) $\in$ *spec.local.qrm* $\longleftrightarrow$ *fst s* = *fst s'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemmas *simps*[*simp*] =
spec.local.qrm.refl
spec.local.qrm.member
spec.local.qrm.inter

⟨*ML*⟩

lemma *smap-snd*:
  shows *spec.smap snd spec.local.interference* = ⊤
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-interference*:
  shows *spec.local P* = *spec.local* (*P* ⊓ *spec.local.interference*)
⟨*proof*⟩

lemma *bot*:
  shows *spec.local* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
  shows *spec.local* ⊤ = ⊤
⟨*proof*⟩

lemma *monotone*:
  shows *mono spec.local*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.local.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.local.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
  = *monotone2monotone*[*OF spec.local.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup*:
  shows *spec.local* (⨆*X*) = (⨆*x*∈*X*. *spec.local x*)
⟨*proof*⟩

lemmas *sup* = *spec.local.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *spec.local* (*P x*))
⟨*proof*⟩

lemma *idle*:
  shows *spec.local spec.idle* = *spec.idle*
⟨*proof*⟩

lemma *action*:
  fixes *F* :: (′*v* × ′*a agent* × (′*ls* × ′*s*) × (′*ls* × ′*s*)) *set*
  shows *spec.local* (*spec.action F*)
        = *spec.action* (*map-prod id* (*map-prod id* (*map-prod snd snd*)) ‘
                                        (*F* ∩ *UNIV* × *spec.local.qrm*))
⟨*proof*⟩

lemma *return*:
  shows *spec.local* (*spec.return v*) = *spec.return v*
⟨*proof*⟩

lemma *bind-le*: — Converse does not hold
  shows *spec.local* (*f* ⋙= *g*) ≤ *spec.local f* ⋙= (λ*v*. *spec.local* (*g v*))
⟨*proof*⟩

lemma *interference*:

shows *spec.local* (*spec.rel* ({*env*} × *UNIV*)) = *spec.rel* ({*env*} × *UNIV*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-le*:
shows *spec.map id sf vf* (*spec.local P*) ≤ *spec.local* (*spec.map id* (*map-prod id sf*) *vf P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
shows *spec.vmap vf* (*spec.local P*) = *spec.local* (*spec.vmap vf P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *smap-snd*:
fixes *P* :: ('*a*, '*ls* × '*t*, '*w*) *spec*
fixes *sf* :: '*s* ⇒ '*t*
fixes *vf* :: '*v* ⇒ '*w*
shows *spec.invmap id sf vf* (*spec.smap snd P*)
= *spec.smap snd* (*spec.invmap id* (*map-prod id sf*) *vf P*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *local*:
fixes *P* :: ('*a agent*, '*ls* × '*t*, '*v*) *spec*
fixes *sf* :: '*s* ⇒ '*t*
shows *spec.invmap id sf vf* (*spec.local P*) = *spec.local* (*spec.invmap id* (*map-prod id sf*) *vf P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
shows *spec.term.none* (*spec.local P*) = *spec.local* (*spec.term.none P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
shows *spec.term.all* (*spec.local P*) = *spec.local* (*spec.term.all P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
assumes *P* ∈ *spec.term.closed* -
shows *spec.local P* ∈ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

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 $cl \equiv spec.map\text{-}invmap.cl$ - - - id snd id
  assumes $spec.local.interference \sqcap P$
      $\leq cl$ $(spec.local.interference \sqcap Q)$
  shows $spec.local$ $P \leq spec.local$ Q
⟨*proof*⟩

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:

⟨*ML*⟩

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 σ = ($\natural$($trace.map$ id snd id σ),
        fst ($trace.init$ σ),
        if T $then$ $map\text{-}option$ ⟨fst ($trace.final$ σ)⟩ ($trace.term$ σ) $else$ $None$)

⟨*ML*⟩

lemma *natural*:
  shows $seq\text{-}ctxt.prj$ T $(\natural\sigma) = seq\text{-}ctxt.prj$ T σ
⟨*proof*⟩

lemma *idle*:
  shows $seq\text{-}ctxt.prj$ T $(trace.T$ s $[]$ $None) = (trace.T$ $(snd$ $s)$ $[]$ $None, fst$ $s, None)$
⟨*proof*⟩

lemmas *simps*[*simp*] =
  seq-ctxt.prj.natural

⟨*ML*⟩

interpretation *seq-ctxt*: *galois.image-vimage seq-ctxt.prj T* **for** *T* ⟨*proof*⟩

⟨*ML*⟩

lemma *partial-sel-equivE*:
  assumes $seq\text{-}ctxt.equivalent$ T σ_1 σ_2
  obtains $trace.init$ $\sigma_1 = trace.init$ σ_2
    and $trace.term$ $\sigma_1 = trace.term$ σ_2
    and $[\![T; \exists v.$ $trace.term$ $\sigma_1 = Some$ $v]\!] \Longrightarrow trace.final$ $\sigma_1 = trace.final$ σ_2
⟨*proof*⟩

lemma *downwards-existsE*:
  assumes $\sigma_1{}' \leq \sigma_1$
  assumes $seq\text{-}ctxt.equivalent$ T σ_1 σ_2
  obtains $\sigma_2{}'$
   where $\sigma_2{}' \leq \sigma_2$
    and $seq\text{-}ctxt.equivalent$ T $\sigma_1{}'$ $\sigma_2{}'$
⟨*proof*⟩

lemma *downwards-existsE2*:
  assumes $\sigma_1{}' \leq \sigma_1$
  assumes $seq\text{-}ctxt.equivalent$ T $\sigma_1{}'$ $\sigma_2{}'$
  obtains σ_2
   where $\sigma_2{}' \leq \sigma_2$

and *seq-ctxt.equivalent* $T\ \sigma_1\ \sigma_2$
⟨*proof*⟩

lemma *map-sf-eq-id*:
assumes *seq-ctxt.equivalent True* $\sigma_1\ \sigma_2$
shows *seq-ctxt.equivalent True* (*trace.map af id vf* σ_1) (*trace.map af id vf* σ_2)
⟨*proof*⟩

lemma *mono*:
assumes $T \Longrightarrow T'$
assumes *seq-ctxt.equivalent* $T'\ \sigma_1\ \sigma_2$
shows *seq-ctxt.equivalent* $T\ \sigma_1\ \sigma_2$
⟨*proof*⟩

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′*)
⟨*proof*⟩

⟨*ML*⟩

definition *cl* :: *bool* ⇒ (′*a*, ′*ls* × ′*s*, ′*v*) *spec* ⇒ (′*a*, ′*ls* × ′*s*, ′*v*) *spec* **where**
cl T P = $\bigsqcup$(*spec.singleton* ‘ $\{\sigma_1.\ \exists\sigma_2.\ \langle\!|\sigma_2|\!\rangle \le P \wedge$ *seq-ctxt.equivalent* $T\ \sigma_1\ \sigma_2\}$)

⟨*ML*⟩

lemma *cl-le-conv*[*spec.singleton.le-conv*]:
shows $\langle\!|\sigma|\!\rangle \le$ *spec.seq-ctxt.cl T P* $\longleftrightarrow$ $(\exists\sigma'.\ \langle\!|\sigma'|\!\rangle \le P \wedge$ *seq-ctxt.equivalent* $T\ \sigma\ \sigma')$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

interpretation *seq-ctxt*: *closure-complete-distrib-lattice-distributive-class spec.seq-ctxt.cl T* **for** *F*
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.seq-ctxt.cl T P* ⟷ *spec.idle* ≤ *P* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*[*simp*]:
shows *spec.seq-ctxt.cl T* ⊥ = ⊥
⟨*proof*⟩

lemma *mono*:
assumes $T' \Longrightarrow T$
assumes $P \le P'$
shows *spec.seq-ctxt.cl T P* ≤ *spec.seq-ctxt.cl T′ P′*
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* (¬ *F*) *T T′*
assumes *st-ord F P P′*
shows *st-ord F* (*spec.seq-ctxt.cl T P*) (*spec.seq-ctxt.cl T′ P′*)

⟨proof⟩

lemma *Sup*:
  shows *spec.seq-ctxt.cl T* ($\bigsqcup$*X*) = $\bigsqcup$(*spec.seq-ctxt.cl T* ' *X*)
⟨proof⟩

lemmas *sup* = *spec.seq-ctxt.cl.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *singleton*:
  shows *spec.seq-ctxt.cl T* ⟨σ⟩ = $\bigsqcup$(*spec.singleton* ' {σ'. *seq-ctxt.equivalent T* σ σ'}) (**is** *?lhs* = *?rhs*)
⟨proof⟩

lemma *idle*: — not *simp* friendly
  shows *spec.seq-ctxt.cl T* (*spec.idle* :: ('*a*, '*ls* × '*s*, '*v*) *spec*)
      = *spec.term.none* (*spec.rel* (*UNIV* × (*UNIV* $\times_R$ *Id*)) :: ('*a*, '*ls* × '*s*, '*w*) *spec*) (**is** *?lhs* = *?rhs*)
⟨proof⟩

lemma *invmap-le*:
  shows *spec.seq-ctxt.cl True* (*spec.invmap af id vf P*) ≤ *spec.invmap af id vf* (*spec.seq-ctxt.cl True P*)
⟨proof⟩

lemma *map-le*:
  shows *spec.map af id vf* (*spec.seq-ctxt.cl True P*) ≤ *spec.seq-ctxt.cl True* (*spec.map af id vf P*)
⟨proof⟩

⟨ML⟩

lemma *cl*:
  shows *spec.term.none* (*spec.seq-ctxt.cl T P*) = *spec.seq-ctxt.cl T* (*spec.term.none P*)
⟨proof⟩

lemma *cl-True-False*:
  shows *spec.seq-ctxt.cl True* (*spec.term.none f*) = *spec.seq-ctxt.cl False* (*spec.term.none f*)
⟨proof⟩

⟨ML⟩

lemma *cl-le*:
  shows *spec.seq-ctxt.cl T* (*spec.term.all P*) ≤ *spec.term.all* (*spec.seq-ctxt.cl T P*)
⟨proof⟩

lemma *cl-False*:
  shows *spec.seq-ctxt.cl False* (*spec.term.all P*) = *spec.term.all* (*spec.seq-ctxt.cl False P*)
⟨proof⟩

⟨ML⟩

lemma *cl-le*:
  shows *spec.seq-ctxt.cl True f* ⤜ (λ*v*. *spec.seq-ctxt.cl T* (*g v*)) ≤ *spec.seq-ctxt.cl T* (*f* ⤜ *g*)
⟨proof⟩

lemma *clL-le*:
  shows *spec.seq-ctxt.cl True f* ⤜ *g* ≤ *spec.seq-ctxt.cl T* (*f* ⤜ *g*)
⟨proof⟩

lemma *clR-le*:
  shows *f* ⤜ (λ*v*. *spec.seq-ctxt.cl T* (*g v*)) ≤ *spec.seq-ctxt.cl T* (*f* ⤜ *g*)
⟨proof⟩

⟨*ML*⟩

lemma *cl-local-le*: — the RHS is the closure that arises from *spec.local*, ignoring the constraint
shows $spec.seq\text{-}ctxt.cl\ T\ P \leq spec.map\text{-}invmap.cl\ \text{-}\ \text{-}\ \text{-}\ id\ snd\ id\ P$
⟨*proof*⟩

lemma *cl-local*:
shows $spec.local\ (spec.seq\text{-}ctxt.cl\ T\ (spec.local.interference \sqcap P))$
$= spec.local\ P$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cl-imp-local-le*:
assumes $spec.local.interference \sqcap P$
$\leq spec.seq\text{-}ctxt.cl\ False\ (spec.local.interference \sqcap Q)$
shows $spec.local\ P \leq spec.local\ Q$
⟨*proof*⟩

lemma *cl-inf-pre*:
shows $spec.pre\ P \sqcap spec.seq\text{-}ctxt.cl\ T\ c = spec.seq\text{-}ctxt.cl\ T\ (spec.pre\ P \sqcap c)$
⟨*proof*⟩

lemma *cl-pre-le-conv*:
shows $spec.seq\text{-}ctxt.cl\ T\ c \leq spec.pre\ P \longleftrightarrow c \leq spec.pre\ P$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *cl-inf-post*:
shows $spec.post\ Q \sqcap spec.seq\text{-}ctxt.cl\ True\ c = spec.seq\text{-}ctxt.cl\ True\ (spec.post\ Q \sqcap c)$
⟨*proof*⟩

lemma *cl-post-le-conv*:
shows $spec.seq\text{-}ctxt.cl\ True\ c \leq spec.post\ Q \longleftrightarrow c \leq spec.post\ Q$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

lemma *cl-action-permuteL-le*:
fixes $F\ :: ('v \times 'a \times ('ls \times 's) \times ('ls \times 's))\ set$
fixes $G\ :: 'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))\ set$
fixes $G' :: ('v' \times 'a \times ('ls \times 's) \times ('ls \times 's))\ set$
fixes $F' :: 'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))\ set$
— F does not change $'s$, can be partial
assumes F: $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!] \Longrightarrow snd\ s' = snd\ s$
— The final state and return value are independent of the order of actions. F' does not change $'s$, cannot be partial
assumes $FGG'F'$: $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a',\ s,\ t) \in F;\ (w,\ a,\ t,\ s') \in G\ v]\!]$
$\Longrightarrow \exists\, v'\ a''\ a'''\ s''\ t'.\ (v',\ a'',\ s,\ t') \in G' \wedge (w,\ a''',\ t',\ s'') \in F'\ v'$
$\wedge\ snd\ s' = snd\ t' \wedge (snd\ s \neq snd\ t' \longrightarrow a'' = a) \wedge (T \longrightarrow fst\ s'' = fst\ s') \wedge snd\ s'' = snd\ t'$
shows $(spec.action\ F \gg\!= (\lambda v.\ spec.action\ (G\ v))) \sqcap spec.pre\ P$
$\leq spec.seq\text{-}ctxt.cl\ T\ (spec.action\ G' \gg\!= (\lambda v.\ spec.action\ (F'\ v)))$ (**is** - ≤ *?rhs*)
⟨*proof*⟩

Secondly: an initial action G that does change the observable state can be swapped with an arbitrary action action F that does not observably change the state.

lemma *cl-action-permuteR-le*:
  fixes G :: $('v \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
  fixes F :: $'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
  fixes F' :: $('v' \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
  fixes G' :: $'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
    — F does not stall if G makes an observable state change
  assumes G: $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G;\ snd\ s' \neq snd\ s]\!]$
  $\Longrightarrow \exists\, v'\ w\ a''\ t\ s''.\ (v',\ a'',\ s,\ t) \in F' \wedge (w,\ a,\ t,\ s'') \in G'\ v' \wedge snd\ t = snd\ s \wedge snd\ s'' = snd\ s'$
    — The final state and return value are independent of the order of actions
  assumes $GFF'G'$: $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a,\ s,\ t) \in G;\ (w,\ a',\ t,\ s') \in F\ v]\!]$
  $\Longrightarrow snd\ s' = snd\ t \wedge (\exists\, v'\ a''\ a'''\ s''\ t'.\ (v',\ a'',\ s,\ t') \in F' \wedge (w,\ a''',\ t',\ s'') \in G'\ v'$
  $\wedge\ snd\ t' = snd\ s \wedge (T \longrightarrow fst\ s'' = fst\ s') \wedge snd\ s'' = snd\ s' \wedge (snd\ s'' \neq snd\ t' \longrightarrow a''' = a))$
  shows $(spec.action\ G \gg\!= (\lambda v.\ spec.action\ (F\ v))) \sqcap spec.pre\ P$
    $\leq spec.seq\text{-}ctxt.cl\ T\ (spec.action\ F' \gg\!= (\lambda v.\ spec.action\ (G'\ v)))$
$\langle proof\rangle$

lemma *cl-action-bind-action-pre-post*:
  fixes F' :: $('v \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
  fixes G' :: $'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
  fixes Q :: $'w \Rightarrow ('ls \times 's)$ *pred*
  assumes $\bigwedge v\ w\ a\ a'\ s\ s'\ s''.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (w,\ a',\ s',\ s'') \in G\ v]\!] \Longrightarrow Q\ w\ s''$
  shows $spec.pre\ P \sqcap spec.seq\text{-}ctxt.cl\ True\ (spec.action\ F \gg\!= (\lambda v.\ spec.action\ (G\ v))) \leq spec.post\ Q$
$\langle proof\rangle$

lemma *cl-rev-kleene-star-action-permute-le*:
  fixes $F\ G$ :: $(unit \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
    — F does not stall if G changes the observable state
  assumes G: $\bigwedge a\ s\ s'.\ [\![((),\ a,\ s,\ s') \in G;\ snd\ s' \neq snd\ s]\!]$
  $\Longrightarrow \exists\, w\ a''\ t\ s''.\ ((),\ a'',\ s,\ t) \in F \wedge ((),\ a,\ t,\ s'') \in G \wedge snd\ t = snd\ s \wedge snd\ s'' = snd\ s'$
    — The final state is independent of order of actions, F does not change $'s$, can be partial
  assumes $GFFG$: $\bigwedge a\ a'\ s\ s'\ t.\ [\![((),\ a,\ s,\ t) \in G;\ ((),\ a',\ t,\ s') \in F]\!]$
  $\Longrightarrow snd\ s' = snd\ t \wedge (\exists\, a''\ a'''\ t'.\ ((),\ a'',\ s,\ t') \in F \wedge ((),\ a''',\ t',\ s') \in G$
  $\wedge\ snd\ t' = snd\ s \wedge (snd\ s' \neq snd\ t' \longrightarrow a''' = a))$
  shows $spec.kleene.rev\text{-}star\ (spec.action\ G) \gg\!= (\lambda\text{-}::unit.\ spec.action\ F)$
    $\leq spec.seq\text{-}ctxt.cl\ True\ (spec.action\ F \gg spec.kleene.rev\text{-}star\ (spec.action\ G))$ (**is** *?lhs spec.kleene.rev-star* $\leq$ *?rhs*)
$\langle proof\rangle$

lemma *cl-local-action-interference-permute-le*: — local actions permute with interference
  fixes F :: $(unit \times 'a\ agent \times ('ls \times 's) \times ('ls \times 's))$ *set*
  fixes r :: $'s\ rel$
    — F does not block
  assumes $\bigwedge s\ ls.\ \exists\, v\ a\ ls'.\ (v,\ a,\ (ls,\ s),\ (ls',\ s)) \in F$
    — F is insensitive to and does not modify the shared state
  assumes $\bigwedge v\ a\ s\ s'\ s''\ ls\ ls'.\ (v,\ a,\ (ls,\ s),\ (ls',\ s')) \in F$
  $\Longrightarrow s' = s \wedge (v,\ a,\ (ls,\ s''),\ (ls',\ s'')) \in F$
  shows $spec.rel\ (A \times (Id \times_R r)) \gg\!= (\lambda\text{-}::unit.\ spec.action\ F)$
    $\leq spec.seq\text{-}ctxt.cl\ True\ (spec.action\ F \gg spec.rel\ (A \times (Id \times_R r)))$
$\langle proof\rangle$

lemma *cl-action-mumble-trailing-le*:
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!]$
  $\Longrightarrow \exists\, a'\ ls'.\ (v,\ a',\ s,\ (ls',\ snd\ s')) \in F'$
  $\wedge\ (snd\ s' \neq snd\ s \longrightarrow a' = a) \wedge (T \longrightarrow ls' = fst\ s')$

shows *spec.action F* ⊓ *spec.pre P* ≤ *spec.seq-ctxt.cl T* (*spec.action F'*)
⟨*proof*⟩

lemma *cl-action-mumbleL-le*:
assumes $\bigwedge w\ a\ s\ s'.\ [\![P\ s;\ (w,\ a,\ s,\ s') \in G]\!]$
$\Longrightarrow \exists\, v\ a'\ a''\ t\ s''.\ (v,\ a',\ s,\ t) \in F' \wedge (w,\ a'',\ t,\ s'') \in G'\ v$
$\wedge\ snd\ t = snd\ s \wedge (T \longrightarrow fst\ s'' = fst\ s')$
$\wedge\ snd\ s'' = snd\ s' \wedge (snd\ s'' \neq snd\ t \longrightarrow a'' = a)$
shows *spec.action G* ⊓ *spec.pre P* ≤ *spec.seq-ctxt.cl T* (*spec.action F'* ⫺ (λ*v. spec.action* (*G' v*)))
⟨*proof*⟩

lemma *cl-action-mumbleR-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G]\!]$
$\Longrightarrow \exists\, w\ a'\ a''\ t.\ (w,\ a',\ s,\ t) \in G' \wedge (v,\ a'',\ t,\ s') \in F'\ w$
$\wedge\ snd\ t = snd\ s' \wedge (snd\ t \neq snd\ s \longrightarrow a' = a)$
shows *spec.action G* ⊓ *spec.pre P* ≤ *spec.seq-ctxt.cl T* (*spec.action G'* ⫺ (λ*v. spec.action* (*F' v*)))
⟨*proof*⟩

lemma *cl-action-mumble-expandL-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!] \Longrightarrow snd\ s' = snd\ s$
assumes $\bigwedge v\ w\ a\ a'\ s\ s'\ s''.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (w,\ a',\ s',\ s'') \in G\ v]\!]$
$\Longrightarrow \exists\, s'''.\ (w,\ a',\ s,\ s''') \in G' \wedge snd\ s''' = snd\ s'' \wedge (T \longrightarrow fst\ s''' = fst\ s'')$
shows (*spec.action F* ⫺ (λ*v. spec.action* (*G v*))) ⊓ *spec.pre P* ≤ *spec.seq-ctxt.cl T* (*spec.action G'*)
⟨*proof*⟩

lemma *cl-action-mumble-expandR-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G;\ snd\ s' \neq snd\ s]\!] \Longrightarrow \exists\, v'\ s''.\ (v',\ a,\ s,\ s'') \in G' \wedge snd\ s'' = snd\ s'$
assumes $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a,\ s,\ t) \in G;\ (w,\ a',\ t,\ s') \in F\ v]\!]$
$\Longrightarrow snd\ s' = snd\ t \wedge (\exists\, a''\ s''.\ (w,\ a'',\ s,\ s'') \in G' \wedge snd\ s'' = snd\ s' \wedge (T \longrightarrow fst\ s'' = fst\ s') \wedge$ $(snd\ s'' \neq snd\ s \longrightarrow a'' = a))$
shows (*spec.action G* ⫺ (λ*v. spec.action* (*F v*))) ⊓ *spec.pre P* ≤ *spec.seq-ctxt.cl T* (*spec.action G'*)
⟨*proof*⟩

⟨*ML*⟩

lemma *init-write-interference-permute-le*:
fixes *P* :: (*'a agent*, *'ls* × *'s*, *'v*) *spec*
shows *spec.local* (*spec.rel* ({*env*} × *UNIV*) ⫺ (λ-::*unit. spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*))
≤ *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ (*spec.rel* ({*env*} × *UNIV*) ⫺ (λ-::*unit. P*)))
⟨*proof*⟩

lemma *init-write-interference2-permute-le*:
fixes *P* :: (*'a agent*, *'ls* × *'s*, *'v*) *spec*
shows *spec.local* (*spec.rel* ($A \times (Id \times_R r)$) ⫺ (λ-::*unit. spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*))
≤ *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ (*spec.rel* ($A \times (Id \times_R r)$) ⫺ (λ-::*unit. P*)))
⟨*proof*⟩

lemma *trailing-local-act*:
fixes *F* :: *'v* ⇒ (*'w* × *'a agent* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
shows *spec.local* (*P* ⫺ (λ*v. spec.action* (*F v*)))
= *spec.local* (*P* ⫺ (λ*v. spec.action* {(*w*, *a*, (*ls*, *s*), (*ls*, *s'*)) |*w a ls s ls' s'*. (*w*, *a*, (*ls*, *s*), (*ls'*, *s'*)) ∈ *F v* ∧ (*a* = *env* ⟶ *ls'* = *ls*)})) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

definition *localize* :: *'ls rel* ⇒ (*'a agent*, *'s*, *'v*) *spec* ⇒ (*'a agent*, *'ls* × *'s*, *'v*) *spec* **where**
localize r P = *spec.rel* ({*env*} × (*r* $\times_R$ *UNIV*) ∪ *range proc* × (*Id* $\times_R$ *UNIV*)) ⊓ *spec.sinvmap snd P*

⟨*ML*⟩

lemma *localize-le*:
assumes *spec.idle* ≤ *P*
shows *spec.idle* ≤ *spec.localize r P*
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:
shows *spec.term.none* (*spec.localize r P*) = *spec.localize r* (*spec.term.none P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:
shows *spec.term.all* (*spec.localize r P*) = *spec.localize r* (*spec.term.all P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:
assumes *P* ∈ *spec.term.closed* -
shows *spec.localize r P* ∈ *spec.term.closed* -
⟨*proof*⟩

⟨*ML*⟩

lemma *singleton*:
fixes σ :: (*'a agent*, *'s*, *'v*) *trace.t*
shows *spec.localize Id* ⦇σ⦈ = (⨆*ls*::*'ls*. ⦇*trace.map id* (*Pair ls*) *id* σ⦈) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *bot*:
shows *spec.localize r* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
shows *spec.localize r* ⊤ = *spec.rel* ({*env*} × (*r* $\times_R$ *UNIV*) ∪ *range proc* × (*Id* $\times_R$ *UNIV*))
⟨*proof*⟩

lemma *Sup*:
shows *spec.localize r* (⨆*X*) = (⨆*x*∈*X*. *spec.localize r x*)
⟨*proof*⟩

lemmas *sup* = *spec.localize.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *mono*:
assumes *r* ⊆ *r'*

assumes $P \leq P'$
shows *spec.localize r P* $\leq$ *spec.localize r' P'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* ($\leq$) *r*
assumes *monotone orda* ($\leq$) *P*
shows *monotone orda* ($\leq$) (λx. *spec.localize* (*r x*) (*P x*))
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* ($\leq$) *P*
shows *mcont luba orda Sup* ($\leq$) (λx. *spec.localize r* (*P x*))
⟨*proof*⟩

lemma *bind*:
shows *spec.localize r* (*f* ⤜ *g*) = *spec.localize r f* ⤜ (λv. *spec.localize r* (*g v*))
⟨*proof*⟩

lemma *action*:
fixes *F* :: ('*v* × '*a agent* × '*s* × '*s*) *set*
shows *spec.localize r* (*spec.action F*)
= *spec.rel* ({*env*} × (*r* $\times_R$ *Id*))
⤜ (λ-::*unit*. *spec.action* ((*map-prod id* (*map-prod id* (*map-prod snd snd*)) −' *F*)
∩ *UNIV* × ({*env*} × (*r* $\times_R$ *UNIV*) ∪ *range proc* × (*Id* $\times_R$ *UNIV*) ∪ *UNIV* × *Id*))
⤜ (λv. *spec.rel* ({*env*} × (*r* $\times_R$ *Id*)) ⤜ (λ-::*unit*. *spec.return v*)))
⟨*proof*⟩

lemma *return*:
shows (*spec.localize r* (*spec.return v*) :: ('*a agent*, '*ls* × '*s*, '*v*) *spec*)
= *spec.rel* ({*env*} × (*r* $\times_R$ *Id*)) ⤜ (λ-::*unit*. *spec.return v*)
⟨*proof*⟩

lemma *rel*:
shows *spec.localize r* (*spec.rel s*)
= *spec.rel* (({*env*} × (*r* $\times_R$ *UNIV*) ∪ *range proc* × (*Id* $\times_R$ *UNIV*))
∩ *map-prod id* (*map-prod snd snd*) −' (*s* ∪ *UNIV* × *Id*))
⟨*proof*⟩

lemma *rel-le*:
shows *spec.localize Id P* $\leq$ *spec.rel* (*UNIV* × (*Id* $\times_R$ *UNIV*))
⟨*proof*⟩

lemma *parallel*:
shows *spec.localize UNIV* (*P* ∥ *Q*) = *spec.localize UNIV P* ∥ *spec.localize UNIV Q*
⟨*proof*⟩

⟨*ML*⟩

lemma *localize-le*:
assumes *Id* ⊆ *r*
shows *spec.action* (*map-prod id* (*map-prod id* (*map-prod snd snd*)) −' *F* ∩ *UNIV* × *UNIV* × (*Id* $\times_R$ *UNIV*))

$\leq$ *spec.localize r* (*spec.action F*)
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:
assumes $P \in$ *spec.interference.closed* ({*env*} × *UNIV*)
shows *spec.localize UNIV P* $\in$ *spec.interference.closed* ({*env*} × *UNIV*)
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:
assumes *Id* $\subseteq$ *r*
shows *spec.local* (*spec.localize r P*) = *P* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *smap-sndL*:
assumes *UNIV* × (*Id* $\times_R$ *UNIV*) $\subseteq$ *r*
shows *spec.smap snd f* ⋙= *g* = *spec.smap snd* (*f* ⋙= (λ*v*. *spec.rel r* ⊓ *spec.sinvmap snd* (*g v*))) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *smap-sndR*:
assumes *UNIV* × (*Id* $\times_R$ *UNIV*) $\subseteq$ *r*
shows *f* ⋙= (λ*v*. *spec.smap snd* (*g v*)) = *spec.smap snd* (*spec.rel r* ⊓ *spec.sinvmap snd f* ⋙= *g*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *localL*:
shows *spec.local f* ⋙= *g* = *spec.local* (*f* ⋙= (λ*v*. *spec.localize Id* (*g v*)))
⟨*proof*⟩

lemma *localR*:
shows *f* ⋙= (λ*v*. *spec.local* (*g v*)) = *spec.local* (*spec.localize Id f* ⋙= *g*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le*:
shows *spec.local* (*spec.cam.cl* ({*env*} × (*s* $\times_R$ *r*)) *P*) $\leq$ *spec.cam.cl* ({*env*} × *r*) (*spec.local P*)
⟨*proof*⟩

lemma *cl*:
assumes *Id* $\subseteq r_l$
shows *spec.local* (*spec.cam.cl* ({*env*} × (r_l $\times_R$ *r*)) *P*)
= *spec.cam.cl* ({*env*} × *r*) (*spec.local P*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
assumes *Id* $\subseteq$ *s*
assumes $P \in$ *spec.cam.closed* ({*env*} × (*s* $\times_R$ *r*))
shows *spec.local P* $\in$ *spec.cam.closed* ({*env*} × *r*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-le*:
  shows *spec.local* (*spec.interference.cl* ({*env*} × (*s* $\times_R$ *r*)) *P*)
     ≤ *spec.interference.cl* ({*env*} × *r*) (*spec.local P*)
⟨*proof*⟩

lemma *cl*:
  assumes *Id* ⊆ *s*
  shows *spec.local* (*spec.interference.cl* ({*env*} × (*s* $\times_R$ *r*)) *P*)
     = *spec.interference.cl* ({*env*} × *r*) (*spec.local P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
  assumes *P* ∈ *spec.interference.closed* ({*env*} × (*Id* $\times_R$ *r*))
  shows *spec.local P* ∈ *spec.interference.closed* ({*env*} × *r*)
⟨*proof*⟩

lemma *local-UNIV*:
  assumes *P* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
  shows *spec.local P* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
⟨*proof*⟩

⟨*ML*⟩

15.4 *spec.local_init*

⟨*ML*⟩

definition *local-init* :: *'a* ⇒ *'ls* ⇒ (*'a agent*, *'ls* × *'s*, *'v*) *spec* ⇒ (*'a agent*, *'s*, *'v*) *spec* **where**
  local-init a ls P = *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*)

⟨*ML*⟩

lemma *local-init-le-conv*:
  shows ⦇σ⦈ ≤ *spec.local-init a ls P*
     ⟷ ⦇σ⦈ ≤ *spec.idle* ∨ (∃σ′. ⦇σ′⦈ ≤ *P*
          ∧ *trace.steps* σ′ ⊆ *spec.local.qrm*
          ∧ ⦇σ⦈ ≤ ⦇*trace.map id snd id* σ′⦈
          ∧ *fst* (*trace.init* σ′) = *ls*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init-le*[*spec.idle-le*]:
  shows *spec.idle* ≤ *spec.local-init a ls P*
⟨*proof*⟩

⟨*ML*⟩

lemma *Sup*:
  shows *spec.local-init a ls* (⨆*X*) = (⨆*x*∈*X*. *spec.local-init a ls x*) ⊔ *spec.idle*
⟨*proof*⟩

lemma *Sup-not-empty*:
  assumes *X* ≠ {}
  shows *spec.local-init a ls* (⨆*X*) = (⨆*x*∈*X*. *spec.local-init a ls x*)

⟨*proof*⟩

lemmas *sup* = *spec.local-init.Sup-not-empty*[**where** $X=\{X,\ Y\}$ **for** X Y, *simplified*]

lemma *bot*:
  shows *spec.local-init a ls* ⊥ = *spec.idle*
⟨*proof*⟩

lemma *top*:
  shows *spec.local-init a ls* ⊤ = (⊤ :: (*'a agent*, *'s*, *'v*) *spec*)
⟨*proof*⟩

lemma *monotone*:
  shows *mono* (*spec.local-init a ls* :: (*'a agent*, *'ls* × *'s*, *'v*) *spec* ⇒ -)
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.local-init.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.local-init.monotone*]

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *P*
  shows *monotone orda* (≤) (λ*x*. *spec.local-init a ls* (*P x*))
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *spec.local-init a ls* (*P x*))
⟨*proof*⟩

lemma *action*:
  fixes *F* :: (*'v* × *'a agent* × (*'ls* × *'s*) × (*'ls* × *'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'*)) ∈ *F* ∧ (*a* = *env* ⟶ *ls'* = *ls*)} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *return*:
  shows *spec.local-init a ls* (*spec.return v*) = *spec.return v*
⟨*proof*⟩

lemma *localize-le*:
  assumes *spec.idle* ≤ *P*
  shows *spec.local-init a ls* (*spec.localize r P*) ≤ *P*
⟨*proof*⟩

lemma *localize*:
  assumes *spec.idle* ≤ *P*
  assumes *Id* ⊆ *r*
  shows *spec.local-init a ls* (*spec.localize r P*) = *P* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *inf-interference*:
  shows *spec.local-init a ls P* = *spec.local-init a ls* (*P* ⊓ *spec.local.interference*)
⟨*proof*⟩

lemma *eq-local*:
  assumes *spec.idle* ≤ *P*
  shows (⨆*ls*. *spec.local-init a ls P*) = *spec.local P*
⟨*proof*⟩

lemma *ag-le*:
shows *spec.local-init a ls* ({|*P*|}, *Id* $\times_R$ *A* ⊢ *UNIV* $\times_R$ *G*, {|λ*v s. Q v* (*snd s*)|})
≤ {|λ*s. P* (*ls*, *s*)|}, *A* ⊢ *G*, {|*Q*|}
⟨*proof*⟩

⟨*ML*⟩

lemma *local-initL*:
shows *spec.local-init a ls f* ⤜ *g* = *spec.local-init a ls* (*f* ⤜ (λ*v. spec.localize Id* (*g v*)))
⟨*proof*⟩

lemma *local-initR*:
shows *f* ⤜ (λ*v. spec.local-init a ls* (*g v*)) = *spec.local-init a ls* (*spec.localize Id f* ⤜ *g*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init*:
fixes *P* :: ('*a agent*, '*ls* × '*t*, '*v*) *spec*
fixes *sf* :: '*s* ⇒ '*t*
shows *spec.sinvmap sf* (*spec.local-init a ls P*)
= *spec.local-init a ls* (*spec.rel* (*UNIV* × (*Id* $\times_R$ *map-prod sf sf* −' *Id*))
⤜ (λ-::*unit. spec.sinvmap* (*map-prod id sf*) *P*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init*:
shows *spec.vmap vf* (*spec.local-init a ls P*) = *spec.local-init a ls* (*spec.vmap vf P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init*:
shows *spec.vinvmap vf* (*spec.local-init a ls P*) = *spec.local-init a ls* (*spec.vinvmap vf P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init*:
shows *spec.term.none* (*spec.local-init a ls P*) = *spec.local-init a ls* (*spec.term.none P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init*:
shows *spec.term.all* (*spec.local-init a ls P*)
= *spec.local-init a ls* (*spec.term.all P*) ⊔ ⨆*range spec.return*
⟨*proof*⟩

⟨*ML*⟩

lemma *local-init*:
assumes *P* ∈ *spec.interference.closed* ({*env*} × (*Id* $\times_R$ *r*))
shows *spec.local-init a ls P* ∈ *spec.interference.closed* ({*env*} × *r*)
⟨*proof*⟩

⟨*ML*⟩

15.5 Hoist to *('s, 'v) prog*

⟨*ML*⟩

lift-definition *local* :: ('*ls* × '*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **is** *spec.local*
⟨*proof*⟩

definition *local-init* :: '*ls* ⇒ ('*ls* × '*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **where**
local-init ls P = *prog.local* (*prog.write* (*map-prod* ⟨*ls*⟩ *id*) ≫ *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*
⟨*proof*⟩

⟨*ML*⟩

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*⟩

⟨*ML*⟩

lemma *Sup*:
shows *prog.local* ($\bigsqcup$*X*) = ($\bigsqcup$*x*∈*X*. *prog.local x*)
⟨*proof*⟩

lemmas *sup* = *prog.local.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*:
shows *prog.local* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
shows *prog.local* ⊤ = ⊤
⟨*proof*⟩

lemma *monotone*:
shows *mono prog.local*
⟨*proof*⟩

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* (≤) (λ*x*. *prog.local* (*P x*))
⟨*proof*⟩

lemma *bind-botR*:
shows *prog.local* (*P* ≫= ⊥) = *prog.local P* ≫= ⊥
⟨*proof*⟩

lemma *action*:
  shows *prog.local* (*prog.action F*) = *prog.action* (*map-prod id* (*map-prod snd snd*) ‘ *F*)
⟨*proof*⟩

lemma *return*:
  shows *prog.local* (*prog.return v*) = *prog.return v*
⟨*proof*⟩

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (=) (*rel-fun cr-prog cr-prog*) (*spec.local-init* ()) *prog.local-init*
⟨*proof*⟩

lemma *Sup*:
  shows *prog.local-init ls* (⨆ *X*) = (⨆ *x*∈*X*. *prog.local-init ls x*)
⟨*proof*⟩

lemmas *sup* = *prog.local-init.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*[*simp*]:
  shows *prog.local-init ls* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
  shows *prog.local-init ls* ⊤ = ⊤
⟨*proof*⟩

lemma *monotone*:
  shows *mono* (*prog.local-init ls*)
⟨*proof*⟩

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* (≤) *P*
  shows *monotone orda* (≤) (λ*x*. *prog.local-init ls* (*P x*))
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *prog.local-init ls* (*P x*))
⟨*proof*⟩

lemma *bind-botR*:
  shows *prog.local-init ls* (*P* ⋙ ⊥) = *prog.local-init ls P* ⋙ ⊥
⟨*proof*⟩

lemma *return*:
  shows *prog.local-init ls* (*prog.return v*) = *prog.return v* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *eq-local*:
  shows (⨆ *ls*. *prog.local-init ls P*) = *prog.local P*
⟨*proof*⟩

⟨*ML*⟩

lemma *localize-alt-def*:
  shows *prog.localize P* = *prog.rel* (*Id* $\times_R$ *UNIV*) ⊓ *prog.sinvmap snd P*
⟨*proof*⟩

⟨*ML*⟩

lemma *Sup*:
  shows *prog.localize* (⨆ *X*) = (⨆ *x*∈*X*. *prog.localize x*)
⟨*proof*⟩

lemmas *sup* = *prog.localize.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*:
  shows *prog.localize* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
  shows *prog.localize* ⊤ = *prog.rel* (*Id* $\times_R$ *UNIV*)
⟨*proof*⟩

lemma *monotone*:
  shows *mono prog.localize*
⟨*proof*⟩

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* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *prog.localize* (*P x*))
⟨*proof*⟩

lemmas *p2s*[*prog.p2s.simps*] = *prog.localize.rep-eq*

lemma *bind*:
  shows *prog.localize* (*f* ≫= *g*) = *prog.localize f* ≫= (λ*v*. *prog.localize* (*g v*))
⟨*proof*⟩

lemma *parallel*:
  shows *prog.localize* (*P* ∥ *Q*) = *prog.localize P* ∥ *prog.localize Q*
⟨*proof*⟩

lemma *rel*:
  fixes *r* :: *'s rel*
  shows *prog.localize* (*prog.rel r*) = *prog.rel* (*Id* $\times_R$ *r*)
⟨*proof*⟩

lemma *action*:
  shows *prog.localize* (*prog.action F*)
      = *prog.action* (*map-prod id* (*map-prod snd snd*) −‘ *F* ∩ *UNIV* × (*Id* $\times_R$ *UNIV*))
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:

fixes P :: ('s, 'v) *prog*
shows *prog.local* (*prog.localize* P :: ('ls × 's, 'v) *prog*) = P
⟨*proof*⟩

⟨*ML*⟩

15.6 Refinement rules

⟨*ML*⟩

We use *localizeA* to hoist assumes similarly to *spec.localize*.

definition *localizeA* :: (*sequential*, 's, 'v) *spec* ⇒ (*sequential*, 'ls × 's, 'v) *spec* **where**
localizeA P = *spec.local.interference* ⊓ *spec.sinvmap snd* P

⟨*ML*⟩

lemma *bot*:
shows *spec.localizeA* ⊥ = ⊥
⟨*proof*⟩

lemma *top*:
shows *spec.localizeA* ⊤ = *spec.local.interference*
⟨*proof*⟩

lemma *ag-assm*:
shows *spec.localizeA* (*ag.assm* A) = *ag.assm* ($Id \times_R A$)
⟨*proof*⟩

⟨*ML*⟩

lemma *localI*: — Introduce local state
fixes A :: (*sequential*, 's, 'v) *spec*
fixes c :: (*sequential*, 'ls × 's, 'v) *spec*
fixes c' :: (*sequential*, 's, 'v) *spec*
fixes P :: 's *pred*
fixes Q :: 'v ⇒ 's *pred*
assumes c ≤ ⦃λs. P (*snd s*)⦄, *spec.localizeA* A ⊩ *spec.sinvmap snd* c', ⦃λv s. Q v (*snd s*)⦄
shows *spec.local* c ≤ ⦃P⦄, A ⊩ c', ⦃Q⦄
⟨*proof*⟩

⟨*ML*⟩

lemma *local-seq-ctxt-cl*:
fixes A :: (*sequential*, 's, 'v) *spec*
fixes P :: 's *pred*
fixes Q :: 'v ⇒ 's *pred*
fixes c :: (*sequential*, 'ls × 's, 'v) *spec*
fixes c' :: (*sequential*, 'ls × 's, 'v) *spec*
assumes *spec.local.interference* ⊓ c
≤ ⦃λs. P (*snd s*)⦄, *spec.localizeA* A ⊩ *spec.seq-ctxt.cl False* (*spec.local.interference* ⊓ c'), ⦃λv s. Q v (*snd* s)⦄
shows *spec.local* c ≤ ⦃P⦄, A ⊩ *spec.local* c', ⦃Q⦄
⟨*proof*⟩

lemma *cl-bind*:
fixes f :: ('a *agent*, 'ls × 's, 'v) *spec*
fixes g :: 'v ⇒ ('a *agent*, 'ls × 's, 'w) *spec*
assumes g: ⋀v. g v ≤ ⦃Q' v⦄, *refinement.spec.bind.res* (*spec.pre* P ⊓ *spec.term.all* A ⊓ *spec.seq-ctxt.cl True* f') A v ⊩ *spec.seq-ctxt.cl* T (g' v), ⦃Q⦄

assumes f: $f \le \{\!|P|\!\}$, *spec.term.all* $A \Vdash$ *spec.seq-ctxt.cl True* f', $\{\!|Q'|\!\}$
shows $f \ggg g \le \{\!|P|\!\}$, $A \Vdash$ *spec.seq-ctxt.cl* T $(f' \ggg g')$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *cl-action-permuteL*:
fixes F :: $('v \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes G :: $'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes G' :: $('v' \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes F' :: $'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes Q :: $'w \Rightarrow ('ls \times 's)$ *pred*
assumes F: $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!] \Longrightarrow snd\ s' = snd\ s$
assumes $FGG'F'$: $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a',\ s,\ t) \in F;\ (w,\ a,\ t,\ s') \in G\ v]\!]$
$\Longrightarrow \exists\, v'\ a''\ a'''\ t'.\ (v',\ a'',\ s,\ t') \in G' \wedge (w,\ a''',\ t',\ s') \in F'\ v'$
$\wedge\ snd\ s' = snd\ t' \wedge (snd\ s \ne snd\ t' \longrightarrow a'' = a)$
assumes Q: $\bigwedge v\ w\ a\ a'\ s\ s'\ s''.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G';\ (w,\ a',\ s',\ s'') \in F'\ v]\!] \Longrightarrow Q\ w\ s''$
shows *spec.action* $F \ggg (\lambda v.$ *spec.action* $(G\ v)) \le \{\!|P|\!\}$, $A \Vdash$ *spec.seq-ctxt.cl* T (*spec.action* $G' \ggg (\lambda v.$ *spec.action* $(F'\ v)))$, $\{\!|Q|\!\}$
⟨*proof*⟩

lemma *cl-action-permuteR*:
fixes G :: $('v \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes F :: $'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes F' :: $('v' \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes G' :: $'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
assumes G: $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G;\ snd\ s' \ne snd\ s]\!]$
$\Longrightarrow \exists\, v'\ w\ a''\ t\ s''.\ (v',\ a'',\ s,\ t) \in F' \wedge (w,\ a,\ t,\ s'') \in G'\ v' \wedge snd\ t = snd\ s \wedge snd\ s'' = snd\ s'$
assumes $GFF'G'$: $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a,\ s,\ t) \in G;\ (w,\ a',\ t,\ s') \in F\ v]\!]$
$\Longrightarrow snd\ s' = snd\ t \wedge (\exists\, v'\ a''\ a'''\ t'.\ (v',\ a'',\ s,\ t') \in F' \wedge (w,\ a''',\ t',\ s') \in G'\ v'$
$\wedge\ snd\ t' = snd\ s \wedge (snd\ s' \ne snd\ t' \longrightarrow a''' = a))$
assumes Q: $\bigwedge v\ w\ a\ a'\ s\ s'\ s''.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F';\ (w,\ a',\ s',\ s'') \in G'\ v]\!] \Longrightarrow Q\ w\ s''$
shows *spec.action* $G \ggg (\lambda v.$ *spec.action* $(F\ v)) \le \{\!|P|\!\}$, $A \Vdash$ *spec.seq-ctxt.cl* T (*spec.action* $F' \ggg (\lambda v.$ *spec.action* $(G'\ v)))$, $\{\!|Q|\!\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *localI*: — Introduce local state
fixes A :: (*sequential*, $'s$, $'v$) *spec*
fixes c :: $('ls \times 's,\ 'v)$ *prog*
fixes c' :: (*sequential*, $'s$, $'v$) *spec*
fixes P :: $'s$ *pred*
fixes Q :: $'v \Rightarrow 's$ *pred*
assumes *prog.p2s* $c \le \{\!|\lambda s.\ P\ (snd\ s)|\!\}$, *spec.localizeA* $A \Vdash$ *spec.sinvmap snd* c', $\{\!|\lambda v\ s.\ Q\ v\ (snd\ s)|\!\}$
shows *prog.p2s* (*prog.local* c) $\le \{\!|P|\!\}$, $A \Vdash c'$, $\{\!|Q|\!\}$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

lemma *data*:
  fixes *A* :: (*sequential*, *'s*, *'v*) *spec*
  fixes *c* :: (*sequential*, *'cls* × *'s*, *'v*) *spec*
  fixes *c'* :: (*sequential*, *'als* × *'s*, *'v*) *spec*
  fixes *sf* :: *'cls* ⇒ *'als*
  assumes *c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *spec.localizeA A* ⊩ *spec.sinvmap* (*map-prod sf id*) *c'*, ⦃λ*v s*. *Q v* (*snd s*)⦄
  shows *spec.local c* ≤ ⦃*P*⦄, *A* ⊩ *spec.local c'*, ⦃*Q*⦄
⟨*proof*⟩

⟨*ML*⟩

lemma *data*:
  fixes *A* :: (*sequential*, *'s*, *'v*) *spec*
  fixes *c* :: (*'cls* × *'s*, *'v*) *prog*
  fixes *c'* :: (*'als* × *'s*, *'v*) *prog*
  fixes *sf* :: *'cls* ⇒ *'als*
  assumes *prog.p2s c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *spec.localizeA A* ⊩ *spec.sinvmap* (*map-prod sf id*) (*prog.p2s c'*), ⦃λ*v s*. *Q v* (*snd s*)⦄
  shows *prog.p2s* (*prog.local c*) ≤ ⦃*P*⦄, *A* ⊩ *prog.p2s* (*prog.local c'*), ⦃*Q*⦄
⟨*proof*⟩

⟨*ML*⟩

15.7 Assume/guarantee

⟨*ML*⟩

lemma *local*:
  fixes *A G* :: *'s rel*
  fixes *P* :: *'s pred*
  fixes *Q* :: *'v* ⇒ *'s pred*
  fixes *c* :: (*sequential*, *'ls* × *'s*, *'v*) *spec*
  assumes *c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *Id* $\times_R$ *A* ⊢ *UNIV* $\times_R$ *G*, ⦃λ*v s*. *Q v* (*snd s*)⦄
  shows *spec.local c* ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
⟨*proof*⟩

lemma *localize-lift*:
  fixes *A G* :: *'s rel*
  fixes *P* :: *'s* ⇒ *bool*
  fixes *Q* :: *'v* ⇒ *'s* ⇒ *bool*
  fixes *c* :: (*sequential*, *'s*, *'v*) *spec*
  notes *inf.bounded-iff*[*simp del*]
  assumes *c*: *c* ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
  shows *spec.localize UNIV c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *UNIV* $\times_R$ *A* ⊢ *Id* $\times_R$ *G*, ⦃λ*v s*::*'ls* × *'s*. *Q v* (*snd s*)⦄

⟨*proof*⟩

⟨*ML*⟩

lemma *local*:
  fixes A G :: $'s$ *rel*
  fixes P :: $'s$ *pred*
  fixes Q :: $'v \Rightarrow 's$ *pred*
  fixes c :: $('ls \times 's, 'v)$ *prog*
  assumes *prog.p2s c* $\leq$ ⦃$\lambda s.$ P (*snd s*)⦄, *Id* $\times_R$ $A \vdash$ *UNIV* $\times_R$ G, ⦃λv $s.$ Q v (*snd s*)⦄
  shows *prog.p2s* (*prog.local c*) $\leq$ ⦃P⦄, $A \vdash G$, ⦃Q⦄
⟨*proof*⟩

lemma *localize-lift*:
  fixes A G :: $'s$ *rel*
  fixes P :: $'s \Rightarrow$ *bool*
  fixes Q :: $'v \Rightarrow 's \Rightarrow$ *bool*
  fixes c :: $('s, 'v)$ *prog*
  assumes *prog.p2s c* $\leq$ ⦃P⦄, $A \vdash G$, ⦃Q⦄
  shows *prog.p2s* (*prog.localize c*) $\leq$ ⦃$\lambda s.$ P (*snd s*)⦄, *UNIV* $\times_R$ $A \vdash$ *Id* $\times_R$ G, ⦃λv $s.$ Q v (*snd s*)⦄
⟨*proof*⟩

⟨*ML*⟩

15.8 Specification inhabitation

⟨*ML*⟩

lemma *localize*:
  assumes P $-s$, $xs\rightarrow$ P'
  assumes *Id* $\subseteq r$
  shows *spec.localize r P* $-$(*ls*, *s*), *map* (*map-prod id* (*Pair ls*)) $xs\rightarrow$ *spec.localize r* P'
⟨*proof*⟩

lemma *local*:
  assumes P $-$(*ls*, *s*), $xs\rightarrow$ *spec.return v*
  assumes *trace.steps'* (*ls*, *s*) *xs* $\subseteq$ *spec.local.qrm*
  shows *spec.local P* $-s$, *map* (*map-prod id snd*) $xs\rightarrow$ *spec.return v*
⟨*proof*⟩

lemma *local-init*:
  assumes P $-$(*ls*, *s*), $xs\rightarrow$ P'
  assumes *trace.steps'* (*ls*, *s*) *xs* $\subseteq$ *spec.local.qrm*
  shows *spec.local-init a ls P* $-s$, *map* (*map-prod id snd*) $xs\rightarrow$ *spec.local-init a* (*fst* (*trace.final'* (*ls*, *s*) *xs*)) P'
⟨*proof*⟩

⟨*ML*⟩

lemma *localize*:
  assumes *prog.p2s P* $-s$, $xs\rightarrow$ *prog.p2s* P'
  shows *prog.p2s* (*prog.localize P*) $-$(*ls*, *s*), *map* (*map-prod id* (*Pair ls*)) $xs\rightarrow$ *prog.p2s* (*prog.localize* P')
⟨*proof*⟩

lemma *local*:
  assumes *prog.p2s P* $-$(*ls*, *s*), $xs\rightarrow$ *spec.return v*
  assumes *trace.steps'* (*ls*, *s*) *xs* $\subseteq$ *spec.local.qrm*
  shows *prog.p2s* (*prog.local P*) $-s$, *map* (*map-prod id snd*) $xs\rightarrow$ *spec.return v*
⟨*proof*⟩

lemma *local-init*:
  assumes *prog.p2s P* $-(ls, s), xs\rightarrow$ *prog.p2s P'*
  assumes *trace.steps'* (ls, s) *xs* $\subseteq$ *spec.local.qrm*
  shows *prog.p2s* (*prog.local-init ls P*) $-s$, *map* (*map-prod id snd*) *xs*$\rightarrow$ *prog.p2s* (*prog.local-init* (*fst* (*trace.final'* (*ls*, *s*) *xs*)) *P'*)
⟨*proof*⟩

⟨*ML*⟩

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*⟩

⟨*ML*⟩

lemma *trailing*:
shows *tmap sf vf (trailing s xs) = trailing (sf s) (tmap sf vf xs)*
⟨*proof*⟩

⟨*ML*⟩

lemma *trailing*:
shows *tlength (trailing s xs)* $\leq$ *tlength xs*
⟨*proof*⟩

⟨*ML*⟩

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*: *trailing s (trepeat s) = trepeat s*
⟨*proof*⟩

lemma *eq-TNil-conv*:
shows *trailing s xs = TNil b* ⟷ *tfinite xs* ∧ *terminal xs = b*
and *TNil b = trailing s xs* ⟷ *tfinite xs* ∧ *terminal xs = b*
and *is-TNil (trailing s xs)* ⟷ *tfinite xs*
⟨*proof*⟩

lemma *eq-TCons-conv*:
shows *trailing s xs = TCons y ys* ⟷ ¬*tfinite xs* ∧ *TCons y ys = trepeat s*
and *TCons y ys = trailing s xs* ⟷ ¬*tfinite xs* ∧ *TCons y ys = trepeat s*
⟨*proof*⟩

lemma *tmap*:
shows *trailing s (tmap sf vf xs) = tmap id vf (trailing s xs)*
⟨*proof*⟩

⟨*ML*⟩

lemma *unique*:
assumes ⋀*s xs. f s xs = (if snd ‘ tset xs* ⊆ {*s*} *then trailing (undefined, s) xs*
else if snd (thd xs) = s then f s (ttl xs)
else TCons (thd xs) (f (snd (thd xs)) (ttl xs)))
shows *f = collapse*
⟨*proof*⟩

lemma *collapse*:
shows *collapse s (collapse s xs) = collapse s xs*
⟨*proof*⟩

lemma *simps*[*simp*]:
shows *TNil*: *collapse s (TNil b) = TNil b*
and *TCons*: *collapse s (TCons x xs) = (if snd x = s then collapse s xs else TCons x (collapse (snd x) xs))*

and *trailing*: *collapse s* (*trailing* (*undefined*, *s*) *xs*) = *trailing* (*undefined*, *s*) *xs*
⟨*proof*⟩

lemma *tshift-stuttering*:
assumes *snd* ‘ *set xs* ⊆ {*s*}
shows *collapse s* (*tshift xs ys*) = *collapse s ys*
⟨*proof*⟩

lemma *infinite-trailing*:
assumes ¬*tfinite xs*
assumes *snd* ‘ *tset xs* ⊆ {*s*′}
shows *collapse s xs* = (*if s* = *s*′ *then trepeat* (*undefined*, *s*′) *else TCons* (*thd xs*) (*trepeat* (*undefined*, *s*′)))
⟨*proof*⟩

lemma *eq-TNil-conv*:
shows *collapse s xs* = *TNil b* ⟷ *tfinite xs* ∧ *snd* ‘ *tset xs* ⊆ {*s*} ∧ *terminal xs* = *b* (**is** *?lhs* ⟷ *?rhs*)
and *TNil b* = *collapse s xs* ⟷ *tfinite xs* ∧ *snd* ‘ *tset xs* ⊆ {*s*} ∧ *terminal xs* = *b* (**is** *?thesis1*)
⟨*proof*⟩

lemma *is-TNil-conv*:
shows *is-TNil* (*collapse s xs*) ⟷ *tfinite xs* ∧ *snd* ‘ *tset xs* ⊆ {*s*} (**is** *?thesis2*)
⟨*proof*⟩

lemma *eq-TConsE*:
assumes *collapse s xs* = *TCons y ys*
obtains
(*trailing-stuttering*) ¬ *tfinite xs*
and *snd* ‘ *tset xs* = {*s*}
and *TCons y ys* = *trepeat* (*undefined*, *s*)
| (*step*) *us ys*′ **where** *xs* = *tshift us* (*TCons y ys*′)
and *snd* ‘ *set us* ⊆ {*s*}
and *snd y* ≠ *s*
and *collapse* (*snd y*) *ys*′ = *ys*
⟨*proof*⟩

lemma *eq-TCons-conv*:
shows *collapse s xs* = *TCons y ys*
⟷ (¬*tfinite xs* ∧ *snd* ‘ *tset xs* = {*s*} ∧ *TCons y ys* = *trepeat* (*undefined*, *s*))
∨ (∃ *xs*′ *ys*′. *xs* = *tshift xs*′ (*TCons y ys*′) ∧ *snd* ‘ *set xs*′ ⊆ {*s*} ∧ *snd y* ≠ *s* ∧ *collapse* (*snd y*) *ys*′ = *ys*) (**is** *?lhs* ⟷ *?rhs*)
and *TCons y ys* = *collapse s xs*
⟷ (¬*tfinite xs* ∧ *snd* ‘ *tset xs* = {*s*} ∧ *TCons y ys* = *trepeat* (*undefined*, *s*))
∨ (∃ *xs*′ *ys*′. *xs* = *tshift xs*′ (*TCons y ys*′) ∧ *snd* ‘ *set xs*′ ⊆ {*s*} ∧ *snd y* ≠ *s* ∧ *collapse* (*snd y*) *ys*′ = *ys*) (**is** *?thesis1*)
⟨*proof*⟩

lemma *tfinite*:
shows *tfinite* (*collapse s xs*) ⟷ *tfinite xs* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *tfinite-conv*:
assumes *collapse s xs* = *collapse s*′ *xs*′
shows *tfinite xs* ⟷ *tfinite xs*′
⟨*proof*⟩

lemma *terminal*:
shows *terminal* (*collapse s xs*) = *terminal xs*
⟨*proof*⟩

lemma *tlength*:
  shows *tlength* (*collapse s xs*) ≤ *tlength xs*
⟨*proof*⟩

lemma *tset-memberD*:
  assumes (*a*, *s'*) ∈ *tset* (*collapse s xs*)
  shows *s'* ∈ *snd* ‘ *tset xs*
⟨*proof*⟩

lemma *tset-memberD2*:
  assumes (*a*, *s'*) ∈ *tset xs*
  shows *s* = *s'* ∨ *s'* ∈ *snd* ‘ *tset* (*collapse s xs*)
⟨*proof*⟩

lemma *tshift*:
  shows *collapse s* (*tshift xs ys*) = *tshift* (*trace.natural' s xs*) (*collapse* (*trace.final' s xs*) *ys*)
⟨*proof*⟩

lemma *trepeat*:
  shows *collapse s* (*trepeat* (*a*, *s*)) = *trepeat* (*undefined*, *s*)
⟨*proof*⟩

lemma *eq-trepeat-conv*:
  shows *trepeat* (*undefined*, *s*) = *collapse s xs* ⟷ ¬*tfinite xs* ∧ *snd* ‘ *tset xs* = {*s*} (**is** *?thesis1*)
    and *collapse s xs* = *trepeat* (*undefined*, *s*) ⟷ ¬*tfinite xs* ∧ *snd* ‘ *tset xs* = {*s*} (**is** *?thesis2*)
⟨*proof*⟩

lemma *treplicate*:
  shows *collapse s* (*treplicate i* (*a*, *s*) *v*) = *TNil v*
⟨*proof*⟩

lemma *eq-tshift-conv*:
  shows *collapse s xs* = *tshift ys zs*
    ⟷ (∃ *xs' xs'' ys'*. *tshift xs' xs''* = *xs* ∧ *trace.natural' s xs'* @ *ys'* = *ys*
        ∧ ((¬*tfinite xs''* ∧ *snd* ‘ *tset xs''* = {*trace.final' s xs'*} ∧ *tshift ys' zs* = *trepeat* (*undefined*, *trace.final' s xs'*))
        ∨ (*ys'* = [] ∧ *collapse* (*trace.final' s xs'*) *xs''* = *zs*))) (**is** *?lhs* ⟷ *?rhs*)
    and *tshift ys zs* = *collapse s xs*
    ⟷ (∃ *xs' xs'' ys'*. *tshift xs' xs''* = *xs* ∧ *trace.natural' s xs'* @ *ys'* = *ys*
        ∧ ((¬*tfinite xs''* ∧ *snd* ‘ *tset xs''* = {*trace.final' s xs'*} ∧ *tshift ys' zs* = *trepeat* (*undefined*, *trace.final' s xs'*))
        ∨ (*ys'* = [] ∧ *collapse* (*trace.final' s xs'*) *xs''* = *zs*))) (**is** *?thesis1*)
⟨*proof*⟩

lemma *eq-collapse-ttake-dropn-conv*:
  shows *collapse s xs* = *collapse s ys*
    ⟷ (∃ *j*. *trace.natural' s* (*fst* (*ttake i xs*)) = *trace.natural' s* (*fst* (*ttake j ys*))
        ∧ *snd* (*ttake i xs*) = *snd* (*ttake j ys*)
        ∧ *collapse* (*trace.final' s* (*fst* (*ttake i xs*))) (*tdropn i xs*)
        = *collapse* (*trace.final' s* (*fst* (*ttake i xs*))) (*tdropn j ys*)) (**is** *?lhs* ⟷ (∃ *j*. *?rhs i j s xs ys*))
⟨*proof*⟩

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*

⟨*proof*⟩

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*⟩

⟨*ML*⟩

definition *natural* :: ('*a*, '*s*, '*v*) *behavior.t* ⇒ ('*a*, '*s*, '*v*) *behavior.t* (‹$\natural_T$›) **where**
  $\natural_T\omega$ = *behavior.B* (*behavior.init* ω) (*collapse* (*behavior.init* ω) (*behavior.rest* ω))

⟨*ML*⟩

lemma *collapse*[*simp*]:
  shows *behavior.sset* (*behavior.B s* (*collapse s xs*)) = *behavior.sset* (*behavior.B s xs*)
⟨*proof*⟩

lemma *natural*[*simp*]:
  shows *behavior.sset* ($\natural_T\omega$) = *behavior.sset* ω
⟨*proof*⟩

lemma *continue*:
  shows *behavior.sset* (σ $@-_B$ *xs*) = *trace.sset* σ ∪ (*case trace.term* σ *of None* ⇒ *snd* ‘ *tset xs* | *Some* - ⇒ {})
⟨*proof*⟩

⟨*ML*⟩

lemma *sel*[*simp*]:
  shows *behavior.init* ($\natural_T\omega$) = *behavior.init* ω
    and *behavior.rest* ($\natural_T\omega$) = *collapse* (*behavior.init* ω) (*behavior.rest* ω)
⟨*proof*⟩

lemma *TNil*:
  shows $\natural_T$(*behavior.B s* (*TNil v*)) = *behavior.B s* (*TNil v*)
⟨*proof*⟩

lemma *tfinite*:
  shows *tfinite* (*behavior.rest* ($\natural_T$ ω)) ⟷ *tfinite* (*behavior.rest* ω)
⟨*proof*⟩

lemma *continue*:
  shows $\natural_T$(σ $@-_B$ *xs*) = ♮σ $@-_B$ (*collapse* (*trace.final* σ) *xs*)
⟨*proof*⟩

lemma *tshift*:
  shows $\natural_T$(*behavior.B s* (*tshift as xs*)) = *behavior.B s* (*collapse s* (*tshift as xs*))
⟨*proof*⟩

lemma *trepeat*:
  shows $\natural_T$(*behavior.B s* (*trepeat* (*a*, *s*))) = *behavior.B s* (*trepeat* (*undefined*, *s*))
⟨*proof*⟩

lemma *treplicate*:
  shows $\natural_T$(*behavior.B s* (*treplicate i* (*a*, *s*) *v*)) = *behavior.B s* (*TNil v*)
⟨*proof*⟩

lemma *map-natural*:

shows $\natural_T(behavior.map\ af\ sf\ vf\ (\natural_T\omega)) = \natural_T(behavior.map\ af\ sf\ vf\ \omega)$
⟨*proof*⟩

lemma *idle*:
assumes $behavior.sset\ \omega \subseteq \{behavior.init\ \omega\}$
shows $\natural_T\omega = behavior.B\ (behavior.init\ \omega)\ (trailing\ (undefined,\ behavior.init\ \omega)\ (behavior.rest\ \omega))$
⟨*proof*⟩

⟨*ML*⟩

interpretation *stuttering*: *galois.image-vimage-idempotent* $\natural_T$
⟨*proof*⟩

⟨*ML*⟩

abbreviation *syn* :: $('a,\ 's,\ 'v)\ behavior.t \Rightarrow ('a,\ 's,\ 'v)\ behavior.t \Rightarrow bool$ (**infix** ‹$\simeq_T$› *50*) **where**
$\omega_1 \simeq_T \omega_2 \equiv behavior.stuttering.equivalent\ \omega_1\ \omega_2$

lemma *map*:
assumes $\omega_1 \simeq_T \omega_2$
shows $behavior.map\ af\ sf\ vf\ \omega_1 \simeq_T behavior.map\ af\ sf\ vf\ \omega_2$
⟨*proof*⟩

lemma *takeE*:
assumes $\omega_1 \simeq_T \omega_2$
obtains j **where** $behavior.take\ i\ \omega_1 \simeq_S behavior.take\ j\ \omega_2$
⟨*proof*⟩

lemma *idle-dropn*:
assumes $behavior.dropn\ i\ \omega = Some\ \omega'$
assumes $behavior.sset\ \omega \subseteq \{behavior.init\ \omega\}$
shows $\omega \simeq_T \omega'$
⟨*proof*⟩

⟨*ML*⟩

lemma *takeE*:
fixes σ :: $('a,\ 's,\ 'v)\ trace.t$
assumes $behavior.take\ i\ \omega \simeq_S \sigma$
obtains $\omega'\ j$ **where** $\omega \simeq_T \omega'$ **and** $\sigma = behavior.take\ j\ \omega'$
⟨*proof*⟩

lemmas *rev-takeE* = *trace.stuttering.equiv.behavior.takeE*[*OF sym*]

⟨*ML*⟩

lemma *takeE*:
fixes ω :: $('a,\ 's,\ 'v)\ behavior.t$
obtains j **where** $\natural(behavior.take\ i\ \omega) = behavior.take\ j\ (\natural_T\omega)$
⟨*proof*⟩

⟨*ML*⟩

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*
⟨*proof*⟩

setup-lifting *type-definition-tls*

instantiation *tls* :: (*type*, *type*, *type*) *complete-boolean-algebra*
begin

lift-definition *bot-tls* :: ('*a*, '*s*, '*v*) *tls* **is** *empty* ⟨*proof*⟩
lift-definition *top-tls* :: ('*a*, '*s*, '*v*) *tls* **is** *UNIV* ⟨*proof*⟩
lift-definition *sup-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *sup* ⟨*proof*⟩
lift-definition *inf-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *inf* ⟨*proof*⟩
lift-definition *less-eq-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩
lift-definition *less-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ *bool* **is** *less* ⟨*proof*⟩
lift-definition *Inf-tls* :: ('*a*, '*s*, '*v*) *tls set* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *Inf* ⟨*proof*⟩
lift-definition *Sup-tls* :: ('*a*, '*s*, '*v*) *tls set* ⇒ ('*a*, '*s*, '*v*) *tls* **is** λ*X*. *Sup X* ⊔ *behavior.stuttering.cl* {} ⟨*proof*⟩
lift-definition *minus-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *minus* ⟨*proof*⟩
lift-definition *uminus-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *uminus* ⟨*proof*⟩

instance
⟨*proof*⟩

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*]

⟨*ML*⟩

lemma *boolean-implication-transfer*[*transfer-rule*]:
shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=))) ($\longrightarrow_B$) ($\longrightarrow_B$)
⟨*proof*⟩

lemma *bot-not-top*:
shows $\bot \neq (\top$ :: ('*a*, '*s*, '*v*) *tls*)
⟨*proof*⟩

⟨*ML*⟩

16.3 Irreducible elements

⟨*ML*⟩

definition *singleton* :: ('*a*, '*s*, '*v*) *behavior.t* ⇒ ('*a*, '*s*, '*v*) *behavior.t set* **where**
singleton ω = *behavior.stuttering.cl* {ω}

lemma *singleton-le-conv*:
shows *raw.singleton* $\sigma_1 \leq$ *raw.singleton* $\sigma_2 \longleftrightarrow \natural_T\sigma_1 = \natural_T\sigma_2$

⟨*proof*⟩

⟨*ML*⟩

lift-definition *singleton* :: $('a, 's, 'v)$ *behavior.t* $\Rightarrow$ $('a, 's, 'v)$ *tls* (‹⦇-⦈$_T$› [0]) **is** *raw.singleton*
⟨*proof*⟩

abbreviation *singleton-behavior-syn* :: $'s \Rightarrow ('a \times 's, 'v)$ *tllist* $\Rightarrow ('a, 's, 'v)$ *tls* (‹⦇-, -⦈$_T$› [0, 0]) **where**
  $⦇s, xs⦈_T \equiv ⦇behavior.B\ s\ xs⦈_T$

⟨*ML*⟩

lemma *Sup-prime*:
  shows *Sup-prime* $⦇\omega⦈_T$
⟨*proof*⟩

lemma *nchotomy*:
  shows $\exists X \in behavior.stuttering.closed.\ x = \bigsqcup (tls.singleton\ `\ X)$
⟨*proof*⟩

lemmas *exhaust* = *bexE*[*OF tls.singleton.nchotomy*]

lemma *collapse*[*simp*]:
  shows $\bigsqcup (tls.singleton\ `\ \{\omega.\ ⦇\omega⦈_T \le P\}) = P$
⟨*proof*⟩

lemmas *not-bot* = *Sup-prime-not-bot*[*OF tls.singleton.Sup-prime*] — Non-triviality

⟨*ML*⟩

lemma *singleton-le-ext-conv*:
  shows $P \le Q \longleftrightarrow (\forall \omega.\ ⦇\omega⦈_T \le P \longrightarrow ⦇\omega⦈_T \le Q)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

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 $⦇\omega⦈_T = ⦇\omega'⦈_T \longleftrightarrow \omega \simeq_T \omega'$
⟨*proof*⟩

lemma *singleton-cong*:
  assumes $\omega \simeq_T \omega'$
  shows $⦇\omega⦈_T = ⦇\omega'⦈_T$
⟨*proof*⟩

⟨*ML*⟩

named-theorems *le-conv* ‹ *simplification rules for* ‹$⦇\sigma⦈_T \le$ *const* ...› ›

lemma *boolean-implication-le-conv*[*tls.singleton.le-conv*]:
  shows $⦇\sigma⦈_T \le P \longrightarrow_B Q \longleftrightarrow (⦇\sigma⦈_T \le P \longrightarrow ⦇\sigma⦈_T \le Q)$
⟨*proof*⟩

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 $\langle\!|s,\ xs|\!\rangle_T \le \langle\!|s,\ collapse\ s\ xs|\!\rangle_T$
    and $snd\ `\ set\ ys \subseteq \{s\} \Longrightarrow \langle\!|s,\ tshift\ ys\ xs|\!\rangle_T = \langle\!|s,\ xs|\!\rangle_T$
    and $\langle\!|s,\ TCons\ (a,\ s)\ xs|\!\rangle_T = \langle\!|s,\ xs|\!\rangle_T$
⟨*proof*⟩

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*]

⟨*ML*⟩

16.4 The idle process

The idle process contains no transitions and does not terminate.

⟨*ML*⟩

definition *idle* :: ('*a*, '*s*, '*v*) *behavior.t set* **where**
  $idle = (\bigcup s.\ raw.singleton\ (behavior.B\ s\ (trepeat\ (undefined,\ s))))$

lemma *idle-alt-def*:
  shows $raw.idle = \{\omega.\ \neg tfinite\ (behavior.rest\ \omega) \wedge behavior.sset\ \omega \subseteq \{behavior.init\ \omega\}\}$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *not-tfinite*:
  assumes $\omega \in raw.idle$
  shows $\neg tfinite\ (behavior.rest\ \omega)$
⟨*proof*⟩

⟨*ML*⟩

lemma *idle*[*iff*]:
  shows $raw.idle \in behavior.stuttering.closed$
⟨*proof*⟩

⟨*ML*⟩

lift-definition *idle* :: ('*a*, '*s*, '*v*) *tls* **is** *raw.idle* ⟨*proof*⟩

lemma *idle-alt-def*:
  shows $tls.idle = (\bigsqcup s.\ \langle\!|s,\ trepeat\ (undefined,\ s)|\!\rangle_T)$
⟨*proof*⟩

⟨*ML*⟩

lemma *idle-le-conv*[*tls.singleton.le-conv*]:
  shows $\langle\!|\omega|\!\rangle_T \le tls.idle \longleftrightarrow \neg tfinite\ (behavior.rest\ \omega) \wedge behavior.sset\ \omega \subseteq \{behavior.init\ \omega\}$
⟨*proof*⟩

⟨*ML*⟩

lemma *minimal-le*:
  shows $\langle\!|s,\ trepeat\ (undefined,\ s)|\!\rangle_T \leq tls.idle$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

definition *state-prop* :: $'s\ pred \Rightarrow ('a,\ 's,\ 'v)\ behavior.t\ set$ **where**
  $state\text{-}prop\ P = \{\omega.\ P\ (behavior.init\ \omega)\}$

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
  $always\ P = -raw.eventually\ (-P)$

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 $\bigwedge j.\ j < i \Longrightarrow$ *the* (*behavior.dropn* $j\ \omega$) $\in P$
shows $\omega \in$ *raw.until* $P\ Q$
⟨*proof*⟩

lemma *eventually-alt-def*:
shows *raw.eventually* $P = \{\omega\ .\ \exists \omega' \in P.\ \exists i.$ *behavior.dropn* $i\ \omega =$ *Some* $\omega'\}$
⟨*proof*⟩

lemma *always-alt-def*:
shows *raw.always* $P = \{\omega\ .\ \forall i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* $\omega' \longrightarrow \omega' \in P\}$
⟨*proof*⟩

lemma *alwaysI*:
assumes $\bigwedge i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* $\omega' \Longrightarrow \omega' \in P$
shows $\omega \in$ *raw.always* P
⟨*proof*⟩

lemma *alwaysD*:
assumes $\omega \in$ *raw.always* P
assumes *behavior.dropn* $i\ \omega =$ *Some* ω'
shows $\omega' \in P$
⟨*proof*⟩

⟨*ML*⟩

lemma *monotone*:
shows *mono raw.state-prop*
⟨*proof*⟩

lemma *simps*:
shows
raw.state-prop ⟨*False*⟩ = {}
raw.state-prop $\bot$ = {}
raw.state-prop ⟨*True*⟩ = *UNIV*
raw.state-prop $\top$ = *UNIV*
$-$ *raw.state-prop* P = *raw.state-prop* ($- P$)
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* ($P \longrightarrow_B Q$)
⟨*proof*⟩

lemma *Inf*:
shows *raw.state-prop* ($\bigsqcap X$) = $\bigcap$(*raw.state-prop* ‘ X)
⟨*proof*⟩

lemma *Sup*:
shows *raw.state-prop* ($\bigsqcup X$) = $\bigcup$(*raw.state-prop* ‘ X)
⟨*proof*⟩

⟨*ML*⟩

lemma *inf-always-le*:
fixes P :: ('a, 's, 'v) *behavior.t set*
assumes $P \in$ *behavior.stuttering.closed*
shows *raw.terminated* $\cap P \subseteq$ *raw.always* P
⟨*proof*⟩

⟨*ML*⟩

lemma *base*:
  shows $\omega \in Q \Longrightarrow \omega \in raw.until\ P\ Q$
    and $Q \subseteq raw.until\ P\ Q$
⟨*proof*⟩

lemma *step*:
  assumes $\omega \in P$
  assumes $behavior.tl\ \omega = Some\ \omega'$
  assumes $\omega' \in raw.until\ P\ Q$
  shows $\omega \in raw.until\ P\ Q$
⟨*proof*⟩

lemmas *intro*[*intro*] =
  raw.until.base
  raw.until.step

lemma *induct*[*case-names base step*, *consumes 1*, *induct set*: *raw.until*]:
  assumes $\omega \in raw.until\ P\ Q$
  assumes *base*: $\bigwedge\omega.\ \omega \in Q \Longrightarrow R\ \omega$
  assumes *step*: $\bigwedge\omega\ \omega'.\ [\![\omega \in P;\ behavior.tl\ \omega = Some\ \omega';\ \omega' \in raw.until\ P\ Q;\ R\ \omega']\!] \Longrightarrow R\ \omega$
  shows $R\ \omega$
⟨*proof*⟩

lemma *mono*:
  assumes $P \subseteq P'$
  assumes $Q \subseteq Q'$
  shows $raw.until\ P\ Q \subseteq raw.until\ P'\ Q'$
⟨*proof*⟩

lemma *botL*:
  shows $raw.until\ \{\}\ Q = Q$
⟨*proof*⟩

lemma *botR*:
  shows $raw.until\ P\ \{\} = \{\}$
⟨*proof*⟩

lemma *untilR*:
  shows $raw.until\ P\ (raw.until\ P\ Q) = raw.until\ P\ Q$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *InfL-not-empty*:
  assumes $X \neq \{\}$
  shows $raw.until\ (\bigcap X)\ Q = (\bigcap x{\in}X.\ raw.until\ x\ Q)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *SupR*:
  shows $raw.until\ P\ (\bigcup X) = \bigcup(raw.until\ P\ `\ X)$
⟨*proof*⟩

lemma *weakenL*:
  shows $raw.until\ UNIV\ P = raw.until\ (-\ P)\ P$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *implication-ordering-le*: — Warford et al. (2020, (16))
  shows $raw.until\ P\ Q \cap raw.until\ (-Q)\ R \subseteq raw.until\ P\ R$

⟨*proof*⟩

lemma *infR-ordering-le*: — Warford et al. (2020, (18))
  shows *raw.until P* (*Q* ∩ *R*) ⊆ *raw.until* (*raw.until P Q*) *R* (**is** *?lhs*⊆ *?rhs*)
⟨*proof*⟩

lemma *untilL*:
  shows *raw.until* (*raw.until P Q*) *Q* ⊆ *raw.until P Q* (**is** *?lhs*⊆ *?rhs*)
⟨*proof*⟩

lemma *alwaysR-le*:
  shows *raw.until P* (*raw.always Q*) ⊆ *raw.always* (*raw.until P Q*) (**is** *?lhs* ⊆ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *neg*:
  shows − (*raw.until P Q* ∪ *raw.always P*) = *raw.until* (− *Q*) (− *P* ∩ − *Q*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *terminated*:
  shows *raw.eventually raw.terminated* = {ω. *tfinite* (*behavior.rest* ω)} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *state-prop*[*intro*]:
  shows *raw.state-prop P* ∈ *behavior.stuttering.closed*
⟨*proof*⟩

lemma *terminated*[*intro*]:
  shows *raw.terminated* ∈ *behavior.stuttering.closed*
⟨*proof*⟩

lemma *until*[*intro*]:
  assumes *P* ∈ *behavior.stuttering.closed*
  assumes *Q* ∈ *behavior.stuttering.closed*
  shows *raw.until P Q* ∈ *behavior.stuttering.closed*
⟨*proof*⟩

lemma *eventually*[*intro*]:
  assumes *P* ∈ *behavior.stuttering.closed*
  shows *raw.eventually P* ∈ *behavior.stuttering.closed*
⟨*proof*⟩

lemma *always*[*intro*]:
  assumes *P* ∈ *behavior.stuttering.closed*
  shows *raw.always P* ∈ *behavior.stuttering.closed*
⟨*proof*⟩

⟨*ML*⟩

definition *valid* :: ('*a*, '*s*, '*v*) *tls* ⇒ *bool* **where**
  valid P ⟷ *P* = ⊤

lift-definition *state-prop* :: '*s pred* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *raw.state-prop* ⟨*proof*⟩

lift-definition *terminated* :: ('*a*, '*s*, '*v*) *tls* **is** *raw.terminated* ⟨*proof*⟩
lift-definition *until* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *raw.until* ⟨*proof*⟩

definition *eventually* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
eventually P = *tls.until* ⊤ *P*

definition *always* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
always P = −*tls.eventually* (−*P*)

definition *release* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
release P Q = −(*tls.until* (−*P*) (−*Q*))

definition *unless* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
unless P Q = *tls.until P Q* ⊔ *tls.always P*

abbreviation (*input*) *always-imp-syn* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
always-imp-syn P Q ≡ *tls.always* ($P \longrightarrow_B Q$)

abbreviation (*input*) *leads-to* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
leads-to P Q ≡ *tls.always-imp-syn P* (*tls.eventually Q*)

open-bundle *syntax*
begin
notation *tls.valid* (‹⊨ -› [*30*] *30*)
notation *tls.state-prop* (‹⟪-⟫› [*0*])
notation *tls.until* (**infix** ‹$\mathcal{U}$› *85*)
notation *tls.eventually* (‹◇-› [*87*] *87*)
notation *tls.always* (‹□-› [*87*] *87*)
notation *tls.release* (**infixr** ‹$\mathcal{R}$› *85*)
notation *tls.unless* (**infixr** ‹$\mathcal{W}$› *85*)
notation *tls.always-imp-syn* (**infixr** ‹$\longrightarrow_\Box$› *75*)
notation *tls.leads-to* (**infixr** ‹↝› *75*)
end

bundle *no-syntax*
begin
no-notation *tls.valid* (‹⊨ -› [*30*] *30*)
no-notation *tls.state-prop* (‹⟪-⟫› [*0*])
no-notation *tls.until* (**infixr** ‹$\mathcal{U}$› *85*)
no-notation *tls.eventually* (‹◇-› [*87*] *87*)
no-notation *tls.always* (‹□-› [*87*] *87*)
no-notation *tls.release* (**infixr** ‹$\mathcal{R}$› *85*)
no-notation *tls.unless* (**infixr** ‹$\mathcal{W}$› *85*)
no-notation *tls.always-imp-syn* (**infixr** ‹$\longrightarrow_\Box$› *75*)
no-notation *tls.leads-to* (**infixr** ‹↝› *75*)
end

lemma *validI*:
assumes ⊤ ≤ *P*
shows ⊨ *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *trans*[*trans*]:
assumes ⊨ *P*
assumes *P* ≤ *Q*
shows ⊨ *Q*

⟨proof⟩

lemma *mp*:
assumes $\models P \longrightarrow_B Q$
assumes $\models P$
shows $\models Q$
⟨proof⟩

lemmas *rev-mp* = *tls.valid.mp*[*rotated*]

⟨ML⟩

lemma *uminus-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!| \omega |\!\rangle_T \le -P \longleftrightarrow \neg \langle\!| \omega |\!\rangle_T \le P$
⟨proof⟩

lemma *state-prop-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!| \omega |\!\rangle_T \le$ *tls.state-prop* $P \longleftrightarrow P$ (*behavior.init* ω)
⟨proof⟩

lemma *terminated-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!| \omega |\!\rangle_T \le$ *tls.terminated* $\longleftrightarrow$ *tfinite* (*behavior.rest* ω) $\wedge$ *behavior.sset* $\omega \subseteq$ {*behavior.init* ω}
⟨proof⟩

lemma *until-le-conv*[*tls.singleton.le-conv*]:
fixes P :: ('*a*, '*s*, '*v*) *tls*
shows $\langle\!| \omega |\!\rangle_T \le P\ \mathcal{U}\ Q \longleftrightarrow (\exists\, i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* ω'
$\wedge \langle\!| \omega' |\!\rangle_T \le Q$
$\wedge (\forall\, j<i.\ \langle\!|$ *the* (*behavior.dropn* $j\ \omega$)$|\!\rangle_T \le P$)) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨proof⟩

lemma *eventually-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!| \omega |\!\rangle_T \le \Diamond P \longleftrightarrow (\exists\, i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* $\omega' \wedge \langle\!| \omega' |\!\rangle_T \le P)$
⟨proof⟩

lemma *always-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!| \omega |\!\rangle_T \le$ *tls.always* $P \longleftrightarrow (\forall\, i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* $\omega' \longrightarrow \langle\!| \omega' |\!\rangle_T \le P)$
⟨proof⟩

⟨ML⟩

interpretation *until*: *closure-complete-lattice-distributive-class tls.until P* **for** *P*
⟨proof⟩

⟨ML⟩

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* $(P\ \mathcal{U}\ Q)\ (P'\ \mathcal{U}\ Q')$
⟨*proof*⟩

lemma *SupL-le*:
shows $(\bigsqcup x{\in}X.\ x\ \mathcal{U}\ R) \le (\bigsqcup X)\ \mathcal{U}\ R$
⟨*proof*⟩

lemma *supL-le*:
shows $P\ \mathcal{U}\ R \sqcup Q\ \mathcal{U}\ R \le (P \sqcup Q)\ \mathcal{U}\ R$
⟨*proof*⟩

lemma *SupR*:
shows $P\ \mathcal{U}\ (\bigsqcup X) = \bigsqcup((\mathcal{U})\ P\ \text{`}\ X)$
⟨*proof*⟩

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 $P\ \mathcal{U}\ Q \sqcap (-Q)\ \mathcal{U}\ R \le P\ \mathcal{U}\ R$
⟨*proof*⟩

lemma *supL-ordering-le*: — Warford et al. (2020, (17))
shows $P\ \mathcal{U}\ (Q\ \mathcal{U}\ R) \le (P \sqcup Q)\ \mathcal{U}\ R$ (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *infR-ordering-le*: — Warford et al. (2020, (18))
shows $P\ \mathcal{U}\ (Q \sqcap R) \le (P\ \mathcal{U}\ Q)\ \mathcal{U}\ R$
⟨*proof*⟩

lemma *boolean-implication-distrib-le*: — Warford et al. (2020, (19))
shows $(P \longrightarrow_B Q)\ \mathcal{U}\ R \le (P\ \mathcal{U}\ R) \longrightarrow_B (Q\ \mathcal{U}\ R)$
⟨*proof*⟩

lemma *excluded-middleR*: — Warford et al. (2020, (23))
shows $\models P\ \mathcal{U}\ Q \sqcup P\ \mathcal{U}\ (-Q)$
⟨*proof*⟩

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*⟩

lemma *absorb*:
shows $P\ \mathcal{U}\ P = P$
⟨*proof*⟩

lemma *absorb-supL*: — Warford et al. (2020, (23))
shows $P \sqcup P\ \mathcal{U}\ Q = P \sqcup Q$
⟨*proof*⟩

lemma *absorb-supR*: — Warford et al. (2020, (23))
shows $Q \sqcup P\ \mathcal{U}\ Q = P\ \mathcal{U}\ Q$

⟨*proof*⟩

lemma *eventually-le*:
shows $P\ \mathcal{U}\ Q \le \Diamond Q$
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *sup-le*: — Warford et al. (2020, (28))
shows $P\ \mathcal{U}\ Q \le P \sqcup Q$
⟨*proof*⟩

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*⟩

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

⟨*ML*⟩

interpretation *eventually*: *closure-complete-lattice-distributive-class tls.eventually*
⟨*proof*⟩

lemma *eventually-alt-def*:
shows $\Diamond P = (-P)\ \mathcal{U}\ P$
⟨*proof*⟩

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.eventually tls.eventually*
⟨*proof*⟩

lemma *bot*:
shows $\Diamond\bot = \bot$
⟨*proof*⟩

lemma *bot-conv*:
shows $\Diamond P = \bot \longleftrightarrow P = \bot$
⟨*proof*⟩

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-le* = *tls.eventually.cl-Inf-le*
lemmas *inf-le* = *tls.eventually.cl-inf-le*

lemma *neg*:
  shows $-\Diamond P = \Box(-P)$
⟨*proof*⟩

lemma *boolean-implication-le*:
  shows $\Diamond P \longrightarrow_B \Diamond Q \le \Diamond(P \longrightarrow_B Q)$
⟨*proof*⟩

lemmas *simps* =
  tls.eventually.bot
  tls.eventually.top
  tls.eventually.expansive
  tls.eventually-def[*symmetric*]

lemma *terminated*:
  shows $\Diamond$*tls.terminated* = $\bigsqcup$(*tls.singleton* ‘ {ω. *tfinite* (*behavior.rest* ω)})
⟨*proof*⟩

lemma *always-imp-le*:
  shows $P \longrightarrow_\Box Q \le \Diamond P \longrightarrow_B \Diamond Q$
⟨*proof*⟩

lemma *until*:
  shows $\Diamond(P\ \mathcal{U}\ Q) = \Diamond Q$
⟨*proof*⟩

⟨*ML*⟩

lemma *always-alt-def*:
  shows $\Box P = P\ \mathcal{W}\ \bot$
⟨*proof*⟩

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.always tls.always*
⟨*proof*⟩

tls.always is an interior operator

lemma *idempotent*[*simp*]:
  shows $\Box\Box P = \Box P$
⟨*proof*⟩

lemma *contractive*:
  shows $\Box P \le P$
⟨*proof*⟩

lemma *monotone*[*iff*]:
  shows *mono tls.always*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tls.always.monotone*]

lemmas *mono*[*trans*] = *monoD*[*OF tls.always.monotone*]

lemma *bot*:
  shows $\Box\bot = \bot$
⟨*proof*⟩

lemma *top*:
  shows $\Box\top = \top$
⟨*proof*⟩

lemma *top-conv*:
  shows $\Box P = \top \longleftrightarrow P = \top$
⟨*proof*⟩

lemma *Sup-le*:
  shows $\bigsqcup(tls.always \text{ ` } X) \le \Box(\bigsqcup X)$
⟨*proof*⟩

lemma *sup-le*:
  shows $\Box P \sqcup \Box Q \le \Box(P \sqcup Q)$
⟨*proof*⟩

lemma *Inf*:
  shows $\Box(\bigsqcap X) = \bigsqcap(tls.always \text{ ` } X)$
⟨*proof*⟩

lemma *inf*:
  shows $\Box(P \sqcap Q) = \Box P \sqcap \Box Q$
⟨*proof*⟩

lemma *neg*:
  shows $-\Box P = \Diamond(-P)$
⟨*proof*⟩

lemma *always-necessitation*:
  assumes $\models P$
  shows $\models \Box P$
⟨*proof*⟩

lemma *valid-conv*:
  shows $\models \Box P \longleftrightarrow \models P$
⟨*proof*⟩

lemma *always-imp-le*:
  shows $P \longrightarrow_\Box Q \le \Box P \longrightarrow_B \Box Q$
⟨*proof*⟩

lemma *eventually-le*:
  shows $\Box P \le \Diamond P$
⟨*proof*⟩

lemma *not-until-le*: — Warford et al. (2020, (81))
  shows $\Box P \le -(Q\ \mathcal{U}\ (-P))$
⟨*proof*⟩

lemmas *simps* =
  tls.always.bot
  tls.always.top

tls.always.contractive
tls.always-alt-def[*symmetric*]

⟨*ML*⟩

lemma *until-alwaysR-le*: — Warford et al. (2020, (140))
shows $P\ \mathcal{U}\ \Box Q \leq \Box(P\ \mathcal{U}\ Q)$
⟨*proof*⟩

lemma *until-alwaysR*: — Warford et al. (2020, (141))
shows $P\ \mathcal{U}\ \Box P = \Box P$
⟨*proof*⟩

lemma *eventually-always-always-eventually-le*: — Warford et al. (2020, (145))
shows $\Diamond\Box P \leq \Box\Diamond P$
⟨*proof*⟩

lemma *always-inf-eventually-eventually-le*:
shows $\Box P \sqcap \Diamond Q \leq \Diamond(P \sqcap Q)$
⟨*proof*⟩

lemma *always-always-imp*: — Kröger and Merz (2008, §2.2: T33 frame)
shows $\models \Box P \longrightarrow_B \Box Q \longrightarrow_B \Box(P \sqcap Q)$
⟨*proof*⟩

lemma *always-eventually-imp*: — Kröger and Merz (2008, §2.2: T34 frame)
shows $\models \Box P \longrightarrow_B \Diamond Q \longrightarrow_B \Diamond(P \sqcap Q)$
⟨*proof*⟩

lemma *always-imp-always-generalization*: — Kröger and Merz (2008, §2.2: T35)
shows $\Box P \longrightarrow_\Box Q \leq \Box P \longrightarrow_B \Box Q$
⟨*proof*⟩

lemma *always-imp-eventually-generalization*: — Kröger and Merz (2008, §2.2: T36)
shows $P \longrightarrow_\Box \Diamond Q \leq \Diamond P \longrightarrow_B \Diamond Q$
⟨*proof*⟩

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 $\Diamond\Box\Diamond P = \Box\Diamond P$
⟨*proof*⟩

lemma *eventually-always-eventually-absorption*: — Kröger and Merz (2008, §2.2: T38)
shows $\Box\Diamond\Box P = \Diamond\Box P$
⟨*proof*⟩

lemma *always-imp-always-eventually-le*: — Warford et al. (2020, (157))
shows $P \longrightarrow_\Box Q \leq \Box\Diamond P \longrightarrow_B \Box\Diamond Q$
⟨*proof*⟩

lemma *always-imp-eventually-always-le*: — Warford et al. (2020, (158))
shows $P \longrightarrow_\Box Q \leq \Diamond\Box P \longrightarrow_B \Diamond\Box Q$
⟨*proof*⟩

lemma *always-eventually-inf-le*:
shows $\Box\Diamond(P \sqcap Q) \leq \Box\Diamond P \sqcap \Box\Diamond Q$ — Warford et al. (2020, (159))
⟨*proof*⟩

lemma *eventually-always-sup-le*:
  shows $\Diamond\Box P \sqcap \Diamond\Box Q \leq \Diamond\Box(P \sqcup Q)$ — Warford et al. (2020, (160))
⟨*proof*⟩

lemma *always-eventually-sup*: — Warford et al. (2020, (161))
  fixes P :: $('a, 's, 'v)$ *tls*
  shows $\Box\Diamond(P \sqcup Q) = \Box\Diamond P \sqcup \Box\Diamond Q$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *eventually-always-inf*: — Warford et al. (2020, (162))
  shows $\Diamond\Box(P \sqcap Q) = \Diamond\Box P \sqcap \Diamond\Box Q$
⟨*proof*⟩

lemma *eventual-latching*: — Warford et al. (2020, (163))
  shows $\Diamond\Box(P \longrightarrow_B \Box Q) = \Diamond\Box(-P) \sqcup \Diamond\Box Q$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)))
      ($\lambda P\ Q.$ *raw.until* $P\ Q \cup$ *raw.always* P)
      tls.unless
⟨*proof*⟩

lemma *neg*: — Warford et al. (2020, (170))
  shows $-(P\ \mathcal{W}\ Q) = (-Q)\ \mathcal{U}\ (-P \sqcap -Q)$
⟨*proof*⟩

lemma *alwaysR-le*: — Warford et al. (2020, (177))
  shows $P\ \mathcal{W}\ \Box Q \leq \Box(P\ \mathcal{W}\ Q)$
⟨*proof*⟩

lemma *sup-le*: — Warford et al. (2020, (206))
  shows $P\ \mathcal{W}\ Q \leq P \sqcup Q$
⟨*proof*⟩

lemma *ordering*: — Warford et al. (2020, (252))
  shows $\models (-P)\ \mathcal{W}\ Q \sqcup (-Q)\ \mathcal{W}\ P$
⟨*proof*⟩

⟨*ML*⟩

lemma *eq-unless-inf-eventually*:
  shows $P\ \mathcal{U}\ Q = (P\ \mathcal{W}\ Q) \sqcap \Diamond Q$
⟨*proof*⟩

lemma *always-strengthen-le*: — Warford et al. (2020, (83))
  shows $\Box P \sqcap (Q\ \mathcal{U}\ R) \leq (P \sqcap Q)\ \mathcal{U}\ (P \sqcap R)$
⟨*proof*⟩

lemma *until-weakI*:
  shows $\Box P \sqcap \Diamond Q \leq P\ \mathcal{U}\ Q$ (**is** *?lhs* ≤ *?rhs*) — Warford et al. (2020, (84))
⟨*proof*⟩

lemma *always-impL*: — Warford et al. (2020, (86))
  shows $P \longrightarrow_{\Box} P' \sqcap P\ \mathcal{U}\ Q \leq P'\ \mathcal{U}\ Q$ (**is** *?thesis1*)
    and $P\ \mathcal{U}\ Q \sqcap P \longrightarrow_{\Box} P' \leq P'\ \mathcal{U}\ Q$ (**is** *?thesis2*)

⟨*proof*⟩

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*⟩

lemma *neg*: — Warford et al. (2020, (173))
  shows $-(P \; \mathcal{U} \; Q) = (-Q) \; \mathcal{W} \; (-P \sqcap -Q)$
⟨*proof*⟩

⟨*ML*⟩

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* $(\bigsqcup X) = \bigsqcup(tls.state\text{-}prop \; ` \; X)$
⟨*proof*⟩

lemma *Inf*:
  shows *tls.state-prop* $(\bigsqcap X) = \bigsqcap(tls.state\text{-}prop \; ` \; X)$
⟨*proof*⟩

lemmas *simps* = *raw.state-prop.simps*[*transferred*]

⟨*ML*⟩

lemma *not-bot*:
  shows *tls.terminated* $\neq \bot$
⟨*proof*⟩

lemma *not-top*:
  shows *tls.terminated* $\neq \top$
⟨*proof*⟩

lemma *inf-always*:
  shows *tls.terminated* $\sqcap \square P =$ *tls.terminated* $\sqcap P$
⟨*proof*⟩

lemma *always-le-conv*:
  shows *tls.terminated* $\leq \square P \longleftrightarrow$ *tls.terminated* $\leq P$
⟨*proof*⟩

lemma *inf-eventually*:
  shows *tls.terminated* $\sqcap \Diamond P =$ *tls.terminated* $\sqcap P$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *eventually-le-conv*:
  shows *tls.terminated* $\leq$ *tls.eventually* $P \longleftrightarrow$ *tls.terminated* $\leq P$
⟨*proof*⟩

lemma *eq-always-terminated*:
  shows *tls.terminated* = $\square$*tls.terminated*
⟨*proof*⟩

⟨*ML*⟩

16.5.1 Leads-to and leads-to-via

So-called *response* properties are of the form $P \longrightarrow_{\Box} \Diamond Q$ (pronounced "P leads to Q", written $P \leadsto Q$) (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 $P \longrightarrow_{\Box} I \,\mathcal{U}\, Q$ to establish liveness properties in a sequential setting.

lemma *leads-to-refl*:
  shows $\models P \leadsto P$
⟨*proof*⟩

lemma *leads-to-mono*:
  assumes $P' \leq P$
  assumes $Q \leq Q'$
  shows $P \leadsto Q \leq P' \leadsto Q'$
⟨*proof*⟩

lemma *leads-to-supL*:
  shows $(P \leadsto R) \sqcap (Q \leadsto R) \leq (P \sqcup Q) \leadsto R$
⟨*proof*⟩

lemma *always-imp-leads-to*:
  shows $P \longrightarrow_{\Box} Q \leq P \leadsto Q$
⟨*proof*⟩

lemma *leads-to-eventually*:
  shows $\Diamond P \sqcap (P \leadsto Q) \leq \Diamond Q$
⟨*proof*⟩

lemma *leads-to-leads-to-via*:
  shows $P \longrightarrow_{\Box} Q \,\mathcal{U}\, R \leq P \leadsto R$
⟨*proof*⟩

lemma *leads-to-trans*:
  shows $P \leadsto Q \sqcap Q \leadsto R \leq P \leadsto R$ (**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

lemma *leads-to-via-weakenR*:
  shows $Q \longrightarrow_{\Box} Q' \sqcap P \longrightarrow_{\Box} I \,\mathcal{U}\, Q \leq P \longrightarrow_{\Box} I \,\mathcal{U}\, Q'$
⟨*proof*⟩

lemma *leads-to-via-supL*: — useful for case distinctions
  shows $P \longrightarrow_{\Box} I \,\mathcal{U}\, Q \sqcap P' \longrightarrow_{\Box} I' \,\mathcal{U}\, Q \leq P \sqcup P' \longrightarrow_{\Box} (I \sqcup I') \,\mathcal{U}\, Q$
⟨*proof*⟩

lemma *leads-to-via-trans*:
  shows $(P \longrightarrow_{\Box} I \,\mathcal{U}\, Q) \sqcap (Q \longrightarrow_{\Box} I' \,\mathcal{U}\, R) \leq P \longrightarrow_{\Box} (I \sqcup I') \,\mathcal{U}\, R$ (**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

lemma *leads-to-via-disj*: — more like a chaining rule
  shows $(P \longrightarrow_{\Box} I \,\mathcal{U}\, Q) \sqcap (Q \longrightarrow_{\Box} I' \,\mathcal{U}\, R) \leq (P \sqcup Q \longrightarrow_{\Box} (I \sqcup I') \,\mathcal{U}\, R)$
⟨*proof*⟩

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* :: $('a, 's, 'v)$ *tls* $\Rightarrow$ $('a, 's, 'v)$ *tls* $\Rightarrow$ $('a, 's, 'v)$ *tls* **where**

$weakly\text{-}fair\ enabled\ taken = \Box enabled \longrightarrow_{\Box} \Diamond taken$

lemma *weakly-fair-def2*:
shows $tls.weakly\text{-}fair\ enabled\ taken = \Box(-(\Box(enabled \sqcap -taken)))$
⟨*proof*⟩

lemma *weakly-fair-def3*:
shows $tls.weakly\text{-}fair\ enabled\ taken = \Diamond\Box enabled \longrightarrow_B \Box\Diamond taken$
⟨*proof*⟩

lemma *weakly-fair-def4*:
shows $tls.weakly\text{-}fair\ enabled\ taken = \Box\Diamond(enabled \longrightarrow_B taken)$
⟨*proof*⟩

⟨*ML*⟩

lemma *mono*:
assumes $P' \leq P$
assumes $Q \leq Q'$
shows $tls.weakly\text{-}fair\ P\ Q \leq tls.weakly\text{-}fair\ P'\ Q'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes $st\text{-}ord\ (\neg F)\ P\ P'$
assumes $st\text{-}ord\ F\ Q\ Q'$
shows $st\text{-}ord\ F\ (tls.weakly\text{-}fair\ P\ Q)\ (tls.weakly\text{-}fair\ P'\ Q')$
⟨*proof*⟩

lemma *weakly-fair-triv*:
shows $\Box\Diamond(-enabled) \leq tls.weakly\text{-}fair\ enabled\ taken$
⟨*proof*⟩

lemma *mp*:
shows $tls.weakly\text{-}fair\ enabled\ taken \sqcap \Box enabled \leq \Diamond taken$
⟨*proof*⟩

⟨*ML*⟩

lemma *weakly-fair*:
shows $\Box(tls.weakly\text{-}fair\ enabled\ taken) = tls.weakly\text{-}fair\ enabled\ taken$
⟨*proof*⟩

⟨*ML*⟩

lemma *weakly-fair*:
shows $\Diamond(tls.weakly\text{-}fair\ enabled\ taken) = tls.weakly\text{-}fair\ enabled\ taken$
⟨*proof*⟩

⟨*ML*⟩

Similarly for strong fairness. van Glabbeek and Höfner (2019) call this "response to persistence" as a generalisation of strong fairness.

definition $strongly\text{-}fair :: ('a, 's, 'v)\ tls \Rightarrow ('a, 's, 'v)\ tls \Rightarrow ('a, 's, 'v)\ tls$ **where**
$strongly\text{-}fair\ enabled\ taken = \Box\Diamond enabled \longrightarrow_{\Box} \Diamond taken$

lemma *strongly-fair-def2*:
shows $tls.strongly\text{-}fair\ enabled\ taken = \Box(-\Box(\Diamond enabled \sqcap -taken))$
⟨*proof*⟩

lemma *strongly-fair-def3*:
  shows *tls.strongly-fair enabled taken* $=$ $\Box\Diamond$*enabled* $\longrightarrow_B$ $\Box\Diamond$*taken*
⟨*proof*⟩

⟨*ML*⟩

lemma *mono*:
  assumes $P' \leq P$
  assumes $Q \leq Q'$
  shows *tls.strongly-fair* P Q $\leq$ *tls.strongly-fair* P' Q'
⟨*proof*⟩

lemma *strengthen*[*strg*]:
  assumes *st-ord* $(\neg F)$ P P'
  assumes *st-ord* F Q Q'
  shows *st-ord* F (*tls.strongly-fair* P Q) (*tls.strongly-fair* P' Q')
⟨*proof*⟩

lemma *supL*: — does not hold for *tls.weakly-fair*
  shows *tls.strongly-fair* (*enabled1* $\sqcup$ *enabled2*) *taken*
    $=$ (*tls.strongly-fair enabled1 taken* $\sqcap$ *tls.strongly-fair enabled2 taken*)
⟨*proof*⟩

lemma *weakly-fair-le*:
  shows *tls.strongly-fair enabled taken* $\leq$ *tls.weakly-fair enabled taken*
⟨*proof*⟩

lemma *always-enabled-weakly-fair-strongly-fair*:
  shows $\Box$*enabled* $\leq$ *tls.weakly-fair enabled taken* $\longleftrightarrow_B$ *tls.strongly-fair enabled taken*
⟨*proof*⟩

⟨*ML*⟩

lemma *strongly-fair*:
  shows $\Box$(*tls.strongly-fair enabled taken*) $=$ *tls.strongly-fair enabled taken*
⟨*proof*⟩

⟨*ML*⟩

lemma *strongly-fair*:
  shows $\Diamond$(*tls.strongly-fair enabled taken*) $=$ *tls.strongly-fair enabled taken*
⟨*proof*⟩

⟨*ML*⟩

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 $('a, 's, 'v)$ *spec*.
Observations:

- our safety closure for $('a, 's, 'v)$ *tls* introduces infinite sequences to stand for the prefixes in $('a, 's, 'v)$ *spec*
  - i.e., the non-termination of trace σ (*trace.term* σ = *None*) is represented by a behavior ending with *trace.final* σ infinitely stuttered
  - Abadi and Lamport (1991, §2.1) consider these behaviors to represent terminating processes

⟨*ML*⟩

definition *to-spec* :: $('a, 's, 'v)$ *behavior.t set* ⇒ $('a, 's, 'v)$ *trace.t set* **where**
  to-spec T = {*behavior.take* i ω |ω i. $\omega \in T$}

definition *from-spec* :: $('a, 's, 'v)$ *trace.t set* ⇒ $('a, 's, 'v)$ *behavior.t set* **where**
  from-spec S = {ω . ∀ i. *behavior.take* i $\omega \in S$}

interpretation *safety*: *galois.powerset raw.to-spec raw.from-spec*
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
  shows *raw.from-spec* {} = {}
⟨*proof*⟩

lemma *singleton*:
  shows *raw.from-spec* (*Safety-Logic.raw.singleton* σ)
    = ⋃(*raw.singleton* ‘ {ω . ∀ i. *behavior.take* i ω ∈ *Safety-Logic.raw.singleton* σ}) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *sup*:
  assumes P ∈ *raw.spec.closed*
  assumes Q ∈ *raw.spec.closed*
  shows *raw.from-spec* ($P \cup Q$) = *raw.from-spec* P ∪ *raw.from-spec* Q
⟨*proof*⟩

⟨*ML*⟩

lemma *singleton*:
  shows *raw.to-spec* (*TLS.raw.singleton* ω)
    = (⋃i. *Safety-Logic.raw.singleton* (*behavior.take* i ω)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cl-altI*:
  assumes ⋀i. $\exists \omega' \in P$. *behavior.take* i ω = *behavior.take* i ω'
  shows ω ∈ *raw.safety.cl* P
⟨*proof*⟩

lemma *cl-altE*:
  assumes ω ∈ *raw.safety.cl* P
  obtains ω' **where** $\omega' \in P$ **and** *behavior.take* i ω = *behavior.take* i ω'
⟨*proof*⟩

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 $raw.safety.cl\ P = \{\omega.\ \forall i.\ \exists \beta.\ behavior.take\ i\ \omega\ @-_B\ \beta \in P\}$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *closed-alt-def*: — If ω is not in P then some prefix of ω has irretrievably gone wrong
shows $raw.safety.closed = \{P.\ \forall \omega.\ \omega \notin P \longrightarrow (\exists i.\ \forall \beta.\ behavior.take\ i\ \omega\ @-_B\ \beta \notin P)\}$
⟨*proof*⟩

lemma *closed-alt-def2*: — Contraposition gives the customary prefix-closure definition
shows $raw.safety.closed = \{P.\ \forall \omega.\ (\forall i.\ \exists \beta.\ behavior.take\ i\ \omega\ @-_B\ \beta \in P) \longrightarrow \omega \in P\}$
⟨*proof*⟩

lemma *closedI2*:
assumes $\bigwedge \omega.\ (\bigwedge i.\ \exists \beta.\ behavior.take\ i\ \omega\ @-_B\ \beta \in P) \Longrightarrow \omega \in P$
shows $P \in raw.safety.closed$
⟨*proof*⟩

lemma *closedE2*:
assumes $P \in raw.safety.closed$
assumes $\bigwedge i.\ \omega \notin P \Longrightarrow \exists \beta.\ behavior.take\ i\ \omega\ @-_B\ \beta \in P$
shows $\omega \in P$
⟨*proof*⟩

⟨*ML*⟩

lemma *state-prop*:
shows $raw.safety.cl\ (raw.state\text{-}prop\ P) = raw.state\text{-}prop\ P$
⟨*proof*⟩

lemma *terminated-iff*:
assumes $\omega \in raw.terminated$
shows $\omega \in raw.safety.cl\ P \longleftrightarrow \omega \in P$ (**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma *terminated*:
shows $raw.safety.cl\ raw.terminated = raw.idle \cup raw.terminated$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *le-terminated-bot*:
assumes $P \in behavior.stuttering.closed$
assumes $raw.safety.cl\ P \subseteq raw.terminated$
shows $P = \{\}$
⟨*proof*⟩

lemma *always-le*:
shows $raw.safety.cl\ (raw.always\ P) \subseteq raw.always\ (raw.safety.cl\ P)$
⟨*proof*⟩

lemma *eventually*:
assumes $P \neq \bot$
shows $raw.safety.cl\ (raw.eventually\ P)$
$= -raw.eventually\ raw.terminated \cup raw.eventually\ P$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

⟨*ML*⟩

lemma *always-eventually*:
assumes $P \in raw.safety.closed$
assumes $\forall i.\ \exists j{\geq}i.\ \exists \beta.\ behavior.take\ j\ \omega\ @-_B\ \beta \in P$

shows $\omega \in P$
⟨*proof*⟩

lemma *sup*:
assumes $P \in$ *raw.safety.closed*
assumes $Q \in$ *raw.safety.closed*
shows $P \cup Q \in$ *raw.safety.closed*
⟨*proof*⟩

lemma *unless*: — Sistla (1994, §3.1) – minimality is irrelevant
assumes $P \in$ *raw.safety.closed*
assumes $Q \in$ *raw.safety.closed*
shows *raw.unless* P $Q \in$ *raw.safety.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *to-spec*:
shows *range raw.to-spec* $\subseteq$ *downwards.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *to-spec*:
shows *raw.to-spec* ‘ *behavior.stuttering.closed* $\subseteq$ *trace.stuttering.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *to-spec*:
shows *raw.to-spec* ‘ *behavior.stuttering.closed* $\subseteq$ *raw.spec.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *from-spec*:
shows *raw.from-spec* ‘ *trace.stuttering.closed*
$\subseteq$ (*behavior.stuttering.closed* :: ('*a*, '*s*, '*v*) *behavior.t set set*)
⟨*proof*⟩

lemma *safety-cl*:
assumes $P \in$ *behavior.stuttering.closed*
shows *raw.safety.cl* $P \in$ *behavior.stuttering.closed*
⟨*proof*⟩

⟨*ML*⟩

lift-definition *to-spec* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *raw.to-spec*
⟨*proof*⟩

lift-definition *from-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *raw.from-spec*
⟨*proof*⟩

interpretation *safety*: *galois.complete-lattice-class tls.to-spec tls.from-spec*
⟨*proof*⟩

⟨*ML*⟩

lemma *singleton*:
notes *spec.singleton.transfer*[*transfer-rule*]
shows *tls.from-spec* (*spec.singleton* σ)
$= \bigsqcup$(*tls.singleton* ‘ {ω . $\forall$ *i*. *behavior.take* *i* $\omega \in$ *Safety-Logic.raw.singleton* σ})
⟨*proof*⟩

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
⟨*proof*⟩

lemmas *Inf* = *tls.safety.upper-Inf*
lemmas *inf* = *tls.safety.upper-inf*

⟨*ML*⟩

lemma *singleton*:
notes *spec.singleton.transfer*[*transfer-rule*]
shows *tls.to-spec* (*tls.singleton* ω) = ($\bigsqcup$*i*. *spec.singleton* (*behavior.take* *i* ω))
⟨*proof*⟩

lemmas *bot* = *tls.safety.lower-bot*

lemmas *Sup* = *tls.safety.lower-Sup*
lemmas *sup* = *tls.safety.lower-sup*

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.safety.cl* *tls.safety.cl*
⟨*proof*⟩

lemma *bot*[*iff*]:
shows *tls.safety.cl* $\bot = \bot$
⟨*proof*⟩

lemma *sup*:
shows *tls.safety.cl* ($P \sqcup Q$) = *tls.safety.cl* $P \sqcup$ *tls.safety.cl* Q
⟨*proof*⟩

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 $P \neq \bot$
shows *tls.safety.cl* ($\Diamond P$) = $-\Diamond$*tls.terminated* $\sqcup \Diamond P$
⟨*proof*⟩

lemma *terminated-iff*:
assumes $\langle\!|\omega|\!\rangle_T \le$ *tls.terminated*
shows $\langle\!|\omega|\!\rangle_T \le$ *tls.safety.cl* $P \longleftrightarrow \langle\!|\omega|\!\rangle_T \le P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *terminated*:
shows *tls.safety.cl* *tls.terminated* = *tls.idle* $\sqcup$ *tls.terminated*
⟨*proof*⟩

lemma *not-terminated*:
shows $tls.safety.cl\ (-\ tls.terminated) = -\ tls.terminated$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *le-terminated-conv*:
shows $tls.safety.cl\ P \leq tls.terminated \longleftrightarrow P = \bot$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
shows *rel-set* (*pcr-tls* (=) (=) (=))
(*behavior.stuttering.closed* ∩ *raw.safety.closed*)
tls.safety.closed (**is** *rel-set* - *?lhs* *?rhs*)
⟨*proof*⟩

lemma *bot*:
shows $\bot \in tls.safety.closed$
⟨*proof*⟩

lemma *sup*:
assumes $P \in tls.safety.closed$
assumes $Q \in tls.safety.closed$
shows $P \sqcup Q \in tls.safety.closed$
⟨*proof*⟩

lemmas *inf* = *tls.safety.closed-inf*

lemma *boolean-implication*:
assumes $-P \in tls.safety.closed$
assumes $Q \in tls.safety.closed$
shows $P \longrightarrow_B Q \in tls.safety.closed$
⟨*proof*⟩

lemma *state-prop*:
shows $tls.state\text{-}prop\ P \in tls.safety.closed$
⟨*proof*⟩

lemma *not-terminated*:
shows $-\ tls.terminated \in tls.safety.closed$
⟨*proof*⟩

lemma *unless*:
assumes $P \in tls.safety.closed$
assumes $Q \in tls.safety.closed$
shows $tls.unless\ P\ Q \in tls.safety.closed$
⟨*proof*⟩

lemma *always*:
assumes $P \in tls.safety.closed$
shows $tls.always\ P \in tls.safety.closed$
⟨*proof*⟩

⟨*ML*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

lemma *to-spec-le-conv*[*tls.singleton.le-conv*]:
notes *spec.singleton.transfer*[*transfer-rule*]
shows $⦇\sigma⦈ \leq$ *tls.to-spec P* $\longleftrightarrow$ ($\exists\,\omega\ i.\ ⦇\omega⦈_T \leq P \wedge \sigma =$ *behavior.take i* ω)
⟨*proof*⟩

lemma *from-spec-le-conv*[*tls.singleton.le-conv*]:
notes *spec.singleton.transfer*[*transfer-rule*]
shows $⦇\omega⦈_T \leq$ *tls.from-spec P* $\longleftrightarrow$ ($\forall\,i.$ ⦇*behavior.take i* ω⦈ $\leq P$)
⟨*proof*⟩

lemma *safety-cl-le-conv*[*tls.singleton.le-conv*]:
shows $⦇\omega⦈_T \leq$ *tls.safety.cl P* $\longleftrightarrow$ ($\forall\,i.\ \exists\,\omega'.\ ⦇\omega'⦈_T \leq P \wedge$ *behavior.take i* $\omega =$ *behavior.take i* ω')
⟨*proof*⟩

⟨*ML*⟩

16.7 Maps

⟨*ML*⟩

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('b, 't, 'w)$ *tls* **where**
map af sf vf P = $\bigsqcup$(*tls.singleton* ‘ *behavior.map af sf vf* ‘ $\{\sigma.\ ⦇\sigma⦈_T \leq P\}$)

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w)$ *tls* $\Rightarrow ('a, 's, 'v)$ *tls* **where**
invmap af sf vf P = $\bigsqcup$(*tls.singleton* ‘ *behavior.map af sf vf* −‘ $\{\sigma.\ ⦇\sigma⦈_T \leq P\}$)

abbreviation *amap* ::$('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('b, 's, 'v)$ *tls* **where**
amap af $\equiv$ *tls.map af id id*
abbreviation *ainvmap* ::$('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v)$ *tls* $\Rightarrow ('a, 's, 'v)$ *tls* **where**
ainvmap af $\equiv$ *tls.invmap af id id*
abbreviation *smap* ::$('s \Rightarrow 't) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('a, 't, 'v)$ *tls* **where**
smap sf $\equiv$ *tls.map id sf id*
abbreviation *sinvmap* ::$('s \Rightarrow 't) \Rightarrow ('a, 't, 'v)$ *tls* $\Rightarrow ('a, 's, 'v)$ *tls* **where**
sinvmap sf $\equiv$ *tls.invmap id sf id*
abbreviation *vmap* ::$('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('a, 's, 'w)$ *tls* **where** — aka *liftM*
vmap vf $\equiv$ *tls.map id id vf*
abbreviation *vinvmap* ::$('v \Rightarrow 'w) \Rightarrow ('a, 's, 'w)$ *tls* $\Rightarrow ('a, 's, 'v)$ *tls* **where**
vinvmap vf $\equiv$ *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*⟩

⟨*ML*⟩

lemma *map-le-conv*[*tls.singleton.le-conv*]:
shows $⦇\omega⦈_T \leq$ *tls.map af sf vf P* $\longleftrightarrow$ ($\exists\,\omega'.\ ⦇\omega'⦈_T \leq P \wedge ⦇\omega⦈_T \leq$ ⦇*behavior.map af sf vf* $\omega'⦈_T$)
⟨*proof*⟩

lemma *invmap-le-conv*[*tls.singleton.le-conv*]:
shows $⦇\omega⦈_T \leq$ *tls.invmap af sf vf P* $\longleftrightarrow$ ⦇*behavior.map af sf vf* $\omega⦈_T \leq P$
⟨*proof*⟩

⟨ML⟩

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* $\langle\!\langle\omega\rangle\!\rangle_T$ = $\langle\!\langle$*behavior.map af sf vf* $\omega\rangle\!\rangle_T$
⟨*proof*⟩

lemma *top*:
  assumes *surj af*
  assumes *surj sf*
  assumes *surj vf*
  shows *tls.map af sf vf* ⊤ = ⊤
⟨*proof*⟩

lemma *id*:
  shows *tls.map id id id P* = *P*
    and *tls.map* (λ*x*. *x*) (λ*x*. *x*) (λ*x*. *x*) *P* = *P*
⟨*proof*⟩

lemma *comp*:
  shows *tls.map af sf vf* ∘ *tls.map ag sg vg* = *tls.map* (*af* ∘ *ag*) (*sf* ∘ *sg*) (*vf* ∘ *vg*) (**is** *?lhs* = *?rhs*)
    and *tls.map af sf vf* (*tls.map ag sg vg P*) = *tls.map* (λ*a*. *af* (*ag a*)) (λ*s*. *sf* (*sg s*)) (λ*v*. *vf* (*vg v*)) *P* (**is** *?thesis1*)
⟨*proof*⟩

lemmas *map* = *tls.map.comp*

⟨ML⟩

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* $\langle\!\langle\omega\rangle\!\rangle_T = \bigsqcup$(*tls.singleton* ‘ {ω′. $\langle\!\langle$*behavior.map af sf vf* $\omega'\rangle\!\rangle_T \leq \langle\!\langle\omega\rangle\!\rangle_T$})
⟨*proof*⟩

lemma *id*:

shows *tls.invmap id id id P* = *P*
and *tls.invmap* (λ*x*. *x*) (λ*x*. *x*) (λ*x*. *x*) *P* = *P*
⟨*proof*⟩

lemma *comp*:
shows *tls.invmap af sf vf* (*tls.invmap ag sg vg P*) = *tls.invmap* (λ*x*. *ag* (*af x*)) (λ*s*. *sg* (*sf s*)) (λ*v*. *vg* (*vf v*)) *P* (**is** *?lhs P* = *?rhs P*)
and *tls.invmap af sf vf* ∘ *tls.invmap ag sg vg* = *tls.invmap* (*ag* ∘ *af*) (*sg* ∘ *sf*) (*vg* ∘ *vf*) (**is** *?thesis1*)
⟨*proof*⟩

lemmas *invmap* = *tls.invmap.comp*

⟨*ML*⟩

lemma *map*:
shows *tls.to-spec* (*tls.map af sf vf P*) = *spec.map af sf vf* (*tls.to-spec P*)
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

lemma *Ax1*:
shows $\models \Box(P \longrightarrow_B Q) \longrightarrow_B \Box P \longrightarrow_B \Box Q$
⟨*proof*⟩

lemma *Ax2*:
shows $\models \Box P \longrightarrow_B P$
⟨*proof*⟩

lemma *Ax3*:
shows $\models \Box P \longrightarrow_B \Box\Box P$
⟨*proof*⟩

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*⟩

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*⟩

lemma *Ax6*:
  assumes $\models P$
  shows $\models \Box P$
⟨*proof*⟩

lemma *Ax8*:
  assumes $\models P$
  assumes $\models P \longrightarrow_B Q$
  shows $\models Q$
⟨*proof*⟩

⟨*ML*⟩

16.9 Tweak syntax

unbundle *tls.no-syntax*
no-notation *tls.singleton* (‹⦇-⦈$_T$›)

⟨*ML*⟩

bundle *extra-syntax*
begin
notation *tls.singleton* (‹⦇-⦈$_T$› [*0*])
notation *tls.from-spec* (‹⦇-⦈› [*0*])
end

⟨*ML*⟩

17 Atomic sections

By restricting the environment to stuttering steps we can consider arbitrary processes to be atomic, i.e., free of interference.

⟨*ML*⟩

definition *atomic* :: $'a \Rightarrow ('a, 's, 'v)$ *spec* $\Rightarrow ('a, 's, 'v)$ *spec* **where**
  atomic a P $=$ *P* $\sqcap$ *spec.rel* $(\{a\} \times$ *UNIV*$)$

⟨*ML*⟩

lemma *atomic-le-conv*[*spec.idle-le*]:
  shows *spec.idle* $\leq$ *spec.atomic a P* $\longleftrightarrow$ *spec.idle* $\leq$ *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *atomic*:
  shows *spec.term.none* (*spec.atomic a P*) $=$ *spec.atomic a* (*spec.term.none P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *atomic*:
  shows *spec.term.all* (*spec.atomic a P*) $=$ *spec.atomic a* (*spec.term.all P*)
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*[*simp*]:
  shows *spec.atomic a* $\bot = \bot$
⟨*proof*⟩

lemma *top*[*simp*]:
  shows *spec.atomic a* $\top$ = *spec.rel* ({*a*} × *UNIV*)
⟨*proof*⟩

lemma *contractive*:
  shows *spec.atomic a P* $\leq$ *P*
⟨*proof*⟩

lemma *idempotent*[*simp*]:
  shows *spec.atomic a* (*spec.atomic a P*) = *spec.atomic a P*
⟨*proof*⟩

lemma *monotone*:
  shows *mono* (*spec.atomic a*)
⟨*proof*⟩

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 *spec.atomic a* ($\bigsqcup$*X*) = $\bigsqcup$(*spec.atomic a* ‘ *X*)
⟨*proof*⟩

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* ($\leq$) (λx. *spec.atomic a* (*P x*))
⟨*proof*⟩

lemma *Inf-not-empty*:
  assumes *X* $\neq$ {}
  shows *spec.atomic a* ($\bigsqcap$*X*) = $\bigsqcap$(*spec.atomic a* ‘ *X*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *action*:
  shows *spec.atomic a* (*spec.action F*) = *spec.action* (*F* ∩ *UNIV* × ({*a*} × *UNIV* ∪ *UNIV* × *Id*))
⟨*proof*⟩

lemma *return*:
  shows *spec.atomic a* (*spec.return v*) = *spec.return v*
⟨*proof*⟩

lemma *bind*:

shows *spec.atomic a* $(f \gg= g)$ = *spec.atomic a f* $\gg=$ $(\lambda v.$ *spec.atomic a* $(g\ v))$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rel*:
shows *spec.atomic a* (*spec.rel r*) = *spec.rel* $(r \cap \{a\} \times UNIV)$
⟨*proof*⟩

lemma *interference*:
shows *spec.atomic* (*proc a*) (*spec.rel* $(\{env\} \times UNIV)$) = *spec.rel* {}
⟨*proof*⟩

⟨*ML*⟩

lemma *cl*:
shows *spec.atomic* (*proc a*) (*spec.cam.cl* $(\{env\} \times UNIV)$ *P*) = *spec.atomic* (*proc a*) *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *cl*:
shows *spec.atomic* (*proc a*) (*spec.interference.cl* $(\{env\} \times UNIV)$ *P*) = *spec.return* () $\gg$ *spec.atomic* (*proc a*) *P*
⟨*proof*⟩

⟨*ML*⟩

lift-definition *atomic* :: $('s, 'v)$ *prog* $\Rightarrow$ $('s, 'v)$ *prog* **is**
$\lambda P.$ *spec.interference.cl* $(\{env\} \times UNIV)$ (*spec.atomic self P*) ⟨*proof*⟩

⟨*ML*⟩

lemma *bot*[*simp*]:
shows *prog.atomic* $\bot = \bot$
⟨*proof*⟩

lemma *contractive*:
shows *prog.atomic P* $\leq$ *P*
⟨*proof*⟩

lemma *idempotent*[*simp*]:
shows *prog.atomic* (*prog.atomic P*) = *prog.atomic P*
⟨*proof*⟩

lemma *monotone*:
shows *mono prog.atomic*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.atomic.monotone*]

lemmas *mono* = *monotoneD*[*OF prog.atomic.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.atomic.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup*:
  shows *prog.atomic* ($\bigsqcup X$) = $\bigsqcup$(*prog.atomic* ‘ *X*)
⟨*proof*⟩

lemmas *sup* = *prog.atomic.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *mcont*:
  shows *mcont Sup* (≤) *Sup* (≤) *prog.atomic*
⟨*proof*⟩

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.atomic.mcont*, *of luba orda P* **for** *luba orda P*]

lemma *Inf-le*:
  shows *prog.atomic* ($\bigsqcap X$) ≤ $\bigsqcap$(*prog.atomic* ‘ *X*)
⟨*proof*⟩

lemmas *inf-le* = *prog.atomic.Inf-le*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *action*:
  shows *prog.atomic* (*prog.action F*) = *prog.action F*
⟨*proof*⟩

lemma *return*:
  shows *prog.atomic* (*prog.return v*) = *prog.return v*
⟨*proof*⟩

lemma *bind-le*:
  shows *prog.atomic* (*f* ≫= *g*) ≤ *prog.atomic f* ≫= (λ*v*. *prog.atomic* (*g v*))
⟨*proof*⟩

⟨*ML*⟩

lemmas *atomic* = *prog.atomic.rep-eq*

⟨*ML*⟩

17.1 Inhabitation

⟨*ML*⟩

lemma *atomic*:
  assumes *P* −*s*, *xs*→ *P*′
  assumes *trace.steps*′ *s xs* ⊆ {*a*} × *UNIV*
  shows *spec.atomic a P* −*s*, *xs*→ *spec.atomic a P*′
⟨*proof*⟩

lemma *atomic-term*:
  assumes *P* −*s*, *xs*→ *spec.return v*
  assumes *trace.steps*′ *s xs* ⊆ {*a*} × *UNIV*
  shows *spec.atomic a P* −*s*, *xs*→ *spec.return v*
⟨*proof*⟩

lemma *atomic-diverge*:
  assumes *P* −*s*, *xs*→ ⊥

assumes *trace.steps′ s xs* ⊆ {*a*} × *UNIV*
shows *spec.atomic a P* −*s*, *xs*→ ⊥
⟨*proof*⟩

⟨*ML*⟩

lemma *atomic-term*:
assumes *prog.p2s P* −*s*, *xs*→ *spec.return v*
assumes *trace.steps′ s xs* ⊆ {*self*} × *UNIV*
shows *prog.p2s* (*prog.atomic P*) −*s*, *xs*→ *spec.return v*
⟨*proof*⟩

lemma *atomic-diverge*:
assumes *prog.p2s P* −*s*, *xs*→ ⊥
assumes *trace.steps′ s xs* ⊆ {*self*} × *UNIV*
shows *prog.p2s* (*prog.atomic P*) −*s*, *xs*→ ⊥
⟨*proof*⟩

⟨*ML*⟩

17.2 Assume/guarantee

⟨*ML*⟩

lemma *atomic*:
assumes *prog.p2s c* ≤ ⦃*P*⦄, *Id* ⊢ *G*, ⦃*Q*⦄
assumes *P*: *stable A P*
assumes *Q*: ⋀*v*. *stable A* (*Q v*)
shows *prog.p2s* (*prog.atomic c*) ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
⟨*proof*⟩

⟨*ML*⟩

18 Exceptions

A sketch of how we might handle exceptions in this framework.

⟨*ML*⟩

type-synonym (′*s*, ′*x*, ′*v*) *exn* = (′*s*, ′*x* + ′*v*) *prog*

definition *action* :: (′*v* × ′*s* × ′*s*) *set* ⇒ (′*s*, ′*x*, ′*v*) *raw.exn* **where**
action = *prog.action* ∘ *image* (*map-prod Inr id*)

definition *return* :: ′*v* ⇒ (′*s*, ′*x*, ′*v*) *raw.exn* **where**
return = *prog.return* ∘ *Inr*

definition *throw* :: ′*x* ⇒ (′*s*, ′*x*, ′*v*) *raw.exn* **where**
throw = *prog.return* ∘ *Inl*

definition *catch* :: (′*s*, ′*x*, ′*v*) *raw.exn* ⇒ (′*x* ⇒ (′*s*, ′*x*, ′*v*) *raw.exn*) ⇒ (′*s*, ′*x*, ′*v*) *raw.exn* **where**
catch f handler = *f* ≫= *case-sum handler raw.return*

definition *bind* :: (′*s*, ′*x*, ′*v*) *raw.exn* ⇒ (′*v* ⇒ (′*s*, ′*x*, ′*v*) *raw.exn*) ⇒ (′*s*, ′*x*, ′*v*) *raw.exn* **where**
bind f g = *f* ≫= *case-sum raw.throw g*

definition *parallel* :: (′*s*, ′*x*, *unit*) *raw.exn* ⇒ (′*s*, ′*x*, *unit*) *raw.exn* ⇒ (′*s*, ′*x*, *unit*) *raw.exn* **where**
parallel P Q = (*P* ≫= *case-sum* ⊥ *prog.return* ∥ *Q* ≫= *case-sum* ⊥ *prog.return*) ≫= *raw.return*

⟨*ML*⟩

lemma *bind*:
  shows *raw.bind* (*raw.bind f g*) *h* = *raw.bind f* (λ*x*. *raw.bind* (*g x*) *h*)
⟨*proof*⟩

lemma *return*:
  shows *returnL*: *raw.bind* (*raw.return v*) = (λ*g*. *g v*)
    and *returnR*: *raw.bind f raw.return* = *f*
⟨*proof*⟩

lemma *throwL*:
  shows *raw.bind* (*raw.throw x*) = (λ*g*. *raw.throw x*)
⟨*proof*⟩

⟨*ML*⟩

lemma *catch*:
  shows *raw.catch* (*raw.catch f* $handler_1$) $handler_2$ = *raw.catch f* (λ*x*. *raw.catch* ($handler_1$ *x*) $handler_2$)
⟨*proof*⟩

lemma *returnL*:
  shows *raw.catch* (*raw.return v*) = (λ*handler*. *raw.return v*)
⟨*proof*⟩

lemma *throw*:
  shows *throwL*: *raw.catch* (*raw.throw x*) = (λ*g*. *g x*)
    and *throwR*: *raw.catch f raw.throw* = *f*
⟨*proof*⟩

⟨*ML*⟩

lemma *commute*:
  shows *raw.parallel P Q* = *raw.parallel Q P*
⟨*proof*⟩

lemma *assoc*:
  shows *raw.parallel P* (*raw.parallel Q R*) = *raw.parallel* (*raw.parallel P Q*) *R*
⟨*proof*⟩

lemma *return*:
  shows *raw.parallel* (*raw.return* ()) *P* = *raw.catch P* (λ*x*. ⊥) (**is** *?thesis1*)
    and *raw.parallel P* (*raw.return* ()) = *raw.catch P* (λ*x*. ⊥) (**is** *?thesis2*)
⟨*proof*⟩

lemma *throw*:
  shows *raw.parallel* (*raw.throw x*) *P* = *raw.bind* (*raw.catch P* (λ*x*. ⊥)) (λ*x*. ⊥) (**is** *?thesis1*)
    and *raw.parallel P* (*raw.throw x*) = *raw.bind* (*raw.catch P* (λ*x*. ⊥)) (λ*x*. ⊥) (**is** *?thesis2*)
⟨*proof*⟩

⟨*ML*⟩

typedef ('*s*, '*x*, '*v*) *exn* = *UNIV* :: ('*s*, '*x*, '*v*) *raw.exn set*
⟨*proof*⟩

setup-lifting *type-definition-exn*

instantiation *exn* :: (*type*, *type*, *type*) *complete-distrib-lattice*
begin

lift-definition *bot-exn* :: (*'s*, *'x*, *'v*) *exn* **is** ⊥ ⟨*proof*⟩
lift-definition *top-exn* :: (*'s*, *'x*, *'v*) *exn* **is** ⊤ ⟨*proof*⟩
lift-definition *sup-exn* :: (*'s*, *'x*, *'v*) *exn* ⇒ (*'s*, *'x*, *'v*) *exn* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *sup* ⟨*proof*⟩
lift-definition *inf-exn* :: (*'s*, *'x*, *'v*) *exn* ⇒ (*'s*, *'x*, *'v*) *exn* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *inf* ⟨*proof*⟩
lift-definition *less-eq-exn* :: (*'s*, *'x*, *'v*) *exn* ⇒ (*'s*, *'x*, *'v*) *exn* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩
lift-definition *less-exn* :: (*'s*, *'x*, *'v*) *exn* ⇒ (*'s*, *'x*, *'v*) *exn* ⇒ *bool* **is** *less* ⟨*proof*⟩
lift-definition *Inf-exn* :: (*'s*, *'x*, *'v*) *exn set* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *Inf* ⟨*proof*⟩
lift-definition *Sup-exn* :: (*'s*, *'x*, *'v*) *exn set* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *Sup* ⟨*proof*⟩

instance ⟨*proof*⟩

end

⟨*ML*⟩

lift-definition *action* :: (*'v* × *'s* × *'s*) *set* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *raw.action* ⟨*proof*⟩
lift-definition *return* :: *'v* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *raw.return* ⟨*proof*⟩
lift-definition *throw* :: *'x* ⇒ (*'s*, *'x*, *'v*) *exn* **is** *raw.throw* ⟨*proof*⟩
lift-definition *catch* :: (*'s*, *'x*, *'v*) *exn* ⇒ (*'x* ⇒ (*'s*, *'x*, *'v*) *exn*) ⇒ (*'s*, *'x*, *'v*) *exn* **is** *raw.catch* ⟨*proof*⟩
lift-definition *bind* :: (*'s*, *'x*, *'v*) *exn* ⇒ (*'v* ⇒ (*'s*, *'x*, *'v*) *exn*) ⇒ (*'s*, *'x*, *'v*) *exn* **is** *raw.bind* ⟨*proof*⟩
lift-definition *parallel* :: (*'s*, *'x*, *unit*) *exn* ⇒ (*'s*, *'x*, *unit*) *exn* ⇒ (*'s*, *'x*, *unit*) *exn* **is** *raw.parallel* ⟨*proof*⟩

adhoc-overloading
  Monad-Syntax.bind ⇌ *exn.bind*
adhoc-overloading
  parallel ⇌ *exn.parallel*

⟨*ML*⟩

lemma *bind*:
  shows *f* ≫= *g* ≫= *h* = *exn.bind f* (λ*x*. *g x* ≫= *h*)
⟨*proof*⟩

lemma *return*:
  shows *returnL*: (≫=) (*exn.return v*) = (λ*g*. *g v*) (**is** *?thesis1*)
    and *returnR*: *f* ≫= *exn.return* = *f* (**is** *?thesis2*)
⟨*proof*⟩

lemma *throwL*:
  shows (≫=) (*exn.throw x*) = (λ*g*. *exn.throw x*)
⟨*proof*⟩

⟨*ML*⟩

lemma *catch*:
  shows *exn.catch* (*exn.catch f* $handler_1$) $handler_2$ = *exn.catch f* (λ*x*. *exn.catch* ($handler_1$ *x*) $handler_2$)
⟨*proof*⟩

lemma *returnL*:
  shows *exn.catch* (*exn.return v*) = (λ*handler*. *exn.return v*)
⟨*proof*⟩

lemma *throw*:
  shows *throwL*: *exn.catch* (*exn.throw x*) = (λ*g*. *g x*)
    and *throwR*: *exn.catch f exn.throw* = *f*

⟨*proof*⟩

⟨*ML*⟩

lemma *commute*:
shows *exn.parallel P Q* = *exn.parallel Q P*
⟨*proof*⟩

lemma *assoc*:
shows *exn.parallel P* (*exn.parallel Q R*) = *exn.parallel* (*exn.parallel P Q*) *R*
⟨*proof*⟩

lemma *return*:
shows *returnL*: *exn.return* () ∥ *P* = *exn.catch P* ⊥
and *returnR*: *P* ∥ *exn.return* () = *exn.catch P* ⊥
⟨*proof*⟩

lemma *throw*:
shows *throwL*: *exn.throw x* ∥ *P* = *exn.catch P* ⊥ ⋙= ⊥
and *throwR*: *P* ∥ *exn.throw x* = *exn.catch P* ⊥ ⋙= ⊥
⟨*proof*⟩

⟨*ML*⟩

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* ⇒ $'a$ *rel* **where**
ceilr P = ⨆{*r*. *stable r P*}

lemma *ceilr-alt-def*:
shows *ceilr P* = {(*s*, *s'*). *P s* ⟶ *P s'*}
⟨*proof*⟩

lemma *ceilrE*[*elim*]:
assumes (*x*, *y*) ∈ *ceilr P*
assumes *P x*
shows *P y*
⟨*proof*⟩

⟨*ML*⟩

named-theorems *simps* ‹*simp rules for* ***const*** ‹*ceilr*››

lemma *bot*[*ceilr.simps*]:
  shows $ceilr\ \bot = UNIV$
⟨*proof*⟩

lemma *top*[*ceilr.simps*]:
  shows $ceilr\ \top = UNIV$
⟨*proof*⟩

lemma *const*[*ceilr.simps*]:
  shows $ceilr\ \langle c\rangle = UNIV$
    and $ceilr\ (P \wedge \langle c\rangle) = (\mathit{if}\ c\ \mathit{then}\ ceilr\ P\ \mathit{else}\ UNIV)$
    and $ceilr\ (\langle c\rangle \wedge P) = (\mathit{if}\ c\ \mathit{then}\ ceilr\ P\ \mathit{else}\ UNIV)$
    and $ceilr\ (P \wedge \langle c\rangle \wedge P') = (\mathit{if}\ c\ \mathit{then}\ ceilr\ (P \wedge P')\ \mathit{else}\ UNIV)$
⟨*proof*⟩

lemma *Id-le*:
  shows $Id \subseteq ceilr\ P$
⟨*proof*⟩

lemmas *refl*[*iff*] = *ceilr.Id-le*[*folded refl-alt-def*]

lemma *trans*[*iff*]:
  shows $trans\ (ceilr\ P)$
⟨*proof*⟩

lemma *stable*[*stable.intro*]:
  shows $stable\ (ceilr\ P)\ P$
⟨*proof*⟩

lemma *largest*[*stable.intro*]:
  assumes $stable\ r\ P$
  shows $r \subseteq ceilr\ P$
⟨*proof*⟩

lemma *disj-subseteq*: — Converse does not hold
  shows $ceilr\ (P \vee Q) \subseteq ceilr\ P \cup ceilr\ Q$
⟨*proof*⟩

lemma *Ex-subseteq*: — Converse does not hold
  shows $ceilr\ (\exists x.\ P\ x) \subseteq (\bigcup x.\ ceilr\ (P\ x))$
⟨*proof*⟩

lemma *conj-subseteq*: — Converse does not hold
  shows $ceilr\ P \cap ceilr\ Q \subseteq ceilr\ (P \wedge Q)$
⟨*proof*⟩

lemma *All-subseteq*: — Converse does not hold
  shows $(\bigcap x.\ ceilr\ (P\ x)) \subseteq ceilr\ (\forall x.\ P\ x)$
⟨*proof*⟩

lemma *const-implies*[*ceilr.simps*]:
  shows $ceilr\ (\langle P\rangle \longrightarrow Q) = (\mathit{if}\ P\ \mathit{then}\ ceilr\ Q\ \mathit{else}\ UNIV)$
⟨*proof*⟩

lemma *Id-proj-on*:
  shows $(\bigcap c.\ ceilr\ (\langle c\rangle = f)) = Id_f$
    and $(\bigcap c.\ ceilr\ (f = \langle c\rangle)) = Id_f$
⟨*proof*⟩

⟨*ML*⟩

lemma *Inter-ceilr*:
  shows $stable\ (\bigcap v.\ ceilr\ (Q\ v))\ (Q\ v)$
⟨*proof*⟩

⟨*ML*⟩

We can internalize the stability conditions; see §19.1.1 for further discussion.

⟨*ML*⟩

lemma *p2s-s2p-ag-ceilr*:
  shows $prog.p2s\ (prog.s2p\ (\{\!|P|\!\},\ ceilr\ P \cap (\bigcap v.\ ceilr\ (Q\ v)) \vdash G,\ \{\!|Q|\!\}))$
  $= \{\!|P|\!\},\ ceilr\ P \cap (\bigcap v.\ ceilr\ (Q\ v)) \vdash G,\ \{\!|Q|\!\}$
⟨*proof*⟩

⟨*ML*⟩

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 (⋙=) duplicates stability goals.

See §22 for an example of using these rules.

⟨*ML*⟩

named-theorems *intro* ‹*safe backchaining intro rules*›

lemma *init*:
  assumes $c \le \{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
  assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
  assumes $A' \subseteq A$ — these rules use *ceilr* which always yields a reflexive relation (*ceilr.refl*
  shows $c \le \{\!|P'|\!\},\ A' \vdash G,\ \{\!|Q|\!\}$
⟨*proof*⟩

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 $\bigwedge v.\ prog.p2s\ (g\ v) \le \{\!\!\{Q'\ v\}\!\!\},\ A_2\ v \vdash G,\ \{\!\!\{Q\}\!\!\}$
assumes $prog.p2s\ f \le \{\!\!\{P\}\!\!\},\ A_1 \vdash G,\ \{\!\!\{Q'\}\!\!\}$
shows $prog.p2s\ (f \ggg g) \le \{\!\!\{P\}\!\!\},\ A_1 \cap (\bigcap v.\ A_2\ v) \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *rev-bind* = *iag.bind*[*rotated*]

lemma *read*[*iag.intro*]:
shows $prog.p2s\ (prog.read\ F) \le \{\!\!\{\lambda s.\ Q\ (F\ s)\ s\}\!\!\},\ ceilr\ (\lambda s.\ Q\ (F\ s)\ s) \cap (\bigcap s.\ ceilr\ (Q\ (F\ s))) \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *return*[*iag.intro*]:
shows $prog.p2s\ (prog.return\ v) \le \{\!\!\{Q\ v\}\!\!\},\ ceilr\ (Q\ v) \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *write*[*iag.intro*]: — this is where *guarantee* obligations arise
shows $prog.p2s\ (prog.write\ F)$
$\le \{\!\!\{(\lambda s.\ Q\ ()\ (F\ s) \wedge (s,\ F\ s) \in G)\}\!\!\},\ ceilr\ (\lambda s.\ Q\ ()\ (F\ s) \wedge (s,\ F\ s) \in G) \cap ceilr\ (Q\ ()) \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *parallel*: — not in the *iag* format; instantiate the first two assumptions
assumes $prog.p2s\ c_1 \le \{\!\!\{P_1\}\!\!\},\ A_1 \vdash G_1,\ \{\!\!\{Q_1\}\!\!\}$
assumes $prog.p2s\ c_2 \le \{\!\!\{P_2\}\!\!\},\ A_2 \vdash G_2,\ \{\!\!\{Q_2\}\!\!\}$
assumes $\bigwedge s.\ [\![Q_1\ ()\ s;\ Q_2\ ()\ s]\!] \Longrightarrow Q\ ()\ s$
assumes $G_2 \subseteq A_1$
assumes $G_1 \subseteq A_2$
assumes $G_1 \cup G_2 \subseteq G$
shows $prog.p2s\ (prog.parallel\ c_1\ c_2) \le \{\!\!\{P_1 \wedge P_2\}\!\!\},\ A_1 \cap A_2 \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *local* = *ag.prog.local* — not in the *iag* format

lemma *if*[*iag.intro*]:
assumes $b \Longrightarrow prog.p2s\ c_1 \le \{\!\!\{P_1\}\!\!\},\ A_1 \vdash G,\ \{\!\!\{Q\}\!\!\}$
assumes $\neg b \Longrightarrow prog.p2s\ c_2 \le \{\!\!\{P_2\}\!\!\},\ A_2 \vdash G,\ \{\!\!\{Q\}\!\!\}$
shows $prog.p2s\ (if\ b\ then\ c_1\ else\ c_2) \le \{\!\!\{if\ b\ then\ P_1\ else\ P_2\}\!\!\},\ A_1 \cap A_2 \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *case-option*[*iag.intro*]:
assumes $x = None \Longrightarrow prog.p2s\ none \leq \{\!\!\{P_n\}\!\!\},\ A_n \vdash G,\ \{\!\!\{Q\}\!\!\}$
assumes $\bigwedge v.\ x = Some\ v \Longrightarrow prog.p2s\ (some\ v) \leq \{\!\!\{P_s\ v\}\!\!\},\ A_s\ v \vdash G,\ \{\!\!\{Q\}\!\!\}$
shows $prog.p2s\ (case\text{-}option\ none\ some\ x) \leq \{\!\!\{case\ x\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v\}\!\!\},\ case\text{-}option\ A_n\ A_s\ x \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *case-sum*[*iag.intro*]:
assumes $\bigwedge v.\ x = Inl\ v \Longrightarrow prog.p2s\ (left\ v) \leq \{\!\!\{P_l\ v\}\!\!\},\ A_l\ v \vdash G,\ \{\!\!\{Q\}\!\!\}$
assumes $\bigwedge v.\ x = Inr\ v \Longrightarrow prog.p2s\ (right\ v) \leq \{\!\!\{P_r\ v\}\!\!\},\ A_r\ v \vdash G,\ \{\!\!\{Q\}\!\!\}$
shows $prog.p2s\ (case\text{-}sum\ left\ right\ x) \leq \{\!\!\{case\text{-}sum\ P_l\ P_r\ x\}\!\!\},\ case\text{-}sum\ A_l\ A_r\ x \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *case-list*[*iag.intro*]:
assumes $x = [] \Longrightarrow prog.p2s\ nil \leq \{\!\!\{P_n\}\!\!\},\ A_n \vdash G,\ \{\!\!\{Q\}\!\!\}$
assumes $\bigwedge v\ vs.\ x = v\ \#\ vs \Longrightarrow prog.p2s\ (cons\ v\ vs) \leq \{\!\!\{P_c\ v\ vs\}\!\!\},\ A_c\ v\ vs \vdash G,\ \{\!\!\{Q\}\!\!\}$
shows $prog.p2s\ (case\text{-}list\ nil\ cons\ x) \leq \{\!\!\{case\text{-}list\ P_n\ P_c\ x\}\!\!\},\ case\text{-}list\ A_n\ A_c\ x \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *while*:
fixes $c :: {}'k \Rightarrow ({}'s,\ {}'k + {}'v)\ prog$
assumes c: $\bigwedge k.\ prog.p2s\ (c\ k) \leq \{\!\!\{P\ k\}\!\!\},\ A \vdash G,\ \{\!\!\{case\text{-}sum\ I\ Q\}\!\!\}$
shows $prog.p2s\ (prog.while\ c\ k) \leq \{\!\!\{\langle\forall\ v\ s.\ I\ v\ s \longrightarrow P\ v\ s\rangle \wedge I\ k\}\!\!\},\ A \cap (\bigcap v.\ ceilr\ (Q\ v)) \vdash G,\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *whenM* = *iag.if*[**where** c_1=*c* **and** A_1=*A* **and** P_1=*P*, *OF* - *iag.return*[**where** *v*=()]] **for** *A c P*

⟨*ML*⟩

19.2 Refinement with relational assumes

Two sets of refinement rules:

- relational assumes
- relational assumes and *prog.sinvmap* (inverse state abstraction)

⟨*ML*⟩

lemma *bind*:
assumes $\bigwedge v.\ prog.p2s\ (g\ v) \leq \{\!\!\{Q'\ v\}\!\!\},\ ag.assm\ A \Vdash prog.p2s\ (g'\ v),\ \{\!\!\{Q\}\!\!\}$
assumes $prog.p2s\ f \leq \{\!\!\{P\}\!\!\},\ ag.assm\ A \Vdash prog.p2s\ f',\ \{\!\!\{Q'\}\!\!\}$
shows $prog.p2s\ (f \gg\!= g) \leq \{\!\!\{P\}\!\!\},\ ag.assm\ A \Vdash prog.p2s\ (f' \gg\!= g'),\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemmas *rev-bind* = *rar.prog.bind*[*rotated*]

lemma *action*:
fixes $F :: ({}'v \times {}'s \times {}'s)\ set$
fixes $F' :: ({}'v \times {}'s \times {}'s)\ set$
assumes Q: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow Q\ v\ s'$
assumes F': $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow (v,\ s,\ s') \in F'$
assumes sP: $stable\ A\ P$
assumes sQ: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.action\ F) \leq \{\!\!\{P\}\!\!\},\ ag.assm\ A \Vdash prog.p2s\ (prog.action\ F'),\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *return*:

assumes *sQ*: *stable A* (*Q v*)
shows *prog.p2s* (*prog.return v*) $\leq$ ⦃*Q v*⦄, *ag.assm A* ⊩ *prog.p2s* (*prog.return v*), ⦃*Q*⦄
⟨*proof*⟩

lemma *parallel-refinement*:
assumes c_1: *prog.p2s* $c_1 \leq$ ⦃P_1⦄, *ag.assm* ($A \cup G_2$) ⊩ *prog.p2s* ($c_1{}' \sqcap$ *prog.rel* G_1), ⦃Q_1⦄
assumes c_2: *prog.p2s* $c_2 \leq$ ⦃P_2⦄, *ag.assm* ($A \cup G_1$) ⊩ *prog.p2s* ($c_2{}' \sqcap$ *prog.rel* G_2), ⦃Q_2⦄
shows *prog.p2s* ($c_1 \parallel c_2$) $\leq$ ⦃$P_1 \wedge P_2$⦄, *ag.assm A* ⊩ *prog.p2s* ($c_1{}' \sqcap$ *prog.rel* $G_1 \parallel c_2{}' \sqcap$ *prog.rel* G_2), ⦃$\lambda v.\ Q_1\ v \wedge Q_2\ v$⦄
⟨*proof*⟩

lemma *parallel*:
assumes *prog.p2s* $c_1 \leq$ ⦃P_1⦄, *ag.assm* ($A \cup G_2$) ⊩ *prog.p2s* $c_1{}'$, ⦃Q_1⦄
assumes *prog.p2s* $c_1 \leq$ ⦃P_1⦄, $A \cup G_2 \vdash G_1$, ⦃$\top$⦄
assumes *prog.p2s* $c_2 \leq$ ⦃P_2⦄, *ag.assm* ($A \cup G_1$) ⊩ *prog.p2s* $c_2{}'$, ⦃Q_2⦄
assumes *prog.p2s* $c_2 \leq$ ⦃P_2⦄, $A \cup G_1 \vdash G_2$, ⦃$\top$⦄
shows *prog.p2s* ($c_1 \parallel c_2$) $\leq$ ⦃$P_1 \wedge P_2$⦄, *ag.assm A* ⊩ *prog.p2s* ($c_1{}' \parallel c_2{}'$), ⦃$\lambda v.\ Q_1\ v \wedge Q_2\ v$⦄
⟨*proof*⟩

lemma *while*:
fixes *c* :: *'k* ⇒ (*'s*, *'k* + *'v*) *prog*
fixes *c'* :: *'k* ⇒ (*'s*, *'k* + *'v*) *prog*
assumes *c*: ⋀*k*. *prog.p2s* (*c k*) $\leq$ ⦃*P k*⦄, *ag.assm A* ⊩ *prog.p2s* (*c' k*), ⦃*case-sum I Q*⦄
assumes *IP*: ⋀*s v*. *I v s* ⟹ *P v s*
assumes *sQ*: ⋀*v*. *stable A* (*Q v*)
shows *prog.p2s* (*prog.while c k*) $\leq$ ⦃*I k*⦄, *ag.assm A* ⊩ *prog.p2s* (*prog.while c' k*), ⦃*Q*⦄
⟨*proof*⟩

lemma *app*:
fixes *xs* :: *'a list*
fixes *f* :: *'a* ⇒ (*'s*, *unit*) *prog*
fixes *P* :: *'a list* ⇒ *'s pred*
assumes ⋀*x ys zs*. *xs* = *ys* @ *x* # *zs* ⟹ *prog.p2s* (*f x*) $\leq$ ⦃*P ys*⦄, *ag.assm A* ⊩ *prog.p2s* (*f' x*), ⦃λ-. *P* (*ys* @ [*x*])⦄
assumes ⋀*ys*. *prefix ys xs* ⟹ *stable A* (*P ys*)
shows *prog.p2s* (*prog.app f xs*) $\leq$ ⦃*P* []⦄, *ag.assm A* ⊩ *prog.p2s* (*prog.app f' xs*), ⦃λ-. *P xs*⦄
⟨*proof*⟩

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*]

⟨*ML*⟩

abbreviation (*input*) *absfn sf c* ≡ *prog.p2s* (*prog.sinvmap sf c*)

lemma *bind*:
assumes ⋀*v*. *prog.p2s* (*g v*) $\leq$ ⦃*Q' v*⦄, *ag.assm A* ⊩ *rair.prog.absfn sf* (*g' v*), ⦃*Q*⦄
assumes *prog.p2s f* $\leq$ ⦃*P*⦄, *ag.assm A* ⊩ *rair.prog.absfn sf f'*, ⦃*Q'*⦄
shows *prog.p2s* (*f* ⪢= *g*) $\leq$ ⦃*P*⦄, *ag.assm A* ⊩ *rair.prog.absfn sf* (*f'* ⪢= *g'*), ⦃*Q*⦄
⟨*proof*⟩

lemmas *rev-bind* = *rair.prog.bind*[*rotated*]

lemma *action*:
fixes *F* :: (*'v* × *'s* × *'s*) *set*
fixes *F'* :: (*'v* × *'t* × *'t*) *set*
fixes *sf* :: *'s* ⇒ *'t*
assumes *Q*: ⋀*v s s'*. ⟦*P s*; (*v*, *s*, *s'*) ∈ *F*⟧ ⟹ *Q v s'*

assumes F': $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow (v,\ sf\ s,\ sf\ s') \in F'$
assumes sP: $stable\ A\ P$
assumes sQ: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.action\ F) \leq \{\!|P|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (prog.action\ F'),\ \{\!|Q|\!\}$
$\langle proof \rangle$

lemma *return*:
assumes sQ: $stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.return\ v) \leq \{\!|Q\ v|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (prog.return\ v),\ \{\!|Q|\!\}$
$\langle proof \rangle$

lemma *parallel*:
fixes $sf :: 's \Rightarrow 't$
assumes $prog.p2s\ c_1 \leq \{\!|P_1|\!\},\ ag.assm\ (A \cup G_2) \Vdash rair.prog.absfn\ sf\ c_1',\ \{\!|Q_1|\!\}$
assumes $prog.p2s\ c_1 \leq \{\!|P_1|\!\},\ A \cup G_2 \vdash G_1,\ \{\!|\top|\!\}$
assumes $prog.p2s\ c_2 \leq \{\!|P_2|\!\},\ ag.assm\ (A \cup G_1) \Vdash rair.prog.absfn\ sf\ c_2',\ \{\!|Q_2|\!\}$
assumes $prog.p2s\ c_2 \leq \{\!|P_2|\!\},\ A \cup G_1 \vdash G_2,\ \{\!|\top|\!\}$
shows $prog.p2s\ (c_1 \parallel c_2) \leq \{\!|P_1 \wedge P_2|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (c_1' \parallel c_2'),\ \{\!|\lambda v.\ Q_1\ v \wedge Q_2\ v|\!\}$
$\langle proof \rangle$

lemma *while*:
fixes $c :: 'k \Rightarrow ('s,\ 'k + 'v)\ prog$
fixes $c' :: 'k \Rightarrow ('t,\ 'k + 'v)\ prog$
fixes $sf :: 's \Rightarrow 't$
assumes c: $\bigwedge k.\ prog.p2s\ (c\ k) \leq \{\!|P\ k|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (c'\ k),\ \{\!|case\text{-}sum\ I\ Q|\!\}$
assumes IP: $\bigwedge s\ v.\ I\ v\ s \Longrightarrow P\ v\ s$
assumes sQ: $\bigwedge v.\ stable\ A\ (Q\ v)$
shows $prog.p2s\ (prog.while\ c\ k) \leq \{\!|I\ k|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (prog.while\ c'\ k),\ \{\!|Q|\!\}$
$\langle proof \rangle$

lemma *app*:
fixes $xs :: 'a\ list$
fixes $f :: 'a \Rightarrow ('s,\ unit)\ prog$
fixes $P :: 'a\ list \Rightarrow 's\ pred$
assumes $\bigwedge x\ ys\ zs.\ xs = ys\ @\ x\ \#\ zs \Longrightarrow prog.p2s\ (f\ x) \leq \{\!|P\ ys|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (f'\ x),\ \{\!|\lambda\text{-}.\ P\ (ys\ @\ [x])|\!\}$
assumes $\bigwedge ys.\ prefix\ ys\ xs \Longrightarrow stable\ A\ (P\ ys)$
shows $prog.p2s\ (prog.app\ f\ xs) \leq \{\!|P\ []|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (prog.app\ f'\ xs),\ \{\!|\lambda\text{-}.\ P\ xs|\!\}$
$\langle proof \rangle$

lemma *if*:
assumes $i \Longrightarrow prog.p2s\ t \leq \{\!|P|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ t',\ \{\!|Q|\!\}$
assumes $\neg i \Longrightarrow prog.p2s\ e \leq \{\!|P'|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ e',\ \{\!|Q|\!\}$
shows $prog.p2s\ (if\ i\ then\ t\ else\ e) \leq \{\!|if\ i\ then\ P\ else\ P'|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (if\ i\ then\ t'\ else\ e'),\ \{\!|Q|\!\}$
$\langle proof \rangle$

lemma *case-option*:
assumes $opt = None \Longrightarrow prog.p2s\ none \leq \{\!|P_n|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ none',\ \{\!|Q|\!\}$
assumes $\bigwedge v.\ opt = Some\ v \Longrightarrow prog.p2s\ (some\ v) \leq \{\!|P_s\ v|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (some'\ v),\ \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}option\ none\ some\ opt) \leq \{\!|case\ opt\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v|\!\},\ ag.assm\ A \Vdash rair.prog.absfn\ sf\ (case\text{-}option\ none'\ some'\ opt),\ \{\!|Q|\!\}$
$\langle proof \rangle$

$\langle ML \rangle$

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.

⟨*ML*⟩

definition *floor* :: $'a\ rel \Rightarrow 'a\ pred \Rightarrow 'a\ pred$ **where** — An interior operator, or a closure in the dual lattice
$floor\ r\ P\ s \longleftrightarrow (\forall s'.\ (s,\ s') \in r^* \longrightarrow P\ s')$

definition *ceiling* :: $'a\ rel \Rightarrow 'a\ pred \Rightarrow 'a\ pred$ **where** — A closure operator
$ceiling\ r\ P\ s \longleftrightarrow (\exists s'.\ (s',\ s) \in r^* \wedge P\ s')$

⟨*ML*⟩

lemma *empty-rel*[*simp*]:
shows $wdp.floor\ \{\}\ P = P$
⟨*proof*⟩

lemma *reflcl*:
shows $wdp.floor\ (r^=) = wdp.floor\ r$
⟨*proof*⟩

lemma *const*:
shows $wdp.floor\ r\ \langle c\rangle = \langle c\rangle$
⟨*proof*⟩

lemma *contractive*:
shows $wdp.floor\ r\ P \leq P$
⟨*proof*⟩

lemma *idempotent*:
shows $wdp.floor\ r\ (wdp.floor\ r\ P) = wdp.floor\ r\ P$
⟨*proof*⟩

lemma *mono*:

assumes $r' \subseteq r$
assumes $P \leq P'$
shows *wdp.floor* r $P \leq$ *wdp.floor* r' P'
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes *st-ord* $(\neg F)$ r r'
assumes *st-ord* F P P'
shows *st-ord* F (*wdp.floor* r P) (*wdp.floor* r' P')
⟨*proof*⟩

lemma *weakest*:
assumes $Q \leq P$
assumes *stable* r Q
shows $Q \leq$ *wdp.floor* r P
⟨*proof*⟩

lemma *Chernoff*:
assumes $P \leq Q$
shows (*wdp.floor* r $P \wedge Q$) $\leq$ *wdp.floor* r Q
⟨*proof*⟩

lemma *floor1*:
assumes $r \subseteq r'$
shows *wdp.floor* r' (*wdp.floor* r P) $=$ *wdp.floor* r' P
⟨*proof*⟩

lemma *floor2*:
assumes $r \subseteq r'$
shows *wdp.floor* r (*wdp.floor* r' P) $=$ *wdp.floor* r' P
⟨*proof*⟩

⟨*ML*⟩

interpretation *ceiling*: *closure-complete-lattice-distributive-class wdp.ceiling* r **for** r
⟨*proof*⟩

⟨*ML*⟩

lemma *empty-rel*[*simp*]:
shows *wdp.ceiling* $\{\}$ $P = P$
⟨*proof*⟩

lemma *reflcl*:
shows *wdp.ceiling* $(r^{=})$ $=$ *wdp.ceiling* r
⟨*proof*⟩

lemma *const*:
shows *wdp.ceiling* r $\langle c \rangle = \langle c \rangle$
⟨*proof*⟩

lemma *mono*:
assumes $r \subseteq r'$
assumes $P \leq P'$
shows *wdp.ceiling* r $P \leq$ *wdp.ceiling* r' P'
⟨*proof*⟩

lemma *strengthen*[*strg*]:

assumes *st-ord* $F\ r\ r'$
assumes *st-ord* $F\ P\ P'$
shows *st-ord* F (*wdp.ceiling* $r\ P$) (*wdp.ceiling* $r'\ P'$)
⟨*proof*⟩

lemma *strongest*:
assumes $P \leq Q$
assumes *stable* $r\ Q$
shows *wdp.ceiling* $r\ P \leq Q$
⟨*proof*⟩

lemma *ceiling1*:
assumes $r \subseteq r'$
shows *wdp.ceiling* r' (*wdp.ceiling* $r\ P$) = *wdp.ceiling* $r'\ P$
⟨*proof*⟩

lemma *ceiling2*:
assumes $r \subseteq r'$
shows *wdp.ceiling* r (*wdp.ceiling* $r'\ P$) = *wdp.ceiling* $r'\ P$
⟨*proof*⟩

⟨*ML*⟩

lemma *floor*:
shows *stable* r (*wdp.floor* $r\ P$)
⟨*proof*⟩

lemma *ceiling*:
shows *stable* r (*wdp.ceiling* $r\ P$)
⟨*proof*⟩

lemma *floor-conv*:
assumes *stable* $r\ P$
shows P = *wdp.floor* $r\ P$
⟨*proof*⟩

lemma *ceiling-conv*:
assumes *stable* $r\ P$
shows P = *wdp.ceiling* $r\ P$
⟨*proof*⟩

⟨*ML*⟩

lemma *floor-alt-def*: — Wickerson et al. (2010, §3)
shows *wdp.floor* $r\ P = \bigsqcup\{Q.\ Q \leq P \land$ *stable* $r\ Q\}$
⟨*proof*⟩

lemma *ceiling-alt-def*: — Wickerson et al. (2010, §3)
shows *wdp.ceiling* $r\ P = \bigsqcap\{Q.\ P \leq Q \land$ *stable* $r\ Q\}$
⟨*proof*⟩

lemma *duality-floor-ceiling*:
shows *wdp.ceiling* $r\ (\neg P) = (\neg$*wdp.floor* $(r^{-1})\ P)$
⟨*proof*⟩

lemma *ceiling-floor*:
assumes $r \subseteq r'$
shows *wdp.ceiling* r (*wdp.floor* $r'\ P$) = *wdp.floor* $r'\ P$

⟨*proof*⟩

lemma *floor-ceiling*:
  assumes $r \subseteq r'$
  shows $wdp.floor\ r\ (wdp.ceiling\ r'\ P) = wdp.ceiling\ r'\ P$
⟨*proof*⟩

lemma *floor-ceilr*:
  shows $wdp.floor\ (ceilr\ P)\ P = P$
⟨*proof*⟩

lemma *ceiling-ceilr*:
  shows $wdp.ceiling\ (ceilr\ P)\ P = P$
⟨*proof*⟩

⟨*ML*⟩

20.1 Assume/Guarantee rules

§3.2 traditional assume/guarantee rules ⟨*ML*⟩

lemma *action*: — arbitrary A
  fixes $F :: ('v \times 's \times 's)\ set$
  assumes Q: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow Q\ v\ s'$
  assumes G: $\bigwedge v\ s\ s'.\ [\![P\ s;\ s \neq s';\ (v,\ s,\ s') \in F]\!] \Longrightarrow (s,\ s') \in G$
  shows $prog.p2s\ (prog.action\ F) \leq \{\!|wdp.floor\ A\ P|\!\},\ A \vdash G,\ \{\!|\lambda v.\ wdp.ceiling\ A\ (Q\ v)|\!\}$
⟨*proof*⟩

lemmas $mono = ag.mono$
lemmas $bind = ag.prog.bind$

etc. – the other rules are stock

⟨*ML*⟩

§4, Appendix C parametric specifications **definition** $pag :: ('s\ rel \Rightarrow 's\ pred) \Rightarrow 's\ rel\ set \Rightarrow 's\ rel \Rightarrow ('s\ rel \Rightarrow 'v \Rightarrow 's\ pred) \Rightarrow (sequential,\ 's,\ 'v)\ spec$ (‹$\{\!|$-$|\!\}$, -/ $\vdash_P$ -, $\{\!|$-$|\!\}$› [0,0,0,0] 100) **where**
  $\{\!|P|\!\},\ As \vdash_P G,\ \{\!|Q|\!\} = (\bigsqcap A{\in}As.\ \{\!|P\ A|\!\},\ A \vdash G,\ \{\!|Q\ A|\!\})$

⟨*ML*⟩

lemma *empty*:
  shows $\{\!|P|\!\},\ \{\} \vdash_P G,\ \{\!|Q|\!\} = \top$
⟨*proof*⟩

lemma *singleton*:
  shows $\{\!|P|\!\},\ \{A\} \vdash_P G,\ \{\!|Q|\!\} = \{\!|P\ A|\!\},\ A \vdash G,\ \{\!|Q\ A|\!\}$
⟨*proof*⟩

lemma *mono*: — strengthening of the WEAKEN rule in Figure 4, needed for the example
  assumes $\bigwedge A.\ A \in As' \Longrightarrow P'\ A \leq P\ A$
  assumes $As' \leq As$
  assumes $G \leq G'$
  assumes $\bigwedge A.\ A \in As' \Longrightarrow Q\ A \leq Q'\ A$
  shows $\{\!|P|\!\},\ As \vdash_P G,\ \{\!|Q|\!\} \leq \{\!|P'|\!\},\ As' \vdash_P G',\ \{\!|Q'|\!\}$
⟨*proof*⟩

lemma *action*: — allow assertions to depend on assume A, needed for the example
  fixes $F :: ('v \times 's \times 's)\ set$

assumes Q: $\bigwedge A\ v\ s\ s'.\ [\![A \in As;\ P\ A\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow Q\ A\ v\ s'$
assumes G: $\bigwedge A\ v\ s\ s'.\ [\![A \in As;\ P\ A\ s;\ s \neq s';\ (v,\ s,\ s') \in F]\!] \Longrightarrow (s,\ s') \in G$
shows *prog.p2s* (*prog.action F*) $\leq$ ⦃λ*A. wdp.floor A* (*P A*)⦄, *As* $\vdash_P$ *G*, ⦃λ*A v. wdp.ceiling A* (*Q A v*)⦄
⟨*proof*⟩

lemmas *sup* = *ag.prog.sup*

lemma *bind*:
assumes $\bigwedge v.$ *prog.p2s* (*g v*) $\leq$ ⦃λ*A. Q′ A v*⦄, *As* $\vdash_P$ *G*, ⦃*Q*⦄
assumes *prog.p2s f* $\leq$ ⦃*P*⦄, *As* $\vdash_P$ *G*, ⦃*Q′*⦄
shows *prog.p2s* (*f* ⪢= *g*) $\leq$ ⦃*P*⦄, *As* $\vdash_P$ *G*, ⦃*Q*⦄
⟨*proof*⟩

lemma *parallel*:
assumes *prog.p2s* $c_1 \leq$ ⦃P_1⦄, (∪) G_2 ‘ $A \vdash_P G_1$, ⦃Q_1⦄
assumes *prog.p2s* $c_2 \leq$ ⦃P_2⦄, (∪) G_1 ‘ $A \vdash_P G_2$, ⦃Q_2⦄
shows *prog.p2s* (*prog.parallel* $c_1\ c_2$)
$\leq$ ⦃λ$R.\ P_1\ (R \cup G_2) \wedge P_2\ (R \cup G_1)$⦄, $A \vdash_P G_1 \cup G_2$, ⦃λ$R\ v.\ Q_1\ (R \cup G_2)\ v \wedge Q_2\ (R \cup G_1)\ v$⦄
⟨*proof*⟩

etc. – the other rules follow similarly

⟨*ML*⟩

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* ≡ *prog.write* ((+) *1*) — atomic increment
abbreviation (*input*) *increases* :: *int rel* **where** *increases* ≡ {(*x*, *x′*). $x \leq x'$}

lemma *ag-incr1*: — the precondition is stable as the rely is very strong
shows *prog.p2s incr* $\leq$ ⦃(=) *c*⦄, {} ⊢ *increases*, ⦃⟨(=) (*c* + *1*)⟩⦄
⟨*proof*⟩

lemma *ag-incr2*: — note the weaker precondition due to the larger assume
shows *prog.p2s incr* $\leq$ ⦃(≤) *c*⦄, *increases* ⊢ *increases*, ⦃⟨(≤) (*c* + *1*)⟩⦄
⟨*proof*⟩

lemma *ag-incr1-par-incr1*:
shows *prog.p2s* (*incr* ∥ *incr*) $\leq$ ⦃λ*x*. $c \leq x$⦄, *increases* ⊢ *increases*, ⦃λ- *x*. $c + 1 \leq x$⦄
⟨*proof*⟩

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* ⊢ *increases*, ⦃⟨*wdp.ceiling A* (λ*s. wdp.ceiling A* ((=) *c*) (*s* − *1*))⟩⦄
⟨*proof*⟩
abbreviation (*input*) *comm-xpp* :: *int rel set* **where**
comm-xpp ≡ {*A*. ∀ *p s. wdp.ceiling A p* (*s* − *1*) = *wdp.ceiling A* (λ*s. p* (*s* − *1*)) *s*}

lemma *pag-incr*: — postcondition can be simplified wrt *comm-xpp*
shows *prog.p2s incr* $\leq$ ⦃λ*A. wdp.ceiling A* ((=) *c*)⦄, *comm-xpp* $\vdash_P$ *increases*, ⦃λ*A*. ⟨*wdp.ceiling A* ((=) (*c* + *1*))⟩⦄
⟨*proof*⟩
lemma
shows *prog.p2s incr* $\leq$ ⦃(=) *c*⦄, {} ⊢ *increases*, ⦃⟨(=) (*c* + *1*)⟩⦄
⟨*proof*⟩

lemma
  shows $prog.p2s\ incr \leq \{\!|(\leq)\ c|\!\}$, $increases \vdash increases$, $\{\!|\langle(\leq)\ (c + 1)\rangle|\!\}$
$\langle proof \rangle$

21 Example: inhabitation

The following is a simple example of showing that a specification is inhabited.

lemma
  shows $\langle\!|0{::}nat,\ [(self,\ 1),\ (self,\ 2)],\ Some\ ()|\!\rangle$
    $\leq prog.p2s\ (prog.while\ \langle prog.write\ ((+)\ 1) \gg (prog.return\ (Inl\ ()) \sqcup prog.return\ (Inr\ ()))\rangle\ ())$
$\langle proof \rangle$
$\langle proof \rangle$

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 $foundE :: nat \Longrightarrow 's\ state$ **where** $foundE \equiv fst_L\ ;_L\ fst_L$
abbreviation $foundO :: nat \Longrightarrow 's\ state$ **where** $foundO \equiv snd_L\ ;_L\ fst_L$

context
  fixes $pred :: 'a \Rightarrow ('s,\ bool)\ prog$
  fixes $predPre :: 's\ pred$

fixes *predP* :: *'a* ⇒ *'s pred*
fixes *A* :: *'s rel*
fixes *array* :: *'a list* ⟹ *'s*
— A guarantee of *Id* indicates that *pred a* is observationally pure.
assumes *iag-pred*: $\bigwedge a.\ prog.p2s\ (pred\ a) \le \{\!|predPre \wedge \langle a\rangle \in SET\ get_{array}|\!\},\ A^{=} \cap Id_{get_{array}} \cap ceilr\ predPre \cap Id_{predP\ a} \vdash Id,\ \{\!|\lambda rv.\ \langle rv\rangle = predP\ a|\!\}$
begin

abbreviation *array'* :: *'a list* ⟹ *'s state* **where** $array' \equiv array\ ;_L\ snd_L$

partial-function (*lfp*) *findP-loop-evens* :: *nat* ⇒ (*'s state*, *unit*) *prog* **where**
findP-loop-evens i =
  do { *fO* ← *prog.read* get_{foundO}
    ; *prog.whenM* (*i* < *fO*)
      (*do* { *v* ← *prog.read* ($\lambda s.\ get_{array'}\ s\ !\ i$)
        ; *b* ← *prog.localize* (*pred v*)
        ; *if b then prog.write* ($\lambda s.\ put_{foundE}\ s\ i$) *else findP-loop-evens* (*i* + *2*)
        })
    }

partial-function (*lfp*) *findP-loop-odds* :: *nat* ⇒ (*'s state*, *unit*) *prog* **where**
findP-loop-odds i =
  do { *fE* ← *prog.read* get_{foundE}
    ; *prog.whenM* (*i* < *fE*)
      (*do* { *v* ← *prog.read* ($\lambda s.\ get_{array'}\ s\ !\ i$)
        ; *b* ← *prog.localize* (*pred v*)
        ; *if b then prog.write* ($\lambda s.\ put_{foundO}\ s\ i$) *else findP-loop-odds* (*i* + *2*)
        })
    }

definition *findP* :: (*'s*, *nat*) *prog* **where**
findP = *prog.local* (
  do { *N* ← *prog.read* (*SIZE* $get_{array'}$)
    ; *prog.write* ($\lambda s.\ put_{foundE}\ s\ N$)
    ; *prog.write* ($\lambda s.\ put_{foundO}\ s\ N$)
    ; (*findP-loop-evens 0* ∥ *findP-loop-odds 1*)
    ; *fE* ← *prog.read* (get_{foundE})
    ; *fO* ← *prog.read* (get_{foundO})
    ; *prog.return* (*min fE fO*)
    })

Relies and guarantees **abbreviation** (*input*) *A'* :: *'s rel* **where** $A' \equiv A^{=} \cap ceilr\ predPre \cap (\bigcap a.\ Id_{predP\ a})$

definition *AE* :: *'s state rel* **where**
$AE = UNIV \times_R A' \cap Id_{get_{array'}} \cap Id_{get_{foundE}} \cap \le_{get_{foundO}}$

definition *GE* :: *'s state rel* **where**
$GE = Id_{snd} \cap Id_{get_{foundO}} \cap \le_{get_{foundE}}$

definition *AO* :: *'s state rel* **where**
$AO = UNIV \times_R A' \cap Id_{get_{array'}} \cap Id_{get_{foundO}} \cap \le_{get_{foundE}}$

definition *GO* :: *'s state rel* **where**
$GO = Id_{snd} \cap Id_{get_{foundE}} \cap \le_{get_{foundO}}$

lemma *AG-refl-trans*:
shows
refl AE

refl AO
trans A $\Longrightarrow$ *trans AE*
trans A $\Longrightarrow$ *trans AO*
refl GE
refl GO
trans GE
trans GO
⟨*proof*⟩

lemma *AG-containment*:
shows $GO \subseteq AE$
and $GE \subseteq AO$
⟨*proof*⟩

lemma *G-containment*:
shows $GE \cup GO \subseteq UNIV \times_R Id$
⟨*proof*⟩

Safety proofs **lemma** *ag-findP-loop-evens*:
shows *prog.p2s* (*findP-loop-evens i*)
$\leq$ ⦃⟨*even i*⟩ ∧ (λ*s*. *predPre* (*snd s*)) ∧ get_{foundE} = *SIZE* $get_{array'}$ ∧ $get_{foundO} \leq$ *SIZE* $get_{array'}$⦄, *AE* ⊢ *GE*,
⦃λ-. (get_{foundE} < *SIZE* $get_{array'}$ ⟶ *localize1 predP* **$$** $get_{array'}$ **!** get_{foundE})
∧ (∀ *j*. ⟨*i* ≤ *j* ∧ *even j*⟩ ∧ ⟨*j*⟩ < *pred-min* get_{foundE} get_{foundO} ⟶ ¬ *localize1 predP* **$$** $get_{array'}$ **!** ⟨*j*⟩)⦄
⟨*proof*⟩

lemma *ag-findP-loop-odds*:
shows *prog.p2s* (*findP-loop-odds i*)
$\leq$ ⦃⟨*odd i*⟩ ∧ (λ*s*. *predPre* (*snd s*)) ∧ get_{foundO} = *SIZE* $get_{array'}$ ∧ $get_{foundE} \leq$ *SIZE* $get_{array'}$⦄, *AO* ⊢ *GO*,
⦃λ-. (get_{foundO} < *SIZE* $get_{array'}$ ⟶ *localize1 predP* **$$** $get_{array'}$ **!** get_{foundO})
∧ (∀ *j*. ⟨*i* ≤ *j* ∧ *odd j*⟩ ∧ ⟨*j*⟩ < *pred-min* get_{foundE} get_{foundO} ⟶ ¬ *localize1 predP* **$$** $get_{array'}$ **!** ⟨*j*⟩)⦄
⟨*proof*⟩

theorem *ag-findP*:
shows *prog.p2s findP*
$\leq$ ⦃*predPre*⦄, $A' \cap Id_{get_{array}}$
⊢ *Id*, ⦃λ*v s*. *v* = (*LEAST i*. *i* < *SIZE* get_{array} *s* ⟶ *predP* (get_{array} *s* ! *i*) *s*)⦄
⟨*proof*⟩

end

We conclude by showing how we can instantiate the above with a *coprime* predicate.

⟨*ML*⟩

type-synonym *'s state* = (*nat* × *nat*) × *'s*

abbreviation *x* :: *nat* $\Longrightarrow$ *'s gcd.state* **where** *x* ≡ fst_L $;_L$ fst_L
abbreviation *y* :: *nat* $\Longrightarrow$ *'s gcd.state* **where** *y* ≡ snd_L $;_L$ fst_L

definition *seq* :: *nat* ⇒ *nat* ⇒ (*'s*, *nat*) *prog* **where**
seq a b =
prog.local (
do { *prog.write* (λ*s*. $put_{gcd.x}$ *s a*)
; *prog.write* (λ*s*. $put_{gcd.y}$ *s b*)
; *prog.while* (λ-.
do { *xv* ← *prog.read* $get_{gcd.x}$

$\quad\quad\quad ;\ yv \leftarrow prog.read\ get_{gcd.y}$
$\quad\quad\quad ;\ if\ xv = yv$
$\quad\quad\quad\quad then\ prog.return\ (Inr\ ())$
$\quad\quad\quad\quad else\ (do\ \{\ (if\ xv < yv$
$\quad\quad\quad\quad\quad\quad then\ prog.write\ (\lambda s.\ put_{gcd.y}\ s\ (yv - xv))$
$\quad\quad\quad\quad\quad\quad else\ prog.write\ (\lambda s.\ put_{gcd.x}\ s\ (xv - yv)))$
$\quad\quad\quad\quad\quad ;\ prog.return\ (Inl\ ())\ \})$
$\quad\quad\quad \})\ ()$
$\quad\quad ;\ prog.read\ get_{gcd.x}$
$\quad\quad \})$

⟨*ML*⟩

lemma *seq*:
shows $prog.p2s\ (gcd.seq\ a\ b) \le \{\!\!\{\langle True\rangle\}\!\!\},\ UNIV \vdash Id,\ \{\!\!\{\lambda v.\ \langle v = gcd\ a\ b\rangle\}\!\!\}$
⟨*proof*⟩

⟨*ML*⟩

definition *findP-coprime* :: (*nat* × *nat list*, *nat*) *prog* **where**
$findP\text{-}coprime = findP\ (\lambda a.\ prog.read\ get_{fst_L} \gg\!\!= gcd.seq\ a \gg\!\!= (\lambda c.\ prog.return\ (c = 1)))\ snd_L$

lemma *ag-findP-coprime'*:
shows *prog.p2s findP-coprime*
$\le \{\!\!\{\langle True\rangle\}\!\!\},\ Id$
$\vdash Id,\ \{\!\!\{\lambda rv\ s.\ rv = (LEAST\ i.\ i < length\ (get_{snd_L}\ s) \longrightarrow coprime\ (get_{fst_L}\ s)\ (get_{snd_L}\ s\ !\ i))\}\!\!\}$
⟨*proof*⟩

lemma *ag-findP-coprime*: — Shuffle the parameter to the precondition, exploiting purity.
shows *prog.p2s findP-coprime*
$\le \{\!\!\{\langle a\rangle = get_{fst_L}\}\!\!\},\ Id$
$\vdash Id,\ \{\!\!\{\lambda rv\ s.\ rv = (LEAST\ i.\ i < length\ (get_{snd_L}\ s) \longrightarrow coprime\ a\ (get_{snd_L}\ s\ !\ i))\}\!\!\}$
⟨*proof*⟩

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

⟨*ML*⟩

record $('a, 's)$ *interface* =
  init :: $('s,\ unit)$ *prog*
  enq :: $'a \Rightarrow ('s,\ unit)$ *prog*
  deq :: $('s,\ 'a\ option)$ *prog*

type-synonym $'a$ *abstract* = $'a$ *set*

definition *abstract* :: $('a,\ 'a\ pending.abstract \times 's)$ *pending.interface* **where**
  abstract =
    ⦇ *pending.interface.init* = *prog.write* (*map-prod* ⟨{}⟩ *id*)
    , *pending.interface.enq* = $\lambda x.$ *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* × $'a$ *list* — a queue

fun *cdeq-update* :: $'a$ *pending.concrete* × $'s \Rightarrow 'a$ *option* × $'a$ *pending.concrete* × $'s$ **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.enq* = $\lambda x.$ *prog.write* (*map-prod* (*map-prod* ((#) x) *id*) *id*)
    , *pending.interface.deq* = *prog.det-action pending.cdeq-update*
    ⦈

abbreviation *absfn'* :: $'a$ *pending.concrete* ⇒ $'a$ *list* **where** — queue as a list
  absfn' s ≡ *snd* s @ *rev* (*fst* s)

definition *absfn* :: $'a$ *pending.concrete* ⇒ $'a$ *pending.abstract* **where**
  absfn s = *set* (*pending.absfn'* s)

⟨*ML*⟩

lemma *init*:
  fixes Q :: *unit* ⇒ $'a$ *pending.abstract* × $'s$ ⇒ *bool*
  fixes A :: $'s$ *rel*
  assumes *stable* ($Id \times_R A$) (Q ())
  shows *prog.p2s* (*pending.init pending.abstract*) ≤ ⦃$\lambda s.$ Q () ({}, *snd* s)⦄, $Id \times_R A \vdash UNIV \times_R Id$, ⦃$Q$⦄
⟨*proof*⟩

lemma *enq*:
  fixes x :: $'a$
  fixes Q :: *unit* ⇒ $'a$ *pending.abstract* × $'s$ ⇒ *bool*
  fixes A :: $'s$ *rel*
  assumes *stable* ($Id \times_R A$) (Q ())
  shows *prog.p2s* (*pending.enq pending.abstract* x) ≤ ⦃$\lambda s.$ Q () (*insert* x (*fst* s), *snd* s)⦄, $Id \times_R A \vdash UNIV \times_R$ Id, ⦃Q⦄
⟨*proof*⟩

lemma *deq*:

fixes *Q* :: *'a option* ⇒ *'a pending.abstract* × *'s* ⇒ *bool*
fixes *A* :: *'s rel*
assumes ⋀*v. stable* (*Id* $\times_R$ *A*) (*Q v*)
shows *prog.p2s* (*pending.deq pending.abstract*) ≤ ⦃λ*s. if fst s* = {} *then Q None s else* (∀ *x X. fst s* = *insert x X* ⟶ *Q* (*Some x*) (*X*, *snd s*))⦄, *Id* $\times_R$ *A* ⊢ *UNIV* $\times_R$ *Id*, ⦃*Q*⦄
⟨*proof*⟩

⟨*ML*⟩

record (*'a*, *'s*) *interface* =
init :: (*'s*, *unit*) *prog*
ins :: *'a* ⇒ (*'s*, *unit*) *prog*
mem :: *'a* ⇒ (*'s*, *bool*) *prog*

type-synonym *'a abstract* = *'a list* — model finite sets

definition *abstract* :: (*'a*, *'s* × *'a set.abstract* × *'t*) *set.interface* **where**
abstract =
⦇ *set.interface.init* = *prog.write* (*map-prod id* (*map-prod* ⟨[]⟩ *id*))
, *set.interface.ins* = λ*x. prog.write* (*map-prod id* (*map-prod* ((#) *x*) *id*))
, *set.interface.mem* = λ*x. prog.read* (λ*s. x* ∈ *set* (*fst* (*snd s*)))
⦈

⟨*ML*⟩

lemma *init*:
fixes *Q* :: *unit* ⇒ *'s* × *'a set.abstract* × *'t* ⇒ *bool*
fixes *A* :: *'s rel*
assumes *stable* (*A* $\times_R$ *Id* $\times_R$ *B*) (*Q* ())
shows *prog.p2s* (*set.init set.abstract*) ≤ ⦃λ*s. Q* () (*fst s*, [], *snd* (*snd s*))⦄, *A* $\times_R$ *Id* $\times_R$ *B* ⊢ *Id* $\times_R$ *UNIV* $\times_R$ *Id*, ⦃*Q*⦄
⟨*proof*⟩

lemma *ins*:
fixes *x* :: *'a*
fixes *Q* :: *unit* ⇒ *'s* × *'a set.abstract* × *'t* ⇒ *bool*
fixes *A* :: *'s rel*
assumes *stable* (*A* $\times_R$ *Id* $\times_R$ *B*) (*Q* ())
shows *prog.p2s* (*set.ins set.abstract x*) ≤ ⦃λ*s. Q* () (*fst s*, *x* # *fst* (*snd s*), *snd* (*snd s*))⦄, *A* $\times_R$ *Id* $\times_R$ *B* ⊢ *Id* $\times_R$ *UNIV* $\times_R$ *Id*, ⦃*Q*⦄
⟨*proof*⟩

lemma *mem*:
fixes *Q* :: *bool* ⇒ *'s* × *'a set.abstract* × *'t* ⇒ *bool*
assumes ⋀*v. stable* (*A* $\times_R$ *Id* $\times_R$ *B*) (*Q v*)
shows *prog.p2s* (*set.mem set.abstract x*) ≤ ⦃λ*s. Q* (*x* ∈ *set* (*fst* (*snd s*))) *s*⦄, *A* $\times_R$ *Id* $\times_R$ *B* ⊢ *Id* $\times_R$ *UNIV* $\times_R$ *Id*, ⦃*Q*⦄
⟨*proof*⟩

⟨*ML*⟩

context
fixes *pending* :: (*'a*, *'p* × *'a set.abstract* × *'s*) *pending.interface*
fixes *f* :: *'a* ⇒ *'a list*
begin

definition *loop* :: *'a pred* ⇒ (*'p* × *'a set.abstract* × *'s*, *'a option*) *prog* **where**
loop p =

```
  prog.while (λ-.
    do { aopt ← pending.deq pending
       ; case aopt of
           None ⇒ prog.return (Inr None)
         | Some x ⇒
             if p x
             then prog.return (Inr (Some x))
             else do { prog.app (λy. do { b ← 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*: $Id \times_R UNIV \times_R Id \subseteq UNIV \times_R UNIV \times_R Id$
⟨*proof*⟩

context
fixes f :: $'a \Rightarrow 'a$ *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 ⟷
$q \subseteq set\ v$
$\wedge\ set\ v \subseteq path\ ``\ \{x_0\}$
$\wedge\ set\ v \cap Collect\ P \subseteq q$
$\wedge\ x_0 \in set\ v$

definition *vclosureP* :: $'a \Rightarrow 'a$ *pending.abstract* $\Rightarrow 'a$ *set.abstract* $\Rightarrow$ *bool* **where**
vclosureP x q v $\longleftrightarrow$ $(x \in set\ v - q \longrightarrow step\ ``\ \{x\} \subseteq set\ v)$

definition *search-postP* :: $'a$ *option* $\Rightarrow$ *bool* **where**
search-postP rv = (*case rv of*
None $\Rightarrow$ *finite* $(path\ ``\ \{x_0\}) \wedge (path\ ``\ \{x_0\} \cap Collect\ P = \{\})$
| *Some y* $\Rightarrow (x_0,\ y) \in path \wedge P\ y)$

abbreviation *aloop-inv s* $\equiv$ *aloop-invP* (*fst s*) (*fst* (*snd s*))
abbreviation *vclosure x s* $\equiv$ *vclosureP x* (*fst s*) (*fst* (*snd s*))
abbreviation *search-post rv* $\equiv$ ⟨*search-postP rv*⟩

lemma *vclosureP-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*⟩

lemma *vclosureP-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*
⟨*proof*⟩

lemma *vclosureP-init*:
shows *vclosureP x* $\{x_0\}$ $[x_0]$
⟨*proof*⟩

lemma *vclosureP-step*:
assumes $\forall\, z.\ x \neq z \longrightarrow vclosureP\ z\ q\ v$
assumes $x \neq z$
shows *vclosureP z* (*insert y q*) (*y* # *v*)
⟨*proof*⟩

lemma *vclosureP-dequeue*:
assumes $\forall\, z.\ vclosureP\ z\ (insert\ x\ q)\ v$
assumes $x \neq z$
shows *vclosureP z q v*
⟨*proof*⟩

lemma *aloop-invPD*:
assumes *aloop-invP q v*
assumes $x \in q$
shows $(x_0,\ x) \in path$
⟨*proof*⟩

lemma *aloop-invP-init*:
shows *aloop-invP* $\{x_0\}$ $[x_0]$
⟨*proof*⟩

lemma *aloop-invP-step*:
assumes *aloop-invP q v*
assumes $(x_0,\ x) \in path$
assumes $y \in set\ (f\ x) - set\ v$
shows *aloop-invP* (*insert y q*) (*y* # *v*)
⟨*proof*⟩

lemma *aloop-invP-dequeue*:
assumes *aloop-invP* (*insert x q*) *v*
assumes $\neg$ *P x*
shows *aloop-invP q v*
⟨*proof*⟩

lemma *search-postcond-None*:
assumes *aloop-invP* {} *v*
assumes $\forall$ *y*. *vclosureP y* {} *v*
shows *search-postP None*
⟨*proof*⟩

lemma *search-postcond-Some*:
assumes *aloop-invP q v*
assumes $x \in q$
assumes *P x*
shows *search-postP* (*Some x*)
⟨*proof*⟩

lemmas *stable-simps* =
prod.sel
split-def
sum.simps

lemma *ag-aloop*:
shows *prog.p2s* (*aloop f P*) $\leq$ ⦃*aloop-inv* $\wedge$ ($\forall$ *x*. *vclosure x*)⦄, *Id* $\times_R$ *Id* $\times_R$ *UNIV* $\vdash$ *UNIV* $\times_R$ *UNIV* $\times_R$ *Id*, ⦃*search-post*⦄
⟨*proof*⟩

lemma *ag-amain*:
shows *prog.p2s* (*amain f P* x_0) $\leq$ ⦃⟨*True*⟩⦄, *Id* $\times_R$ *Id* $\times_R$ *UNIV* $\vdash$ *UNIV* $\times_R$ *UNIV* $\times_R$ *Id*, ⦃*search-post*⦄
⟨*proof*⟩

lemma *ag-asearch*:
shows *prog.p2s* (*asearch f P* x_0 :: (*'s*, *'a option*) *prog*) $\leq$ ⦃⟨*True*⟩⦄, *UNIV* $\vdash$ *Id*, ⦃*search-post*⦄
⟨*proof*⟩

Refinement **abbreviation** $A \equiv$ *ag.assm* (*Id* $\times_R$ *Id* $\times_R$ *UNIV*)
abbreviation *absfn c* $\equiv$ *prog.sinvmap* (*map-prod pending.absfn id*) *c*
abbreviation *p2s-absfn c* $\equiv$ *prog.p2s* (*absfn c*)

— visited set: reflexive
lemma *ref-set-init*:
shows *prog.p2s* (*set.init set.abstract*) $\leq$ ⦃λs. *True*⦄, $A \Vdash$ *p2s-absfn* (*set.init set.abstract*), ⦃$\lambda v\ s$. *True*⦄
⟨*proof*⟩

lemma *ref-set-ins*:
shows *prog.p2s* (*set.ins set.abstract x*) $\leq$ ⦃λs. *True*⦄, $A \Vdash$ *p2s-absfn* (*set.ins set.abstract x*), ⦃$\lambda v\ s$. *True*⦄
⟨*proof*⟩

lemma *ref-set-mem*:
shows *prog.p2s* (*set.mem set.abstract x*) $\leq$ ⦃λs. *True*⦄, $A \Vdash$ *p2s-absfn* (*set.mem set.abstract x*), ⦃$\lambda v\ s$. *True*⦄
⟨*proof*⟩
lemma *ref-queue-init*:
shows *prog.p2s* (*pending.init pending.concrete*) $\leq$ ⦃λs. *True*⦄, $A \Vdash$ *p2s-absfn* (*pending.init pending.abstract*), ⦃$\lambda v\ s$. *True*⦄
⟨*proof*⟩

lemma *ref-queue-enq*:
shows *prog.p2s* (*pending.enq pending.concrete x*) ≤ ⦃λ*s. True*⦄, *A* ⊩ *p2s-absfn* (*pending.enq pending.abstract x*), ⦃λ*v s. True*⦄
⟨*proof*⟩

lemma *ref-queue-deq*:
shows *prog.p2s* (*pending.deq pending.concrete*) ≤ ⦃λ*s. True*⦄, *A* ⊩ *p2s-absfn* (*pending.deq pending.abstract*), ⦃λ*v s. True*⦄
⟨*proof*⟩
lemma *ref-bfs-loop*:
shows *prog.p2s* (*loop pending.concrete f P*) ≤ ⦃λ*s. True*⦄, *A* ⊩ *p2s-absfn* (*loop pending.abstract f P*), ⦃λ*v s. True*⦄
⟨*proof*⟩

lemma *ref-bfs-main*:
shows *prog.p2s* (*main pending.concrete f P x*) ≤ ⦃⟨*True*⟩⦄, *A* ⊩ *p2s-absfn* (*amain f P x*), ⦃λ*v s. True*⦄
⟨*proof*⟩

theorem *ref-bfs*:
shows *bfs f P x* ≤ *asearch f P x*
⟨*proof*⟩

theorem *bfs-post-le*:
shows *prog.p2s* (*bfs f P* x_0) ≤ *spec.post* (*search-post*)
⟨*proof*⟩

end

24 Observations about safety closure

We demonstrate that *Sup* does not distribute in ('*a*, '*s*, '*v*) *tls* as it does in ('*a*, '*s*, '*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* :: *nat* ⇒ ('*a* × *nat*, '*v*) *tllist* **where**
bnats i = *TCons* (*undefined*, *i*) (*bnats* (*Suc i*))

definition *bnat* :: ('*a*, *nat*, '*v*) *behavior.t* **where**
bnat = *behavior.B 0* (*bnats 1*)

definition *tnats* :: *nat* ⇒ *nat* ⇒ ('*a* × *nat*) *list* **where**
tnats i j = *map* (*Pair undefined*) (*upt i j*)

definition *tnat* :: *nat* ⇒ ('*a*, *nat*, '*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*
⟨*proof*⟩

lemma *take-tnats*:
shows *take i* (*tnats j k*) = *tnats j* (*min* (*i* + *j*) *k*)
⟨*proof*⟩

lemma *take-tnat*:

shows *trace.take i* (*tnat j*) = *tnat* (*min i j*)
⟨*proof*⟩

lemma *mono-tnat*:
shows *mono tnat*
⟨*proof*⟩

lemma *final′-tnats*:
shows *trace.final′ i* (*tnats j k*) = (*if j* < *k then k* − *1 else i*)
⟨*proof*⟩

lemma *sset-tnat*:
shows *trace.sset* (*tnat i*) = {*j*. *j* ≤ *i*}
⟨*proof*⟩

lemma *natural′-tnats*:
shows *trace.natural′ i* (*tnats* (*Suc i*) (*Suc j*)) = *tnats* (*Suc i*) (*Suc j*)
⟨*proof*⟩

lemma *natural-tnat*:
shows ♮(*tnat i*) = *tnat i*
⟨*proof*⟩

lemma *ttake-bnats*:
shows *ttake i* (*bnats j*) = (*tnats j* (*i* + *j*), *None*)
⟨*proof*⟩

lemma *take-bnat-tnat*:
shows *behavior.take i bnat* = *tnat i*
⟨*proof*⟩

unbundle *tls.extra-syntax*

definition *bnat-approx* :: (*unit*, *nat*, *unit*) *spec set* **where**
bnat-approx = {⦇*behavior.take i bnat*⦈ |*i*. *True*}

lemma *bnat-approx-alt-def*:
shows *bnat-approx* = {⦇*tnat i*⦈ |*i*. *True*}
⟨*proof*⟩

lemma *not-tls-from-spec-Sup-distrib*: — *tls.from-spec* is not *Sup*-distributive
shows ¬*tls.from-spec* (⨆ *bnat-approx*) ≤ ⨆(*tls.from-spec* ‘ *bnat-approx*) (**is** ¬*?lhs* ≤ *?rhs*)
⟨*proof*⟩

definition *bnat′* :: (*unit*, *nat*, *unit*) *tls set* **where**
bnat′ = *tls.from-spec* ‘ *bnat-approx*

lemma *not-tls-safety-cl-Sup-distrib*: — *tls.safety.cl* is not *Sup*-distributive
shows ¬*tls.safety.cl* (⨆ *bnat′*) ≤ ⨆(*tls.safety.cl* ‘ *bnat′*)
⟨*proof*⟩

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*
⟨*proof*⟩

Negation does not preserve *tls.safety.closed* **notepad**
begin

⟨*proof*⟩

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

⟨*ML*⟩

lemma *dense-alt-def*: — Lynch (1996, §8.5.3 Liveness Property)
  shows $(raw.safety.dense :: ('a, 's, 'v)\ behavior.t\ set\ set)$
    $= \{P.\ \forall\sigma.\ \exists xsv.\ \sigma\ @-_B\ xsv \in P\}$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

⟨*ML*⟩

definition *live* :: $('a, 's, 'v)$ *tls set* **where**
  $live = tls.safety.dense$

⟨*ML*⟩

lemma *not-bot*:
  shows $\bot \notin tls.live$
⟨*proof*⟩

lemma *top*:
  shows $\top \in tls.live$
⟨*proof*⟩

lemma *live-le*:
  assumes $P \in tls.live$
  assumes $P \leq Q$
  shows $Q \in tls.live$
⟨*proof*⟩

lemma *inf-safety-eq-top*: — Lynch (1996, Theorem 8.8)
  shows $tls.live \sqcap tls.safety.closed = \{\top\}$
⟨*proof*⟩

lemma *terminating*:
  shows $tls.eventually\ tls.terminated \in tls.live$
⟨*proof*⟩

However this definition of liveness does not endorse traditional *response* properties.

corec *alternating* :: *bool* ⇒ (*'a* × *bool*, *'b*) *tllist* **where**
  alternating b = *TCons* (*undefined*, *b*) (*alternating* (¬*b*))

abbreviation (*input*) *A b* ≡ *behavior.B b* (*tls.live.alternating* (¬*b*))

lemma *dropn-alternating*:
  shows *behavior.dropn i* (*tls.live.A b*) = *Some* (*tls.live.A* (*if even i then b else* ¬*b*))
⟨*proof*⟩

notepad
begin

⟨*proof*⟩

end

⟨*ML*⟩

The famous decomposition **definition** *Safe* :: (*'a*, *'s*, *'v*) *tls* ⇒ (*'a*, *'s*, *'v*) *tls* **where**
  Safe P = *tls.safety.cl P*

definition *Live* :: (*'a*, *'s*, *'v*) *tls* ⇒ (*'a*, *'s*, *'v*) *tls* **where**
  Live P = *P* ⊔ −*tls.safety.cl P*

lemma *decomp*:
  shows *P* = *tls.Safe P* ⊓ *tls.Live P*
⟨*proof*⟩

⟨*ML*⟩

lemma *Safe*:
  shows *tls.Safe P* ∈ *tls.safety.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *Live*:
  shows *tls.Live P* ∈ *tls.live*
⟨*proof*⟩

⟨*ML*⟩

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* :: $'a \times 'a \Rightarrow 'a$ *list*
  fixes *index* :: $'a \times 'a \Rightarrow 'a \Rightarrow nat$
  assumes *index*: $i \in set\ (interval\ b) \Longrightarrow interval\ b\ !\ index\ b\ i = i$
  assumes *interval*: $map\ (index\ b)\ (interval\ b) = [0..<length\ (interval\ b)]$

lemma *index-length*:
  assumes $i \in set\ (interval\ b)$
  shows $index\ b\ i < length\ (interval\ b)$
⟨*proof*⟩

lemma *distinct-interval*:
  shows *distinct* (*interval b*)
⟨*proof*⟩

lemma *inj-on-index*:
  shows *inj-on* (*index b*) (*set* (*interval b*))
⟨*proof*⟩

lemma *index-eq-conv*:
  assumes $i \in set\ (interval\ b)$
  assumes $j \in set\ (interval\ b)$
  shows $index\ b\ i = index\ b\ j \longleftrightarrow i = j$
⟨*proof*⟩

lemma *index-inv-into*:
  assumes $i < length\ (interval\ b)$
  shows $inv\text{-}into\ (set\ (interval\ b))\ (index\ b)\ i \in set\ (interval\ b)$
⟨*proof*⟩

lemma *linear-order-on*:
  shows $linear\text{-}order\text{-}on\ (set\ (interval\ b))\ \{(i, j).\ \{i, j\} \subseteq set\ (interval\ b) \wedge index\ b\ i \leq index\ b\ j\}$
⟨*proof*⟩

lemma *interval-map*:
  shows $map\ (\lambda i.\ f\ (interval\ b\ !\ i))\ [0..<length\ (interval\ b)] = map\ f\ (interval\ b)$
⟨*proof*⟩

lemma *index-forE*:
  assumes $i < length\ (interval\ b)$
  obtains *j* **where** $j \in set\ (interval\ b)$ **and** $index\ b\ j = i$
⟨*proof*⟩

instantiation *unit* :: *Ix*

begin

definition *interval-unit* = (λ(*x*::*unit*, *y*::*unit*). [()])
definition *index-unit* = (λ(*x*::*unit*, *y*::*unit*) -::*unit*. *0*::*nat*)

instance ⟨*proof*⟩

end

instantiation *nat* :: *Ix*
begin

definition *interval-nat* = (λ(*l*, *u*::*nat*). [*l*..<*Suc u*]) — bounds are inclusive
definition *index-nat* = (λ(*l*, *u*::*nat*) *i*::*nat*. *i* − *l*)

lemma *upt-minus*:
  shows *map* (λ*i*. *i* − *l*) [*l*..<*u*] = [*0*..<*u* − *l*]
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *int* :: *Ix*
begin

definition *interval-int* = (λ(*l*, *u*::*int*). [*l*..*u*]) — bounds are inclusive
definition *index-int* = (λ(*l*, *u*::*int*) *i*::*int*. *nat* (*i* − *l*))

lemma *upto-minus*:
  shows *map* (λ*i*. *nat* (*i* − *l*)) [*l*..*u*] = [*0*..<*nat* (*u* − *l* + *1*)]
⟨*proof*⟩

instance ⟨*proof*⟩

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'*))))
⟨*proof*⟩

instance
⟨*proof*⟩

end

⟨*ML*⟩

lemma *interval-conv*:
shows $(x,\ y) \in set\ (interval\ b) \longleftrightarrow x \in set\ (interval\ (fst\text{-}bounds\ b)) \wedge y \in set\ (interval\ (snd\text{-}bounds\ b))$
⟨*proof*⟩

⟨*ML*⟩

type-synonym $'i\ square = ('i,\ 'i)\ two\text{-}dim$

definition $square :: 'i{::}Ix\ Ix.square \Rightarrow bool$ **where**
$square = (\lambda((l,\ l'),\ (u,\ u')).\ Ix.interval\ (l,\ u) = Ix.interval\ (l',\ u'))$

⟨*ML*⟩

lemma *conv*:
assumes *Ix.square b*
shows $i \in set\ (Ix.interval\ (fst\text{-}bounds\ b)) \longleftrightarrow i \in set\ (Ix.interval\ (snd\text{-}bounds\ b))$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

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* ⇀ *heap.rep list* — partial map from addresses to structured encoded values

abbreviation *empty* :: *heap.t* **where**
empty ≡ *Map.empty*

primrec *apply-write* :: *heap.write* ⇒ *heap.t* ⇒ *heap.t* **where**
apply-write (*heap.Write addr i x*) *s* = *s*(*addr* ↦ (*the* (*s addr*))[*i* := *x*])

class *rep* = — the class of representable types
assumes *ex-inj*: ∃ *to-heap-rep* :: ′*a* ⇒ *heap.rep*. *inj to-heap-rep*

⟨*ML*⟩

lemma *countable-classI*[*intro*!]:
shows *OFCLASS*(′a::*countable*, *heap.rep-class*)
⟨*proof*⟩

definition *to* :: ′*a*::*heap.rep* ⇒ *heap.rep* **where**
to = (*SOME f*. *inj f*)

definition *from* :: *heap.rep* ⇒ ′*a*::*heap.rep* **where**
from = *inv* (*heap.rep.to* :: ′*a* ⇒ *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*
⟨*proof*⟩

lemma *surj-from*[*simp*]: *surj heap.rep.from*
⟨*proof*⟩

lemma *to-split*[*simp*]: *heap.rep.to x* = *heap.rep.to y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *from-to*[*simp*]:
shows *heap.rep.from* (*heap.rep.to x*) = *x*
⟨*proof*⟩

instance *unit* :: *heap.rep* ⟨*proof*⟩

instance *bool* :: *heap.rep* ⟨*proof*⟩

instance *nat* :: *heap.rep* ⟨*proof*⟩

instance *int* :: *heap.rep* ⟨*proof*⟩

instance *char* :: *heap.rep* ⟨*proof*⟩

instance *String.literal* :: *heap.rep* ⟨*proof*⟩

instance *typerep* :: *heap.rep* ⟨*proof*⟩

⟨*ML*⟩

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 intutive 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 $-rs$
definition *Id-on* :: *'a::heap.addr-of set ⇒ heap.t rel* (‹*heap.Id*$_{-}$›) **where**
  $heap.Id_{rs} = (\bigcap r{\in}rs.\ heap.rep\text{-}inv\text{-}rel\ r \cap Id_{\lambda s.\ s\ (heap.addr\text{-}of\ r)})$

— allows arbitrary changes to *rs* provided the *rep-inv* of *rs* is respected. requires addresses in −*heap.addr-of* ' *rs* to be unchanged
definition *modifies* :: *'a::heap.addr-of set ⇒ heap.t rel* (‹*heap.modifies*$_{-}$›) **where**
  $heap.modifies_{rs} = (\bigcap r{\in}rs.\ heap.rep\text{-}inv\text{-}rel\ r) \cap \{(s,\ s').\ \forall r{\in}-heap.addr\text{-}of\ `\ rs.\ s\ r = s'\ r\}$

⟨*ML*⟩

lemma *cong*:
  assumes *s (heap.addr-of r) = s' (heap.addr-of r')*
  shows *heap.get r s = heap.get r' s'*
⟨*proof*⟩

lemma *Id-on-proj-cong*:
  assumes $(s,\ s') \in heap.Id_{\{r\}}$
  shows *heap.get r s = heap.get r s'*

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-eq*:
  shows *heap.get r* (*heap.set r v s*) = *v*
⟨*proof*⟩

lemma *set-neq*:
  assumes *heap.addr-of r* ≠ *heap.addr-of r′*
  shows *heap.get r* (*heap.set r′ v s*) = *heap.get r s*
⟨*proof*⟩

⟨*ML*⟩

lemma *cong*:
  assumes *heap.addr-of r* = *heap.addr-of r′*
  assumes *v* = *v′*
  assumes ⋀*r′*. *r′* ≠ *heap.addr-of r* ⟹ *s r′* = *s′ r′*
  shows *heap.set r v s* = *heap.set r′ v′ s′*
⟨*proof*⟩

lemma *empty*:
  shows *heap.set r v* (*heap.empty*) = [*heap.addr-of r* ↦ *map heap.rep.to v*]
⟨*proof*⟩

lemma *fun-upd*:
  shows *heap.set r v* (*fun-upd s a w*) = (*fun-upd s a w*)(*heap.addr-of r* ↦ *map heap.rep.to v*)
⟨*proof*⟩

lemma *same*:
  shows *heap.set r v* (*heap.set r w s*) = *heap.set r v s*
⟨*proof*⟩

lemma *twist*:
  assumes *heap.addr-of r* ≠ *heap.addr-of r′*
  shows *heap.set r v* (*heap.set r′ w s*) = *heap.set r′ w* (*heap.set r v s*)
⟨*proof*⟩

⟨*ML*⟩

lemma *cong*[*cong*]:
  fixes *P* :: *heap.rep list pred*
  assumes ⋀*v*. *s r* = *Some v* ⟹ *P v* = *P′ v*
  assumes *r* = *r′*
  assumes *s r* = *s′ r′*
  shows *heap.obj-at P r s* ⟷ *heap.obj-at P′ r′ s′*
⟨*proof*⟩

lemma *split*:
  shows *Q* (*heap.obj-at P r s*) ⟷ (*s r* = *None* ⟶ *Q False*) ∧ (∀ *v*. *s r* = *Some v* ⟶ *Q* (*P v*))
⟨*proof*⟩

lemma *split-asm*:
  shows *Q* (*heap.obj-at P r s*) ⟷ ¬ ((*s r* = *None* ∧ ¬*Q False*) ∨ (∃ *v*. *s r* = *Some v* ∧ ¬ *Q* (*P v*)))

⟨*proof*⟩

lemmas *splits* = *heap.obj-at.split heap.obj-at.split-asm*

lemma *empty*:
 shows ¬*heap.obj-at P r heap.empty*
⟨*proof*⟩

lemma *set*:
 shows *heap.obj-at P r* (*heap.set r′ v s*)
 $\longleftrightarrow$ (*r* = *heap.addr-of r′* ∧ *P* (*map heap.rep.to v*)) ∨ (*r* ≠ *heap.addr-of r′* ∧ *heap.obj-at P r s*)
⟨*proof*⟩

lemma *fun-upd*:
 shows *heap.obj-at P r* (*fun-upd s a* (*Some w*)) = (*if r* = *a then P w else heap.obj-at P r s*)
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

lemma *empty*[*simp*]:
 shows $heap.Id_{\{\}}$ = *UNIV*
⟨*proof*⟩

lemma *sup*:
 shows $heap.Id_{X \cup Y} = heap.Id_X \cap heap.Id_Y$
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*[*simp*]:
 shows $heap.modifies_{\{\}}$ = *Id*
⟨*proof*⟩

lemma *rep-inv-rel-le*:
 shows $heap.modifies_{rs}$ ⊆ (⋂ *r*∈*rs*. *heap.rep-inv-rel r*)
⟨*proof*⟩

lemma *rep-inv*:
 assumes (*s*, *s′*) ∈ $heap.modifies_{\{a\}}$
 shows *heap.rep-inv a s*
 and *heap.rep-inv a s′*
⟨*proof*⟩

lemma *Id-conv*:
 shows (*s*, *s*) ∈ $heap.modifies_{rs}$ $\longleftrightarrow$ (∀ *r*∈*rs*. (*s*, *s*) ∈ *heap.rep-inv-rel r*)
⟨*proof*⟩

lemma *eqI*:
  assumes $(s, s') \in heap.modifies_{rs}$
  assumes $\bigwedge r.\ [\![r \in rs;\ heap.rep\text{-}inv\ r\ s;\ heap.rep\text{-}inv\ r\ s']\!] \Longrightarrow s\ (heap.addr\text{-}of\ r) = s'\ (heap.addr\text{-}of\ r)$
  shows $s = s'$
⟨*proof*⟩

⟨*ML*⟩

lemma *Id-on-frame-cong*:
  assumes $\bigwedge s\ s'.\ (\bigwedge r.\ r \in rs \Longrightarrow heap.rep\text{-}inv\ r\ s \wedge heap.rep\text{-}inv\ r\ s' \wedge s\ (heap.addr\text{-}of\ r) = s'\ (heap.addr\text{-}of\ r))$
$\Longrightarrow P\ s \longleftrightarrow P'\ s'$
  shows $stable\ heap.Id_{rs}\ P \longleftrightarrow stable\ heap.Id_{rs}\ P'$
⟨*proof*⟩

lemma *Id-on-frameI*:
  assumes $\bigwedge s\ s'.\ (\bigwedge r.\ r \in rs \Longrightarrow heap.rep\text{-}inv\ r\ s \wedge heap.rep\text{-}inv\ r\ s' \wedge s\ (heap.addr\text{-}of\ r) = s'\ (heap.addr\text{-}of\ r))$
$\Longrightarrow P\ s \longleftrightarrow P\ s'$
  shows $stable\ heap.Id_{rs}\ P$
⟨*proof*⟩

lemma *Id-on-rep-invI*[*stable.intro*]:
  assumes $r \in rs$
  shows $stable\ heap.Id_{rs}\ (heap.rep\text{-}inv\ r)$
⟨*proof*⟩

⟨*ML*⟩

25.1 References

datatype *'a ref* = *Ref* (*addr-of*: *heap.addr*)

instantiation *ref* :: (*heap.rep*) *heap.addr-of*
begin

definition *addr-of-ref* :: *'a ref* ⇒ *heap.addr* **where**
  addr-of-ref = *ref.addr-of*

definition *rep-val-inv-ref* :: *'a ref* ⇒ *heap.rep list pred* **where**
  rep-val-inv-ref r vs ⟷ (*case vs of* [*v*] ⇒ *heap.rep.to* (*heap.rep.from v* :: *'a*) = *v* | -⇒ *False*)

instance ⟨*proof*⟩

end

instance *ref* :: (*heap.rep*) *heap.rep*
⟨*proof*⟩

⟨*ML*⟩

definition *get* :: *'a::heap.rep ref* ⇒ *heap.t* ⇒ *'a* **where**
  get r s = *hd* (*heap.get r s*)

definition *set* :: *'a::heap.rep ref* ⇒ *'a* ⇒ *heap.t* ⇒ *heap.t* **where**
  set r v s = *heap.set r* [*v*] *s*

definition *alloc* :: *'a* ⇒ *heap.t* ⇒ (*'a::heap.rep ref* × *heap.t*) *set* **where**
  alloc v s = {(*r*, *Ref.set r v s*) |*r*. ¬*heap.present r s*}

lemma *addr-of*:
  shows *heap.addr-of* (*Ref r*) = *r*
⟨*proof*⟩

⟨*ML*⟩

lemma *fun-upd*:
  shows *Ref.get r* (*fun-upd s a* (*Some* [*w*]))
    = (*if heap.addr-of r* = *a then heap.rep.from w else Ref.get r s*)
⟨*proof*⟩

lemma *set-eq*:
  shows *Ref.get r* (*Ref.set r v s*) = *v*
⟨*proof*⟩

lemma *set-neq*:
  fixes *r* :: *'a::heap.rep ref*
  fixes *r'* :: *'b::heap.rep ref*
  assumes *addr-of r* ≠ *addr-of r'*
  shows *Ref.get r* (*Ref.set r' v s*) = *Ref.get r s*
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
  shows *Ref.set r v* (*heap.empty*) = [*heap.addr-of r* ↦ [*heap.rep.to v*]]
⟨*proof*⟩

lemma *fun-upd*:
  shows *Ref.set r v* (*fun-upd s a w*) = (*fun-upd s a w*)(*heap.addr-of r* ↦ [*heap.rep.to v*])
⟨*proof*⟩

lemma *same*:
  shows *Ref.set r v* (*Ref.set r w s*) = *Ref.set r v s*
⟨*proof*⟩

lemma *obj-at-conv*:
  fixes *a* :: *heap.addr*
  fixes *r* :: *'a::heap.rep ref*
  fixes *v* :: *'a*
  fixes *P* :: *heap.rep list pred*
  shows *heap.obj-at P a* (*Ref.set r v s*) ⟷ (*a* = *heap.addr-of r* ∧ *P* [*heap.rep.to v*])
      ∨ (*a* ≠ *heap.addr-of r* ∧ *heap.obj-at P a s*)
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

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-val-inv*

datatype *'a one-dim-array* = *Array* (*addr-of*: *heap.addr*)

instantiation *one-dim-array* :: (*type*) *heap.addr-of*
begin

definition *addr-of-one-dim-array* :: *'a one-dim-array* ⇒ *heap.addr* **where**
addr-of-one-dim-array = *addr-of*

definition *rep-val-inv-one-dim-array* :: *'a one-dim-array* ⇒ *heap.rep list pred* **where**
[*simp*]: *rep-val-inv-one-dim-array a vs* ⟷ *True*

instance ⟨*proof*⟩

end

⟨*ML*⟩

definition *get* :: *'a::heap.rep one-dim-array* ⇒ *nat* ⇒ *heap.t* ⇒ *'a* **where**
get a i s = *heap.get a s* ! *i*

definition *set* :: *'a::heap.rep one-dim-array* ⇒ *nat* ⇒ *'a* ⇒ *heap.t* ⇒ *heap.t* **where**
set a i v s = *heap.set a* ((*heap.get a s*)[*i*:=*v*]) *s*

definition *alloc* :: *'a list* ⇒ *heap.t* ⇒ (*'a::heap.rep one-dim-array* × *heap.t*) *set* **where**
alloc av s = {(*a*, *heap.set a av s*) |*a*. ¬*heap.present a s*}

definition *list-for* :: *'a::heap.rep one-dim-array* ⇒ *heap.t* ⇒ *'a list* **where**
list-for a = *heap.get a*

⟨*ML*⟩

lemma *weak-cong*:
assumes $i = i'$
assumes $a = a'$
assumes *s* (*heap.addr-of a*) = *s'* (*heap.addr-of a'*)
shows *ODArray.get a i s* = *ODArray.get a' i' s'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *set-eq*:
assumes *i* < *length* (*the* (*s* (*heap.addr-of a*)))

shows *ODArray.get a i* (*ODArray.set a i v s*) = *v*
⟨*proof*⟩

lemma *set-neq*:
assumes $i \neq j$
shows *ODArray.get a i* (*ODArray.set a j v s*) = *ODArray.get a i s*
⟨*proof*⟩

⟨*ML*⟩

25.2.2 User-facing arrays

datatype ('*i*, '*a*) *array* = *Array* (*bounds*: ('*i* × '*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* ⇒ *heap.addr* **where**
addr-of-array = *addr-of* ∘ *array.arr*

definition *rep-val-inv-array* :: ('*a*, '*b*) *array* ⇒ *heap.rep list pred* **where**
rep-val-inv-array a vs ⟷
length vs = *length* (*Ix.interval* (*array.bounds a*))
∧ (∀ *v*∈*set vs*. *heap.rep.to* (*heap.rep.from v* :: '*b*) = *v*)

instance ⟨*proof*⟩

end

instance *array* :: (*countable*, *type*) *heap.rep*
⟨*proof*⟩

⟨*ML*⟩

abbreviation (*input*) *square* :: ('*i*::*Ix* × '*i*, '*a*) *array* ⇒ *bool* **where**
square a ≡ *Ix.square* (*array.bounds a*)

abbreviation (*input*) *index* :: ('*i*::*Ix*, '*a*) *array* ⇒ '*i* ⇒ *nat* **where**
index a ≡ *Ix.index* (*array.bounds a*)

abbreviation (*input*) *interval* :: ('*i*::*Ix*, '*a*) *array* ⇒ '*i list* **where**
interval a ≡ *Ix.interval* (*array.bounds a*)

definition *get* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ *heap.t* ⇒ '*a* **where**
get a i = *ODArray.get* (*array.arr a*) (*Array.index a i*)

definition *set* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ '*a* ⇒ *heap.t* ⇒ *heap.t* **where**
set a i v = *ODArray.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 = *ODArray.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**
$Array.Id_{a,\ is} = heap.rep\text{-}inv\text{-}rel\ a \cap \{(s,\ s').\ \forall i{\in}is.\ Array.get\ a\ i\ s = Array.get\ a\ i\ s'\}$

definition *modifies* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i set* ⇒ *heap.t rel* (‹*Array.modifies*$_{\_, \_}$›) **where**
$Array.modifies_{a,\ is}$
$= heap.modifies_{\{a\}} \cap \{(s,\ s').\ \forall i{\in}set\ (Array.interval\ a) - is.\ Array.get\ a\ i\ s = Array.get\ a\ i\ s'\}$

lemma *simps*[*simp*]:
shows *heap.addr-of* (*array.arr a*) = *heap.addr-of a*
and *heap.addr-of* ∘ *array.arr* = *heap.addr-of*
⟨*proof*⟩

⟨*ML*⟩

lemma *set-eq*:
assumes *heap.rep-inv a s*
assumes $i \in set\ (Array.interval\ a)$
shows *Array.get a i* (*Array.set a i v s*) = *v*
⟨*proof*⟩

lemma *set-neq*:
assumes $i \in set\ (Array.interval\ a)$
assumes $j \in set\ (Array.interval\ a)$
assumes $i \neq j$
shows *Array.get a j* (*Array.set a i v s*) = *Array.get a j s*
⟨*proof*⟩

lemma *Id-on-proj-cong*:
assumes $a = a'$
assumes $i = i'$
assumes $(s,\ s') \in Array.Id_{a',\ \{i'\}}$
assumes $i' \in set\ (Array.interval\ a)$
shows $Array.get\ a\ i\ s = Array.get\ a'\ i'\ s'$
⟨*proof*⟩

lemma *weak-cong*:
assumes $a = a'$
assumes $i = i'$
assumes $s\ (heap.addr\text{-}of\ a) = s'\ (heap.addr\text{-}of\ a')$
shows $Array.get\ a\ i\ s = Array.get\ a'\ i'\ s'$
⟨*proof*⟩

lemma *weak-Id-on-proj-cong*:
assumes $i = i'$
assumes $a = a'$
assumes $(s,\ s') \in heap.Id_{\{a'\}}$
shows $Array.get\ a\ i\ s = Array.get\ a'\ i'\ s'$
⟨*proof*⟩

lemma *ext*:
assumes *heap.rep-inv a s*
assumes *heap.rep-inv a s'*
assumes $\forall i{\in}set\ (Ix\text{-}class.interval\ (array.bounds\ a)).\ Array.get\ a\ i\ s = Array.get\ a\ i\ s'$
shows $s\ (heap.addr\text{-}of\ a) = s'\ (heap.addr\text{-}of\ a)$
⟨*proof*⟩

⟨*ML*⟩

lemma *cong-deref*:
  assumes $a = a'$
  assumes $i = i'$
  assumes $v = v'$
  assumes $s\ r = s'\ r'$
  assumes $r = r'$
  shows $Array.set\ a\ i\ v\ s\ r = Array.set\ a'\ i'\ v'\ s'\ r'$
⟨*proof*⟩

lemma *same*:
  shows $Array.set\ a\ i\ v\ (Array.set\ a\ i\ v'\ s) = Array.set\ a\ i\ v\ s$
⟨*proof*⟩

⟨*ML*⟩

lemma *ex-bij-betw*:
  fixes $a :: ('i{::}Ix,\ 'a{::}heap.rep)\ array$
  fixes $b :: 'j{::}Ix \times 'j$
  assumes $Array.coerce\ a\ b = Some\ a'$
  obtains f **where** $map\ f\ (Array.interval\ a) = Ix.interval\ b$
⟨*proof*⟩

lemma *ex-bij-betw2*:
  fixes $a :: ('i{::}Ix,\ 'a{::}heap.rep)\ array$
  fixes $b :: 'j{::}Ix \times 'j$
  assumes $Array.coerce\ a\ b = Some\ a'$
  obtains f **where** $map\ f\ (Ix.interval\ b) = Array.interval\ a$
⟨*proof*⟩

⟨*ML*⟩

lemma *set*:
  assumes *heap.rep-inv a s*
  shows *heap.rep-inv a (Array.set a i v s)*
⟨*proof*⟩

⟨*ML*⟩

lemma *heap-modifies-le*:
  shows $Array.modifies_{a,\ is} \subseteq heap.modifies_{\{a\}}$
⟨*proof*⟩

lemma *heap-rep-inv-rel-le*:
  shows $Array.modifies_{a,\ is} \subseteq heap.rep\text{-}inv\text{-}rel\ a$
⟨*proof*⟩

lemma *empty*:
  shows $Array.modifies_{a,\ \{\}} = Id \cap heap.rep\text{-}inv\text{-}rel\ a$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *mono*:
  assumes $is \subseteq js$
  shows $Array.modifies_{a,\ is} \subseteq Array.modifies_{a,\ js}$
⟨*proof*⟩

lemma *INTER*:
shows $Array.modifies_{a,\ \bigcap x\in X.\ f\ x} = (\bigcap x\in X.\ Array.modifies_{a,\ f\ x}) \cap heap.modifies_{\{a\}}$
⟨*proof*⟩

lemma *Inter*:
shows $Array.modifies_{a,\ \bigcap X} = (\bigcap x\in X.\ Array.modifies_{a,\ x}) \cap heap.modifies_{\{a\}}$
⟨*proof*⟩

lemma *inter*:
shows $Array.modifies_{a,\ is} \cap Array.modifies_{a,\ js} = Array.modifies_{a,\ is \cap js}$
⟨*proof*⟩

lemma *UNION-subseteq*:
shows $(\bigcup x\in X.\ Array.modifies_{a,\ I\ x}) \subseteq Array.modifies_{a,\ (\bigcup x\in X.\ I\ x)}$
⟨*proof*⟩

lemma *union-subseteq*:
shows $Array.modifies_{a,\ is} \cup Array.modifies_{a,\ js} \subseteq Array.modifies_{a,\ is \cup js}$
⟨*proof*⟩

lemma *Diag-subseteq*:
assumes $\bigwedge s.\ P\ s \Longrightarrow heap.rep\text{-}inv\ a\ s$
shows $Diag\ P \subseteq Array.modifies_{a,\ is}$
⟨*proof*⟩

lemma *get*:
assumes $(s,\ s') \in Array.modifies_{a,\ is}$
assumes $i \in set\ (Array.interval\ a) - is$
shows $Array.get\ a\ i\ s' = Array.get\ a\ i\ s$
⟨*proof*⟩

lemma *set*:
assumes $heap.rep\text{-}inv\ a\ s$
shows $(s,\ Array.set\ a\ i\ v\ s) \in heap.modifies_{\{a\}}$
⟨*proof*⟩

lemma *Array-set*:
assumes $heap.rep\text{-}inv\ a\ s$
assumes $i \in set\ (Array.interval\ a) \cap is$
shows $(s,\ Array.set\ a\ i\ v\ s) \in Array.modifies_{a,\ is}$
⟨*proof*⟩

lemma *Array-set-conv*:
assumes $i \in set\ (Array.interval\ a) \cap is$
shows $(s,\ Array.set\ a\ i\ v\ s) \in Array.modifies_{a,\ is} \longleftrightarrow heap.rep\text{-}inv\ a\ s$ (**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

⟨*ML*⟩

lemmas *simps'* =
Array.rep-inv.set
Array.get.set-eq

⟨*ML*⟩

lemma *Id-on-le*:
shows $heap.Id_{\{a\}} \subseteq Array.Id_{a,\ is}$
⟨*proof*⟩

⟨ML⟩

lemma *empty*:
  shows $Array.Id_{a,\ \{\}} = heap.rep\text{-}inv\text{-}rel\ a$
⟨*proof*⟩

lemma *mono*:
  assumes $is \subseteq js$
  shows $Array.Id_{a,\ js} \subseteq Array.Id_{a,\ is}$
⟨*proof*⟩

lemma *insert*:
  shows $Array.Id_{a,\ insert\ i\ is} = Array.Id_{a,\ \{i\}} \cap Array.Id_{a,\ is}$
⟨*proof*⟩

lemma *union*:
  shows $Array.Id_{a,\ is \cup js} = Array.Id_{a,\ is} \cap Array.Id_{a,\ js}$
⟨*proof*⟩

lemma *rep-inv-rel*:
  shows $Array.Id_{a,\ is} \subseteq heap.rep\text{-}inv\text{-}rel\ a$
⟨*proof*⟩

lemma *eq-heap-Id-on*:
  assumes $set\ (Array.interval\ a) \subseteq is$
  shows $Array.Id_{a,\ is} = heap.Id_{\{a\}}$
⟨*proof*⟩

⟨ML⟩

25.2.3 Stability

⟨ML⟩

lemma *get*[*stable.intro*]:
  assumes $a \in as$
  shows $stable\ heap.Id_{as}\ (\lambda s.\ P\ (Array.get\ a\ i\ s))$
⟨*proof*⟩

lemma *get-chain*: — difficult to apply
  assumes $\bigwedge v.\ stable\ heap.Id_{as}\ (P\ v)$
  assumes $a \in as$
  shows $stable\ heap.Id_{as}\ (\lambda s.\ P\ (Array.get\ a\ i\ s)\ s)$
⟨*proof*⟩

⟨ML⟩

lemma *get*[*stable.intro*]:
  assumes $i \in is$
  shows $stable\ Array.Id_{a,\ is}\ (\lambda s.\ P\ (Array.get\ a\ i\ s))$
⟨*proof*⟩

lemma *get-chain*: — difficult to apply
  assumes $\bigwedge v.\ stable\ Array.Id_{a,\ is}\ (P\ v)$
  assumes $i \in is$
  shows $stable\ Array.Id_{a,\ is}\ (\lambda s.\ P\ (Array.get\ a\ i\ s)\ s)$
⟨*proof*⟩

⟨*ML*⟩

lemma *rep-inv*[*stable.intro*]:
  shows *stable* $Array.Id_{a,\ is}$ (*heap.rep-inv a*)
⟨*proof*⟩

⟨*ML*⟩

lemma *rep-inv*[*stable.intro*]:
  shows *stable* $Array.modifies_{a,\ is}$ (*heap.rep-inv a*)
⟨*proof*⟩

⟨*ML*⟩

lemma *get*[*stable.intro*]:
  assumes $i \in set\ (Array.interval\ a) - is$
  shows *stable* $Array.modifies_{a,\ is}$ $(\lambda s.\ P\ (Array.get\ a\ i\ s))$
⟨*proof*⟩

lemma *get-chain*: — difficult to apply
  assumes $\bigwedge v.\ stable\ Array.modifies_{a,\ is}\ (P\ v)$
  assumes $i \in set\ (Array.interval\ a) - is$
  shows *stable* $Array.modifies_{a,\ is}$ $(\lambda s.\ P\ (Array.get\ a\ i\ s)\ s)$
⟨*proof*⟩

⟨*ML*⟩

26 A concurrent variant of Imperative HOL

We model programs operating on sequentially-consistent memory with the type (*heap.t*, *'v*) *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 imp* = (*heap.t*, *'v*) *prog*

⟨*ML*⟩

definition *raise* :: *String.literal* ⇒ *'a imp* **where** — the literal is just decoration
  raise s = ⊥

definition *assert* :: *bool* ⇒ *unit imp* **where**
  assert P = (*if P then prog.return* () *else prog.raise STR ''assert''*)

⟨*ML*⟩

definition *ref* :: *'a::heap.rep* ⇒ *'a ref imp* **where**
  ref v = *prog.action* {(*r*, *s*, *s'*). (*r*, *s'*) ∈ *Ref.alloc v s*}

definition *lookup* :: *'a::heap.rep ref* ⇒ *'a imp* (‹!-› *61*) **where**
  lookup r = *prog.read* (*Ref.get r*)

definition *update* :: *'a ref* ⇒ *'a::heap.rep* ⇒ *unit imp* (‹- := -› *62*) **where**
update r v = *prog.write* (*Ref.set r v*)

⟨*ML*⟩

definition *new* :: (*'i* × *'i*) ⇒ *'a* ⇒ (*'i::Ix*, *'a::heap.rep*) *array imp* **where**
new b v = *prog.action* {(*Array b a*, *s*, *s'*) |*a s s'*. (*a*, *s'*) ∈ *ODArray.alloc* (*replicate* (*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'*) ∈ *ODArray.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'*) ∈ *ODArray.alloc xs s*}

definition *nth* :: (*'i::Ix*, *'a::heap.rep*) *array* ⇒ *'i* ⇒ *'a imp* **where**
nth a i = *prog.read* (λ*s*. *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*
⟨*proof*⟩

⟨*ML*⟩

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*]

⟨*ML*⟩

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* ⇀ (*Haskell*) *Heap.Prog* - -
code-monad *prog.bind Haskell*
code-printing constant *prog.return* ⇀ (*Haskell*) *return*
code-printing constant *prog.raise* ⇀ (*Haskell*) *error*
code-printing constant *prog.parallel* ⇀ (*Haskell*) *Heap.parallel*

Intermediate operation avoids invariance problem in *Scala* (similar to value restriction)

⟨*ML*⟩

definition *ref'* **where**
  [*code del*]: *ref'* = *prog.Ref.ref*

lemma [*code*]:
  prog.Ref.ref x = *Ref.ref' x*
⟨*proof*⟩

⟨*ML*⟩

code-printing type-constructor *ref* ⇀ (*Haskell*) *Heap.Ref* -
code-printing constant *Ref* ⇀ (*Haskell*) *error/ bare Ref*
code-printing constant *Ref.ref'* ⇀ (*Haskell*) *Heap.newIORef*
code-printing constant *prog.Ref.lookup* ⇀ (*Haskell*) *Heap.readIORef*
code-printing constant *prog.Ref.update* ⇀ (*Haskell*) *Heap.writeIORef*
code-printing constant *HOL.equal* :: *'a ref* ⇒ *'a ref* ⇒ *bool* ⇀ (*Haskell*) **infix** *4* ==
code-printing class-instance *ref* :: *HOL.equal* ⇀ (*Haskell*) −

The target language only has to provide one-dimensional arrays indexed by *integer*.

⟨*ML*⟩

definition *new'* :: *integer* ⇒ *'a* ⇒ *'a::heap.rep one-dim-array imp* **where**
  new' k v = *prog.action* {(*a*, *s*, *s'*) |*a s s'*. (*a*, *s'*) ∈ *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* ⤜ *prog.return* ∘ *Array b*

⟨proof⟩

definition *make′* :: *integer* ⇒ (*integer* ⇒ ′*a*) ⇒ ′*a*::*heap.rep one-dim-array imp* **where**
make′ k f = *prog.action* {(*a*, *s*, *s′*) |*a s s′*. (*a*, *s′*) ∈ *ODArray.alloc* (*map* (*f* ∘ *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*))) (λ*i*. *f* (*Ix.interval b* ! *nat-of-integer i*))
≫= *prog.return* ∘ *Array b*
⟨proof⟩

definition *of-list′* :: *integer* ⇒ ′*a list* ⇒ ′*a*::*heap.rep one-dim-array imp* **where**
of-list′ k xs = *prog.action* {(*a*, *s*, *s′*) |*a s s′*. *nat-of-integer k* ≤ *length xs* ∧ (*a*, *s′*) ∈ *ODArray.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* ≫= *prog.return* ∘ *Array b*
⟨proof⟩

definition *nth′* :: ′*a*::*heap.rep one-dim-array* ⇒ *integer* ⇒ ′*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*))
⟨proof⟩

definition *upd′* :: ′*a*::*heap.rep one-dim-array* ⇒ *integer* ⇒ ′*a*::*heap.rep* ⇒ *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*
⟨proof⟩

⟨ML⟩

code-printing type-constructor *one-dim-array* ⇀ (*Haskell*) *Heap.Array/ -*
code-printing constant *one-dim-array.Array* ⇀ (*Haskell*) *error/ bare Array*
code-printing constant *prog.Array.new′* ⇀ (*Haskell*) *Heap.newArray*
code-printing constant *prog.Array.make′* ⇀ (*Haskell*) *Heap.newFunArray*
code-printing constant *prog.Array.of-list′* ⇀ (*Haskell*) *Heap.newListArray*
code-printing constant *prog.Array.nth′* ⇀ (*Haskell*) *Heap.readArray*
code-printing constant *prog.Array.upd′* ⇀ (*Haskell*) *Heap.writeArray*
code-printing constant *HOL.equal* :: (′*i*, ′*a*) *array* ⇒ (′*i*, ′*a*) *array* ⇒ *bool* ⇀ (*Haskell*) **infix** *4* ==
code-printing class-instance *array* :: *HOL.equal* ⇀ (*Haskell*) −

26.2 Value-returning parallel

definition *parallelP′* :: ′*a*::*heap.rep imp* ⇒ ′*b*::*heap.rep imp* ⇒ (′*a* × ′*b*) *imp* **where**
parallelP′ P_1 P_2 = *do* {
r_1 ← *prog.Ref.ref undefined*

; $r_2 \leftarrow$ *prog.Ref.ref undefined*
; $((P_1 \gg= prog.Ref.update\ r_1) \parallel (P_2 \gg= prog.Ref.update\ r_2))$
; $v_1 \leftarrow prog.Ref.lookup\ r_1$
; $v_2 \leftarrow prog.Ref.lookup\ r_2$
; *prog.return* (v_1, v_2)
}

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*)
⟨*proof*⟩

lemma *apply-writes-present*:
  assumes *heap.present r s*
  shows *heap.present r* (*apply-writes wb s*)
⟨*proof*⟩

⟨*ML*⟩

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*)

adhoc-overloading
  Monad-Syntax.bind ⇌ *raw.bind*

definition *prim-return* :: *'a* ⇒ *'a raw.tso* **where**

$prim\text{-}return\ v = (\lambda wb.\ prog.return\ (v,\ wb))$

⟨*ML*⟩

lemma *mono*:
assumes $f \le f'$
assumes $\bigwedge x.\ g\ x \le g'\ x$
shows $raw.bind\ f\ g \le raw.bind\ f'\ g'$
⟨*proof*⟩

lemma *strengthen*[*strg*]:
assumes $st\text{-}ord\ F\ f\ f'$
assumes $\bigwedge x.\ st\text{-}ord\ F\ (g\ x)\ (g'\ x)$
shows $st\text{-}ord\ F\ (raw.bind\ f\ g)\ (raw.bind\ f'\ g')$
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes $monotone\ orda\ (\le)\ F$
assumes $\bigwedge x.\ monotone\ orda\ (\le)\ (\lambda y.\ G\ y\ x)$
shows $monotone\ orda\ (\le)\ (\lambda f.\ raw.bind\ (F\ f)\ (G\ f))$
⟨*proof*⟩

lemma *botL*:
shows $raw.bind\ \bot\ g = \bot$
⟨*proof*⟩

lemma *bind*:
fixes f :: - *raw.tso*
shows $f \ggg g \ggg h = f \ggg (\lambda x.\ g\ x \ggg h)$
⟨*proof*⟩

lemma *prim-return*:
shows *prim-returnL*: $raw.bind\ (raw.prim\text{-}return\ v) = (\lambda g.\ g\ v)$
and *prim-returnR*: $f \ggg raw.prim\text{-}return = f$
⟨*proof*⟩

lemma *supL*:
fixes g :: - ⇒ - *raw.tso*
shows $f_1 \sqcup f_2 \ggg g = (f_1 \ggg g) \sqcup (f_2 \ggg g)$
⟨*proof*⟩

lemma *supR*:
fixes f :: - *raw.tso*
shows $f \ggg (\lambda v.\ g_1\ v \sqcup g_2\ v) = (f \ggg g_1) \sqcup (f \ggg g_2)$
⟨*proof*⟩

lemma *SUPL*:
fixes X :: - *set*
fixes f :: - ⇒ - *raw.tso*
shows $(\bigsqcup x{\in}X.\ f\ x) \ggg g = (\bigsqcup x{\in}X.\ f\ x \ggg g)$
⟨*proof*⟩

lemma *SUPR*:
fixes X :: - *set*
fixes f :: - *raw.tso*
shows $f \ggg (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \ggg g\ x) \sqcup (f \ggg \bot)$
⟨*proof*⟩

lemma *SUPR-not-empty*:
  fixes f :: - *raw.tso*
  assumes $X \neq \{\}$
  shows $f \gg= (\lambda v. \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg= g\ x)$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *f*
  assumes $\bigwedge v.$ *mcont luba orda Sup* (≤) ($\lambda x.$ *g x v*)
  shows *mcont luba orda Sup* (≤) ($\lambda x.$ *raw.bind* (*f x*) (*g x*))
⟨*proof*⟩

⟨*ML*⟩

interpretation *kleene*: *kleene raw.prim-return* () $\lambda x\ y.$ *raw.bind x* ⟨*y*⟩
⟨*proof*⟩

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*

⟨*ML*⟩

definition *cl* :: *'v raw.tso* ⇒ *'v raw.tso* **where**
  cl P = *raw.commit-writes* ≫ *P* ≫= ($\lambda v.$ *raw.commit-writes* ≫ *raw.prim-return v*)

⟨*ML*⟩

definition *action* :: (*write-buffer* ⇒ (*'v* × *write-buffer* × *heap.t* × *heap.t*) *set*) ⇒ *'v raw.tso* **where**
  action F = *raw.tso.cl* ($\lambda wb.$ *prog.action* {((*v*, *wb* @ *ws*), *ss'*) |*v ss' ws*. (*v*, *ws*, *ss'*) ∈ *F wb*})

definition *return* :: *'v* ⇒ *'v raw.tso* **where**
  return v = *raw.action* ⟨{*v*} × {[]} × *Id*⟩

definition *guard* :: (*write-buffer* ⇒ *heap.t pred*) ⇒ *unit raw.tso* **where**
  guard g = *raw.action* ($\lambda wb.$ {()} × {[]} × *Diag* (*g wb*))

definition *MFENCE* :: *unit raw.tso* **where**
  MFENCE = *raw.guard* ($\lambda wb\ s.$ *wb* = [])

definition *vmap* :: (*'v* ⇒ *'w*) ⇒ *'v raw.tso* ⇒ *'w raw.tso* **where**
  vmap vf P = ($\lambda wb.$ *prog.vmap* (*map-prod vf id*) (*P wb*))

— Parallel composition
definition *t2p* :: *'v raw.tso* ⇒ (*heap.t*, *'v*) *prog* **where**
  t2p P = *P* [] ≫= (λ(*v*, *wb*). *raw.MFENCE wb* ≫ *prog.return v*)

— Jagadeesan et al. (2012, p184 rule PAR-CMD): perform MFENCE before fork
definition *parallel* :: *unit raw.tso* ⇒ *unit raw.tso* ⇒ *unit raw.tso* **where**
  parallel P Q = *raw.MFENCE* ≫ ⟨(*raw.t2p P* ∥ *raw.t2p Q*) ≫ *prog.return* ((), [])⟩

lemma *return-alt-def*:
  shows *raw.return* = ($\lambda v.$ *raw.tso.cl* (*raw.prim-return v*))
⟨*proof*⟩

⟨ML⟩

lemma *return-le*:
  shows *raw.prim-return* () ≤ *raw.commit-writes*
⟨*proof*⟩

lemma *return-le′*:
  shows *prog.return* ((), *wb*) ≤ *raw.commit-writes wb*
⟨*proof*⟩

lemma *commit-writes*:
  shows *raw.commit-writes* ≫ *raw.commit-writes* = *raw.commit-writes*
⟨*proof*⟩

lemma *Nil*:
  shows *raw.commit-writes* [] = *prog.return* ((), []) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Cons*:
  shows *raw.commit-writes* (*w* # *wb*)
    = (*raw.commit-write* [*w*] ≫ *raw.commit-writes wb*) ⊔ *raw.prim-return* () (*w* # *wb*)
⟨*proof*⟩

lemma *Cons-le*:
  shows *raw.commit-write* [*w*] ≫ *raw.commit-writes wb* ≤ *raw.commit-writes* (*w* # *wb*)
⟨*proof*⟩

⟨ML⟩

lemma *prim-return-Nil-le*:
  shows ⦇*s*, [], *Some* ((), *wb*)⦈ ≤ *prog.p2s* (*raw.prim-return* () *wb*)
⟨*proof*⟩

⟨ML⟩

lemma *noop-le*:
  shows ⦇*s*, [], *Some* ((), *wb*)⦈ ≤ *prog.p2s* (*raw.commit-writes wb*)
⟨*proof*⟩

lemma *wb-suffix*:
  assumes ⦇*s*, *xs*, *Some* ((), *wb′*)⦈ ≤ *prog.p2s* (*raw.commit-writes wb*)
  shows *suffix wb′ wb*
⟨*proof*⟩

⟨ML⟩

lemma *bind-commit-writes-absorbL*:
  fixes *P* :: ′*v raw.tso*
  shows *raw.commit-writes* ≫ *raw.tso.cl P* = *raw.tso.cl P*
⟨*proof*⟩

lemma *bind-commit-writes-absorb-unitR*:
  fixes *P* :: *unit raw.tso*
  shows *raw.tso.cl P* ≫ *raw.commit-writes* = *raw.tso.cl P*
⟨*proof*⟩

lemma *bind-commit-writes-absorbR*:
  fixes *P* :: ′*v raw.tso*

shows *raw.tso.cl P* ⤜ (λ*v*. *raw.commit-writes* ≫ *raw.prim-return v*) = *raw.tso.cl P*
⟨*proof*⟩

lemma *bot*:
shows *raw.tso.cl* ⊥ = *raw.commit-writes* ⤜ ⊥
⟨*proof*⟩

lemma *prim-return*:
shows *raw.tso.cl* (*raw.prim-return v*) = *raw.commit-writes* ≫ *raw.prim-return v*
⟨*proof*⟩

lemma *Nil*:
shows *raw.tso.cl P* [] = *P* [] ⤜ (λ*v*. *raw.commit-writes* (*snd v*) ⤜ (λ*w*. *prog.return* (*fst v*, *snd w*)))
⟨*proof*⟩

lemma *commit*:
fixes *wb* :: *write-buffer*
shows *raw.commit-write* [*w*] ≫ *f wb* ≤ *raw.tso.cl f* (*w* # *wb*)
⟨*proof*⟩

⟨*ML*⟩

interpretation *tso*: *closure-complete-distrib-lattice-distributive-class raw.tso.cl*
⟨*proof*⟩

⟨*ML*⟩

lemma *bind*:
fixes *f* :: ′*v raw.tso*
assumes *f* ∈ *raw.tso.closed*
shows *raw.tso.cl* (*f* ⤜ *g*) = *f* ⤜ (λ*v*. *raw.tso.cl* (*g v*))
⟨*proof*⟩

⟨*ML*⟩

lemma *commit-writes-absorbL*:
assumes *f* ∈ *raw.tso.closed*
shows *raw.commit-writes* ≫ *f* = *f*
⟨*proof*⟩

lemma *commit-writes-absorb-unitR*:
assumes *f* ∈ *raw.tso.closed*
shows *f* ≫ *raw.commit-writes* = *f*
⟨*proof*⟩

lemma *returnL*:
assumes *g v* ∈ *raw.tso.closed*
shows *raw.return v* ⤜ *g* = *g v*
⟨*proof*⟩

lemma *returnR*:
assumes *f* ∈ *raw.tso.closed*
shows *f* ⤜ *raw.return* = *f*
⟨*proof*⟩

⟨*ML*⟩

lemma *commit-writes*:

shows *raw.commit-writes* ∈ *raw.tso.closed*
⟨*proof*⟩

lemma *bind*[*intro*]:
fixes *f* :: *'v raw.tso*
fixes *g* :: *'v* ⇒ *'w raw.tso*
assumes *f* ∈ *raw.tso.closed*
assumes ⋀*x*. *g x* ∈ *raw.tso.closed*
shows *f* ≫= *g* ∈ *raw.tso.closed*
⟨*proof*⟩

lemma *action*[*intro*]:
shows *raw.action F* ∈ *raw.tso.closed*
⟨*proof*⟩

lemma *guard*[*intro*]:
shows *raw.guard g* ∈ *raw.tso.closed*
⟨*proof*⟩

lemma *MFENCE*[*intro*]:
shows *raw.MFENCE* ∈ *raw.tso.closed*
⟨*proof*⟩

lemma *parallel*[*intro*]:
assumes *P* ∈ *raw.tso.closed*
assumes *Q* ∈ *raw.tso.closed*
shows *raw.parallel P Q* ∈ *raw.tso.closed*
⟨*proof*⟩

lemma *vmap*[*intro*]:
assumes *P* ∈ *raw.tso.closed*
shows *raw.vmap vf P* ∈ *raw.tso.closed*
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
shows *raw.action* ⊥ = *raw.tso.cl* ⊥
⟨*proof*⟩

lemma *monotone*:
shows *mono raw.action*
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *st-monotone*[*OF raw.action.monotone*]
lemmas *mono* = *monotoneD*[*OF raw.action.monotone*]

lemma *Sup*:
shows *raw.action* (⨆ *Fs*) = ⨆(*raw.action* ‘ *Fs*) ⊔ *raw.tso.cl* ⊥ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *sup*:
shows *raw.action* (*F* ⊔ *G*) = *raw.action F* ⊔ *raw.action G*
⟨*proof*⟩

⟨*ML*⟩

lemma *return-le*:

shows *raw.guard g* ≤ *raw.return* ()
⟨*proof*⟩

lemma *monotone*:
shows *mono* (*raw.guard* :: (*write-buffer* ⇒ *heap.t pred*) ⇒ -)
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *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*⟩

⟨*ML*⟩

lemma *MFENCE-alt-def*:
shows *raw.MFENCE* = *raw.commit-writes* ≫ (λ*wb. prog.action* ({((), *wb*)} × *Diag* ⟨*wb* = []⟩))
⟨*proof*⟩

⟨*ML*⟩

lemma *Nil*:
shows *raw.MFENCE* [] = *prog.return* ((), [])
⟨*proof*⟩

⟨*ML*⟩

lemma *MFENCE*:
shows *prog.p2s* (*raw.MFENCE wb*) ≤ ⦃*P*⦄, *A* ⊩ *prog.p2s* (*raw.MFENCE wb*), ⦃λ*v s. snd v* = []⦄
⟨*proof*⟩

⟨*ML*⟩

lemma *MFENCEL*:
shows *raw.MFENCE wb* ⋙= *g* = *raw.MFENCE wb* ≫ *g* ((), []) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *MFENCE-return*:
shows *raw.MFENCE wb* ≫ *prog.return* ((), []) = *raw.MFENCE wb*
⟨*proof*⟩

lemma *MFENCE-MFENCE*:
shows *raw.MFENCE* ≫ *raw.MFENCE* = *raw.MFENCE*
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
shows *raw.t2p* ⊥ = ⊥
⟨*proof*⟩

lemma *cl-bot*:
shows *raw.t2p* (*raw.tso.cl* ⊥) = ⊥
⟨*proof*⟩

lemma *monotone*:

shows *mono raw.t2p*
⟨*proof*⟩

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 $raw.t2p\ (\bigsqcup X) = \bigsqcup(raw.t2p\ `\ X)$
⟨*proof*⟩

lemma *sup*:
shows $raw.t2p\ (P \sqcup Q) = raw.t2p\ P \sqcup raw.t2p\ Q$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
fixes *P* :: - ⇒ - *raw.tso*
assumes *mcont luba orda Sup* (≤) *F*
shows *mcont luba orda Sup* (≤) (λ*x*. *raw.t2p* (*F x*))
⟨*proof*⟩

lemma *return*:
shows *raw.t2p* (*raw.return v*) = *prog.return v*
⟨*proof*⟩

lemma *MFENCE-bind*:
shows *raw.t2p* (*raw.MFENCE* ⋙= *P*) = *raw.t2p* (*P* ())
⟨*proof*⟩

lemma *bind-return-unit*:
shows *raw.t2p* (λ*wb*. *prog.bind P* (λ-::*unit*. *prog.return* ((), []))) = *P*
⟨*proof*⟩

⟨*ML*⟩

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))
shows *raw.parallel P Q* = *raw.parallel Q P*
⟨*proof*⟩

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))
shows *raw.parallel P* (*raw.parallel Q R*) = *raw.parallel* (*raw.parallel P Q*) *R*
⟨*proof*⟩

lemma *mono*:
assumes $P \le P'$
assumes $Q \le Q'$
shows *raw.parallel P Q* ≤ *raw.parallel P′ Q′*
⟨*proof*⟩

lemma *botL*:
shows *raw.parallel* (*raw.tso.cl* ⊥) *f* = *raw.MFENCE* ≫ *f* ≫ *raw.MFENCE* ≫ *raw.tso.cl* ⊥
⟨*proof*⟩

lemma *returnL*:
shows *raw.parallel* (*raw.return* ()) *P* = *raw.MFENCE* ⋙= (λ-. *P*) ⋙= (λ-. *raw.MFENCE*)
⟨*proof*⟩

lemma *SupL-not-empty*:

assumes $\forall x \in X.\ x \in raw.tso.closed$
assumes $Q \in raw.tso.closed$
assumes $X \neq \{\}$
shows $raw.parallel\ (\bigsqcup X \sqcup raw.tso.cl\ \bot)\ Q = (\bigsqcup P \in X.\ raw.parallel\ P\ Q) \sqcup raw.tso.cl\ \bot$
⟨*proof*⟩

⟨*ML*⟩

typedef *'v tso* = *raw.tso.closed* :: *'v raw.tso set*
morphisms *t2p' Abs-tso*
⟨*proof*⟩

setup-lifting *type-definition-tso*

instantiation *tso* :: (*type*) *complete-distrib-lattice*
begin

lift-definition *bot-tso* :: *'v tso* **is** *raw.tso.cl* ⊥ ⟨*proof*⟩
lift-definition *top-tso* :: *'v tso* **is** ⊤ ⟨*proof*⟩
lift-definition *sup-tso* :: *'v tso* ⇒ *'v tso* ⇒ *'v tso* **is** *sup* ⟨*proof*⟩
lift-definition *inf-tso* :: *'v tso* ⇒ *'v tso* ⇒ *'v tso* **is** *inf* ⟨*proof*⟩
lift-definition *less-eq-tso* :: *'v tso* ⇒ *'v tso* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩
lift-definition *less-tso* :: *'v tso* ⇒ *'v tso* ⇒ *bool* **is** *less* ⟨*proof*⟩
lift-definition *Inf-tso* :: *'v tso set* ⇒ *'v tso* **is** *Inf* ⟨*proof*⟩
lift-definition *Sup-tso* :: *'v tso set* ⇒ *'v tso* **is** $\lambda X.\ Sup\ X \sqcup raw.tso.cl\ \bot$ ⟨*proof*⟩

instance ⟨*proof*⟩

end

⟨*ML*⟩

lift-definition *bind* :: *'v tso* ⇒ (*'v* ⇒ *'w tso*) ⇒ *'w tso* **is** *raw.bind* ⟨*proof*⟩
lift-definition *action* :: (*write-buffer* ⇒ (*'v* × *write-buffer* × *heap.t* × *heap.t*) *set*) ⇒ *'v tso* **is** *raw.action* ⟨*proof*⟩
lift-definition *MFENCE* :: *unit tso* **is** *raw.MFENCE* ⟨*proof*⟩
lift-definition *parallel* :: *unit tso* ⇒ *unit tso* ⇒ *unit tso* **is** *raw.parallel* ⟨*proof*⟩
lift-definition *vmap* :: (*'v* ⇒ *'w*) ⇒ *'v tso* ⇒ *'w tso* **is** *raw.vmap* ⟨*proof*⟩

lift-definition *t2p* :: *'v tso* ⇒ (*heap.t*, *'v*) *prog* **is** *raw.t2p* ⟨*proof*⟩

adhoc-overloading
Monad-Syntax.bind ⇌ *tso.bind*
adhoc-overloading
parallel ⇌ *tso.parallel*

definition *return* :: *'v* ⇒ *'v tso* **where**
return v = *tso.action* ⟨{*v*} × {[]} × *Id*⟩

definition *guard* :: (*write-buffer* ⇒ *heap.t pred*) ⇒ *unit tso* **where**
guard g = *tso.action* (λ*wb*. {()} × {[]} × *Diag* (*g wb*))

abbreviation (*input*) *read* :: (*heap.t* ⇒ *'v*) ⇒ *'v tso* **where**
read f ≡ *tso.action* (λ*wb*. {(*f* (*apply-writes wb s*), [], *s*, *s*) |*s*. *True*})

abbreviation (*input*) *write* :: (*heap.t* ⇒ *heap.write*) ⇒ *unit tso* **where**
write f ≡ *tso.action* ⟨{((), [*f s*], *s*, *s*) |*s*. *True*}⟩

lemma *return-alt-def*:

shows *tso.return v* = *tso.read* ⟨*v*⟩
⟨*proof*⟩

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*]

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (=) *cr-tso raw.return tso.return*
⟨*proof*⟩

⟨*ML*⟩

lemma *empty*:
shows *bot*: *tso.action* ⊥ = ⊥
and *tso.action* (λ-. {}) = ⊥
⟨*proof*⟩

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* (⨆ *Fs*) = ⨆ (*tso.action* ‘ *Fs*)
⟨*proof*⟩

lemmas *sup* = *tso.action.Sup*[**where** *Fs*={*F*, *G*} **for** *F G*, *simplified*]

⟨*ML*⟩

lemmas *if-distrL* = *if-distrib*[**where** *f*=λ*f*. *tso.bind f 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 ⋀*x*. *st-ord F* (*g x*) (*g′ x*)
shows *st-ord F* (*tso.bind f g*) (*tso.bind f′ g′*)
⟨*proof*⟩

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *raw.bind.mono2mono*[*transferred*]

lemma *bind*: — Jagadeesan et al. (2012, §5 (2))
shows *f* ⤜ *g* ⤜ *h* = *tso.bind f* (λ*x*. *g x* ⤜ *h*)
⟨*proof*⟩

lemma *return*: — Jagadeesan et al. (2012, §5 (1))
shows *returnL*: *tso.return v* ⤜ *g* = *g v*
and *returnR*: *f* ⤜ *tso.return* = *f*
⟨*proof*⟩

lemma *botL*:
  shows *tso.bind* ⊥ *g* = ⊥
⟨*proof*⟩

lemma *botR-le*:
  shows *tso.bind f* ⟨⊥⟩ ≤ *f* (**is** *?thesis1*)
    and *tso.bind f* ⊥ ≤ *f* (**is** *?thesis2*)
⟨*proof*⟩

lemma
  fixes *f* :: - *tso*
  fixes f_1 :: - *tso*
  shows *supL*: $(f_1 \sqcup f_2) \ggg= g = (f_1 \ggg= g) \sqcup (f_2 \ggg= g)$
    and *supR*: $f \ggg= (\lambda x.\ g_1\ x \sqcup g_2\ x) = (f \ggg= g_1) \sqcup (f \ggg= g_2)$
⟨*proof*⟩

lemma *SUPL*:
  fixes *X* :: - *set*
  fixes *f* :: - ⇒ - *tso*
  shows $(\bigsqcup x{\in}X.\ f\ x) \ggg= g = (\bigsqcup x{\in}X.\ f\ x \ggg= g)$
⟨*proof*⟩

lemma *SUPR*:
  fixes *X* :: - *set*
  fixes *f* :: - *tso*
  shows $f \ggg= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \ggg= g\ x) \sqcup (f \ggg= \bot)$
⟨*proof*⟩

lemma *SupR*:
  fixes *X* :: - *set*
  fixes *f* :: - *tso*
  shows $f \gg (\bigsqcup X) = (\bigsqcup x{\in}X.\ f \gg x) \sqcup (f \ggg= \bot)$
⟨*proof*⟩

lemma *SUPR-not-empty*:
  fixes *f* :: - *tso*
  assumes $X \neq \{\}$
  shows $f \ggg= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \ggg= g\ x)$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *f*
  assumes ⋀*v*. *mcont luba orda Sup* (≤) (λ*x*. *g x v*)
  shows *mcont luba orda Sup* (≤) (λ*x*. *tso.bind* (*f x*) (*g x*))
⟨*proof*⟩

⟨*ML*⟩

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (=) *cr-tso raw.guard tso.guard*
⟨*proof*⟩

lemma *bot*:
  shows *tso.guard* ⊥ = ⊥
    and *tso.guard* (λ- - .*False*) = ⊥
⟨*proof*⟩

lemma *top*:

shows *tso.guard* ⊤ = *tso.return* () (**is** *?thesis1*)
  and *tso.guard* (λ-. ⊤) = *tso.return* () (**is** *?thesis2*)
  and *tso.guard* (λ- -. *True*) = *tso.return* () (**is** *?thesis3*)
⟨*proof*⟩

lemma *return-le*:
 shows *tso.guard g* ≤ *tso.return* ()
⟨*proof*⟩

lemma *monotone*:
 shows *mono tso.guard*
⟨*proof*⟩

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'
⟨*proof*⟩

⟨*ML*⟩

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))
 shows *tso.parallel P Q* = *tso.parallel Q P*
⟨*proof*⟩

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))
 shows *tso.parallel P* (*tso.parallel Q R*) = *tso.parallel* (*tso.parallel P Q*) *R*
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
 assumes *monotone orda* (≤) *F*
 assumes *monotone orda* (≤) *G*
 shows *monotone orda* (≤) (λ*f*. *tso.parallel* (*F f*) (*G f*))
⟨*proof*⟩

lemma *bot*:
 shows *parallel-botL*: *tso.parallel* ⊥ *f* = *tso.MFENCE* ≫ *f* ≫ *tso.MFENCE* ⋙ ⊥ (**is** *?thesis1*)
  and *parallel-botR*: *tso.parallel f* ⊥ = *tso.MFENCE* ≫ *f* ≫ *tso.MFENCE* ⋙ ⊥ (**is** *?thesis2*)
⟨*proof*⟩

lemma *return*: — Jagadeesan et al. (2012, unnumbered)
 shows *returnL*: *tso.return* () ∥ *P* = *tso.MFENCE* ≫ *P* ≫ *tso.MFENCE* (**is** *?thesis1*)
  and *returnR*: *P* ∥ *tso.return* () = *tso.MFENCE* ≫ *P* ≫ *tso.MFENCE* (**is** *?thesis2*)
⟨*proof*⟩

lemma *Sup-not-empty*:

fixes *X* :: *unit tso 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*⟩

lemma *sup*:
fixes *P* :: *unit tso*
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
⟨*proof*⟩

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *P*
assumes *mcont luba orda Sup* (≤) *Q*
shows *mcont luba orda Sup* (≤) (λ*x. tso.parallel* (*P x*) (*Q x*))
⟨*proof*⟩

⟨*ML*⟩

lemmas *MFENCE-MFENCE* = *raw.bind.MFENCE-MFENCE*[*transferred*]

⟨*ML*⟩

lemma *monotone*:
shows *mono* (λ*t. t2p′ t wb*)
⟨*proof*⟩

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*) = *raw.return v*
⟨*proof*⟩

⟨*ML*⟩

Combinators ⟨*ML*⟩

abbreviation *guardM* :: *bool* ⇒ *unit tso* **where**
guardM b ≡ *if b then* ⊥ *else tso.return* ()

abbreviation *unlessM* :: *bool* ⇒ *unit tso* ⇒ *unit tso* **where**
unlessM b c ≡ *if b then tso.return* () *else c*

abbreviation *whenM* :: *bool* ⇒ *unit tso* ⇒ *unit tso* **where**
whenM b c ≡ *if b then c else tso.return* ()

definition *app* :: (*′a* ⇒ *unit tso*) ⇒ *′a list* ⇒ *unit tso* **where** — Haskell's *mapM-*
app f xs = *foldr* (λ*x m. f x* ≫ *m*) *xs* (*tso.return* ())

primrec *fold-mapM* :: (*′a* ⇒ *′b tso*) ⇒ *′a list* ⇒ *′b list tso* **where**
fold-mapM f [] = *tso.return* []
| *fold-mapM f* (*x* # *xs*) = *do* {
y ← *f x*;
ys ← *fold-mapM f xs*;
tso.return (*y* # *ys*)

}

— Jagadeesan et al. (2012, §5 (6) is *tso.while.simps*)
partial-function (*lfp*) *while* :: ($'k \Rightarrow ('k + 'v)$ *tso*) $\Rightarrow$ $'k \Rightarrow 'v$ *tso* **where**
while c k = *c k* ⪢= (λ*rv*. *case rv of Inl k'* ⇒ *while c k'* | *Inr v* ⇒ *tso.return v*)

abbreviation (*input*) *while'* :: ((*unit* + $'v$) *tso*) ⇒ $'v$ *tso* **where**
while' c ≡ *tso.while* ⟨*c*⟩ ()

definition *raise* :: *String.literal* ⇒ $'v$ *tso* **where**
raise s = ⊥

definition *assert* :: *bool* ⇒ *unit tso* **where**
assert P = (*if P then tso.return* () *else tso.raise STR ''assert''*)

declare *tso.raise-def*[*code del*]

⟨*ML*⟩

lemma *bot*:
shows *tso.fold-mapM* ⊥ = (λ*xs*. *case xs of* [] ⇒ *tso.return* [] | - ⇒ ⊥)
⟨*proof*⟩

lemma *append*:
shows *tso.fold-mapM f* (*xs* @ *ys*) = *tso.fold-mapM f xs* ⪢= (λ*xs*. *tso.fold-mapM f ys* ⪢= (λ*ys*. *tso.return* (*xs* @ *ys*)))
⟨*proof*⟩

⟨*ML*⟩

lemma *bot*:
shows *tso.app* ⊥ = (λ*xs*. *case xs of* [] ⇒ *tso.return* () | - ⇒ ⊥)
and *tso.app* (λ-. ⊥) = (λ*xs*. *case xs of* [] ⇒ *tso.return* () | - ⇒ ⊥)
⟨*proof*⟩

lemma *Nil*:
shows *tso.app f* [] = *tso.return* ()
⟨*proof*⟩

lemma *Cons*:
shows *tso.app f* (*x* # *xs*) = *f x* ≫ *tso.app f xs*
⟨*proof*⟩

lemmas *simps* =
tso.app.bot
tso.app.Nil
tso.app.Cons

lemma *append*:
shows *tso.app f* (*xs* @ *ys*) = *tso.app f xs* ≫ *tso.app f ys*
⟨*proof*⟩

lemma *monotone*:
shows *mono* (λ*f*. *tso.app f xs*)
⟨*proof*⟩

lemmas *strengthen*[*strg*] = *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 $(\bigsqcup f{\in}X.\ tso.app\ f\ xs) \le tso.app\ (\bigsqcup X)\ xs$
⟨*proof*⟩

⟨*ML*⟩

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.

⟨*ML*⟩

definition *ref* :: *'a::heap.rep* ⇒ *'a ref tso* **where**
  ref v = *tso.action* (λ*wb*. {(*r*, [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)], *s*, *s'*)
    |*r s s' v'*. (*r*, *s'*) ∈ *Ref.alloc v' s*})

definition *lookup* :: *'a::heap.rep ref* ⇒ *'a tso* (‹!-› *61*) **where**
  lookup r = *tso.read* (*Ref.get r*)

definition *update* :: *'a ref* ⇒ *'a::heap.rep* ⇒ *unit tso* (‹- := -› *62*) **where**
  update r v = *tso.write* ⟨*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)⟩

declare *tso.Ref.ref-def*[*code del*]
declare *tso.Ref.lookup-def*[*code del*]
declare *tso.Ref.update-def*[*code del*]

⟨*ML*⟩

27.2 Inhabitation

In order to obtain compositional rules we need to make the write buffer explicit.

⟨*ML*⟩

definition *t2s* :: *write-buffer* ⇒ *'v tso* ⇒ (*sequential*, *heap.t*, *'v* × *write-buffer*) *spec* **where**
  t2s wb P = *prog.p2s* (*tso.t2p' P wb*)

⟨*ML*⟩

lemma *t2s-commit*:
  assumes ⦇*heap.apply-write w s*, *xs*, *v*⦈ ≤ *tso.t2s wb f*
  shows ⦇*s*, (*self*, *heap.apply-write w s*) # *xs*, *v*⦈ ≤ *tso.t2s* (*w* # *wb*) *f*
⟨*proof*⟩

⟨*ML*⟩

lemma *t2s-le*:
  shows *spec.idle* ≤ *tso.t2s wb P*
⟨*proof*⟩

⟨*ML*⟩

lemmas *minimal*[*iff*] = *order.trans*[*OF spec.idle.minimal-le spec.idle.tso.t2s-le*]

⟨*ML*⟩

lemma *t2s-le*:
  shows *spec.rel* ({*env*} × *UNIV*) ⋙ (λ-::*unit*. *tso.t2s wb P*) ≤ *tso.t2s wb P*
⟨*proof*⟩

⟨*ML*⟩

lemma *t2p*[*prog.p2s.simps*]:
  shows *prog.p2s* (*tso.t2p P*)
    = *tso.t2s* [] *P* ⋙ (λ*vwb*. *prog.p2s* (*raw.MFENCE* (*snd vwb*) ≫ *prog.return* (*fst vwb*)))
⟨*proof*⟩

⟨*ML*⟩

lemma *bind*:
  shows *tso.t2s wb* (*f* ⋙ *g*) = *tso.t2s wb f* ⋙ (λ*x*. *tso.t2s* (*snd x*) (*g* (*fst x*)))
⟨*proof*⟩

lemma *parallel*:
  shows *tso.t2s* [] (*P* ∥ *Q*) = *prog.p2s* ((*tso.t2p P* ∥ *tso.t2p Q*) ≫ *prog.return* ((), []))
⟨*proof*⟩

lemma *return*:
  shows *tso.t2s* [] (*tso.return v*) = *prog.p2s* (*prog.return* (*v*, []))
⟨*proof*⟩

⟨*ML*⟩

Inhabitation rules. ⟨*ML*⟩

lemma *bind*:
  assumes *tso.t2s wb f* −*s*, *xs*→ *tso.t2s wb′ f′*
  shows *tso.t2s wb* (*f* ⋙ *g*) −*s*, *xs*→ *tso.t2s wb′* (*f′* ⋙ *g*)
⟨*proof*⟩

lemma *commit*:
  shows *tso.t2s* (*w* # *wb*) *f* −*s*, [(*self*, *heap.apply-write w s*)]→ *tso.t2s wb f*
⟨*proof*⟩

⟨*ML*⟩

lemma *ref*:
  fixes *r* :: ′*a*::*heap.rep ref*
  fixes *s* :: *heap.t*
  fixes *v* :: ′*a*
  fixes *v′* :: ′*a*
  assumes ¬*heap.present r s*
  shows *tso.t2s wb* (*tso.Ref.ref v*)
      −*s*, [(*self*, *Ref.set r v′ s*)]→
        tso.t2s (*wb* @ [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)]) (*tso.return r*) (**is** *?lhs* −*s*, *?step*→ *?rhs*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

⟨*ML*⟩

lemmas *bind′* = *inhabits.trans*[*OF inhabits.tso.bind*]
lemmas *commit′* = *inhabits.trans*[*OF inhabits.tso.commit*]

⟨*ML*⟩

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-atomicmodifyioref-work

27.3.1 Haskell

Adaption layer

code-printing code-module *TSOHeap* ⇀ (*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)

⟨*ML*⟩

definition *ref′* **where**
  [*code del*]: *ref′* = *tso.Ref.ref*

lemma [*code*]:
  tso.Ref.ref x = *tso.Ref.ref′ x*
  ⟨*proof*⟩

⟨*ML*⟩

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 ⟪*heap.empty*, *?xs*, *Some* (*0*, *0*)⟫ ≤ *prog.p2s* (*tso.t2p iwp2-3-a*)
⟨*proof*⟩

thm *iwp2-3-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 = (λ*a* (*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 (λ*k*. ||(*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 (λ*k*. ||(*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*
⟨*proof*⟩

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* ⊆ *ks* ∧ *fst ij* ∉ *set p* ∧ *snd ij* ∉ *set p* ∧ *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-path-weight* :: *'i matrix* ⇒ *'i* × *'i* ⇒ *'i set* ⇒ *nat* **where**
min-path-weight m ij ks = *Min* (*path-weight m ij* ‘ *paths ij ks*)

context
fixes *a* :: (*'i*::*Ix* × *'i*, *nat*) *array*
fixes *m* :: *'i matrix*
begin

definition *fw-p-inv* :: *'i* × *'i* ⇒ *'i set* ⇒ *heap.t pred* **where** — process invariant
fw-p-inv ij ks = (*heap.rep-inv a* ∧ *Array.get a ij* = ⟨*min-path-weight m ij ks*⟩)

definition *fw-inv* :: *'i set* ⇒ *heap.t pred* **where** — loop invariant
fw-inv ks = (∀ *ij*. ⟨*ij*∈*set* (*Array.interval a*)⟩ ⟶ *fw-p-inv ij ks*)

definition *fw-pre* :: *heap.t pred* **where** — overall precondition
fw-pre = (⟨*Array.square a*⟩ ∧ *heap.rep-inv a*
∧ (∀ *ij*. ⟨*ij*∈*set* (*Array.interval a*)⟩ ⟶ *Array.get a ij* = ⟨*m ij*⟩))

definition *fw-post* :: *unit* ⇒ *heap.t pred* **where** — overall postcondition
fw-post - = *fw-inv* (*set* (*Ix.interval* (*fst-bounds* (*array.bounds a*))))

end

⟨*ML*⟩

lemma *I*:
assumes *set p* ⊆ *ks*
assumes *i* ∉ *set p*
assumes *j* ∉ *set p*
assumes *distinct p*
shows *p* ∈ *paths* (*i*, *j*) *ks*
⟨*proof*⟩

lemma *Nil*:
shows [] ∈ *paths ij ks*
⟨*proof*⟩

lemma *empty*:
shows *paths ij* {} = {[]}
⟨*proof*⟩

lemma *not-empty*:
shows *paths ij ks* ≠ {}
⟨*proof*⟩

lemma *monotone*:
shows *mono* (*paths ij*)
⟨*proof*⟩

lemmas *mono* = *monoD*[*OF paths.monotone*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF paths.monotone*]

lemma *finite*:
assumes *finite ks*
shows *finite* (*paths ij ks*)
⟨*proof*⟩

lemma *unused*:
assumes *p* ∈ *paths ij* (*insert k ks*)
assumes *k* ∉ *set p*
shows *p* ∈ *paths ij ks*
⟨*proof*⟩

lemma *decompE*:
assumes *p* ∈ *paths* (*i*, *j*) (*insert k ks*)
assumes *k* ∈ *set p*
obtains *r s*
where *p* = *r* @ *k* # *s*
and *r* ∈ *paths* (*i*, *k*) *ks* **and** *s* ∈ *paths* (*k*, *j*) *ks*
and *distinct* (*r* @ *s*) **and** *i* ∉ *set* (*r* @ *k* # *s*) **and** *j* ∉ *set* (*r* @ *k* # *s*)
⟨*proof*⟩

⟨*ML*⟩

lemma *append*:
  shows *path-weight m ij (xs @ y # ys) = path-weight m (fst ij, y) xs + path-weight m (y, snd ij) ys*
⟨*proof*⟩

⟨*ML*⟩

lemmas *min-path-weightI = trans[OF min-path-weight-def Min-eqI]*

⟨*ML*⟩

lemma *fw-update*:
  assumes *m*: *min-path-weight m (i, k) ks + min-path-weight m (k, j) ks < min-path-weight m (i, j) ks*
  assumes *finite ks*
  shows *min-path-weight m (i, j) (insert k ks)*
    = min-path-weight m (i, k) ks + min-path-weight m (k, j) ks (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *return*:
  assumes *m*: ¬(*min-path-weight m (i, k) ks + min-path-weight m (k, j) ks < min-path-weight m (i, j) ks*)
  assumes *finite ks*
  shows *min-path-weight m (i, j) (insert k ks) = min-path-weight m (i, j) ks*
⟨*proof*⟩

⟨*ML*⟩

lemma *Id-on-fw-inv*:
  shows *stable heap.*$Id_{\{a\}}$ *(fw-inv a m ys)*
⟨*proof*⟩

lemma *Id-on-fw-p-inv*:
  shows *stable heap.*$Id_{\{a\}}$ *(fw-p-inv a m ij ks)*
⟨*proof*⟩

lemma *modifies-fw-p-inv*:
  assumes *ij ∈ set (Array.interval a) − is*
  shows *stable Array.*$modifies_{a,\ is}$ *(fw-p-inv a m ij ks)*
⟨*proof*⟩

⟨*ML*⟩

lemma *fw-p-inv-cong*:
  assumes $a = a'$
  assumes $m = m'$
  assumes $ij = ij'$
  assumes $ks = ks'$
  assumes *s (heap.addr-of a) = s′ (heap.addr-of a′)*
  shows *fw-p-inv a m ij ks s = fw-p-inv a′ m′ ij′ ks′ s′*
⟨*proof*⟩

lemma *fw-p-invD*:
  assumes *fw-p-inv a m ij ks s*
  shows *heap.rep-inv a s*
    and *Array.get a ij s = min-path-weight m ij ks*
⟨*proof*⟩

lemma *fw-p-inv-fw-update*:

assumes *finite ks*
assumes $ij \in set$ (*Array.interval a*)
assumes *fw-p-inv a m ij ks s*
assumes *min-path-weight m* (*fst ij*, *k*) *ks* + *min-path-weight m* (*k*, *snd ij*) *ks* < *min-path-weight m ij ks*
shows *fw-p-inv a m ij* (*insert k ks*) (*Array.set a ij* (*min-path-weight m* (*fst ij*, *k*) *ks* + *min-path-weight m* (*k*, *snd ij*) *ks*) *s*)
⟨*proof*⟩

lemma *fw-p-inv-return*:
assumes *finite ks*
assumes *fw-p-inv a m ij ks s*
assumes ¬(*min-path-weight m* (*fst ij*, *k*) *ks* + *min-path-weight m* (*k*, *snd ij*) *ks* < *min-path-weight m ij ks*)
shows *fw-p-inv a m ij* (*insert k ks*) *s*
⟨*proof*⟩

⟨*ML*⟩

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 *insert k ks* ⊆ *set* (*Ix.interval* (*fst-bounds* (*array.bounds a*)))
assumes *Array.square a*
assumes *ij*: $ij \in set$ (*Array.interval a*)
defines $\bigwedge ij.\ G\ ij \equiv Array.modifies_{a,\ \{ij\ |\text{-::}unit.\ k \notin \{fst\ ij,\ snd\ ij\}\}}$
defines $A \equiv heap.Id_{\{a\}} \cup \bigcup (G\ `\ (set\ (Array.interval\ a) - \{ij\}))$
shows *prog.p2s* (*fw-update a ij k*)
≤ ⦃*fw-p-inv a m ij ks* ∧ *fw-p-inv a m* (*fst ij*, *k*) *ks* ∧ *fw-p-inv a m* (*k*, *snd ij*) *ks*⦄, *A*
⊢ *G ij*, ⦃λ-. *fw-p-inv a m ij* (*insert k ks*)⦄
⟨*proof*⟩

lemma *fw-chaotic*:
fixes *a* :: ('*i*::*Ix* × '*i*, *nat*) *array*
fixes *m* :: '*i matrix*
shows *prog.p2s* (*fw-chaotic a*) ≤ ⦃*fw-pre a m*⦄, $heap.Id_{\{a\}} \vdash heap.modifies_{\{a\}}$, ⦃*fw-post a m*⦄
⟨*proof*⟩

⟨*ML*⟩

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 Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

W.-P. de Roever, 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. Trefler. 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 fysiky*, 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.