CIMP

Peter Gammie

October 27, 2022

Abstract

CIMP extends the small imperative language IMP with control non-determinism and constructs for synchronous message passing.

Contents

1 Point-free notation
2 Infinite Sequences
  2.1 Decomposing safety and liveness
3 Linear Temporal Logic
  3.1 Leads-to and leads-to-via
  3.2 Fairness
  3.3 Safety and liveness
4 CIMP syntax and semantics
  4.1 Syntax
  4.2 Process semantics
  4.3 System steps
  4.4 Control predicates
5 State-based invariants
  5.0.1 Relating reachable states to the initial programs
  5.1 Simple-minded Hoare Logic/VCG for CIMP
    5.1.1 VCG rules
    5.1.2 Cheap non-interference rules
6 One locale per process
7 Example: a one-place buffer
8 Example: an unbounded buffer
9 Concluding remarks
References

1 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.

abbreviation (input)
  pred-K :: 'b ⇒ 'a ⇒ 'b (⟨-⟩) where
  ⟨f⟩ ≡ λs. f

abbreviation (input)
  pred-not :: ('a ⇒ bool) ⇒ 'a ⇒ bool (¬ · [40] 40) where
\(\neg a \equiv \lambda s. \neg a s\)

**abbreviation (input)**
\[
pred-conj :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infixr} \ \land \ 35) \ \text{where} \nonumber \]
\[
a \land b \equiv \lambda s. \ a \ s \land b \ s \nonumber \]

**abbreviation (input)**
\[
pred-disj :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infixr} \ \lor \ 30) \ \text{where} \nonumber \]
\[
a \lor b \equiv \lambda s. \ a \ s \lor b \ s \nonumber \]

**abbreviation (input)**
\[
pred-implies :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infixr} \ \rightarrow \ 25) \ \text{where} \nonumber \]
\[
a \rightarrow b \equiv \lambda s. \ a \ s \rightarrow b \ s \nonumber \]

**abbreviation (input)**
\[
pred-iff :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infixr} \ \leftrightarrow \ 25) \ \text{where} \nonumber \]
\[
a \leftrightarrow b \equiv \lambda s. \ a \ s \leftrightarrow b \ s \nonumber \]

**abbreviation (input)**
\[
pred-eq :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infix} = 40) \ \text{where} \nonumber \]
\[
a = b \equiv \lambda s. \ a \ s = b \ s \nonumber \]

**abbreviation (input)**
\[
pred-member :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \ \text{set}) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infix} \ \in \ 40) \ \text{where} \nonumber \]
\[
a \in b \equiv \lambda s. \ a \ s \in b \ s \nonumber \]

**abbreviation (input)**
\[
pred-neq :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infix} \neq 40) \ \text{where} \nonumber \]
\[
a \neq b \equiv \lambda s. \ a \ s \neq b \ s \nonumber \]

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

**abbreviation (input)**
\[
pred-less :: ('a \Rightarrow 'b:\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infix} \ < \ 40) \ \text{where} \nonumber \]
\[
a < b \equiv \lambda s. \ a \ s < b \ s \nonumber \]

**abbreviation (input)**
\[
pred-le :: ('a \Rightarrow 'b:\text{ord}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{infix} \leq 40) \ \text{where} \nonumber \]
\[
a \leq b \equiv \lambda s. \ a \ s \leq b \ s \nonumber \]

**abbreviation (input)**
\[
pred-plus :: ('a \Rightarrow 'b:\text{plus}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \ (\text{infixl} \ + 65) \ \text{where} \nonumber \]
\[
a + b \equiv \lambda s. \ a \ s + b \ s \nonumber \]

**abbreviation (input)**
\[
pred-minus :: ('a \Rightarrow 'b:\text{minus}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \ (\text{infixl} \ - 65) \ \text{where} \nonumber \]
\[
a - b \equiv \lambda s. \ a \ s - b \ s \nonumber \]

**abbreviation (input)**
\[
fun-fanout :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c \ (\text{infix} \ \bowtie \ 35) \ \text{where} \nonumber \]
\[
f \bowtie g \equiv \lambda x. \ (f \ x, \ g \ x) \nonumber \]

**abbreviation (input)**
\[
pred-all :: ('b \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool} \ (\text{binder} \ \forall \ 10) \ \text{where} \nonumber \]
\[
\forall x. \ P \ x \equiv \lambda s. \ \forall x. \ P \ x \ s \nonumber \]
2 Infinite Sequences

Infinite sequences and some operations on them.

More application specific.

We use the customary function-based representation.
type-synonym ′a seq = nat ⇒ ′a

type-synonym ′a seq-pred = ′a seq ⇒ bool

definition suffix :: ′a seq ⇒ nat ⇒ ′a seq (infixl |s 60) where
  σ |s i ≡ λj. σ (j + i)

primrec stake :: nat ⇒ ′a seq ⇒ ′a list where
  stake 0 σ = []
| stake (Suc n) σ = σ 0 # stake n (σ |s 1)

primrec shift :: ′a list ⇒ ′a seq ⇒ ′a seq (infixr @− 65) where
  shift [] σ = σ
| shift (x # xs) σ = (λi. case i of 0 ⇒ x | Suc i ⇒ shift xs σ)

abbreviation interval-syn (‘−→ [69, 0] 70) where
  σ(i → j) ≡ stake j (σ |s i)

lemma suffix-eval: (σ |s i) j = σ (j + i)

unfolding suffix-def by simp

lemma suffix-plus: σ |s n |s m = σ |s (m + n)
unfolding by (simp add: add.assoc)

lemma suffix-commute: ((σ |s n) |s m) = ((σ |s m) |s n)
by (simp add: suffix-plus add.commute)

lemma suffix-plus-com: σ |s m |s n = σ |s (m + n)
proof −
  have σ |s n |s m = σ |s (m + n) by (rule suffix-plus)
  then show σ |s m |s n = σ |s (m + n) by (simp add: suffix-commute)
qed

lemma suffix-zero: σ |s 0 = σ
unfolding by simp

lemma comp-suffix: f ◦ σ |s i = (f ◦ σ) |s i
unfolding comp-def by simp

lemmas suffix-simps[simp] =
  comp-suffix
  suffix-eval
  suffix-plus-com
  suffix-zero

lemma length-stake[simp]: length (stake n s) = n
by (induct n arbitrary: s) auto

lemma shift-simps[simp]:
  (xs @− σ) 0 = (if xs = [] then σ 0 else hd xs)
| (xs @− σ) |s Suc 0 = (if xs = [] then σ |s Suc 0 else tl xs @− σ)
by (induct xs) auto

lemma stake-nil[simp]:
  stake i σ = [] ≡ i = 0
by (cases i; clarsimp)

lemma stake-shift:
stake $i \ (\sigma \ @ \ - \) = \ \text{take} \ i \ \sigma \ \text{stake} \ (\ i \ - \ \text{length} \ \sigma \ ) \ \sigma$

by (induct $i$ arbitrary: $\sigma$) (auto simp: neq-Nil-conv)

lemma shift-snth-less[simp]:
  assumes $i < \text{length} \ \sigma$
  shows $(\sigma \ @ \ - \) \ i = \ \sigma \ ! \ i$
using assms
proof (induct $i$ arbitrary: $\sigma$)
  case (Suc $i$ $\sigma$)
  then show $?case$
  by (cases $\sigma$) simp-all
qed (simp add: hd-conv-nth nth-tl)

lemma shift-snth-ge[simp]:
  assumes $i \geq \text{length} \ \sigma$
  shows $(\sigma \ @ \ - \) \ i = \ \sigma \ (\ i \ - \ \text{length} \ \sigma)$
using assms
proof (induct $i$ arbitrary: $\sigma$)
  case (Suc $i$ $\sigma$)
  then show $?case$
  by (cases $\sigma$) simp-all
qed simp

lemma shift-snth:
$(\sigma \ @ \ - \) \ i = (\text{if} \ i < \text{length} \ \sigma \ \text{then} \ \sigma \ ! \ i \ \text{else} \ \sigma \ (\ i \ - \ \text{length} \ \sigma))$
by simp

lemma suffix-shift:
$(\sigma \ @ \ - \) \ i = \ \text{drop} \ i \ \sigma \ @ \ (\sigma \ |\ i \ - \ \text{length} \ \sigma)$
proof (induct $i$ arbitrary: $\sigma$)
  case (Suc $i$ $\sigma$)
  then show $?case$
  by (cases $\sigma$) simp-all
qed simp

lemma stake-nth[simp]:
  assumes $i < j$
  shows stake $j$ $\sigma$ ! $i$ = $\sigma$ ! $i$
using assms by (induct $j$ arbitrary: $\sigma$ ! $i$) (simp-all add: nth-Cons')

lemma stake-suffix-id:
  stake $i$ $\sigma$ @ ( $\sigma$ | $i$ ) = $\sigma$
by (induct $i$) (simp-all add: fun-eq-iff shift-snth split: nat.splits)

lemma id-stake-suffix-suffix:
  $\sigma = (\text{stake} \ i \ \sigma \ @ \ [\sigma \ i]) \ @ \ - \ (\sigma \ |\ i \ Suc \ i)$
using stake-suffix-id
apply (metis Suc-diff-le append-Nil2 diff-is-0-eq length-stake lessI nat.simps(3) nat-le-linear shift-snth stake-nil stake-shift take-Suc-conv-app-nth)
done

lemma stake-add[simp]:
  stake $i$ $\sigma$ @ stake $j$ ($\sigma$ | $i$ ) = stake ($i \ + \ j$) $\sigma$
apply (induct $i$ arbitrary: $\sigma$)
apply simp
apply auto
apply (metis One-nat-def plus-1-eq-Suc suffix-plus-com)
done

lemma stake-append: stake $n$ ($\sigma$ @ $s$) = take (min ($\text{length}$ $\sigma$) $n$) $\sigma$ @ stake ($n \ - \ \text{length} \ \sigma$) $s$
proof (induct $n$ arbitrary: $\sigma$)
  case (Suc $n$)
  then show $?case$
  apply clarsimp
  apply (cases $\sigma$)
  done
apply auto
done
done

lemma stake-shift-stake-shift:
  stake i σ @− stake j (σ |ₘ i) @− β = stake (i + j) σ @− β
apply (induct i arbitrary: σ)
apply simp
apply auto
apply (metis One-nat-def plus-1-eq-Suc suffix-plus-com)
done

lemma stake-suffix-drop:
  stake i (σ |ₘ j) = drop j (stake (i + j) σ)
by (metis append-eq-conv-conj length-stake semiring-normalization-rules(24) stake-add)

lemma stake-suffix:
  assumes i ≤ j
  shows stake j σ @− u |ₘ i = σ(i → j − i) @− u
by (simp add: assms stake-suffix-drop suffix-shift)

2.1 Decomposing safety and liveness
Famously properties on infinite sequences can be decomposed into safety and liveness properties Alpern and Schneider (1985); Schneider (1987). See Kindler (1994) for an overview.

definition safety :: 'a seq-pred ⇒ bool where
  safety P ←→ (∀ σ. ¬P σ −→ (∃ i. ∀ β. ¬P (stake i σ @− β)))

lemma safety-def2: — Contraposition gives the customary prefix-closure definition
  safety P ←→ (∀ σ. (∀ i. ∃ β. P (stake i σ @− β)) −→ P σ)
unfolding safety-def by blast

definition liveness :: 'a seq-pred ⇒ bool where
  liveness P ←→ (∀ α. ∃ σ. P (α @− σ))

lemmas safetyI = iffD2[OF safety-def, rule-format]
lemmas safetyI2 = iffD2[OF safety-def2, rule-format]
lemmas livenessI = iffD2[OF liveness-def, rule-format]

lemma safety-False:
  shows safety (λσ. False)
by (rule safetyI) simp

lemma safety-True:
  shows safety (λσ. True)
by (rule safetyI) simp

lemma safety-state-prop:
  shows safety (λσ. P (σ 0))
by (rule safetyI) auto

lemma safety-invariant:
  shows safety (λσ. ∀ i. P (σ i))
apply (rule safetyI)
apply clarsimp
apply (metis length-stake lessI shift-snth-less stake-nth)
done
lemma safety-transition-relation:
  shows safety (\lambda \sigma. \forall i. (\sigma i, \sigma (i + 1)) \in R)
apply (rule safetyI)
apply clarsimp
apply (metis (no-types, opaque-lifting) Suc-eq-plus1 add.left-neutral add-Suc-right add-diff-cancel-left' le-add1 list.sel(1) list.simps(3) shift.simps(1) stake.simps(2) stake-suffix suffix-def)
done

lemma safety-conj:
  assumes safety P
  assumes safety Q
  shows safety (P \land Q)
using assms unfolding safety-def by blast

lemma safety-always-eventually[simplified]:
  assumes safety P
  assumes \forall i. \exists j \geq i. \exists \beta. P (\sigma(0 \rightarrow j) @ - \beta)
  shows P \sigma
using assms unfolding safety-def2
apply --
apply (drule-tac x=\sigma in spec)
apply clarsimp
apply (drule-tac x=i in spec)
apply clarsimp
apply (rule-tac x=(stake j \sigma @ - \beta) | s i in exI)
apply (simp add: stake-shift-stake-shift stake-suffix)
done

lemma safety-disj:
  assumes safety P
  assumes safety Q
  shows safety (P \lor Q)
unfolding safety-def2 using assms
by (metis safety-always-eventually add-diff-cancel-right' diff-le-self le-add-same-cancel2)

The decomposition is given by a form of closure.

definition M_p :: 'a seq-pred \Rightarrow 'a seq-pred where
  M_p P = (\lambda \sigma. \forall i. \exists \beta. P (stake i \sigma @ - \beta))

definition Safe :: 'a seq-pred \Rightarrow 'a seq-pred where
  Safe P = (P \lor M_p P)

definition Live :: 'a seq-pred \Rightarrow 'a seq-pred where
  Live P = (P \lor \neg M_p P)

lemma decomp:
  P = (Safe P \land Live P)
unfolding Safe-def Live-def by blast

lemma safe:
  safety (Safe P)
unfolding Safe-def safety-def M_p-def
apply clarsimp
apply (simp add: stake-shift)
apply (rule-tac x=i in exI)
apply clarsimp
apply (rule-tac x=i in exI)
apply clarsimp
lemma live:
\[ \text{liveness} \ (\text{Live} \ P) \]

proof (rule livenessI)

fix \( \alpha \)

have \( (\exists \beta. \ P (\alpha @- \beta)) \lor \neg(\exists \beta. \ P (\alpha @- \beta)) \) by blast

also have \( \exists \beta. \ P (\alpha @- \beta) \lor (\forall \gamma. \ P (\alpha @- \gamma)) \) by blast

also have \( \ldots \longrightarrow (\exists \beta. \ P (\alpha @- \beta) \lor (\exists i. \ i = \text{length} \alpha \land (\forall \gamma. \ P (\alpha @- \beta) \oplus \neg \gamma))))) \) by (simp add: stake-shift)

also have \( \ldots \longrightarrow (\exists \beta. \ P (\alpha @- \beta) \lor (\forall \gamma. \neg P (\alpha @- \beta))) \) by blast

finally have \( \exists \beta. \ P (\alpha @- \beta) \lor (\exists i. \ i = \text{length} \alpha \land (\forall \gamma. \neg P (\alpha @- \beta) \oplus \neg \gamma))))) \) by blast

then show \( \exists \sigma. \ \text{Live} \ (\alpha @- \sigma) \) unfolding Live-def M_p-def by simp

qed

Sistla (1994) proceeds to give a topological analysis of fairness. An \textit{absolute} liveness property is a liveness property whose complement is stable.

\begin{definition}
\textbf{absolute-liveness :: \'}a seq-pred \Rightarrow bool where \ — closed under prepending any finite sequence
\end{definition}

\begin{definition}
\textbf{stable :: \'}a seq-pred \Rightarrow bool where \ — closed under suffixes
\end{definition}

lemma absolute-liveness-liveness:

assumes \( \text{absolute-liveness} \ P \)

shows \( \text{liveness} \ P \)

using assms unfolding absolute-liveness-def liveness-def by blast

lemma stable-absolute-liveness:

assumes \( P \sigma \)

assumes \( \neg P \sigma \) — extra hypothesis

shows \( \text{stable} \ P \longrightarrow \text{absolute-liveness} \ (\neg P) \)

using assms unfolding stable-def absolute-liveness-def

apply auto

apply (metis cancel-comm-monoid-add-class.diff-cancel drop-eq-Nil order-refl shift.simps(1) suffix-shift suffix-zero)

apply (metis stake-suffix-id)

done

\begin{definition}
\textbf{fairness :: \'}a seq-pred \Rightarrow bool where
\end{definition}

\begin{definition}
\textbf{fairness} :: \'}a seq-pred \Rightarrow bool where
\end{definition}

\begin{lemma}
\textbf{fairness-safety:

assumes \( \text{fairness} \ F \)

shows \( (\forall \sigma. \ F \sigma \longrightarrow P \sigma) \longrightarrow (\forall \sigma. \ P \sigma) \)

apply rule

using assms

apply clarsimp

unfolding safety-def fairness-def stable-def absolute-liveness-def

apply clarsimp

apply blast+

done

8
3 Linear Temporal Logic

To talk about liveness we need to consider infinitary behaviour on sequences. Traditionally future-time linear temporal logic (LTL) is used to do this Manna and Pnueli (1991); Owicki and Lamport (1982).

The following is a straightforward shallow embedding of the now-traditional anchored semantics of LTL Manna and Pnueli (1988). Some of it is adapted from the sophisticated TLA development in the AFP due to Grov and Merz (2011).

Unlike Lamport (2002), include the next operator, which is convenient for stating rules. Sometimes it allows us to ignore the system, i.e. to state rules as temporally valid (LTL-valid) rather than just temporally program valid (LTL-cimp-), in Jackson’s terminology.

**definition** state-prop :: ('a ⇒ bool) ⇒ 'a seq-pred ([·]) where

\[ [P] = (\lambda \sigma. P (\sigma 0)) \]

**definition** next :: 'a seq-pred ⇒ 'a seq-pred (∗- [80] 80) where

\[ (∗P) = (\lambda \sigma. P (\sigma |s 1)) \]

**definition** always :: 'a seq-pred ⇒ 'a seq-pred (∗- [80] 80) where

\[ (∗P) = (\lambda \sigma. \forall i. P (\sigma |s i)) \]

**definition** until :: 'a seq-pred ⇒ 'a seq-pred ⇒ 'a seq-pred (infixr U 30) where

\[ (P U Q) = (\lambda \sigma. \exists i. Q (\sigma |s i) \land (\forall k<i. P (\sigma |s k))) \]

**definition** eventually :: 'a seq-pred ⇒ 'a seq-pred (∗- [80] 80) where

\[ (∗P) = (\langle True \rangle U P) \]

**definition** release :: 'a seq-pred ⇒ 'a seq-pred ⇒ 'a seq-pred (infixr R 30) where

\[ (P R Q) = (\neg (\neg P U \neg Q)) \]

**definition** unless :: 'a seq-pred ⇒ 'a seq-pred ⇒ 'a seq-pred (infixr W 30) where

\[ (P W Q) = ((P U Q) \lor \boxdot P) \]

**abbreviation** (input) pred-always-imp-syn :: 'a seq-pred ⇒ 'a seq-pred ⇒ 'a seq-pred (infixr ↦ 25) where

\[ P \dashrightarrow Q \equiv \boxdot(P \rightarrow Q) \]

**lemmas** defs =

state-prop-def
always-def
eventually-def
next-def
release-def
unless-def
until-def

**lemma** suffix-state-prop[simp]:

shows \[ [P] (\sigma |s i) = P (\sigma i) \]

**unfolding** defs by simp

**lemma** alwaysI[intro]:

assumes \( \forall i. P (\sigma |s i) \)

shows \( (∗P) \sigma \)

**unfolding** defs using assms by blast

**lemma** alwaysD:

assumes \( (∗P) \sigma \)

shows \( P (\sigma |s i) \)

**using** assms unfolding **defs** by blast
lemma *alwaysE*: \[ (\Box P) \sigma \Rightarrow (P |s_i) \Rightarrow Q \Rightarrow Q \]
unfolding *defs* by *blast*

lemma *always-induct*:

assumes *P* \(\sigma\)
assumes \((\Box (P \rightarrow \Diamond P)) \sigma\)
shows \((\Box P) \sigma\)
proof (rule *alwaysI*)

fix *i* from *assms* show \((P |s_i)\)
unfolding *defs* by (induct *i*) simp-all
qed

lemma *seq-comp*:

fixes \(\sigma::'a\) seq
fixes \(P::'b\) seq-pred
fixes \(f::'a\Rightarrow'b\)
shows \((\Box (f \circ \sigma)) \leftrightarrow (\Box (\lambda \sigma. P (f \circ \sigma))) \sigma\)
\((\Diamond (f \circ \sigma)) \leftrightarrow (\Diamond (\lambda \sigma. P (f \circ \sigma))) \sigma\)
\((P \cup Q) (f \circ \sigma) \leftrightarrow ((\lambda \sigma. P (f \circ \sigma)) \cup (\lambda \sigma. Q (f \circ \sigma))) \sigma\)
\((P \wedge Q) (f \circ \sigma) \leftrightarrow ((\lambda \sigma. P (f \circ \sigma)) \wedge (\lambda \sigma. Q (f \circ \sigma))) \sigma\)

unfolding *defs* by simp-all

lemma *nextI [intro]*:

assumes \((P (\sigma |s_i))\)
shows \((\Diamond P) \sigma\)
using *assms* unfolding *defs* by simp

lemma *untilI [intro]*:

assumes \((Q (\sigma |s_i))\)
assumes \(\forall k<i. (P (\sigma |s_k))\)
shows \((P \cup Q) \sigma\)

unfolding *defs* using *assms* by blast

lemma *untilE*:

assumes \((P \cup Q) \sigma\)
obtains \(*i*\) where \((Q (\sigma |s_i))\) and \(\forall k<i. (P (\sigma |s_k))\)

using *assms* unfolding *until-def* by blast

lemma *eventuallyI [intro]*:

assumes \((P (\sigma |s_i))\)
shows \((\Diamond P) \sigma\)

unfolding *eventually-def* using *assms* by blast

lemma *eventuallyE [elim]*:

assumes \((\Diamond P) \sigma\)
obtains \(*i*\) where \((P (\sigma |s_i))\)

using *assms* unfolding *defs* by (blast elim: *untilE*)

lemma *unless-alwaysI*:

assumes \((\Box P) \sigma\)
shows \((P \wedge Q) \sigma\)

using *assms* unfolding *defs* by blast

lemma *unless-untilI*:

assumes \((Q (\sigma |s_j))\)
assumes \(\forall i. i < j \Rightarrow (P (\sigma |s_i))\)
shows \( (P \mathrel{W} Q) \sigma \)

unfolding defs using assms by blast

lemma always-imp-refl[iff]:
shows \( (P \mathrel{\rightarrow} P) \sigma \)

unfolding defs by blast

lemma always-imp-trans:
assumes \( (P \mathrel{\rightarrow} Q) \sigma \)
assumes \( (Q \mathrel{\rightarrow} R) \sigma \)
shows \( (P \mathrel{\rightarrow} R) \sigma \)

using assms unfolding defs by blast

lemma always-imp-mp:
assumes \( (P \mathrel{\rightarrow} Q) \sigma \)
assumes \( P \sigma \)
shows \( Q \sigma \)

using assms unfolding defs by (metis suffix-zero)

lemma always-imp-mp-suffix:
assumes \( (P \mathrel{\rightarrow} Q) \sigma \)
assumes \( P (\sigma | s i) \)
shows \( Q (\sigma | s i) \)

using assms unfolding defs by metis

Some basic facts and equivalences, mostly sanity.

lemma necessitation:
\[
(\forall s. P s) \Rightarrow (\Box P) \sigma \\
(\exists s. P s) \Rightarrow (\Diamond P) \sigma \\
(\forall s. P s) \Rightarrow (P W Q) \sigma \\
(\exists s. Q s) \Rightarrow (P U Q) \sigma
\]

unfolding defs by auto

lemma cong:
\[
(\forall s. P s = P' s) \Rightarrow [P] = [P'] \\
(\forall \sigma. P \sigma = P' \sigma) \Rightarrow (\Box P) = (\Box P') \\
(\forall \sigma. P \sigma = P' \sigma) \Rightarrow (\Diamond P) = (\Diamond P') \\
(\forall \sigma. P \sigma = P' \sigma) \Rightarrow (P W Q) = (P' W Q') \\
(\forall \sigma. Q \sigma = Q' \sigma) \Rightarrow (P U Q) = (P' U Q') \\
(\forall \sigma. Q \sigma = Q' \sigma) \Rightarrow (P W Q) = (P' W Q')
\]

unfolding defs by auto

lemma norm[simp]:
\[
[\langle False \rangle ] = \langle False \rangle \\
[\langle True \rangle ] = \langle True \rangle \\
(\neg [p]) = [\neg p] \\
([p] \land [q]) = [p \land q] \\
([p] \lor [q]) = [p \lor q] \\
([p] \Rightarrow [q]) = [p \Rightarrow q] \\
([p] \sigma \land [q] \sigma) = [p \land q] \sigma \\
([p] \sigma \lor [q] \sigma) = [p \lor q] \sigma \\
([p] \sigma \Rightarrow [q] \sigma) = [p \Rightarrow q] \sigma
\]

(\Box \langle False \rangle ) = \langle False \rangle \\
(\Box \langle True \rangle ) = \langle True \rangle \\
(\Diamond \langle False \rangle ) = \langle False \rangle \\
(\Diamond \langle True \rangle ) = \langle True \rangle
\[(\neg \Box P) \sigma = (\Diamond (\neg P)) \sigma\]
\[(\Box \Box P) = (\Box P)\]

\[(\Diamond (False)) = (False)\]
\[(\Diamond (True)) = (True)\]
\[(\neg\Diamond P) = (\Box (\neg P))\]
\[(\Diamond\Diamond P) = (\Diamond P)\]

\[(P \ W (False)) = (\Box P)\]

\[\neg(P \ U Q)) \sigma = (\neg P \ R \neg Q) \sigma\]
\[(\Box (False) \ U P) = P\]
\[(P \ U (False)) = (False)\]
\[(P \ U (True)) = (True)\]
\[(\Box (True) \ U P) = (\Diamond P)\]
\[(P \ U (P \ U Q)) = (P \ U Q)\]

\[\neg(P \ R Q)) \sigma = (\neg P \ U \neg Q) \sigma\]
\[(\Box (False) \ R P) = (\Box P)\]
\[(P \ R (False)) = (False)\]
\[(\Box (True) \ R P) = P\]
\[(P \ R (True)) = (True)\]

unfolding defs

apply (auto simp: fun-eq-iff)
apply (metis suffix-plus suffix-zero)
apply (metis suffix-plus suffix-zero)
  apply fastforce
  apply force
apply (metis add.commute add-diff-inverse-nat less-diff-conv2 not-le)
apply (metis add.right-neutral not-less0)
  apply force
  apply fastforce

done

lemma always-conj-distrib: \[\Box(P \land Q)) = (\Box P \land \Box Q)\]

unfolding defs by auto

lemma eventually-disj-distrib: \[\Diamond(P \lor Q)) = (\Diamond P \lor \Diamond Q)\]

unfolding defs by auto

lemma always-eventually[elim!]:
assumes (\Box P) \sigma
shows (\Diamond P) \sigma

using assms unfolding defs by auto

lemma eventually-imp-conv-disj: \[\Diamond(P \rightarrow Q)) = (\Diamond(\neg P) \lor \Diamond Q)\]

unfolding defs by auto

lemma eventually-imp-distrib:
\[(\Diamond(P \rightarrow Q)) = (\Box P \rightarrow \Diamond Q)\]

unfolding defs by auto

lemma unfold:
\[(\Box P) \sigma = (P \land \Box P) \sigma\]
\[(\Diamond P) \sigma = (P \lor \Box P) \sigma\]
\[(P \ W Q) \sigma = (Q \lor (P \land \Box (P \ W Q))) \sigma\]
\[(P \ U Q) \sigma = (Q \lor (P \land \Box (P \ U Q))) \sigma\]
\[(P \mathcal{R} Q) \sigma = (Q \land (P \lor \Diamond (P \mathcal{R} Q))) \sigma\]

unfolding defs
apply –
apply (metis (full-types) add.commute add-diff-inverse-nat less-one suffix-plus suffix-zero)
apply (metis (full-types) One-at-def add.right-neutral add-Suc-right lessI less-Suc-eq-0-disj suffix-plus suffix-zero)

apply auto
apply fastforce
apply (metis gr0-conv-Suc nat-neq-iff not-less-0 suffix-plus suffix-zero)
apply (metis gr0-conv-Suc nat-neq-iff not-less0 suffix-plus suffix-zero)
apply auto
apply fastforce
apply (case-tac i; auto)
apply force
using less-Suc-eq-0-disj apply fastforce
apply (case-tac i; auto)
done

lemma mono:
\\[(\Box P) \sigma; \land \sigma. P \sigma \Rightarrow P' \sigma] \Rightarrow (\Box P') \sigma\\
\\[(\Diamond P) \sigma; \land \sigma. P \sigma \Rightarrow P' \sigma] \Rightarrow (\Diamond P') \sigma\\
\\[(P \mathcal{U} Q) \sigma; (P \Rightarrow Q') \sigma \Rightarrow (P' \mathcal{U} Q') \sigma\\
\\[(P \mathcal{W} Q) \sigma; (P \Rightarrow Q') \sigma \Rightarrow (P' \mathcal{W} Q') \sigma\\

unfolding defs by force+

lemma always-imp-mono:
\\[(\Box P) \sigma; (P \Rightarrow P') \sigma] \Rightarrow (\Box P') \sigma\\
\\[(\Diamond P) \sigma; (P \Rightarrow P') \sigma] \Rightarrow (\Diamond P') \sigma\\
\\[(P \mathcal{U} Q) \sigma; (P \Leftarrow Q') \sigma \Rightarrow (P' \mathcal{U} Q') \sigma\\
\\[(P \mathcal{W} Q) \sigma; (P \Leftarrow Q') \sigma \Rightarrow (P' \mathcal{W} Q') \sigma\\

unfolding defs by force+

lemma next-conj-distrib:
\\((P \mathcal{W} Q)) = (\bigcirc P \land P' Q)\\
unfolding defs by auto

lemma next-disj-distrib:
\\((P \mathcal{U} Q)) = (\bigcirc P \lor Q)\\
unfolding defs by auto

lemma until-next-distrib:
\\((P \mathcal{U} Q)) = (P \mathcal{U} \Bigcirc Q)\\
unfolding defs by (auto simp: fun-eq-iff)

lemma until-imp-eventually:
\\((P \mathcal{U} Q) \Rightarrow \Diamond Q) \sigma\\
unfolding defs by auto

lemma until-until-disj:
assumes (P \mathcal{U} Q \mathcal{U} R) \sigma
shows ((P \lor Q) \mathcal{U} R) \sigma
using assms unfoldingdefs
apply clarsimp
apply (metis (full-types) add-diff-inverse-nat add-left-cancel-less)
done

lemma unless-unless-disj:
  assumes (P W Q W R) σ
  shows ((P ∨ Q) W R) σ
using assms unfoldingdefs
apply auto
apply (metis add.commute add-diff-inverse-nat leI less-diff-conv2)
apply (metis add-diff-inverse-nat)
done

lemma until-conj-distrib:
  ((P ∧ Q) U R) = ((P U R) ∧ (Q U R))
unfoldingdefs
apply (auto simp: fun-eq-iff)
apply (metis dual-order.strict-trans nat-neq-iff)
done

lemma until-disj-distrib:
  (P U (Q ∨ R)) = ((P U Q) ∨ (P U R))
unfoldingdefs by (auto simp: fun-eq-iff)

lemma eventually-until:
  (◇P) = (¬P U P)
unfoldingdefs
apply (auto simp: fun-eq-iff)
apply (case-tac P (x |s 0))
apply blast
apply (drule (1) ex-least-nat-less)
apply (metis le-simps(2))
done

lemma eventually-until-eventually:
  (◇(P U Q)) = (◇Q)
unfoldingdefs by force

lemma eventually-unless-until:
  ((P W Q) ∧ ◇Q) = (P U Q)
unfoldingdefs by force

lemma eventually-always-imp-always-eventually:
  assumes (◇□P) σ
  shows (□◇P) σ
using assms unfoldingdefs by (metis suffix-commute)

lemma eventually-always-next-stable:
  assumes (◇P) σ
  assumes (P ↩ ◇P) σ
  shows (◇□P) σ
using assms by (metis (no-types) eventuallyI alwaysD always-induct eventuallyE norm(15))

lemma next-stable-imp-eventually-always:
  assumes (P ↩ ◇P) σ
  shows (◇P → ◇□P) σ
using assms eventually-always-next-stable by blast
lemma always-eventually-always:
\[ \Diamond \Box P = \Box \Diamond P \]

unfoldingdefs by(clarsimp simp:fun-eq-iff)(metisadd.left-commute semiring-normalization-rules(25))

lemma stable-unless:
assumes \((P \leftrightarrow P \lor Q))\sigma\)
shows \((P \leftrightarrow P \lor Q))\sigma\)
usingassmsunfoldingdefs
apply-
apply(ruleccontr)
apply clarsimp
apply (drule (1)ex-least-nat-less[where \(P=\lambda j. \neg P | s i + j\)for i, simplified])
apply clarsimp
apply (metisadd Suc-right le-less less-Suc-eq)
done

lemma unless-induct: — Rule \textsc{wait} from Manna and Pnueli (1995, Fig 3.3)
assumes I: \((I \leftrightarrow (I \lor R))\sigma\)
assumes P: \((P \leftrightarrow I \lor R))\sigma\)
assumes Q: \((I \leftrightarrow Q))\sigma\)
shows \((P \leftrightarrow Q \lor R))\sigma\)
apply (intro alwaysI impI)
apply (erule impE[OF alwaysD[OF P]]
apply (erule disjE)
apply (rule always-imp-mono(\(\{\text{where }P=I \text{ and } Q=R\}\)])
apply (erule mp[OF alwaysD[OF stable-unless[OF I]]])
apply blast
apply (simpadd: Q alwaysD)
done

3.1 Leads-to and leads-to-via

Most of our assertions will be of the form \(\lambda s. A s \rightarrow (\Diamond C) s\) (pronounced “\(A\) leads to \(C\)” or \(\lambda s. A s \rightarrow (B \cup C) s\) (“\(A\) leads to \(C\) via \(B\)”).

Most of these rules are due to Jackson (1998) who used leads-to-via in a sequential setting. Others are due to Manna and Pnueli (1991).

The leads-to-via connective is similar to the “ensures” modality of Chandy and Misra (1989, §3.4.4).

abbreviation(input)
leads-to::’a seq-pred \Rightarrow ’a seq-pred \Rightarrow ’a seq-pred (infixr ~ 25) where
\(P ~ Q \equiv P \leftrightarrow \Diamond Q\)

lemma leads-to-refl:
shows \((P \sim P))\sigma\)
by (metis\{no-types, lifting\}necessitation(1)unfold(2))

lemma leads-to-trans:
assumes \((P \sim Q))\sigma\)
assumes \((Q \sim R))\sigma\)
shows \((P \sim R))\sigma\)
usingassmsunfoldingdefs by clarsimp (metis semiring-normalization-rules(25))

lemma leads-to-eventuallyE:
assumes \((P \sim Q))\sigma\)
assumes \((\Diamond P))\sigma\)
shows ($\Diamond Q$) $\sigma$
using **assms unfolding** **defs by** **auto**

**lemma** **leads-to-mono:**
assumes ($P' \leftrightarrow P$) $\sigma$
assumes ($Q \leftrightarrow Q'$) $\sigma$
assumes ($P \leadsto Q$) $\sigma$
shows ($P' \leadsto Q'$) $\sigma$
using **assms unfolding** **defs by** **clarsimp blast**

**lemma** **leads-to-eventually:**
  shows ($P \leadsto Q \longrightarrow \Diamond P \longrightarrow \Diamond Q$) $\sigma$
by **(metis (no-types, lifting) alwaysI unfold(2))**

**lemma** **leads-to-disj:**
assumes ($P \leadsto R$) $\sigma$
assumes ($Q \leadsto R$) $\sigma$
shows (($P \lor Q) \leadsto R$) $\sigma$
using **assms unfolding** **defs by** **simp**

**lemma** **leads-to-leads-to-viaE:**
 shows (($P \leadsto P \cup Q) \longrightarrow P \leadsto Q$) $\sigma$
unfolding **defs by** **clarsimp blast**

**lemma** **leads-to-via-concl-weaken:**
assumes ($R \leftrightarrow R'$) $\sigma$
assumes ($P \leftrightarrow Q \cup R$) $\sigma$
shows ($P \leftrightarrow Q \cup R'$) $\sigma$
using **assms unfolding** **LTL.defs by** **force**

**lemma** **leads-to-via-trans:**
assumes ($A \leftrightarrow B \cup C$) $\sigma$
assumes ($C \leftrightarrow D \cup E$) $\sigma$
shows ($A \leftrightarrow (B \lor D) \cup E$) $\sigma$

**proof** (**rule alwaysI**, **rule impI**)  
fix i assume $A$ ($\sigma \mid_s i$) with **assms show** (($B \lor D) \cup E$) ($\sigma \mid_s i$)  
  apply –  
  apply (**erule alwaysE**[**where** $i=i$])  
  apply **clarsimp**  
  apply (**erule untilE**)  
  apply **clarsimp**  
  apply (**erule (1) always-imp-mp-suffix**)  
  apply (**erule untilE**)  
  apply **clarsimp**  
  apply (**rule-tac i=ia + iaa in** untilI; **simp add: ac-simps**)  
  apply (**metis (full-types) add.assoc leI le-Suc-ex nat-add-left-cancel-less**)  
  done

**qed**

**lemma** **leads-to-via-disj:** — useful for case distinctions  
assumes ($P \leftrightarrow Q \cup R$) $\sigma$
assumes ($P' \leftrightarrow Q' \cup R$) $\sigma$
shows ($P \lor P' \leftrightarrow (Q \lor Q') \cup R$) $\sigma$
using **assms unfolding** **defs by** **(auto 10 0)**

**lemma** **leads-to-via-disj':** — more like a chaining rule  
assumes ($A \leftrightarrow B \cup C$) $\sigma$
assumes ($C \leftrightarrow D \cup E$) $\sigma$
shows \((A \lor C \iff (B \lor D) \cup E) \sigma\)

**proof**(rule alwaysI, rule impI, erule disjE)

fix \(i\) assume \(A (\sigma |_s i)\) with assumptions show \(((B \lor D) \cup E) (\sigma |_s i)\)

\[-\]
  \begin{itemize}
  \item apply –
  \item apply (erule alwaysE[where \(i=i\])
  \item apply clarsimp
  \item apply (erule untilE)
  \item apply clarsimp
  \item apply (drule \((1)\) always-imp-mp-suffix)
  \item apply (erule untilE)
  \item apply clarsimp
  \item apply (rule-tac \(i=ia + iaa\) in untilE; simp add: ac-simps)
  \item apply (metis (full-types) add.assoc leI le-Suc-ex nat-add-left-cancel-less)
  \end{itemize}

\(\text{done}\)

next

fix \(i\) assume \(C (\sigma |_s i)\) with assumptions(2) show \(((B \lor D) \cup E) (\sigma |_s i)\)

\[-\]
  \begin{itemize}
  \item apply –
  \item apply (erule alwaysE[where \(i=i\])
  \item apply (simp add: mono)
  \end{itemize}

\(?case\)

**qed**

**lemma** leads-to-via-stable-augmentation:

assumes stable: \((P \land Q \iff \circ Q) \sigma\)

assumes \(U: (A \iff P \cup C) \sigma\)

shows \(((A \land Q) \iff P \cup (C \land Q)) \sigma\)

**proof**(intro alwaysI impI, elim conjE)

fix \(i\) assume \(AP: A (\sigma |_s i) Q (\sigma |_s i)\)

have \(Q (\sigma |_s (j + i))\) if \(Q (\sigma |_s i)\) and \(\forall k<j. P (\sigma |_s (k + i))\) for \(j\)

using that stable by \((induct \(j\); force simp; defs)\)

with \(U\ AP\ show\ \(P \cup (\lambda\sigma. C \land Q) \sigma\) (\sigma |_s i)\)

unfolding \(defs\) by clarsimp (metis (full-types) add.commute)

**qed**

**lemma** leads-to-via-wf:

assumes \(wf R\)

assumes \(indhyp: \land t. (A \land [\delta = \{t\}] \iff B \cup (A \land [\delta \otimes \{t\} \in \{R\} \lor C)) \sigma\)

shows \((A \iff B \cup C) \sigma\)

**proof**(intro alwaysI impI)

fix \(i\) assume \(A (\sigma |_s i)\) with \(\langle wf R \rangle\) show \((B \cup C) (\sigma |_s i)\)

**proof**(induct \(\delta (\sigma |_s i)\) arbitrary; \(i\))

\(\text{case (less } i\text{) with } \text{indhyp[where } t=\delta (\sigma |_s i)\text{]} \text{ show } ?case\)

\[-\]
  \begin{itemize}
  \item apply –
  \item apply (drule alwaysD[where \(i=\)]
  \item apply clarsimp
  \item apply (rename-tac \(j\))
  \item apply (erule disjE; clarsimp)
  \item apply (drule-tac \(x=i + j\) in meta-spec; clarsimp)
  \item apply (erule untilE; clarsimp)
  \item apply (rename-tac \(j k\))
  \item apply (rule-tac \(i=j + k\) in untilI)
  \item apply (simp add: add.assoc)
  \item apply clarsimp
  \item apply (metis add.assoc add.commute add-diff-inverse-nat less-diff-conv2 not-le)
  \item apply auto
  \end{itemize}

\(\text{done}\)

**qed**
The well-founded response rule due to Manna and Pnueli (2010, Fig 1.23: \textsc{WELL} (well-founded response)), generalised to an arbitrary set of assertions and sequence predicates.

- \( W1 \) generalised to be contingent.
- \( W2 \) is a well-founded set of assertions that by \( W1 \) includes \( P \)

**Lemma** \textit{leads-to-uf}:

- \textit{fixes} \( Is :: ('a seq-pred \times ('a \Rightarrow 'b)) set \)
- \textit{assumes} \( uf :: (R :: 'b \Rightarrow) \)
- \textit{assumes} \( W1 :: \Box (\exists \varphi. [\langle \varphi \in fst ' Is \rangle] \wedge (P \Rightarrow \varphi)) \sigma \)
- \textit{assumes} \( W2 :: \forall (\varphi, \delta) \in Is. \exists (\varphi', \delta') \in insert (Q, \delta 0) Is. \forall t. (\varphi \wedge \delta = (t)) \Rightarrow \varphi' \wedge [\delta' \odot (t) \in (R)] \sigma \)
- \textit{shows} \( (P \Rightarrow Q) \sigma \)

**Proof**

- \textit{have} \( (\varphi \wedge \delta = (t)) \Rightarrow Q \) \( \sigma \) if \( (\varphi, \delta) \in Is \) for \( \varphi \delta t \)
  - \textit{using} \( uf R \) that \( W2 \)
  - \textit{unfolding} \textsc{ltl.defs split-def}
  - \textit{apply} \textit{clarsimp}
  - \textit{apply} \textit{(metis (no-types, opaque-lifting) ab-semigroup-add-class.add-ac(1) fist-eqD snd-conv surjective-pairing)}

\textbf{done} with \( W1 \) show \( \textit{?thesis} \)

- \textit{apply} \textit{−}
- \textit{apply} \textit{(rule alwaysI)}
- \textit{apply} \textit{clarsimp simp: fun-eq-iff}
- \textit{unfolding} \textit{defs}
  - \textit{apply} \textit{auto}
  - \textit{apply} \textit{(metis (full-types) add.left-commute semiring-normalization-rules(25))}

\textbf{done}

**3.2 Fairness**

A few renderings of weak fairness. \textit{van Glabbeek and Höfner (2019)} call this "response to insistence" as a generalisation of weak fairness.

**Definition** \textit{weakly-fair} :: 'a seq-pred \Rightarrow 'a seq-pred \Rightarrow 'a seq-pred where

\( \textit{weakly-fair enabled taken} = (\Box enabled \leftrightarrow \Diamond taken) \)

**Lemma** \textit{weakly-fair-def2}:

- \textit{shows} \( \textit{weakly-fair enabled taken} = \Box (\neg \Box (enabled \land 
\neg \textit{taken})) \)

**Unfolding** \textit{weakly-fair-def by (metis (full-types) always-conj-distrib norm(18))}

**Lemma** \textit{weakly-fair-def3}:

- \textit{shows} \( \textit{weakly-fair enabled taken} = (\Diamond \Box \textit{enabled} \rightarrow \Box \Diamond \textit{taken}) \)

**Unfolding** \textit{weakly-fair-def2}

- \textit{apply} \textit{clarsimp simp: fun-eq-iff}

**Unfolding** \textit{defs}

- \textit{apply} \textit{auto}

- \textit{apply} \textit{(metis (full-types) add.left-commute semiring-normalization-rules(25))}

\textbf{done}

**Lemma** \textit{weakly-fair-def4}:

- \textit{shows} \( \textit{weakly-fair enabled taken} = \Box \Diamond (\textit{enabled} \rightarrow \textit{taken}) \)

**Using** \textit{weakly-fair-def2 by force
lemma mp-weakly-fair:
  assumes weakly-fair enabled taken σ
  assumes (□enabled) σ
  shows (◊taken) σ
using assms unfolding weakly-fair-def using always-mp-mp by blast

lemma always-weakly-fair:
  shows □(weakly-fair enabled taken) = weakly-fair enabled taken
unfolding weakly-fair-def by simp

lemma eventually-weakly-fair:
  shows ◊(weakly-fair enabled taken) = weakly-fair enabled taken
unfolding weakly-fair-def2 by (simp add: always-eventually-always)

lemma weakly-fair-weaken:
  assumes (enabled' ↪ enabled) σ
  assumes (taken ↪ taken') σ
  shows (weakly-fair enabled taken ↪ weakly-fair enabled' taken') σ
using assms unfolding weakly-fair-def defs by simp blast

lemma weakly-fair-unless-until:
  shows (weakly-fair enabled taken ∧ (enabled ↦ enabled W taken)) = (enabled ↦ enabled U taken)
unfolding defs weakly-fair-def
apply (auto simp: fun-eq-iff)
apply (metis add.right-neutral)
done

lemma stable-leads-to-eventually:
  assumes (enabled ↦ □(enabled ∨ taken)) σ
  shows (enabled ↦ (□enabled ∨ ◊taken)) σ
using assms unfolding defs
apply −
apply (rule ccontr)
pert apply clarsimp
apply (drule (1) ex-least-nat-less[where P=λj. ¬ enabled (σ |s i + j) for i, simplified])
pert apply clarsimp
apply (metis add-Suc-right leI less-irrefl-nat)
done

lemma weakly-fair-stable-leads-to:
  assumes (weakly-fair enabled taken) σ
  assumes (enabled ↦ □(enabled ∨ taken)) σ
  shows (enabled ↦ taken) σ
using stable-leads-to-eventually[OF assms(2)] assms(1) unfolding defs weakly-fair-def
by (auto simp: fun-eq-iff)

lemma weakly-fair-stable-leads-to-via:
  assumes (weakly-fair enabled taken) σ
  assumes (enabled ↦ □(enabled ∨ taken)) σ
  shows (enabled ↦ enabled U taken) σ
using stable-unless[OF assms(2)] assms(1) by (metis (mono-tags) weakly-fair-unless-until)

Similarly for strong fairness. van Glabbeek and Höfner (2019) call this 'response to persistence' as a generalisation of strong fairness.

definition strongly-fair :: 'a seq-pred ⇒ 'a seq-pred ⇒ 'a seq-pred where
  strongly-fair enabled taken = (□◊enabled ↦ ◊taken)

lemma strongly-fair-def2:
\begin{align*}
\text{strongly-fair enabled taken} &= \lozenge(\neg \lozenge (\Diamond \text{enabled} \land \neg \text{taken})) \\
\text{unfolding strongly-fair-def by (metis weakly-fair-def weakly-fair-def2)}
\end{align*}

\text{lemma strongly-fair-def3:}
\begin{align*}
\text{strongly-fair enabled taken} &= (\lozenge \Diamond \text{enabled} \rightarrow \lozenge \Diamond \text{taken}) \\
\text{unfolding strongly-fair-def2 by (metis (full-types) always-eventually-always weakly-fair-def2 weakly-fair-def3)}
\end{align*}

\text{lemma always-strongly-fair:}
\begin{align*}
\lozenge (\text{strongly-fair enabled taken}) &= \text{strongly-fair enabled taken} \\
\text{unfolding strongly-fair-def by simp}
\end{align*}

\text{lemma eventually-strongly-fair:}
\begin{align*}
\Diamond (\text{strongly-fair enabled taken}) &= \text{strongly-fair enabled taken} \\
\text{unfolding strongly-fair-def2 by (simp add: always-eventually-always)}
\end{align*}

\text{lemma strongly-fair-disj-distrib:} — not true for weakly-fair
\begin{align*}
\text{strongly-fair} (\text{enabled1} \lor \text{enabled2}) &= (\text{strongly-fair enabled1 taken} \land \text{strongly-fair enabled2 taken}) \\
\text{unfolding strongly-fair-def2 defs}
\end{align*}

\text{apply (auto simp: fun-eq-iff)}
\begin{align*}
\text{apply blast} \\
\text{apply blast} \\
\text{apply (metis (full-types) semiring-normalization-rules(25))}
\end{align*}
\text{done}

\text{lemma strongly-fair-imp-weakly-fair:}
\begin{align*}
\text{assumes strongly-fair enabled taken } \sigma \\
\text{shows weakly-fair enabled taken } \sigma \\
\text{using assms unfolding strongly-fair-def3 weakly-fair-def3 by (simp add: eventually-always-imp-always-eventually)}
\end{align*}

\text{lemma always-enabled-weakly-fair-strongly-fair:}
\begin{align*}
\text{assumes} (\Box \text{enabled}) \sigma \\
\text{shows weakly-fair enabled taken } \sigma = \text{strongly-fair enabled taken } \sigma \\
\text{using assms by (metis strongly-fair-def3 strongly-fair-imp-weakly-fair unfold(2) weakly-fair-def3)}
\end{align*}

\subsection{3.3 Safety and liveness}

Sistla (1994) shows some characterisations of LTL formulas in terms of safety and liveness. Note his \((U)\) is actually \((W)\).

See also Chang, Manna, and Pnueli (1992).

\text{lemma safety-state-prop:}
\begin{align*}
\text{shows safety } [P] \\
\text{unfolding defs by (rule safety-state-prop)}
\end{align*}

\text{lemma safety-Next:}
\begin{align*}
\text{assumes safety } P \\
\text{shows safety } (\Diamond P) \\
\text{using assms unfolding defeats safety-def}
\end{align*}
\text{apply clarsimp}
\begin{align*}
\text{apply (metis (mono-tags) One-nat-def list.sel(3) nat.simps(3) stake.simps(2))}
\end{align*}
\text{done}

\text{lemma safety-unless:}
\begin{align*}
\text{assumes safety } P \\
\text{assumes safety } Q \\
\text{shows safety } (P W Q) \\
\text{proof(rule safetyI2)}
\end{align*}
\text{fix } \sigma \text{ assume } X : \exists \beta. (P W Q) (\text{stake i } \sigma @ - \beta) \text{ for } i
then show \( (P \land W \land Q) \sigma \)

proof (cases \( \forall i. j. \exists j. P (\sigma(i \rightarrow j) @\beta) \))

| case True

| with \( \langle \text{safety } P \rangle \) have \( \forall i. P (\sigma |_s i) \) unfolding safety-def2 by blast

| then show ?thesis by (blast intro: unless-alwaysI)

| next

| case False

| then obtain \( k k' \) where \( \forall \beta. \neg P (\sigma(k \rightarrow k') @\beta) \) by clarsimp

| then have \( \forall i u. k + k' \leq i \rightarrow \neg P ((\text{stake } i \sigma @- u |_s k) \land (\forall p < m. P (\sigma(p \rightarrow i - p) @- u))] \)

| unfolding def[s] using leE by blast

| then have \( \forall i u. k + k' \leq i \land (P \land W \land Q) (\text{stake } i \sigma @- u) \rightarrow (\exists m \leq k. Q ((\text{stake } i \sigma @- u) |_s m) \land (\forall p < m. P ((\text{stake } i \sigma @- u) |_s p))) \)

| lemma safety-always:

| assumes \( \text{safety } P \)

| shows \( \text{safety } (\Box P) \)

| using def[s] by (metis \text{norm}(20) safety-def safety-unless)

| lemma absolute-liveness-eventually:

| shows \( \text{absolute-liveness } P \iff (\exists \sigma. P \sigma) \land P = \Diamond P \)

| unfolding absolute-liveness-def def[s]

| by (metis cancel-comm-monoid-add-class.diff-cancel drop-eq-Nil order-refl shift.simps(1) stake-suffix-id suffix-shift suffix-zero)

| lemma stable-always:

| shows \( \text{stable } P \iff (\exists \sigma. P \sigma) \land P = \Box P \)

| unfolding stable-def def[s] by (metis suffix-zero)

To show that weakly-fair is a fairness property requires some constraints on \text{enabled} and \text{taken}:

- it is reasonable to assume they are state formulas
- \text{taken} must be satisfiable

| lemma fairness-weakly-fair:

| assumes \( \exists s. \text{taken } s \)

| shows \( \text{fairness } (\text{weakly-fair } [\text{enabled}] [\text{taken}]) \)

| unfolding fairness-def stable-def absolute-liveness-def weakly-fair-def

| using def[s]

| apply auto

| apply (rule-tac \( x=\lambda s \in \text{exI} \))

| apply fastforce

| apply (simp add: alwaysD)

| apply (rule-tac \( x=\lambda s \in \text{exI} \))

| apply fastforce

| apply (metis (full-types) absolute-liveness-def absolute-liveness-eventually eventually-weakly-fair weakly-fair-def)

| done
lemmas

shows fairness (strongly-fair [enabled] [taken])

unfolding fairness-def stable-def absolute-liveness-def strongly-fair-def using assms
apply auto
apply (rule-tac x=λ- .s in exI)
apply fastforce
apply (simp add: alwaysD)
apply (rule-tac x=λ- .s in exI)
apply fastforce
apply (metis (full-types) absolute-liveness-def absolute-liveness-evenually eventually-weakly-fair weakly-fair-def)
done

4 CIMP syntax and semantics

We define a small sequential programming language with synchronous message passing primitives for describing the individual processes. This has the advantage over raw transition systems in that it is programmer-readable, includes sequential composition, supports a program logic and VCG (§5.1), etc. These processes are composed in parallel at the top-level.

CIMP is inspired by IMP, as presented by Winskel (1993) and Nipkow and Klein (2014), and the classical process algebras CCS (Milner 1980, 1989) and CSP (Hoare 1985). Note that the algebraic properties of this language have not been developed.

As we operate in a concurrent setting, we need to provide a small-step semantics (§4.2), which we give in the style of structural operational semantics (SOS) as popularised by Plotkin (2004). The semantics of a complete system (§4.3) is presently taken simply to be the states reachable by interleaving the enabled steps of the individual processes, subject to message passing rendezvous. We leave a trace or branching semantics to future work.

This theory contains all the trusted definitions. The soundness of the other theories supervenes upon this one.

4.1 Syntax

Programs are represented using an explicit (deep embedding) of their syntax, as the semantics needs to track the progress of multiple threads of control. Each (atomic) basic command (§?) is annotated with a location, which we use in our assertions (§4.4). These locations need not be unique, though in practice they likely will be.

Processes maintain local states of type state. These can be updated with arbitrary relations of state ⇒ state set with LocalOp, and conditions of type s ⇒ bool are similarly shallowly embedded. This arrangement allows the end-user to select their own level of atomicity.

The sequential composition operator and control constructs are standard. We add the infinite looping construct Loop so we can construct single-state reactive systems; this has implications for fairness assertions.

type-synonym 's bexp = 's ⇒ bool

datatype ('answer, 'location, 'question, 'state) com
  = Request 'location' state ⇒ 'question' answer ⇒ 'state' ⇒ 'state set' (\[-\] Request - - [0, 70, 70] 71)
  | Response 'location' 'question' ⇒ 'state' ⇒ ('state × 'answer) set (\[-\] Response - [0, 70] 71)
  | LocalOp 'location' state ⇒ 'state set' (\[-\] LocalOp - [0, 70] 71)
  | Cond1 'location' state bexp ('answer', 'location', 'question', 'state') com (\[-\] IF - THEN - FI [0, 0] 71)
  | Cond2 'location' state bexp ('answer', 'location', 'question', 'state') com (\[-\] IF -/ ELSE -/ FI [0, 0, 0, 0] 71)
  | Loop ('answer', 'location', 'question', 'state') com
  | While 'location' state bexp ('answer', 'location', 'question', 'state') com (\[-\] WHILE -/ DO -/ OD [0, 0] 71)
  | Seq ('answer', 'location', 'question', 'state') com (infixr ;; 69)
We provide a one-armed conditional as it is the common form and avoids the need to discover a label for an internal SKIP and/or trickier proofs about the VCG.

In contrast to classical process algebras, we have local state and distinct request and response actions. These provide an interface to Isabelle/HOL’s datatypes that avoids the need for binding (ala the π-calculus of Milner (1989)) or large non-deterministic sums (ala CCS (Milner 1980, §2.8)). Intuitively the requester poses a 'question with a Request command, which upon rendezvous with a responder’s Response command receives an 'answer. The 'question is a deterministic function of the requester’s local state, whereas responses can be non-deterministic. Note that CIMP does not provide a notion of channel; these can be modelled by a judicious choice of 'question.

We also provide a binary external choice operator (⊕) (infix (⊕)). Internal choice can be recovered in combination with local operations (see Milner (1980, §2.3)).

We abbreviate some common commands: SKIP is a local operation that does nothing, and the floor brackets simplify deterministic LocalOps. We also adopt some syntax magic from Makarius’s Hoare and Multiquote theories in the Isabelle/HOL distribution.

abbreviation SKIP-syn (\{\} / SKIP [0] 71) where
\(\{\}\) SKIP \equiv \(\{\}\) LocalOp (\(\lambda\). \{s\})

abbreviation (input) DetLocalOp :: 'location \Rightarrow ('state \Rightarrow 'state)
\Rightarrow ('answer, 'location, 'question, 'state) com (\{\} [-] [0, 0] 71) where
\(\{\}\) \{f\} \equiv \(\{\}\) LocalOp (\(\lambda\). \{f s\})

syntax
-quote :: 'b \Rightarrow ('a \Rightarrow 'b) ((\rightarrow) [0] 1000)
-antiquote :: ('a \Rightarrow 'b) \Rightarrow 'b ((\rightarrow) [0] 1000)
-Arrive :: 'location \Rightarrow idt \Rightarrow 'b \Rightarrow ('answer, 'location, 'question, 'state) com ((\{\} \rightarrow) [0, 0, 70] 71)
-NonDetAssign :: 'location \Rightarrow idt \Rightarrow 'b set \Rightarrow ('answer, 'location, 'question, 'state) com ((\{\} \rightarrow) [0, 0, 70] 71)

abbreviation (input) NonDetAssign :: 'location \Rightarrow ('val \Rightarrow 'val) \Rightarrow 'state \Rightarrow 'state) \Rightarrow ('state \Rightarrow 'val set)
\Rightarrow ('answer, 'location, 'question, 'state) com where
NonDetAssign l upd es \equiv \(\{\}\) LocalOp (\(\lambda\). \{ upd (e) s \mid e. e \in es s \})

translations
\[\{\}\] \(x. e \Rightarrow \text{CONST DetLocalOp} l \equiv \langle\text{(-update-name} x (\lambda.. c))\rangle\]
\[\{\}\] \(x \in es \Rightarrow \text{CONST NonDetAssign} l \equiv \langle\text{(-update-name} x \langle\text{es}\rangle)\rangle\)

parse-translation
let
fun antiquote-tr i (Const \(\{\}\) syntax-const -antiquote\}, -) \$ (t as Const \(\{\}\) syntax-const -antiquote\}, -) \$ -) = skip-antiquote-tr i t
\[\text{antiquote-tr i (Const \(\{\}\) syntax-const -antiquote\}, -) \$ t) =\]
\[\text{antiquote-tr i t \$ Bound i}\]
\[\text{antiquote-tr i (t \$ u) = antiquote-tr i t \$ antiquote-tr i u}\]
\[\text{antiquote-tr i (Abs (x, T, t)) = Abs (x, T, antiquote-tr (i + 1) t)}\]
\[\text{antiquote-tr - a = a}\]

and skip-antiquote-tr i ((c as Const \(\{\}\) syntax-const -antiquote\}, -) \$ t) =
\[c \$ skip-antiquote-tr i t\]
\[\text{skip-antiquote-tr i t = antiquote-tr i t;}\]

fun quote-tr [t] = Abs (s, dummyT, antiquote-tr 0 (Term.incr-boundvars 1 t))
\[\text{quote-tr ts = raise TERM (quote-tr ts);}\]
in [(\{\} syntax-const -quote\}, K quote-tr] end
4.2 Process semantics

Here we define the semantics of a single process’s program. We begin by defining the type of externally-visible behaviour:

datatype (’answer, ’question) seq-label
= sl-Internal (τ)
| sl-Send ’question ’answer («-, »)
| sl-Receive ’question ’answer (»-, »)

We define a labelled transition system (an LTS) using an execution-stack style of semantics that avoids special treatment of the SKIPs introduced by a traditional small step semantics (such as Winskel (1993, Chapter 14)) when a basic command is executed. This was suggested by Thomas Sewell; Pitts (2002) gave a semantics to an ML-like language using this approach.

We record the location of the command that was executed to support fairness constraints.

type-synonym (’answer, ’location, ’question, ’state) local-state
= (’answer, ’location, ’question, ’state) com list × ’location option × ’state

inductive
small-step :: (’answer, ’location, ’question, ’state) local-state
⇒ (’answer, ’question) seq-label
⇒ (’answer, ’location, ’question, ’state) local-state ⇒ bool (¬ ⇒ [55, 0, 56])

where
\[ \alpha = \text{action } s; s' \in \text{val } \beta s \] \(⇒ (\llbracket l \rrbracket \ \text{Request action val # } cs, -, s) ⇔ s_\alpha, \beta_s (cs, \text{Some } l, s') \)
\[ (s', \beta) \in \text{action } \alpha s \] \(⇒ (\llbracket l \rrbracket \ \text{Response action # } cs, -, s) ⇔ s_\alpha, \beta_s (cs, \text{Some } l, s') \)
\[ s' \in R s \] \(⇒ (\llbracket l \rrbracket \ \text{LocalOp } R \# cs, -, s) ⇔_\tau (cs, \text{Some } l, s') \)
\[ b s \] \(⇒ (\llbracket l \rrbracket \ \text{IF } b \ \text{THEN } c \ FI \# cs, -, s) ⇔_\tau (c \# cs, \text{Some } l, s) \)
\[ ¬b s \] \(⇒ (\llbracket l \rrbracket \ \text{IF } b \ \text{THEN } c \ FI \# cs, -, s) ⇔_\tau (cs, \text{Some } l, s) \)
\[ (c \# \text{LOOP DO } c \ OD \# cs, s) ⇔_\alpha (c s', s') \] \(⇒ (\text{LOOP DO } c \ OD \# cs, s) ⇔_\alpha (c s', s') \)
\[ b s \] \(⇒ (\llbracket l \rrbracket \ \text{WHILE } b \ DO \ c \ OD \# cs, -, s) ⇔_\tau (c \# \llbracket l \rrbracket \ \text{WHILE } b \ DO \ c \ OD \# cs, \text{Some } l, s) \)
\[ ¬b s \] \(⇒ (\llbracket l \rrbracket \ \text{WHILE } b \ DO \ c \ OD \# cs, -, s) ⇔_\tau (cs, \text{Some } l, s) \)
\[ (c_1 \# c_2 \# cs, s) ⇔_\alpha (c s', s') \] \(⇒ (c_1; c_2 \# cs, s) ⇔_\alpha (c s', s') \)
\[ \text{Choose1: } (c_1 \# cs, s) ⇔_\alpha (c s', s') \] \(⇒ (c_1 \oplus c_2 \# cs, s) ⇔_\alpha (cs', s') \)
\[ \text{Choose2: } (c_2 \# cs, s) ⇔_\alpha (c s', s') \] \(⇒ (c_1 \oplus c_2 \# cs, s) ⇔_\alpha (cs', s') \)

The following projections operate on local states. These should not appear to the end-user.

abbreviation cPGM :: (’answer, ’location, ’question, ’state) local-state ⇒ (’answer, ’location, ’question, ’state)
com list where
cPGM ≡ fst

abbreviation cTKN :: (’answer, ’location, ’question, ’state) local-state ⇒ ’location option
where
cTKN s ≡ fst (snd s)

abbreviation cLST :: (’answer, ’location, ’question, ’state) local-state ⇒ ’state
where
cLST s ≡ snd (snd s)

4.3 System steps

A global state maps process names to process’ local states. One might hope to allow processes to have distinct types of local state, but there remains no good solution yet in a simply-typed setting; see Schirmer and Wenzel.
An execution step of the overall system is either any enabled internal $\tau$ step of any process, or a communication rendezvous between two processes. For the latter to occur, a Request action must be enabled in process $p1$, and a Response action in (distinct) process $p2$, where the request/response labels $\alpha$ and $\beta$ (semantically) match.

We also track global communication history here to support assertional reasoning (see §5).

**record** $(\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteright\textquoteleft state}) \text{ global-state} = $

\begin{align*}
& \text{GST :: (\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteright\textquoteleft state}) \text{ global-state}} \\
& \text{HST :: (\text{\textquoteleft answer, \textquoteleft question\textquoteright\textquoteleft history}}
\end{align*}

**inductive** — This is a predicate of the current state, so the successor state comes first.

**system-step :: \textquoteleft proc set**

\begin{align*}
& \Rightarrow (\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteright\textquoteleft state}) \text{ system-state} \\
& \Rightarrow (\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteright\textquoteleft state}) \text{ system-state} \\
& \Rightarrow \text{bool}
\end{align*}

\begin{align*}
\text{where} \\
& \text{LocalStep: } [ \text{GST sh p} \rightarrow_\tau \text{ls1}; \text{GST sh'} = (\text{GST sh})(p := \text{ls1}); \text{HST sh'} = \text{HST sh} ] \implies \text{system-step } \{p\} \text{ sh'} \text{ sh} \\
& \text{CommunicationStep: } [ \text{GST sh p} \rightarrow_{s_{\alpha}} \beta_{\text{ls1}}; \text{GST sh q} \rightarrow_{s_{\alpha}} \beta_{\text{ls2}}; p \neq q; \text{GST sh'} = (\text{GST sh})(p := \text{ls1}', q := \text{ls2}'); \text{HST sh'} = \text{HST sh} \odot [(\alpha, \beta)] ] \implies \text{system-step } \{p, q\} \text{ sh'} \text{ sh}
\end{align*}

In classical process algebras matching communication actions yield $\tau$ steps, which aids nested parallel composition and the restriction operation (Milner 1980, §2.2). As CIMP does not provide either we do not need to hide communication labels. In CCS/CSP it is not clear how one reasons about the communication history, and it seems that assertional reasoning about these languages is not well developed.

We define predicates over communication histories and system states. These are uncurried to ease composition.

**type-synonym** $(\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteleft state}) \text{ state-pred}$

\begin{align*}
& = (\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteleft state}) \text{ system-state} \Rightarrow \text{bool}
\end{align*}

The LST operator (written as a postfix $\downarrow$) projects the local states of the processes from a $(\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteleft state}) \text{ system-state}$, i.e. it discards control location information.

Conversely the LSTP operator lifts predicates over local states into predicates over $(\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteleft state}) \text{ system-state}$. Predicates that do not depend on control locations were termed **universal assertions** by Levin and Gries (1981, §3.6).

**type-synonym** $(\text{\textquoteleft proc, \textquoteleft state\textquoteright\textquoteleft state}) \text{ local-state-pred}$

\begin{align*}
& = (\text{\textquoteleft proc, \textquoteleft state\textquoteright\textquoteleft state}) \text{ local-states} \Rightarrow \text{bool}
\end{align*}

**definition** $\text{LST :: (\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteleft state}) \text{ system-state} \Rightarrow (\text{\textquoteleft proc, \textquoteleft state\textquoteright\textquoteleft local-states (\downarrow [1000] 1000) where}}$

\begin{align*}
& \text{s}_{\downarrow} = c\text{LST} \odot \text{GST s}
\end{align*}

**abbreviation** $(\text{input}) \text{LSTP :: (\text{\textquoteleft proc, \textquoteleft state\textquoteright\textquoteleft local-state-pred}$

\begin{align*}
& \Rightarrow (\text{\textquoteleft answer, \textquoteleft location, \textquoteleft proc, \textquoteleft question, \textquoteleft state\textquoteright\textquoteleft state}) \text{ state-pred where}$
\end{align*}

$LSTP P \equiv \lambda s. P s_{\downarrow}$
4.4 Control predicates

Following Lamport (1980), we define the at predicate, which holds of a process when control resides at that location. Due to non-determinism processes can be at a set of locations; it is more like “a statement with this location is enabled”, which incidentally handles non-unique locations. Lamport’s language is deterministic, so he doesn’t have this problem. This also allows him to develop a stronger theory about his control predicates.

type-synonym 'location label = 'location set

primrec
atC :: ('answer, 'location, 'question, 'state) com ⇒ 'location label
where
atC ([]| Request action val) = {l}
atC ([]| Response action) = {l}
atC ([]| LocalOp f) = {l}
atC ([]| IF - THEN - FI) = {l}
atC ([]| IF - THEN - ELSE - FI) = {l}
atC ([]| WHILE - DO - OD) = {l}
atC (LOOP DO c OD) = atC c
atC (c1; c2) = atC c1 ∪ atC c2

primrec atCs :: ('answer, 'location, 'question, 'state) com list ⇒ 'location label
where
atCs [] = {}
atCs (c # -) = atC c

We provide the following definitions to the end-user. AT maps process names to a predicate that is true of locations where control for that process resides, and the abbreviation at provides a conventional way to use it. The constant atS specifies that control for process p resides at one of the given locations. This stands in for, and generalises, the in predicate of Lamport (1980).

definition AT :: ('answer, 'location, 'proc, 'question, 'state) system-state ⇒ 'proc ⇒ 'location label
where
AT s p = atCs (cPGM (GST s p))

abbreviation at :: 'proc ⇒ 'location ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
at p l s ≡ l ∈ AT s p

definition atS :: 'proc ⇒ 'location set ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
atS p ls s = (∃l∈ls. at p l s)

definition atLs :: 'proc ⇒ 'location label set ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
atLs p labels s = (AT s p ∈ labels)

abbreviation (input) atL :: 'proc ⇒ 'location label ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
atL p label ≡ atLs p {label}

definition atPLs :: ('proc × 'location label) set ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
atPLs pls = (∀ p label. ⟨p, label⟩ ∈ pls) → atL p label

The constant taken provides a way of identifying which transition was taken. It is somewhat like Lamport’s after, but not quite due to the presence of non-determinism here. This does not work well for invariants or preconditions.

definition taken :: 'proc ⇒ 'location ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
taken p l s ←→ cTKN (GST s p) = Some l

A process is terminated if it not at any control location.

**abbreviation** (input) terminated :: 'proc ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred where terminated p ≡ atL p {]

A complete system consists of one program per process, and a (global) constraint on their initial local states. From these we can construct the set of initial global states and all those reachable by system steps (§4.3).

**type-synonym** ('answer, 'location, 'proc, 'question, 'state) programs

= 'proc ⇒ ('answer, 'location, 'question, 'state) com

**record** ('answer, 'location, 'proc, 'question, 'state) pre-system =

PGMs :: ('answer, 'location, 'proc, 'question, 'state) programs

INIT :: ('proc, 'state) local-state-pred

**definition**

initial-state :: ('answer, 'location, 'proc, 'question, 'state, 'ext) pre-system-ext ⇒ ('answer, 'location, 'proc, 'question, 'state) global-state

⇒ bool

where

initial-state sys s = ((∀ p. cPGM (s p) = [PGMs sys p] ∧ cTKN (s p) = None) ∧ INIT sys (cLST ◦ s))

We construct infinite runs of a system by allowing stuttering, i.e., arbitrary repetitions of states following Lamport (2002, Chapter 8), by taking the reflexive closure of the system-step relation. Therefore terminated programs infinitely repeat their final state (but note our definition of terminated processes in §4.4).

Some accounts define stuttering as the finite repetition of states. With or without this constraint prerun contains junk in the form of unfair runs, where particular processes do not progress.

**definition**

system-step-reflclp :: ('answer, 'location, 'proc, 'question, 'state) system-state seq-pred

where

system-step-reflclp σ ←→ (λsh sh'. ∃ pls. system-step pls pls' sh' sh)⇒ (σ 0) (σ 1)

**definition**

prerun :: ('answer, 'location, 'proc, 'question, 'state, 'ext) pre-system-ext ⇒ ('answer, 'location, 'proc, 'question, 'state) system-state seq-pred

where

prerun sys = ((λσ. initial-state sys (GST (σ 0)) ∧ HST (σ 0) = [])) ∧ □system-step-reflclp

**definition** — state-based invariants only

prerun-valid :: ('answer, 'location, 'proc, 'question, 'state, 'ext) pre-system-ext ⇒ ('answer, 'location, 'proc, 'question, 'state) system-state seq-pred ⇒ bool (- ⊨ pre - [11, 0] 11)

where

(sys ⊨ pre φ) ←→ (∀ σ. prerun sys σ → (□[φ]) σ)

A run of a system is a prerun that satisfies the FAIR requirement. Typically this would include weak fairness for every transition of every process.

**record** ('answer, 'location, 'proc, 'question, 'state) system =

('answer, 'location, 'proc, 'question, 'state) pre-system

+ FAIR :: ('answer, 'location, 'proc, 'question, 'state) system-state seq-pred

**definition**

run :: ('answer, 'location, 'proc, 'question, 'state) system ⇒ ('answer, 'location, 'proc, 'question, 'state) system-state seq-pred

where

run sys = (prerun sys ∧ FAIR sys)

**definition**

valid :: ('answer, 'location, 'proc, 'question, 'state) system ⇒ ('answer, 'location, 'proc, 'question, 'state) system-state seq-pred ⇒ bool (- ⊨ - [11, 0] 11)
where
\[(\text{sys} \models \varphi) \iff (\forall \sigma. \text{run \ sys} \sigma \rightarrow \varphi \sigma)\]

5 State-based invariants

We provide a simple-minded verification condition generator (VCG) for this language, providing support for establishing state-based invariants. It is just one way of reasoning about CIMP programs and is proven sound w.r.t. to the CIMP semantics.

Our approach follows Lamport (1980); Lamport and Schneider (1984) (and the later Lamport (2002)) and closely related work by Apt, Francez, and de Roever (1980), Cousot and Cousot (1980) and Levin and Gries (1981), who suggest the incorporation of a history variable. Cousot and Cousot (1980) apparently contains a completeness proof. Lamport mentions that this technique was well-known in the mid-80s when he proposed the use of prophecy variables\(^2\). See also de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001) for an extended discussion of some of this.

declare small-step.intros[intro]

inductive-cases small-step-inv:
\(\{(\text{[]} \| \text{[]=} \phi \}\) \rightarrow a s'
\(\{(\text{[]=} \phi)\) \rightarrow a s'
\(\{(\text{LocalOp \ R \[]=} \phi, \text{ls})\) \rightarrow a s'
\(\{(\text{IF \ b \ THEN \ c \[]=} \phi, \text{ls})\) \rightarrow a s'
\(\{(\text{IF \ b \ THEN \ c1 \ ELSE \ c2 \[]=} \phi, \text{ls})\) \rightarrow a s'
\(\{(\text{WHILE \ b \ DO \ c \[]=} \phi, \text{ls})\) \rightarrow a s'
\(\{(\text{LOOP \ DO \ c \[]=} \phi, \text{ls})\) \rightarrow a s'

lemma small-step-stuck:
\(\neg (\text{[]}, s) \rightarrow_a c'\)
by (auto elim: small-step.cases)

declare system-step.intros[intro]

By default we ask the simplifier to rewrite atS using ambient AT information.

lemma atS-state-weak-cong[cong]:
\(AT s p = AT s' p \implies atS p \ ls s \leftarrow atS p \ ls s'\)
by (auto simp: atS-def)

We provide an incomplete set of basic rules for label sets.

lemma atS-simps:
\(\neg atS p \ {\text{[]}} s\)
\(atS p \ {\text{[]}} s \leftarrow at p \ l s\)
\(\{at p \ l s; \ l \in \ls\} \implies atS p \ ls s\)
\((\forall l. at p \ l s \rightarrow l \notin \ls) \implies \neg atS p \ ls s\)
by (auto simp: atS-def)

lemma atS-mono:
\(\{atS p \ ls s; \ ls \subseteq \ls'\} \implies atS p \ ls s'\)
by (auto simp: atS-def)

lemma atS-un:
\(atS p \ (l \cup l') s \leftarrow atS p \ l s \lor atS p \ l' s\)
by (auto simp: atS-def)

lemma atLs-disj-union[simp]:
\(atLs p \ label0 \lor atLs p \ label1) = atLs p (label0 \cup label1)\)
unfolding atLs-def by simp

\(^2\)https://lamport.azurewebsites.net/pubs/pubs.html
lemma atLs-insert-disj:
\[ atLs p \ (insert l \ label0) = (atL p \ l \lor atLs p \ label0) \]
by simp

lemma small-step-terminated:
\[ s \xrightarrow{s} s' \Rightarrow atCs \ (fst \ s) = \{} \Rightarrow atCs \ (fst \ s') = \{} \]
by (induct pred: small-step) auto

lemma atC-not-empty:
\[ atC \ c \neq \{} \]
by (induct c) auto

lemma atCs-empty:
\[ atCs \ cs = \{} \longleftrightarrow cs = [] \]
by (induct cs) (auto simp: atC-not-empty)

lemma terminated-no-commands:
assumes terminated p sh
shows \( \exists \ s. \ GST \ sh \ p = ([], s) \)
using assms unfolding atLs-def AT-def by (metis atCs-empty prod.collapse singletonD)

lemma terminated-GST-stable:
assumes system-step q sh' sh
assumes terminated p sh
shows GST sh p = GST sh' p
using assms by (auto dest: terminated-no-commands simp: small-step-stuck elim: system-step.cases)

lemma terminated-stable:
assumes system-step q sh' sh
assumes terminated p sh
shows terminated p sh'
using assms unfolding atLs-def AT-def
by (fastforce split: if-splits prod.splits
dest: small-step-terminated elim!: system-step.cases)

lemma system-step-pls-nonempty:
assumes system-step pls sh' sh
shows pls \neq \{}
using assms by cases simp-all

lemma system-step-no-change:
assumes system-step ps sh' sh
assumes p \notin ps
shows GST sh' p = GST sh p
using assms by cases simp-all

lemma initial-stateD:
assumes initial-state sys s
shows \( AT \ ((GST = s, HST = []) \downarrow) = \) atC \circ PGMs sys \land INIT sys ((\(GST = s, HST = []\)) \downarrow \land (\forall p \ l. \ \neg taken
p \ l \ (GST = s, HST = [])) \)
using assms unfolding initial-state-def split-def o-def LST-def AT-def taken-def by simp

lemma initial-states-initial[iff]:
assumes initial-state sys s
shows at p l ((\(GST = s, HST = []\)) \longleftrightarrow l \in atC \ (PGMs sys p) \)
using assms unfolding initial-state-def split-def AT-def by simp
definition

reachable-state :: ('answer, 'location, 'proc, 'question, 'state, 'ext) pre-system-ext
⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred

where

reachable-state sys s ←→ (∃ σ i. prerun sys σ ∧ σ i = s)

lemma reachable-stateE:

assumes reachable-state sys sh
assumes ⋀ σ i. prerun sys σ ⇒ P (σ i)
shows P sh
using assms unfolding reachable-state-def by blast

lemma prerun-reachable-state:

assumes prerun sys σ
shows reachable-state sys (σ i)
using assms unfolding prerun-def LTL.defs system-step-reflclp-def reachable-state-def by auto

lemma reachable-state-induct[consumes 1, case-names init LocalStep CommunicationStep, induct set: reachable-state]:

assumes r: reachable-state sys sh
assumes i: ⋀ s. initial-state sys s ⇒ P (GST = s, HST = [])
assumes l: ⋀ sh ls' p. [reachable-state sys sh; P sh; GST sh p →τ ls'] ⇒ P (GST = (GST sh)(p := ls'), HST = HST sh)
assumes c: ⋀ sh ls1' ls2' p1 p2 α β.
[reachable-state sys sh; P sh;
 GST sh p1 →้าง (α, β) ls1'; GST sh p2 →้าง (α, β) ls2'; p1 ≠ p2 ]
⇒ P (GST = (GST sh)(p1 := ls1', p2 := ls2'), HST = HST sh @ [(α, β)])

shows P sh
using r
proof (rule reachable-stateE)
fix σ i assume prerun sys σ show P (σ i)
proof (induct i)
  case 0 from ‹prerun sys σ› show ?case
   unfolding prerun-def by (metis (full-types) i old.unit.exhaust system-state.surjective)
next
  case (Suc i) with ‹prerun sys σ› show ?case
   unfolding prerun-def LTL.defs system-step-reflclp-def reachable-state-def
   apply clarsimp
   apply (drule-tac x=i in spec)
   apply (erule disjE; clarsimp)
   apply (erule system-step.cases; clarsimp)
   apply (metis (full-types) ‹prerun sys σ› l old.unit.exhaust prerun-reachable-state system-state.surjective)
   apply (metis (full-types) ‹prerun sys σ› c old.unit.exhaust prerun-reachable-state system-state.surjective)
done
qed
qed

lemma prerun-valid-TrueI:

shows sys |_{pre} (True)
unfolding prerun-valid-def by simp

lemma prerun-valid-conjI:

assumes sys |_{pre} P
assumes sys |_{pre} Q
shows sys |_{pre} P ∧ Q
using assms unfolding prerun-valid-def always-def by simp
lemma valid-prerun-lift:
  assumes sys |\pre I
  shows sys | □[I]
using assms unfolding prerun-valid-def valid-def run-def by blast

lemma prerun-valid-induct:
  assumes σ. prerun sys σ \implies [I] σ
  assumes σ. prerun sys σ \implies ([I] \iff (\bigcirc[I])) σ
  shows sys |\pre I
unfolding prerun-valid-def using assms by (simp add: always-induct)

lemma prerun-validI:
  assumes σ. reachable-state sys s \implies I s
  shows sys |\pre I
unfolding prerun-valid-def using assms by (simp add: alwaysI prerun-reachable-state)

lemma prerun-validE:
  assumes reachable-state sys s
  assumes sys |\pre I
  shows I s
using assms unfolding prerun-valid-def by (metis alwaysE reachable-stateE suffix-state-prop)

5.0.1 Relating reachable states to the initial programs

To usefully reason about the control locations presumably embedded in the single global invariant, we need to link the programs we have in reachable state s to the programs in the initial states. The fragments function decomposes the program into statements that can be directly executed (§??). We also compute the locations we could be at after executing that statement as a function of the process's local state.

Eliding the bodies of IF and WHILE statements yields smaller (but equivalent) proof obligations.

type-synonym ('answer, 'location, 'question, 'state) loc-comp = 'state ⇒ 'location set

fun lconst :: 'location set ⇒ ('answer, 'location, 'question, 'state) loc-comp where
  lconst lp s = lp

definition lcond :: 'location set ⇒ 'location set ⇒ 'state bexp ⇒ ('answer, 'location, 'question, 'state) loc-comp where
  lcond lp lp' b s = (if b s then lp else lp')

lemma lcond-split:
  Q (lcond lp lp' b s) \iff (b s \implies Q lp) \land (\neg b s \implies Q lp')
unfolding lcond-def by (simp split: if-splits)

lemma lcond-split asm:
  Q (lcond lp lp' b s) \iff \neg ((b s \land \neg Q lp) \lor (\neg b s \land \neg Q lp'))
unfolding lcond-def by (simp split: if-splits)

lemmas lcond splits = lcond-split lcond-split asm

fun
  fragments :: ('answer, 'location, 'question, 'state) com ⇒ 'location set
  ⇒ ( ('answer, 'location, 'question, 'state) com × ('answer, 'location, 'question, 'state) loc-comp ) set
where
  fragments (\lif b THEN c FI) aft
lemma fragmentsL-basic-com
| assumes (c', aft') ∈ fragments c aft
| shows basic-com c'
using assms by (induct c arbitrary: aft') (auto intro: basic-com.intros)

lemma fragmentsL-basic-com:
| assumes (c', aft') ∈ fragmentsL cs
shows basic-com c'
using assms
apply (induct cs)
apply simp
apply (case-tac cs)
apply (auto simp: fragments-basic-com)
done

To reason about system transitions we need to identify which basic statement gets executed next. To that end we factor out the recursive cases of the small-step semantics into contexts, which isolate the basic-com commands with immediate externally-visible behaviour. Note that non-determinism means that more than one basic-com can be enabled at a time.

The representation of evaluation contexts follows Berghofer (2012). This style of operational semantics was originated by Felleisen and Hieb (1992).

type-synonym ('answer, 'location, 'question, 'state) ctxt
  = (('answer, 'location, 'question, 'state) com ⇒ ('answer, 'location, 'question, 'state) com)
  × ('answer, 'location, 'question, 'state) com ⇒ ('answer, 'location, 'question, 'state) com list

inductive-set
ctxt :: ('answer, 'location, 'question, 'state) ctxt set

where
C-Hole: (id, {}) ∈ ctxt
| C-Loop: (E, fctxt) ∈ ctxt ⇒ (λc1. LOOP DO E c1 OD, λc1. fctxt c1 @ [LOOP DO E c1 OD]) ∈ ctxt
| C-Seq: (E, fctxt) ∈ ctxt ⇒ (λc1. E c1; c2, λc1. fctxt c1 @ [c2]) ∈ ctxt
| C-Choose1: (E, fctxt) ∈ ctxt ⇒ (λc1. E c1 ⊕ c2, fctxt) ∈ ctxt
| C-Choose2: (E, fctxt) ∈ ctxt ⇒ (λc2. c1 ⊕ E c2, fctxt) ∈ ctxt

We can decompose a small step into a context and a basic-com.

fun
decompose-com :: ('answer, 'location, 'question, 'state) com
  ⇒ (('answer, 'location, 'question, 'state) com)
  × ('answer, 'location, 'question, 'state) ctxt set

where
decompose-com (LOOP DO c1 OD) = { (c, λt. LOOP DO ictxt t OD, λt. fctxt t @ [LOOP DO ictxt t OD]) | c fctxt ictxt. (c, ictxt, fctxt) ∈ decompose-com c1 }
| decompose-com (c1;; c2) = { (c, λt. ictxt t;; c2, λt. fctxt t @ [c2]) | c fctxt ictxt. (c, ictxt, fctxt) ∈ decompose-com c1 }
| decompose-com (c1 ⊕ c2) = { (c, λt. ictxt t ⊕ c2, fctxt) | c fctxt ictxt. (c, ictxt, fctxt) ∈ decompose-com c1 }
  ∪ { (c, λt. c1 ⊕ ictxt t, fctxt) | c fctxt ictxt. (c, ictxt, fctxt) ∈ decompose-com c2 }
| decompose-com c = {((c, id, {})})

definition
decomposeLS :: ('answer, 'location, 'question, 'state) local-state
  ⇒ (('answer, 'location, 'question, 'state) com
  × (('answer, 'location, 'question, 'state) com ⇒ ('answer, 'location, 'question, 'state) com)
  × (('answer, 'location, 'question, 'state) com ⇒ ('answer, 'location, 'question, 'state) com list) ) set

where
decomposeLS s = (case cPGM s of c # - ⇒ decompose-com c | - ⇒ {})

lemma ctxt-inj:
  assumes (E, fctxt) ∈ ctxt
  assumes E x = E y
  shows x = y
  using assms by (induct set: ctxt) auto

lemma decompose-com-non-empty: decompose-com c ≠ {} by (induct c) auto
lemma decompose-com-basic-com:
assumes \((c', \text{ctxts}) \in \text{decompose-com } c\)
shows \(\text{basic-com } c'\)
using \(\text{assms by (induct } c \text{ arbitrary: } c' \text{ ctxts)} \) (auto intro: basic-com.intros)

lemma decomposeLS-basic-com:
assumes \((c', \text{ctxts}) \in \text{decomposeLS } s\)
shows \(\text{basic-com } c'\)
using \(\text{assms unfolding decomposeLS-def by (simp add: decompose-com-basic-com split: list.splits)}\)

lemma decompose-com-ctxt:
assumes \((c', \text{ctxts}) \in \text{decompose-com } c\)
shows \(\text{ctxts} \in \text{ctxt}\)
using \(\text{assms by (induct } c \text{ arbitrary: } c' \text{ ctxts)} \) (auto intro: ctxt.intros)

lemma decompose-com-ictxt:
assumes \((c', \text{ictxt, fctxt}) \in \text{decompose-com } c\)
shows \(\text{ictxt } c = c\)
using \(\text{assms by (induct } c \text{ arbitrary: } c' \text{ ictxt fctxt) auto}\)

lemma decompose-com-small-step:
assumes as: \((c' \# fctxt c' @ cs, s) \Rightarrow s'\)
assumes ds: \((c', \text{ictxt, fctxt}) \in \text{decompose-com } c\)
shows \((c \# cs, s) \Rightarrow s'\)
using \(\text{decompose-com-ctxt[OF ds] as decompose-com-ictxt[OF ds]}\)
by (induct ictxt fctxt arbitrary: c cs)
\(\text{cases s', fastforce simp: fun-eq-iff dest: ctxt-inj} +\)

theorem context-decompose:
\(s \Rightarrow s' \iff (\exists (c, \text{ictxt, fctxt}) \in \text{decomposeLS } s.\)
\(\text{cPGM } s = \text{ictxt } c \# \text{tl } (\text{cPGM } s)\)
\(\land (c \# fctxt c @ \text{tl } (\text{cPGM } s), \text{cTKN } s, \text{cLST } s) \Rightarrow s'\)
\(\land (\forall l \in \text{atC } c. \text{cTKN } s' = \text{Some } l)) \) (is \(?lhs = ?rhs\))

proof (rule iffI)
assume \(?lhs then show \(?rhs\)
unfolding decomposeLS-def
proof (induct rule: small-step.induct)
case (Choose1 c1 cs s \(\alpha\) cs' s' c2) then show \(?case\)
apply clarsimp
apply (rename-tac c ictxt fctxt)
apply (rule-tac x=(c, \(\lambda\) t. ictxt t \(\oplus\) c2, fctxt) in bexI)
apply auto
done
next
case (Choose2 c2 cs s \(\alpha\) cs' s' c1) then show \(?case\)
apply clarsimp
apply (rename-tac c ictxt fctxt)
apply (rule-tac x=(c, \(\lambda\) t. c1 \(\oplus\) ictxt t, fctxt) in bexI)
apply auto
done
qed fastforce+
next
assume \(?rhs then show \(?lhs\)
unfolding decomposeLS-def
by (cases s) (auto split: list.splits dest: decompose-com-small-step)
qed

While we only use this result left-to-right (to decompose a small step into a basic one), this equivalence shows
that we lose no information in doing so.

Decomposing a compound command preserves fragments too.

that we lose no information in doing so.

Decomposing a compound command preserves fragments too.

fun

\[
\text{loc-compC} :: (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ com}
\]

\[
\Rightarrow (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ com list}
\]

\[
\Rightarrow (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ loc-comp}
\]

where

\[
\text{loc-compC} (\boxed{\text{IF } \text{b } \text{THEN } \text{c } \text{Fi}}) \text{ cs} = \text{lcond } (\text{atC c} ) (\text{atCs cs}) \text{ b}
\]

\[
\text{loc-compC} (\boxed{\text{IF } \text{b } \text{THEN } \text{c1 } \text{ELSE } \text{c2 } \text{Fi}}) \text{ cs} = \text{lcond } (\text{atC c1} ) (\text{atC c2}) \text{ b}
\]

\[
\text{loc-compC} (\boxed{\text{LOOP } \text{DO } \text{c } \text{OD}}) \text{ cs} = \text{lconst } (\text{atC c})
\]

\[
\text{loc-compC} (\boxed{\text{WHILE } \text{b } \text{DO } \text{c } \text{OD}}) \text{ cs} = \text{lcond } (\text{atC c} ) (\text{atCs cs}) \text{ b}
\]

\[
\text{loc-compC} \ c = \text{ lconst } (\text{atC c})
\]

\[
\text{loc-compC} (\boxed{\text{IF } \text{b } \text{THEN } \text{c1 } \text{ELSE } \text{c2 } \text{Fi}})
\]

\[
\text{cs} = \text{lcond } (\text{atC c1} ) (\text{atC c2}) \text{ b}
\]

\[
\text{loc-compC } c \text{ cs} = \text{ lconst } (\text{atC c})
\]

\[
\text{lemma decompose-fragments:}
\]

\[
\text{assumes} \ \ (c, \text{ictxt}, \text{fctxt}) \in \text{decompose-com } c0
\]

\[
\text{shows} \ (c, \text{loc-compC } c (\text{fctxt } c @ \text{cs})) \in \text{fragments } c0 \ (\text{atCs cs})
\]

\[
\text{using} \ \ \text{assms}
\]

\[
\text{proof}(\text{induct } c0 \text{ arbitrary: } c \text{ictxt fctxt cs})
\]

\[
\text{case} \ (\text{Loop } c01 \ c \text{ictxt fctxt cs})
\]

\[
\text{from Loops.prems Loops.hyps(1)[where } \text{cs=ictxt c # cs]} \ \text{show} \ ?\text{case by } (\text{auto simp: decompose-com-ictxt})
\]

\[
\text{next}
\]

\[
\text{case} \ (\text{Seq } c01 \ c02 \ c \text{ictxt fctxt cs})
\]

\[
\text{from Seq.prems Seq.hyps(1)[where } \text{cs=c02 # cs]} \ \text{show} \ ?\text{case by auto}
\]

\[
\text{qed} \ \text{auto}
\]

\[
\text{lemma at-decompose:}
\]

\[
\text{assumes} \ (c, \text{ictxt}, \text{fctxt}) \in \text{decompose-com } c0
\]

\[
\text{shows} \ \text{atC c} \subseteq \text{atC c0}
\]

\[
\text{using} \ \text{assms by } (\text{induct } c0 \text{ arbitrary: } c \text{ictxt fctxt; fastforce})
\]

\[
\text{lemma at-decomposeLS:}
\]

\[
\text{assumes} \ (c, \text{ictxt}, \text{fctxt}) \in \text{decomposeLS } s
\]

\[
\text{shows} \ \text{atC c} \subseteq \text{atCs } (\text{cPGM } s)
\]

\[
\text{using} \ \text{assms unfolding } \text{decomposeLS-def by } (\text{auto simp: at-decompose split: list.splits})
\]

\[
\text{lemma decomposeLS-fragmentsLS:}
\]

\[
\text{assumes} \ (c, \text{ictxt}, \text{fctxt}) \in \text{decomposeLS } s
\]

\[
\text{shows} \ (c, \text{loc-compC } c (\text{fctxt } c @ \text{tl (cPGM } s))) \in \text{fragmentsLS } s
\]

\[
\text{using} \ \text{assms}
\]

\[
\text{proof}(\text{cases } \text{cPGM } s)
\]

\[
\text{case} \ (\text{Cons } d \text{ ds})
\]

\[
\text{with} \ \text{assms decompose-fragments[where } \text{cs=ds]} \ \text{show} \ ?\text{thesis}
\]

\[
\text{by} \ (\text{cases } \text{ds}) (\text{auto simp: decomposeLS-def})
\]

\[
\text{qed} \ (\text{simp add: decomposeLS-def})
\]

\[
\text{lemma small-step-loc-compC:}
\]

\[
\text{assumes basic-com } c
\]

\[
\text{assumes} \ (c \# \text{cs}, \text{ls}) \rightarrow \alpha \text{ ls'}
\]

\[
\text{shows} \ \text{loc-compC } c \text{ cs (snd ls)} = \text{atCs } (\text{cPGM } \text{ls'})
\]

\[
\text{using} \ \text{assms by } (\text{fastforce elim: basic-com.cases elim: small-step-inv split: lcond-splits})
\]

The headline result allows us to constrain the initial and final states of a given small step in terms of the original programs, provided the initial state is reachable.

\[
\text{theorem decompose-small-step:}
\]

\[
\text{assumes GST } sh \ p \rightarrow\alpha \text{ ps'}
\]

\[
\text{assumes reachable-state sys } sh
\]

\[
\text{obtains } c \text{ cs aft}
\]
where \((c, \text{aft}) \in \text{fragments (PGMs sys p)}\) \{\}
and \(\text{at} \exists c \subseteq \text{at} (\exists \text{PGM} (\text{GST} \text{ sh p}))\)
and \((\text{cLST} (\text{GST} \text{ sh p})) = \text{at} \exists (\exists \text{PGM} \text{ ps'})\)
and \((c \neq \text{cs}, \text{cTKN} (\text{GST} \text{ sh p}), \text{cLST} (\text{GST} \text{ sh p})) \rightarrow \text{ps'}\)
and \(\forall l\in\text{at} \exists c. \text{cTKN} \text{ ps'} = \text{Some l}\)

using \text{assms}
apply –
apply \((\text{frule iffD1[OF context-decompose]}))\)
apply \text{clarsimp}
apply \((\text{frule decomposeLS-fragmentsLS})\)
apply \((\text{frule at-decomposeLS})\)
apply \((\text{frule (1) subsetD[OF reachable-state-fragmentsLS]}))\)
apply \((\text{frule decomposeLS-basic-com})\)
apply \((\text{frule (1) small-step-loc-compC})\)
apply \text{simp}
done

Reasoning by induction over the reachable states with \text{decompose-small-step} is quite tedious. We provide a very simple VCG that generates friendlier local proof obligations in \S\text{5.1}.

5.1 Simple-minded Hoare Logic/VCG for CIMP

We do not develop a proper Hoare logic or full VCG for CIMP: this machinery merely packages up the subgoals that arise from induction over the reachable states (\S\text{5}). This is somewhat in the spirit of \text{Ridge (2009)}. Note that this approach is not compositional: it consults the original system to find matching communicating pairs, and \text{aft} tracks the labels of possible successor statements. More serious Hoare logics are provided by \text{Cousot and Cousot (1989); Lamport (1980); Lamport and Schneider (1984).}

Intuitively we need to discharge a proof obligation for either \text{Requests} or \text{Responses} but not both. Here we choose to focus on \text{Requests} as we expect to have more local information available about these.

\text{inductive vcg :: \langle\text{′answer, ′location, ′proc, ′question, ′state}\rangle\ \text{programs}\)
⇒ \text{′proc}\)
⇒ \(\langle\text{′answer, ′location, ′question, ′state}\rangle\ \text{loc-comp}\)
⇒ \(\langle\text{′answer, ′location, ′proc, ′question, ′state}\rangle\ \text{state-pred}\)
⇒ \(\langle\text{′answer, ′location, ′question, ′state}\rangle\ \text{com}\)
⇒ \(\langle\text{′answer, ′location, ′proc, ′question, ′state}\rangle\ \text{state-pred}\)
⇒ \text{bool} (∀, ∀, - |- / \{\}_/ / \{\}_ ) \{11,0,0,0,0,0\} 11\)

where
[∀aft′ action′ s ps ′ ps′ l β s′ p′.]
[pre s; \{\{l\}_\} Request action′, aft′) \in \text{fragments (coms p)} \{\}; p \neq p′;
ps′ ∈ \text{val} β (s↓ p); (p′ps′, β) ∈ \text{action′ (action (s↓ p))} (s′↓ p);]
[at p l s; at p′ l′ s;]
[AT s′ = (AT s)(p := aft (s↓ p), p′ := aft′ (s′↓ p′));]
[s′↓ p = s↓ (p := ps′, p′ := p′ps′);
\text{taken p l s};]
[HST s′ = HST s @ [[\text{action (s↓ p)}, β]]];
\forall p'' ∈ \{-\{p,p\}\}. \text{GST s'' s'} = \text{GST s'' p''}
|⇒ \text{post s'}]
⇒ \text{coms, p, aft † \{pre\} \{\{l\}_\} Request action val \{\{post\}\}
[∀s ps s'.]
[pre s; ps′ ∈ f (s↓ p);]
at p l s;
[AT s′ = (AT s)(p := aft (s↓ p));]
[s′↓ p = s↓ (p := ps′);
\text{taken p l s};]
[HST s′ = HST s;
\forall p'' ∈ \{-\{p\}\}. \text{GST s'' s'} = \text{GST s'' p''} ]
We abbreviate invariance with one-sided validity syntax.

**abbreviation** valid-inv (\(\cdot\), \(\cdot\), \(-\) \(\lnot\) / \(\cdot\) / \(-\) \{11,0,0,0,0\} 11) where

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ Response action } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ c1 }::\text{ c2 } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ LOOP DO c OD } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ c1 } \odot\text{ c2 } \{\text{post}\}
\]

We abbreviate invariance with one-sided validity syntax.

**abbreviation** valid-inv (\(\cdot\), \(\cdot\), \(-\) \(\lnot\) / \(\cdot\) / \(-\) \{11,0,0,0,0\} 11) where

\[
\text{coms}, \ p, \ aft \vdash \{I\} \cdot c \equiv \text{coms}, \ p, \ aft \vdash \{I\} \cdot c \{I\}
\]

**inductive-cases** vcg-inv:

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ Request action val } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ LocalOp f } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ IF b THEN t FI } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ IF b THEN t ELSE e FI } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ WHILE b DO c OD } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ LOOP DO c OD } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ Response action } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ c1 }::\text{ c2 } \{\text{post}\}
\]

\[
\text{coms}, \ p, \ aft \vdash \{\text{pre}\} \{l\} \text{ Choose c1 c2 } \{\text{post}\}
\]

We tweak fragments by omitting Responses, yielding fewer obligations

**fun**

\[
\text{vcg-fragments'} :: (\text{'answer}, \text{'location}, \text{'question}, \text{'state}) \text{ com} \\
\Rightarrow \text{'location set}
\]

37
\[
\Rightarrow (\text{('answer', 'location', 'question', 'state) com} \\
\times (\text{('answer', 'location', 'question', 'state) loc-comp } \text{ set})
\]

where

\[
\text{vcg-fragments'} (\text{[]} \text{ Response action}) \text{ aft } = \{\}
\]

\[
\text{vcg-fragments'} (\text{[]} \text{ IF b THEN c FI}) \text{ aft}
\]

\[
= \text{vcg-fragments'} \text{ c aft}
\]

\[
\cup \{ (\text{[]} \text{ IF b THEN c' FI, lcond (atC c) aft b}) | c'. True \}
\]

\[
\text{vcg-fragments'} (\text{[]} \text{ IF b THEN c1 ELSE c2 FI}) \text{ aft}
\]

\[
= \text{vcg-fragments'} \text{ c2 aft } \cup \text{vcg-fragments'} \text{ c1 aft}
\]

\[
\cup \{ (\text{[]} \text{ IF b THEN c1' ELSE c2' FI, lcond (atC c1) (atC c2) b}) | c1' c2'. True \}
\]

\[
\text{vcg-fragments'} (\text{LOOP DO c OD}) \text{ aft } = \text{vcg-fragments'} \text{ c (atC c)}
\]

\[
\text{vcg-fragments'} (\text{c1 :: c2}) \text{ aft } = \text{vcg-fragments'} \text{ c2 aft } \cup \text{vcg-fragments'} \text{ c1 (atC c2)}
\]

\[
\text{vcg-fragments'} (\text{c1 \text{ OD}}) \text{ aft } = \text{vcg-fragments'} \text{ c aft } \cup \text{vcg-fragments'} \text{ c2 aft}
\]

\[
\text{vcg-fragments'} \text{ c aft } = \{(c, \text{lconst aft})\}
\]

abbreviation

\[
\text{vcg-fragments} :: (\text{('answer', 'location', 'question', 'state) com} \\
\Rightarrow (\text{('answer', 'location', 'question', 'state) com})
\times (\text{('answer', 'location', 'question', 'state) loc-comp } \text{ set})
\]

where

\[
\text{vcg-fragments} \text{ c } \equiv \text{vcg-fragments'} \text{ c } \{\}
\]

fun isResponse :: (\text{('answer', 'location', 'question', 'state) com } \Rightarrow \text{ bool}) \text{ where}

\[
\text{isResponse} (\text{[]} \text{ Response action}) \longleftarrow \text{True}
\]

\[
\text{isResponse} - \longleftarrow \text{False}
\]

lemma fragments-vcg-fragments':

\[
\llbracket c, \text{aft} \rrbracket \in \text{fragments c' aft'} \cdot \neg \text{isResponse c} \Rightarrow (c, \text{aft}) \in \text{vcg-fragments'} \text{ c' aft'}
\]

by (induct c' arbitrary: aft') auto

lemma vcg-fragments'-fragments:

\[
\text{vcg-fragments'} \text{ c' aft'} \subseteq \text{fragments c' aft'}
\]

by (induct c' arbitrary: aft') (auto 10 0)

lemma VCG-step:

assumes \( V: \land p. \forall (c, \text{aft}) \in \text{vcg-fragments} (\text{PGMs sys p}). \text{PGMs sys p, aft } \vdash \llbracket \text{pre} \rrbracket \text{ c } \llbracket \text{post} \rrbracket \)

assumes \( S: \text{system-step p sh' sh} \)

assumes \( R: \text{reachable-state sys sh} \)

assumes \( P: \text{pre sh} \)

shows \( \text{post sh'} \)

using \( S \)

proof

\( \begin{cases} \text{case LocalStep with P show } \text{?thesis} \\
\text{apply} - \\
\text{apply (erule decompose-small-step(OF - R))} \\
\text{apply (frule fragments-basic-com)} \\
\text{apply (erule basic-com.cases)} \\
\text{apply (fastforce dest!: fragments-vcg-fragments' V[rule-format] elim: vcg-inv elim!: small-step-inv simp: LST-def AT-def taken-def fun-eq-iff})} \\
\text{done} \end{cases} \)

next

\( \begin{cases} \text{case CommunicationStep with P show } \text{?thesis} \\
\text{apply} - \\
\text{apply (erule decompose-small-step(OF - R))} \\
\text{apply (erule decompose-small-step(OF - R))} \end{cases} \)
subgoal for $c \text{ cs aft} c' \text{ cs' aft'}$
apply (frule fragments-basic-com \{where $c' = c'$\})
apply (frule fragments-basic-com \{where $c = c'$\})
apply (elim basic-com.cases; clarsimp elim!: small-step-inv)
apply (drule fragments-vcg-fragments')
apply (fastforce dest!: V[rule-format]
  elim: vcg-inv elim!: small-step-inv
  simp: LST-def AT-def taken-def fun-eq-iff)+
done
done
done
qed

The user sees the conclusion of $V$ for each element of $\text{vcg-fragments}$.

lemma VCG-step-inv-stable:
  assumes $V$: $\forall p. \forall (c, \text{ aft}) \in \text{vcg-fragments} (\text{PGMs sys p}). \text{PGMs sys, p, aft } \vdash \{ |I| \} c$
  assumes $\text{prerun sys } \sigma$
  shows $(|I| \leftrightarrow \circ |I|) \sigma$
apply (rule alwaysI)
apply clarsimp
apply (rule nextI)
apply clarsimp
using assms(2) unfolding $\text{prerun-def}$
apply clarsimp
apply (erule_tac $i=i$ in alwaysE)
unfolding system-step-refclp-def
apply clarsimp
apply (erule disjE; clarsimp)
using VCG-step[where $\text{pre}=I$ and $\text{post}=I$] $V$ assms(2) $\text{prerun-reachable-state}$
apply blast
done

lemma VCG:
  assumes $I$: $\forall s. \text{initial-state sys s } \rightarrow I (\emptyset \text{ GST = s, HST = []})$
  assumes $V$: $\forall p. \forall (c, \text{ aft}) \in \text{vcg-fragments} (\text{PGMs sys p}). \text{PGMs sys, p, aft } \vdash \{ |I| \} c$
  shows $\text{sys } \vdash_{\text{pre}} I$
apply (rule prerun-valid-induct)
apply (clarsimp simp: $\text{prerun-def state-prop-def}$)
apply (metis (full-types) $I$ old.unit.exhaust system-state.surjective)
using VCG-step-inv-stable[OF $V$] apply blast
done

lemmas VCG-valid = valid-prerun-lift[OF VCG, of sys $I$] for sys $I$

5.1.1 VCG rules

We can develop some (but not all) of the familiar Hoare rules; see Lamport (1980) and the seL4/l4.verified lemma buckets for inspiration. We avoid many of the issues Lamport mentions as we only treat basic (atomic) commands.

context
fixes coms :: ('answer, 'location, 'proc, 'question, 'state) programs
fixes p :: 'proc
fixes aft :: ('answer, 'location, 'question, 'state) loc-comp
begin

abbreviation valid-syn :: ('answer, 'location, 'proc, 'question, 'state) state-pred
  \Rightarrow ('answer, 'location, 'question, 'state) com
  \Rightarrow ('answer, 'location, 'proc, 'question, 'state) state-pred \Rightarrow bool where
valid-syn $P \ c \ Q \equiv \text{coms, p, aft } \vdash \{ P \} c \{ Q \}$
notation valid-syn (\{\cdot\}/ -/ \{\cdot\})

abbreviation
valid-inv-syn :: (answer, location, proc, question, state) state-pred
⇒ (answer, location, question, state) com ⇒ bool
where
valid-inv-syn P c ≡ \{P\} c \{\{P\}\}

notation valid-inv-syn (\{\cdot\}/ -)

lemma vcg-True:
\{P\} c \{(True)\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemma vcg-conj:
\[ \forall s. P s \Longrightarrow Q s; \{P\} c \{Q\} \] \Longrightarrow \{P\} c \{\wedge R\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemma vcg-pre-imp:
\[ \forall s. P s \Longrightarrow Q s; \{P\} c \{Q\} \] \Longrightarrow \{P\} c \{R\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemmas vcg-pre = vcg-pre-imp[rotated]

lemma vcg-post-imp:
\[ \forall s. Q s \Longrightarrow R s; \{P\} c \{Q\} \] \Longrightarrow \{P\} c \{R\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemma vcg-prop[intro]:
\{(P)\} c
by (cases c) (fastforce intro: vcg.intros)+

lemma vcg-drop-imp:
assumes \{P\} c \{Q\}
sows \{P\} c \{R \Longrightarrow Q\}
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemma vcg-conj-lift:
assumes x: \{P\} c \{Q\}
assumes y: \{P\} c \{Q\}
sows \{P \wedge P\} c \{Q \wedge Q\}
apply (rule vcg-conj)
apply (rule vcg-pre[OF x], simp)
apply (rule vcg-pre[OF y], simp)
done

lemma vcg-disj-lift:
assumes x: \{P\} c \{Q\}
assumes y: \{P\} c \{Q\}
sows \{P \vee P\} c \{Q \vee Q\}
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemma vcg-imp-lift:
assumes \{P\} c \{\neg P\}
assumes \{Q\} c \{Q\}
sows \{P \vee Q\} c \{P \Longrightarrow Q\}
by (simp only: imp-conv-disj vcg-disj-lift[OF assms])
lemma vcg-ex-lift:
assumes $\forall x. \{ P x \} \ c \ \{ Q x \}$
shows $\{ \lambda s. \exists x. P x s \} \ c \ \{ \lambda s. \exists x. Q x s \}$
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-all-lift:
assumes $\forall x. \{ P x \} \ c \ \{ Q x \}$
shows $\{ \lambda s. \exists x. P x s \} \ c \ \{ \lambda s. \exists x. Q x s \}$
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-name-pre-state:
assumes $\forall s. P s = \Rightarrow \{ (=) s \} \ c \ \{ Q \}$
shows $\{ P \} \ c \ \{ Q \}$
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-lift-comp:
assumes $f: \forall P. \{ \lambda s. P (f s :: 'a :: type) \} \ c$
assumes $P: \forall x. \{ Q x \} \ c \ \{ P x \}$
shows $\{ \lambda s. Q (f s) s \} \ c \ \{ \lambda s. P (f s) s \}$
apply (rule vcg-name-pre-state)
apply (rename-tac s)
apply (rule vcg-pre)
apply (rule vcg-post-imp[rotated])
apply (rule vcg-conj-lift)
apply (rule-tac x=f s in P)
apply (rule-tac P=\lambda fs. fs = f s in f)
apply simp
apply simp
done

5.1.2 Cheap non-interference rules

These rules magically construct VCG lifting rules from the easier to prove eq-imp facts. We don’t actually use these in the GC, but we do derive fun-upd equations using the same mechanism. Thanks to Thomas Sewell for the requisite syntax magic.

As these eq-imp facts do not usefully compose, we make the definition asymmetric (i.e., $g$ does not get a bundle of parameters).

Note that these are effectively parametricity rules.

definition eq-imp :: ('a ⇒ 'b ⇒ 'c) ⇒ ('b ⇒ 'e) ⇒ bool where
eq-imp $f \ g \equiv (\forall s s'. (\forall x. f x s = f x s') \Rightarrow (g s = g s'))$

lemma eq-impD:
[ eq-imp $f \ g; \forall x. f x s = f x s' \] \Rightarrow g s = g s'
by (simp add: eq-imp-def)

lemma eq-imp-vcg:
assumes $g: \text{eq-imp} \ f \ g$
assumes $f: \forall x P. \{ P \circ (f x) \} \ c$
shows $\{ P \circ g \} \ c$
apply (rule vcg-name-pre-state)
apply (rename-tac s)
apply (rule vcg-pre)
apply (rule vcg-post-imp[rotated])
apply (rule vcg-all-lift[where 'a='a])
apply \( \text{rule-tac } x=x \) and \( P=\lambda f.s. \; f \; x \; s \; \text{in } f[\text{rule-format}] \)
apply simp
apply (frule eq-impD[where \( f=f \), \( \text{OF } g \)])
apply simp
apply simp
done

lemma eq-imp-vcg-LST:
assumes \( g: \text{eq-imp } f \; g \)
assumes \( f: \forall \; x \; P. \; (P \circ (f \; x) \circ \text{LST}) \; c \)
shows \( (P \circ g \circ \text{LST}) \; c \)
apply (rule vcg-name-pre-state)
apply (rename-tac s)
apply (rule vcg-pre)
apply (rule vcg-post-imp[rotated])
apply (rule vcg-all-lift[where \( \text{\'a=\'a} \)])
apply (rule-tac \( x=x \) and \( P=\lambda f.s. \; f \; x \; s \) \; \text{in } f[\text{rule-format}] \))
apply simp
apply simp
apply simp
done

lemma eq-imp-fun-upd:
assumes \( g: \text{eq-imp } f \; g \)
assumes \( f: \forall \; x. \; f \; x \; (s(fld := \text{val})) = f \; x \; s \)
shows \( g \; s(fld := \text{val}) = g \; s \)
apply (rule eq-impD[\( \text{OF } g \)])
apply (rule f)
done

lemma curry-forall-eq:
\( (\forall \; f. \; P \; f) = (\forall \; f. \; P \; (\text{case-prod } f)) \)
by (metis case-prod-curry)

lemma pres-tuple-vcg:
\( (\forall \; P. \; (P \circ (\lambda s. \; (f \; s \; g \; s))) \; c) \)
\( \iff \; ((\forall \; P. \; (P \circ f \; c) \land (\forall \; P. \; (P \circ g \; c)c)) \)
apply (simp add: curry-forall-eq o-def)
apply safe
apply fast
apply fast
apply (rename-tac P)
apply (rule-tac \( f=f \) and \( P=\lambda f.s. \; P \; f.s \; (g \; s) \) \; \text{in } \text{vcg-lift-comp; simp})
done

lemma pres-tuple-vcg-LST:
\( (\forall \; P. \; (P \circ (\lambda s. \; (f \; s \; g \; s)) \circ \text{LST}) \; c) \)
\( \iff \; ((\forall \; P. \; (P \circ f \; \text{LST} \; c) \land (\forall \; P. \; (P \circ g \; \text{LST} \; c)) \)
apply (simp add: curry-forall-eq o-def)
apply safe
apply fast
apply fast
apply (rename-tac P)
apply (rule-tac \( f=\lambda s. \; f \; s \) ↓ and \( P=\lambda f.s. \; P \; f.s \; (g \; s) \) \; \text{for } g \; \text{in } \text{vcg-lift-comp; simp})
done

no-notation valid-syn \( (\{\_\} / \; -/ \; \{\_\}) \)

42
6 One locale per process

A sketch of what we’re doing in ConcurrentGC, for quicker testing. FIXME write some lemmas that further exercise the generated thms.

locale P1
begin

definition com :: (unit, string, unit, nat) com where
com = {\"A\"} WHILE (\(<\) 0) DO {\"B\"} \[\lambda s \rightarrow s - 1\] OD

intern-com com-def
print-theorems

locset-definition loop = \{B\}
print-theorems

thm locset-cache

definition assertion = atS False loop

end

thm locset-cache
locale P2
begin

thm locset-cache

definition com :: (unit, string, unit, nat) com where
com = {\"C\"} WHILE (\(<\) 0) DO {\"A\"} \[Suc\] OD

intern-com com-def
locset-definition loop = \{A\}
print-theorems

end

thm locset-cache

primrec coms :: bool ⇒ (unit, string, unit, nat) com where
coms False = P1.com
| coms True = P2.com

7 Example: a one-place buffer

To demonstrate the CIMP reasoning infrastructure, we treat the trivial one-place buffer example of Lamport and Schneider (1984, §3.3). Note that the semantics for our language is different to Lamport and Schneider’s, who treated a historical variant of CSP (i.e., not the one in Hoare (1985)).

We introduce some syntax for fixed-topology (static channel-based) scenarios.

abbreviation
rcv-syn :: 'location ⇒ 'channel ⇒ ('val ⇒ 'state ⇒ 'state)
The following adapts Kai Engelhardt’s, from his notes titled Proving an Asynchronous Message Passing Program Correct, 2011. The history variable tracks the causality of the system, which I feel is missing in Lamport’s treatment. We tack on Lamport’s invariant so we can establish Etern-pred.

These definitions largely follow Lamport and Schneider (1984). We have three processes communicating over two channels. We enumerate program locations.

datatype ex-chname = ξ12 | ξ23
type-synonym ex-val = nat
type-synonym ex-ch = ex-chname × ex-val
datatype ex-loc = r12 | r23 | s23 | s12
datatype ex-proc = p1 | p2 | p3

type-synonym ex-pgm = (unit, ex-loc, ex-ch, ex-val) com
type-synonym ex-pred = (unit, ex-loc, ex-proc, ex-ch, ex-val) state-pred
type-synonym ex-state = (unit, ex-loc, ex-proc, ex-ch, ex-val) system-state
type-synonym ex-sys = (unit, ex-loc, ex-proc, ex-ch, ex-val) system
type-synonym ex-history = (ex-ch × unit) list

We further specialise these for our particular example.

primrec

ex-coms :: ex-proc ⇒ ex-pgm

where

ex-coms p1 = \{s12\} ξ12<id
| ex-coms p2 = LOOP DO \{r12\} ξ12<<(λv -. v) ;; \{s23\} ξ23<id OD
| ex-coms p3 = \{r23\} ξ23<<(λv -. v)

Each process starts with an arbitrary initial local state.

abbreviation ex-init :: (ex-proc ⇒ ex-val) ⇒ bool where

ex-init ≡ (True)

abbreviation sys :: ex-sys where

sys ≡ \{PGMs = ex-coms, INIT = ex-init, FAIR = (True)\}

The following adapts Kai Engelhardt’s, from his notes titled Proving an Asynchronous Message Passing Program Correct, 2011. The history variable tracks the causality of the system, which I feel is missing in Lamport’s treatment. We tack on Lamport’s invariant so we can establish Etern-pred.

abbreviation

filter-on-channel :: ex-chname ⇒ ex-state ⇒ ex-val list (|- [100] 101)

where

|ch ≡ map (snd ∘ fst) ∘ filter (|= ch ∘ fst ∘ fst) ∘ HST

definition IL :: ex-pred where

IL = pred-conjoin [ at p1 s12 → LIST-NULL |ξ12 , terminated p1 → |ξ12 = (λs. [s↓ p1]) , at p2 r12 → |ξ12 = |ξ23 , at p2 s23 → |ξ12 = |ξ23 @ (λs. [s↓ p2]) ∧ (λs. s↓ p1 = s↓ p2) , at p3 r23 → LIST-NULL |ξ23 , terminated p3 → |ξ23 = (λs. [s↓ p2]) ∧ (λs. s↓ p1 = s↓ p3) ]

If p3 terminates, then it has p1’s value. This is stronger than Lamport and Schneider’s as we don’t ask that the first process has also terminated.
definition Etern-pred :: ex-pred where
Etern-pred = (terminated p3 →→ (λs. s↓ p1 = s↓ p3))

Proofs from here down.

lemma correct-system:
  assumes IL sh
  shows Etern-pred sh
using assms unfolding Etern-pred-def IL-def by simp

lemma IL-p1: ex-coms, p1, lconst {} ⊢ {IL} {s12} ξ12◁(λs. s)
apply (rule vcg.intros)
apply (rename-tac p')
apply (case-tac p'; clarsimp simp: IL-def atLs-def)
done

lemma IL-p2: ex-coms, p2, lconst {r12} ⊢ {IL} {s23} ξ23◁(λs. s)
apply (rule vcg.intros)
apply (rename-tac p')
apply (case-tac p'; clarsimp simp: IL-def)
done

lemma IL: sys |=_{pre} IL
apply (rule VCG)
apply (clarsimp simp: IL-def atLs-def dest!: initial-stateD)
apply (rename-tac p)
apply (case-tac p; clarsimp simp: IL-p1 IL-p2)
done

lemma IL-valid: sys |= □[IL]
by (rule valid-prerun-lift[OF IL])

8 Example: an unbounded buffer

This is more literally Kai Engelhardt’s example from his notes titled Proving an Asynchronous Message Passing Program Correct, 2011.

datatype ex-chname = ξ12 | ξ23
type-synonym ex-val = nat
type-synonym ex-ls = ex-val list
type-synonym ex-ch = ex-chname × ex-val
datatype ex-loc = c1 | r12 | r23 | s23 | s12
datatype ex-proc = p1 | p2 | p3
type-synonym ex-pgm = (unit, ex-loc, ex-ch, ex-ls) com
type-synonym ex-pred = (unit, ex-loc, ex-proc, ex-ch, ex-ls) state-pred
type-synonym ex-state = (unit, ex-loc, ex-proc, ex-ch, ex-ls) state-pred
type-synonym ex-sys = (unit, ex-loc, ex-proc, ex-ch, ex-ls) system-state
type-synonym ex-history = (ex-ch × unit) list

The local state for the producer process contains all values produced; consider that ghost state.

abbreviation (input) snoc :: 'a ⇒ 'a list ⇒ 'a list where snoc x xs ≡ xs @ [x]

primrec ex-coms :: ex-proc ⇒ ex-pgm where
  ex-coms p1 = LOOP DO {c1} LocalOp (λxs. {snoc x xs | x. True}) ;; {s12} ξ12◁(last, id) OD
  | ex-coms p2 = LOOP DO {r12} ξ12▷snoc
                 ⊕ {c1} IF (λs. length s > 0) THEN {s23} ξ12▷(hd, tl) FI
   OD
ex-coms p3 = LOOP DO \( r_{23} \) \( \xi_{23} \) snoc OD

abbreviation ex-init :: (ex-proc ⇒ ex-ls) ⇒ bool where
ex-init s ≡ ∀ p. s p = []

abbreviation sys :: ex-sys where
sys ≡ \{ PGMs = ex-coms, INIT = ex-init, FAIR = \{True\} \}

abbreviation
filter-on-channel :: ex-chname ⇒ ex-state ⇒ ex-val list (⇂ - [100] 101) where
\( ch ≡ \text{map}(\text{snd} ◦ \text{fst}) ◦ \text{filter}(\text{=} ch ◦ \text{fst} ◦ \text{fst}) ◦ \text{HST} \)

definition I-pred :: ex-pred where
I-pred = \text{pred-conjoin}\[
\text{at } p1 \text{ c1} → |\xi_{12} = (\lambda s. s \downarrow p1) \n\text{, at } p1 \text{ s12} → (\lambda s. \text{length}(s \downarrow p1) > 0 \wedge \text{butlast}(s \downarrow p1)) = (|\xi_{12}| s) \n\text{, } |\xi_{12} ≤ (\lambda s. s \downarrow p1) \n\text{, } |\xi_{12} = |\xi_{23} @ (\lambda s. s \downarrow p2) \n\text{, at } p2 \text{ s23} → (\lambda s. \text{length}(s \downarrow p2) > 0) \n\text{, } (\lambda s. s \downarrow p3) = |\xi_{23} \]
\]
The local state of \( p3 \) is some prefix of the local state of \( p1 \).

definition Etern-pred :: ex-pred where
Etern-pred ≡ \( \lambda s. s \downarrow p3 ≤ s \downarrow p1 \)

lemma correct-system:
assumes I-pred s
shows Etern-pred s
using asms unfolding Etern-def I-pred-def less-eq-list-def prefix-def by clarsimp

lemma p1-c1 [simplified, intro]:
ex-coms, p1, lconst \{s12\} ⊢ \{I-pred\} \{c1\} \text{LocalOp}(\lambda xs. \{ \text{snoct xs} | x. True \})
apply (rule vcg.intros)
apply (clarsimp simp: I-pred-def atS-def)
done

lemma p1-s12 [simplified, intro]:
ex-coms, p1, lconst \{c1\} ⊢ \{I-pred\} \{s12\} \text{\xi}_{12}@(\text{last, id})
apply (rule vcg.intros)
apply (rename-tac p')
apply (case-tac p'; clarsimp)
apply (clarsimp simp: I-pred-def atS-def)
apply (clarsimp simp: I-pred-def atS-def)
apply (clarsimp simp: I-pred-def atS-def)
done

lemma p2-s23 [simplified, intro]:
ex-coms, p2, lconst \{c1, r12\} ⊢ \{I-pred\} \{s23\} \text{\xi}_{12}@(\text{hd, tl})
apply (rule vcg.intros)
apply (rename-tac p')
apply (case-tac p'; clarsimp)
apply (clarsimp simp: I-pred-def atS-def)
done

lemma p2-pi4 [intro]:
ex-coms, p2, lcond \{s23\} \{c1, r12\} (\lambda s. \text{s} ≠ []) ⊢ \{I-pred\} \{c1\} \text{IF}(\lambda s. \text{s} ≠ []) \text{THEN} c' \text{ FI}
apply (rule vcg.intros)
apply (clarsimp simp: I-pred-def atS-def split: lcond-splits)
done
lemma \(I\): \(\text{sys} \models_{\text{pre}} \text{I-pred}\)
apply (rule VCG)
apply (clarsimp dest!: initial-stateD simp: I-pred-def atS-def)
apply (rename-tac p)
apply (case-tac p; auto)
done

lemma \(I\)-valid: \(\text{sys} \models \Box [\text{I-pred}]\)
by (rule valid-prerun-lift[OF I])

9 Concluding remarks

Previously Nipkow and Prensa Nieto (1999); Prensa Nieto (2002, 2003)\(^3\) have developed the classical Owicky/Gries and Rely-Guarantee paradigms for the verification of shared-variable concurrent programs in Isabelle/HOL. These have been used to show the correctness of a garbage collector (Prensa Nieto and Esparza 2000).

We instead use synchronous message passing, which is significantly less explored. de Boer, de Roever, and Hannemann (1999); ? provide compositional systems for \textit{terminating} systems. We have instead adopted Lamport’s paradigm of a single global invariant and local proof obligations as the systems we have in mind are tightly coupled and it is not obvious that the proofs would be easier on a decomposed system; see ?, §1.6.6 for a concurring opinion. Unlike the generic sequential program verification framework Simpl (Schirmer 2004), we do not support function calls, or a sophisticated account of state spaces. Moreover we do no meta-theory beyond showing the simple VCG is sound (§5.1).

References


\(^3\)The theories are in \$ISABELLE/src/HOL/Hoare_Parallel.


