CIMP

Peter Gammie

September 13, 2023

Abstract

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

Contents

1 Point-free notation 1

2 Infinite Sequences 3
  2.1 Decomposing safety and liveness .................................................. 6

3 Linear Temporal Logic 9
  3.1 Leads-to and leads-to-via ............................................................. 15
  3.2 Fairness ................................................................................. 18
  3.3 Safety and liveness ................................................................. 20

4 CIMP syntax and semantics 22
  4.1 Syntax ................................................................................. 22
  4.2 Process semantics ................................................................. 23
  4.3 System steps ......................................................................... 24
  4.4 Control predicates ............................................................... 25

5 State-based invariants 27
  5.0.1 Relating reachable states to the initial programs ................................. 31
  5.1 Simple-minded Hoare Logic/VCG for CIMP ............................................. 36
    5.1.1 VCG rules ....................................................................... 39
    5.1.2 Cheap non-interference rules ..................................................... 41

6 One locale per process 42

7 Example: a one-place buffer 43

8 Example: an unbounded buffer 45

9 Concluding remarks 47

References 49

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Infinite sequences and some operations on them.

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
sigma |s i ≡ λj. sigma (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 i)

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

lemma suffix-eval: (sigma |s i) j = sigma (j + i)
unfolding suffix-def by simp

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

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

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

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

lemma comp-suffix: f σ |s i = (f σ) |s i
unfolding suffix-def 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:
\[\text{stake } i \ (w @- \sigma) = \text{take } i \ w @ \text{stake } (i - \text{length } w) \ \sigma\]

by \ ((\text{induct } i \ \text{arbitrary}: \ w) \ (\text{auto simp: neq-Nil-conv})

**Lemma** shift-snth-less[simp]:
- assumes \(i < \text{length } xs\)
- shows \((xs @- \sigma) \ i = xs \ ! i\)

using \ assms

proof\((\text{induct } i \ \text{arbitrary}: \ xs)\)
  case \(\text{Suc } i \ xs\) then show \ ?case \ by \ (\text{cases } xs) \ \text{simp-all}
qed \ (\text{simp add: hd-conv-nth nth-tl})

**Lemma** shift-snth-ge[simp]:
- assumes \(i \geq \text{length } xs\)
- shows \((xs @- \sigma) \ i = \sigma \ (i - \text{length } xs)\)

using \ assms

proof\((\text{induct } i \ \text{arbitrary}: \ xs)\)
  case \(\text{Suc } i \ xs\) then show \ ?case \ by \ (\text{cases } xs) \ \text{simp-all}
qed \ simp

**Lemma** shift-snth:
\((xs @- \sigma) \ i = (\text{if } i < \text{length } xs \ \text{then } xs \ ! i \ \text{else } \sigma \ (i - \text{length } xs))\)

by \ simp

**Lemma** suffix-shift:
\((xs @- \sigma) \ |_s \ i = \text{drop } i \ xs @- \ (\sigma |_s \ i - \text{length } xs)\)

proof\((\text{induct } i \ \text{arbitrary}: \ xs)\)
  case \(\text{Suc } i \ xs\) then show \ ?case \ by \ (\text{cases } xs) \ \text{simp-all}
qed \ simp

**Lemma** stake-nth[simp]:
- assumes \(i < j\)
- shows \text{stake } j \ s \ ! i = s \ i\)

using \ assms \ by \ ((\text{induct } j \ \text{arbitrary}: \ s \ i) \ (\text{simp-all add: nth-Cons}')

**Lemma** stake-suffix-id:
\(\text{stake } i \ \sigma @- \ (\sigma |_s \ i) = \sigma\)

by \ ((\text{induct } i) \ (\text{simp-all add: fun-eq-iff shift-snth split: nat.splits})

**Lemma** id-stake-suffix-suffix:
\(\sigma = (\text{stake } i \ \sigma @ \ [\sigma |_s i]) @- \ (\sigma |_s \ \text{Suc } i)\)

using \ stake-suffix-id

apply \ (\text{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]:
\(\text{stake } i \ \sigma @ \ \text{stake } j \ (\sigma |_s \ i) = \text{stake } (i + j) \ \sigma\)

apply \ ((\text{induct } i \ \text{arbitrary}: \ \sigma) \ \text{simp})
apply \ auto
apply \ (\text{metis One-nat-def plus-1 eq Suc suffix-plus-com})
done

**Lemma** stake-append: \(\text{stake } n \ (u @- s) = \text{take } (\text{min } (\text{length } u) \ n) \ u @ \ \text{stake } (n - \text{length } u) \ s\)

proof \ ((\text{induct } n \ \text{arbitrary}: \ u)\)
  case \(\text{Suc } n\) then show \ ?case
    apply \ clarsimp
    apply \ (\text{cases } u)
  done
apply auto
done
qed auto

lemma stake-shift-stake-shift:
  stake i σ @− stake j (σ |_ 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 |_[ 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 |_[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 (λσ. ∀ i. (σ i, σ (i + 1)) ∈ 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 ∧ Q)
using assms unfolding safety-def by blast

lemma safety-always-eventually[simplified]:
  assumes safety P
  assumes ∀ i. ∃ j≥i. ∃ β. P (σ(0 → j) ⊢ β)
  shows P σ
using assms unfolding safety-def2
apply
apply (drule-tac x=σ in spec)
apply clarsimp
apply (drule-tac x=i in spec)
apply clarsimp
apply (rule-tac x=(stake j σ ⊢ β) | 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 ∨ 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 ⇒ 'a seq-pred where
  M_p P = (λσ. ∀ i. ∃ β. P (stake i σ ⊢ β))

definition Safe :: 'a seq-pred ⇒ 'a seq-pred where
  Safe P = (P ∨ M_p P)

definition Live :: 'a seq-pred ⇒ 'a seq-pred where
  Live P = (P ∨ ¬M_p P)

lemma decomp:
  P = (Safe P ∧ 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:
  liveness (Live P)
proof (rule livenessI)
  fix α
  have (∃ β. P (α ⊕− β)) ∨ ¬(∃ β. P (α ⊕− β)) by blast
  also have ?this ↔ (∃ β. P (α ⊕− β)) ∨ (∀ γ. ¬P (α ⊕− γ)) by blast
  also have . . . ↔ (∃ β. P (α ⊕− β) ∨ (∃ i. i = length α ∧ (∀ γ. ¬P (stake i (α ⊕− β) ⊕− γ)))) by (simp add: stake-shift)
  finally have . . . ←→ (∃ β. P (α ⊕− β) ∨ (∃ i. i = length α ∧ (∀ γ. ¬P (stake i (α ⊕− β) ⊕− γ)))) by blast
  then show ∃ σ. Live P (α ⊕− σ)
qed

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

definition absolute-liveness :: 'a seq-pred ⇒ bool where — closed under prepending any finite sequence
  absolute-liveness P ←→ (∃ σ. P σ) ∧ (∀ σ α. P σ −→ P (α @− σ))

definition stable :: 'a seq-pred ⇒ bool where — closed under suffixes
  stable P ←→ (∃ σ. P σ) ∧ (∃ i. P σ −→ P (σ @− s i))

lemma absolute-liveness-liveness:
  assumes absolute-liveness P
  shows liveness P
using assms unfolding absolute-liveness-def liveness-def by blast

lemma stable-absolute-liveness:
  assumes P σ
  assumes ¬P σ' — extra hypothesis
  shows stable P ←→ absolute-liveness (¬ 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

definition fairness :: 'a seq-pred ⇒ bool where
  fairness P ←→ stable P ∧ absolute-liveness P

lemma fairness-safety:
  assumes safety P
  assumes fairness F
  shows (∀ σ. F σ −→ P σ) ←→ (∀ σ. P σ)
apply rule
using assms
apply clarsimp
unfolding safety-def fairness-def stable-def absolute-liveness-def
apply clarsimp
apply blast+
done
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.

\[
\text{definition state-prop :: } (\text{'a }\Rightarrow\text{ bool}) \Rightarrow \text{'a seq-pred }[[\cdot]]\text{ where} \\
\lceil P \rceil = (\lambda \sigma. \ P (\sigma \ 0))
\]

\[
\text{definition next :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\circ \ 80 \ 80]]\text{ where} \\
(\circ P) = (\lambda \sigma. \ P (\sigma \ |s \ 1))
\]

\[
\text{definition always :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\square \ 80 \ 80]]\text{ where} \\
(\square P) = (\lambda \sigma. \ \forall i. \ P (\sigma \ |s \ i))
\]

\[
\text{definition until :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\text{infixr } U \ 30]]\text{ where} \\
(P \ U \ Q) = (\lambda \sigma. \ \exists i. \ Q (\sigma \ |s \ i) \land (\forall k<i. \ P (\sigma \ |s \ k)))
\]

\[
\text{definition eventually :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\diamond \ 80 \ 80]]\text{ where} \\
(\diamond P) = (\langle \text{True} \rangle \ U \ P)
\]

\[
\text{definition release :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\text{infixr } R \ 30]]\text{ where} \\
(P \ R \ Q) = (\neg (\neg P \ U \ \neg Q))
\]

\[
\text{definition unless :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\text{infixr } W \ 30]]\text{ where} \\
(P \ W \ Q) = ((P \ U \ Q) \lor \square P)
\]

\[
\text{abbreviation (input)} \\
\text{pred-always-imp-syn :: } \text{'a seq-pred }\Rightarrow\text{'a seq-pred }\Rightarrow\text{'a seq-pred }[[\text{infixr } \hookrightarrow \ 25]]\text{ where} \\
P \hookrightarrow Q \equiv \square (P \rightarrow Q)
\]

\text{lemmas def s =} \\
\text{state-prop-def} \\
\text{always-def} \\
\text{eventually-def} \\
\text{next-def} \\
\text{release-def} \\
\text{unless-def} \\
\text{until-def}

\[
\text{lemma suffix-state-prop[simp]}:\ \\
\text{shows } [P] (\sigma \ |s \ i) = P (\sigma \ i)
\]

\text{unfolding def s by simp}

\[
\text{lemma alwaysI[intro]}:\ \\
\text{assumes } \land i. \ P (\sigma \ |s \ i) \\
\text{shows } (\square P) \ \sigma
\]

\text{unfolding def s using assms by blast}

\[
\text{lemma alwaysD}: \\
\text{assumes } (\square P) \ \sigma \\
\text{shows } P (\sigma \ |s \ i)
\]

\text{using assms unfolding def s by blast}
lemma alwaysE: \( [(\Box P) \sigma; P (\sigma \mid_s i) \Rightarrow Q] \Rightarrow Q \)
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 (\( \sigma \mid_s i \))
    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 (2 P) (f \( \circ \sigma \)) \leftrightarrow ((\lambda \sigma. P (f \circ \sigma))) \( \sigma \)
unfolding defs by simp-all

lemma nextI[intro]:
  assumes P (\( \sigma \mid_s Suc 0 \))
  shows (\( \Diamond P \)) \( \sigma \)
using assms unfolding defs by simp

lemma untilI[intro]:
  assumes Q (\( \sigma \mid_s i \))
  assumes \( \forall k < i. P (\sigma \mid_s k) \)
  shows (P U Q) \( \sigma \)
unfolding defs using assms by blast

lemma untilE:
  assumes (P U Q) \( \sigma \)
  obtains \( i \) where Q (\( \sigma \mid_s i \)) and \( \forall k < i. P (\sigma \mid_s k) \)
using assms unfolding until-def by blast

lemma eventuallyI[intro]:
  assumes P (\( \sigma \mid_s i \))
  shows (\( \Diamond P \)) \( \sigma \)
unfolding eventually-def using assms by blast

lemma eventuallyE[elim]:
  assumes (\( \Diamond P \)) \( \sigma \)
  obtains \( i \) where P (\( \sigma \mid_s i \))
using assms unfolding defs by (blast elim: untilE)

lemma unless-alwaysI:
  assumes (\( \Box P \)) \( \sigma \)
  shows (P W Q) \( \sigma \)
using assms unfolding defs by blast

lemma unless-untilI:
  assumes Q (\( \sigma \mid_s j \))
  assumes \( \forall i. i < j \Rightarrow P (\sigma \mid_s i) \)
shows $(P \mathbin{W} Q) \sigma$

unfolding defs using assms by blast

lemma always-imp-refl iff:
shows $(P \leftrightarrow P) \sigma$

unfolding defs by blast

lemma always-imp-trans:
assumes $(P \leftrightarrow Q) \sigma$
assumes $(Q \leftrightarrow R) \sigma$
shows $(P \leftrightarrow R) \sigma$
using assms unfolding defs by blast

lemma always-imp-mp:
assumes $(P \leftrightarrow Q) \sigma$
assumes $P \sigma$
shows $Q \sigma$
using assms unfolding defs by (metis suffix-zero)

lemma always-imp-mp-suffix:
assumes $(P \leftrightarrow Q) \sigma$
assumes $P \sigma (\sigma \mid s \ i)$
shows $Q (\sigma \mid s \ i)$
using assms unfolding defs by metis

Some basic facts and equivalences, mostly sanity.

lemma necessitation:
$\\langle \forall s. P s \rangle \implies (\Box P) \sigma$
$\\langle \forall s. P s \rangle \implies (\Diamond P) \sigma$
$\\langle \forall s. P s \rangle \implies (P \mathbin{W} Q) \sigma$
$\\langle \forall s. Q s \rangle \implies (P \mathbin{U} Q) \sigma$

unfolding defs by auto

lemma cong:
$\\langle \forall s. P s = P' s \rangle \implies [P] = [P']$
$\\langle \forall \sigma. P \sigma = P' \sigma \rangle \implies (\Box P) = (\Box P')$
$\\langle \forall \sigma. P \sigma = P' \sigma \rangle \implies (\Diamond P) = (\Diamond P')$
$\\langle \forall \sigma. P \sigma = P' \sigma : \\forall \sigma. Q \sigma = Q' \sigma \rangle \implies (P \mathbin{U} Q) = (P' \mathbin{U} Q')$
$\\langle \forall \sigma. P \sigma = P' \sigma : \\forall \sigma. Q \sigma = Q' \sigma \rangle \implies (P \mathbin{W} Q) = (P' \mathbin{W} Q')$

unfolding defs by auto

lemma norm[simp]:
$\langle \\langle \text{False} \rangle \rangle = \langle \text{False} \rangle$
$\langle \\langle \text{True} \rangle \rangle = \langle \text{True} \rangle$

$\langle \neg [p] \rangle = [\neg p]$
$\langle [p] \wedge [q] \rangle = [p \wedge q]$
$\langle [p] \vee [q] \rangle = [p \vee q]$
$\langle [p] \implies [q] \rangle = [p \implies q]$
$\langle [p] \sigma \wedge [q] \sigma \rangle = [p \wedge q] \sigma$
$\langle [p] \sigma \vee [q] \sigma \rangle = [p \vee q] \sigma$
$\langle [p] \sigma \implies [q] \sigma \rangle = [p \implies q] \sigma$

$\langle \diamond (\text{False}) \rangle = \langle \text{False} \rangle$
$\langle \diamond (\text{True}) \rangle = \langle \text{True} \rangle$

$\langle \Box (\text{False}) \rangle = \langle \text{False} \rangle$
$\langle \Box (\text{True}) \rangle = \langle \text{True} \rangle$
\[(\neg \Box P) \sigma = (\Diamond (\neg P)) \sigma\]
\[(\Box \Box P) = (\Box P)\]

\[(\Diamond (\false)) = (\false)\]
\[(\Diamond (\true)) = (\true)\]
\[(\neg \Diamond P) = (\Box (\neg P))\]
\[(\Diamond \Diamond P) = (\Diamond P)\]

\[(P \mathcal{W} (\false)) = (\Box P)\]

\[(\neg (P \mathcal{U} Q)) \sigma = (\neg P \mathcal{R} \neg Q) \sigma\]
\[(P \mathcal{U} (\false)) = (\false)\]
\[(P \mathcal{U} (\true)) = (\true)\]
\[(P \mathcal{U} (\false)) = (\Diamond P)\]
\[(P \mathcal{U} (P \mathcal{U} Q)) = (P \mathcal{U} Q)\]

unfoldingdefs
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:\[(\Box(P \land Q)) = (\Box P \land \Box Q)\]
unfoldingdefsbyauto

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

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

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

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

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

unfolding defs
apply –
apply (metis (full-types) add.commute add-diff-inverse-nat less-one suffix-plus suffix-zero)
apply (metis (full-types) One-nat-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-eq suffix-zero)
apply (metis suffix-zero)
apply force
using less-Suc-eq-0-disj apply fastforce
apply (metis gr0-conv-Suc nat-neq-iff not-less0 suffix-zero)

apply fastforce
apply (case-tac i; auto)
apply force
using less-Suc-eq-0-disj apply force

apply force
using less-Suc-eq-0-disj apply fastforce
apply fastforce
apply (case-tac i; auto)
done

lemma mono:
\[((\Box P) \, \sigma) \land P \sigma \Rightarrow P' \sigma\] \Rightarrow \((\Box P') \, \sigma\)
\[((\Diamond P) \, \sigma) \land P \sigma \Rightarrow P' \sigma\] \Rightarrow \((\Diamond P') \, \sigma\)
\[((P \mathbin{\bigoplus} Q) \, \sigma) \land P \sigma \Rightarrow P' \sigma; \land Q \sigma \Rightarrow Q' \sigma\] \Rightarrow \((P' \mathbin{\bigoplus} Q') \, \sigma\)
\[((P \mathbin{\bigotimes} Q) \, \sigma) \land P \sigma \Rightarrow P' \sigma; \land Q \sigma \Rightarrow Q' \sigma\] \Rightarrow \((P' \mathbin{\bigotimes} Q') \, \sigma\)

unfolding defs by force+

lemma always-imp-mono:
\[((\Box P) \, \sigma) \Rightarrow (P \Rightarrow P') \, \sigma\] \Rightarrow \((\Box P') \, \sigma\)
\[((\Diamond P) \, \sigma) \Rightarrow (P \Rightarrow P') \, \sigma\] \Rightarrow \((\Diamond P') \, \sigma\)
\[((P \mathbin{\bigoplus} Q) \, \sigma) \Rightarrow (P \leftrightarrow P') \, \sigma; (Q \leftrightarrow Q') \, \sigma\] \Rightarrow \((P' \mathbin{\bigoplus} Q') \, \sigma\)
\[((P \mathbin{\bigotimes} Q) \, \sigma) \Rightarrow (P \leftrightarrow P') \, \sigma; (Q \leftrightarrow Q') \, \sigma\] \Rightarrow \((P' \mathbin{\bigotimes} Q') \, \sigma\)

unfolding defs by force+

lemma next-conj-distrib:
\((\circ (P \land Q)) = (\circ P \land \circ Q)\)

unfolding defs by auto

lemma next-disj-distrib:
\((\circ (P \lor Q)) = (\circ P \lor \circ Q)\)

unfolding defs by auto

lemma until-next-distrib:
\((\circ (P \mathbin{\bigoplus} Q)) = (\circ P \mathbin{\bigoplus} Q)\)

unfolding defs by (auto simp: fun-eq-iff)

lemma until-imp-eventually:
\(((P \mathbin{\bigoplus} Q) \Rightarrow \circ Q) \, \sigma\)

unfolding defs by auto

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

lemma unless-unless-disj:
  assumes \((P \land Q) \land R)\ \sigma
  shows \((P \lor Q) \land R)\ \sigma
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 \land Q) \lor R) = ((P \lor R) \land (Q \lor R))
unfoldingdefs
apply (auto simp: fun-eq_iff)
apply (metis dual-order.strict-trans nat-neq_iff)
done

lemma until-disj-distrib:
  \((P \lor (Q \lor R)) = ((P \lor Q) \lor (P \lor R))
unfoldingdefs by (auto simp: fun-eq_iff)

lemma eventually-until:
  \(\Diamond P) = (\neg P \lor 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:
  \(\Diamond(\Diamond P) = (\Diamond Q)
unfoldingdefs by force

lemma eventually-unless-until:
  \(((P \lor Q) \land \Diamond Q = (P \lor Q)
unfoldingdefs by force

lemma eventually-always-imp-always-eventually:
  assumes \(\Diamond \square P) \ \sigma
  shows \(\square \Diamond P) \ \sigma
using assms unfoldingdefs by (metis suffix-commute)

lemma eventually-always-next-stable:
  assumes \(\Diamond P) \ \sigma
  assumes \(P \hookrightarrow \Diamond P) \ \sigma
  shows \(\Diamond \square P) \ \sigma
using assms by (metis (no-types) eventuallyI alwaysD always-induct eventuallyE norm(15))

lemma next-stable-imp-eventually-always:
  assumes \(P \hookrightarrow \Diamond P) \ \sigma
  shows \(\Diamond P \hookrightarrow \Diamond \square P) \ \sigma
using assms eventually-always-next-stable by blast
lemma always-eventually-always:
\(\boxdot\Box P = \Box\Box P\)

unfolding defs by (clarsimp simp: fun-eq-iff) (metis add.left-commute semiring-normalization-rules(25))

lemma stable-unless:
assumes \((P \rightarrow \circ (P \lor Q))\) \(\sigma\)
shows \((P \rightarrow (P \lor W \lor Q))\) \(\sigma\)
using assms unfolding defs
apply –
apply (rule ccontr)
apply clarsimp
apply (drule (1) ex-least-nat-less[where \(P = \lambda j. \neg P\) \(s i + j\) for \(i\), simplified])
apply clarsimp
apply (metis add-Suc-right le-less less-Suc-eq)
done

lemma unless-induct: — Rule \(\texttt{WAIT}\) from Manna and Pnueli (1995, Fig 3.3)
assumes \(I: (I \rightarrow \circ (I \lor R))\) \(\sigma\)
assumes \(P: (P \rightarrow I \lor R)\) \(\sigma\)
assumes \(Q: (I \rightarrow Q)\) \(\sigma\)
shows \((P \rightarrow Q \lor W \lor R)\) \(\sigma\)
apply (intro alwaysI impI)
apply (erule impE[where \(P=I\) and \(Q=R\)])
apply (erule always-imp-mono(4))
apply (erule mp[where \(P=I\) and \(Q=R\)])
apply simp add: Q alwaysD
apply blast
apply (simp add: unfold)
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 “\(A \text{ leads to } C\)”) or \(\lambda s. A s \rightarrow (B \cup C) s\) (“\(A \text{ leads to } C \text{ via } B\)”).

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 \(\rightsquigarrow\) 25) where
\(P \rightsquigarrow Q \equiv P \leftrightarrow \diamond Q\)

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

lemma leads-to-trans:
assumes \((P \rightsquigarrow Q)\) \(\sigma\)
assumes \((Q \rightsquigarrow R)\) \(\sigma\)
shows \((P \rightsquigarrow R)\) \(\sigma\)
using assms unfolding defs by clarsimp (metis semiring-normalization-rules(25))

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

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

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

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

**lemma** **leads-to-via-trans**:  
assumes $(A \rightarrow B \cup C) \sigma$
assumes $(C \rightarrow D \cup E) \sigma$
shows $(A \rightarrow (B \lor D) \cup E) \sigma$
**proof**(rule alwaysI, rule impI)
  fix $i$ assume $A \ (\sigma |_s i)$ with **assms show** $((B \lor D) \cup E) \ (\sigma |_s i)$
    apply –
    apply (erule alwaysE[where $i=i$])
    apply clarsimp
    apply (erule untilE)
    apply clarsimp
    apply (drule (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 \((A \lor C) \leftrightarrow (B \lor D) \cup E) \sigma\)

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

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 (drule (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

next

fix \(i\) assume \(C (\sigma \mid_s i)\) with \(assms(2)\) show \(((B \lor D) \cup E) (\sigma \mid_s i)\)

apply –
apply (erule alwaysE[where \(i=i\)])
apply (simp add: mono)
done

qed

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

assumes \(stable: (P \land Q \leftrightarrow \circ Q) \sigma\)
assumes \(U: (A \leftrightarrow P \cup C) \sigma\)
shows \(((A \land Q) \leftrightarrow P \cup (C \land Q)) \sigma\)

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

fix \(i\) assume \(AP: A (\sigma \mid_s i) Q (\sigma \mid_s i)\)

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

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

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

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

qed

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

assumes \(wf R\)
assumes \(indhyp: \forall t. (A \land [\delta = \langle t \rangle]) \leftrightarrow B \cup (A \land [\delta \otimes \langle t \rangle \in \langle R \rangle \lor C)) \sigma\)
shows \((A \leftrightarrow B \cup C) \sigma\)

**proof** (intro alwaysI implI)

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

prove (induct \(\delta (\sigma \mid_i)\) arbitrary: \(i\))

\(case\) (less \(i\)) with \(indhyp[where \ t=\delta (\sigma \mid_i)]\) show \(?case\)

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

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

- $W_1$ generalised to be contingent.
- $W_2$ is a well-founded set of assertions that by $W_1$ includes $P$

\textbf{lemma} \textit{leads-to-uf}:

\textbf{fixes} \textit{Is} :: ('a seq-pred × ('a ⇒ 'b)) \textit{set} \\
\textbf{assumes} \textit{wf} (R :: 'b rel) \\
\textbf{assumes} $W_1$: $\bigcirc(\exists \varphi. \lfloor \varphi\in\text{fst ' Is}\rfloor \land (P \longrightarrow \varphi)) \sigma$ \\
\textbf{assumes} $W_2$: $\forall (\varphi, \delta)\in\text{Is}. \exists (\varphi', \delta')\in\text{insert (Q, \delta0)} \text{Is}. \forall t. (\varphi \land [\delta \equiv (t)] \leadsto \varphi' \land [\delta' \otimes (t) \in (R)] \sigma$ \\
\textbf{shows} $(P \leadsto Q) \sigma$

\textbf{proof} -- \\
\textbf{have} $(\varphi \land [\delta \equiv (t)] \leadsto Q) \sigma$ if $(\varphi, \delta) \in \text{Is}$ for $\varphi \delta t$ \\
\textbf{proof} -- \\
\textbf{using} $(\text{wf } R)$ that $W_2$ \\
\textbf{unfolding} \textit{LTL}.defs split-def \\
\textbf{apply} \textit{clarsimp} \\
\textbf{apply} $(\text{metis (no-types, opaque-lifting) ab-semigroup-add-class.add-ac(1) fst-eqD snd-conv surjective-pairing})$ \\
\textbf{done}

with $W_1$ \textbf{show} ?thesis \\
\textbf{apply} -- \\
\textbf{apply} $(\text{rule alwaysI})$ \\
\textbf{apply} \textit{clarsimp} \\
\textbf{apply} $(\text{erule-tac i=i in alwaysE})$ \\
\textbf{apply} \textit{clarsimp} \\
\textbf{using} $\textit{alwaysD}$ $\textit{suffix-state-prop}$ \textbf{apply} \textit{blast} \\
\textbf{done}

\textbf{qed}

\textbf{3.2 Fairness}

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

\textbf{definition} \textit{weakly-fair} :: 'a seq-pred ⇒ 'a seq-pred ⇒ 'a seq-pred \textbf{where} \\
\textit{weakly-fair} \textit{enabled taken} $= (\Box\text{enabled} \leftrightarrow \Diamond\text{taken})$

\textbf{lemma} \textit{weakly-fair-def2}:

\textbf{shows} $\text{weakly-fair enabled taken} = \Box(\neg\Box(\text{enabled} \land \neg\text{taken}))$

\textbf{unfolding} \textit{weakly-fair-def} \textbf{by} $(\text{metis (full-types) always-conj-distrib norm(18)})$

\textbf{lemma} \textit{weakly-fair-def3}:

\textbf{shows} $\text{weakly-fair enabled taken} = (\Diamond\Box\text{enabled} \rightarrow \Box\Diamond\text{taken})$

\textbf{unfolding} \textit{weakly-fair-def2} \\
\textbf{apply} $(\textit{clarsimp simp: fun-eq-iff})$

\textbf{unfolding} \textit{defs} \\
\textbf{apply} \textit{auto} \\
\textbf{apply} $(\text{metis (full-types) add.left-commute semiring-normalization-rules(25)})$

\textbf{done}

\textbf{lemma} \textit{weakly-fair-def4}:

\textbf{shows} $\text{weakly-fair enabled taken} = \Box\Diamond(\text{enabled} \rightarrow \text{taken})$

\textbf{using} $\textit{weakly-fair-def2}$ \textbf{by} \textit{force}
lemma \( mp\text{-weakly-fair} \):
  assumes weakly-fair enabled taken \( \sigma \)
  assumes \((\square \text{enabled}) \sigma\)
  shows \((\Diamond \text{taken}) \sigma\)
using asms unfolding weakly-def using always-imp-mp by blast

lemma always-weakly-fair:
  shows \( \Box (\text{weakly-fair enabled taken}) = \text{weakly-fair enabled taken} \)
unfolding weakly-fair-def by simp

lemma eventually-weakly-fair:
  shows \( \Diamond (\text{weakly-fair enabled taken}) = \text{weakly-fair enabled taken} \)
unfolding weakly-fair-def2 by (simp add: always-eventually-always)

lemma weakly-fair-weaken:
  assumes \((\text{enabled} \rightarrow \rightarrow \text{enabled}) \sigma\)
  assumes \((\text{taken} \rightarrow \rightarrow \text{taken}) \sigma\)
  shows \((\text{weakly-fair enabled taken} \rightarrow \text{weakly-fair enabled'} \rightarrow \text{taken}' \sigma\)
using asms unfolding weakly-fair-def defs by simp blast

lemma weakly-fair-unless-until:
  shows \( (\text{weakly-fair enabled taken} \land (\text{enabled} \rightarrow \text{enabled} \text{ W taken})) = (\text{enabled} \rightarrow \text{enabled} \text{ U taken}) \)
unfolding defs weakly-fair-def
apply (auto simp: fun-eq-iff)
apply (metis add.right-neutral)
done

lemma stable-leads-to-eventually:
  assumes \((\text{enabled} \leftrightarrow \Diamond (\text{enabled} \lor \text{taken})) \sigma\)
  shows \((\text{enabled} \leftrightarrow (\Box \text{enabled} \lor \Diamond \text{taken})) \sigma\)
using asms unfolding defs
apply –
apply (rule ccontr)
apply clarsimp
apply (drule (1) ex-least-nat-less[where \( P=\lambda j. \neg \text{enabled} (\sigma \mid s + j) \) for \( i \), simplified])
apply clarsimp
apply (metis add-Suc-right leI less-irrefl-nat)
done

lemma weakly-fair-stable-leads-to:
  assumes weakly-fair enabled taken \( \sigma \)
  assumes \((\text{enabled} \leftrightarrow \Diamond (\text{enabled} \lor \text{taken})) \sigma\)
  shows \((\text{enabled} \rightarrow \text{enabled} \text{ U taken}) \sigma\)
using stable-leads-to-eventually[OF asms(2)] asms(1) unfolding defs weakly-fair-def
by (auto simp: fun-eq-iff)

lemma weakly-fair-stable-leads-to-via:
  assumes weakly-fair enabled taken \( \sigma \)
  assumes \((\text{enabled} \leftrightarrow \Diamond (\text{enabled} \lor \text{taken})) \sigma\)
  shows \((\text{enabled} \leftrightarrow \Diamond \text{taken}) \sigma\)
using stable-unless[OF asms(2)] asms(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 = (\( \Box \Diamond \text{enabled} \leftrightarrow \Diamond \text{taken}\)

lemma strongly-fair-def2:
\[\text{strongly-fair enabled taken} = \Box (\neg \Box (\Diamond \text{enabled} \land \neg \text{taken}))\]

unfolding strongly-fair-def by (metis weakly-fair-def weakly-fair-def2)

lemma strongly-fair-def3:
\[\text{strongly-fair enabled taken} = (\Box \Diamond \text{enabled} \rightarrow \Box \Diamond \text{taken})\]

unfolding strongly-fair-def2 by (metis (full-types) always-eventually-always weakly-fair-def2 weakly-fair-def3)

lemma always-strongly-fair:
\[\Box (\text{strongly-fair enabled taken}) = \text{strongly-fair enabled taken}\]

unfolding strongly-fair-def by simp

lemma eventually-strongly-fair:
\[\Diamond (\text{strongly-fair enabled taken}) = \text{strongly-fair enabled taken}\]

unfolding strongly-fair-def2 by (simp add: always-eventually-always)

lemma strongly-fair-disj-distrib: — not true for weakly-fair
\[\text{strongly-fair} (\text{enabled1} \lor \text{enabled2}) \text{ taken} = (\text{strongly-fair enabled1 taken} \land \text{strongly-fair enabled2 taken})\]

unfolding strongly-fair-def defs
apply (auto simp: fun-eq-iff)
apply blast
apply blast
apply (metis (full-types) semiring-normalization-rules(25))
done

lemma strongly-fair-imp-weakly-fair:
assumes strongly-fair enabled taken \(\sigma\)
shows weakly-fair enabled taken \(\sigma\)
using assms unfolding strongly-fair-def3 weakly-fair-def3 by (simp add: eventually-always-imp-always-eventually)

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

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).

lemma safety-state-prop:
shows safety \([P]\)
unfolding defs by (rule safety-state-prop)

lemma safety-Next:
assumes safety \(P\)
shows safety \((\Diamond P)\)
using assms unfolding defs safety-def
apply clarsimp
apply (metis (mono-tags) One-nat-def list.sel(3) nat.simps(3) stake.simps(2))
done

lemma safety-unless:
assumes safety \(P\)
assumes safety \(Q\)
shows safety \((P W Q)\)
proof (rule safetyI2)
  fix \(\sigma\) assume \(X: \exists \beta. (P W Q)\) (stake \(i\) \(\sigma\) \(\triangleleft \beta\)) for \(i\)
then show \((P \land Q) \sigma\)

proof (cases \(\forall i, j. \exists \beta. P(i \rightarrow j) @- \beta\))
case True
  with \(\langle\text{safety P}\rangle \, \forall i. \, P(\sigma | s \, i)\) 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 \(\forall i. \, k + k' \leq i \rightarrow \neg P((\text{stake } i \, \sigma @- u) | s \, k)\)
  by (metis add.commute diff-add stake-shift-stake-shift stake-suffix-drop suffix-shift)
then have \(\forall i. \, k + k' \leq i \land (P \land Q) ((\text{stake } i \, \sigma @- u) \rightarrow (\exists m \leq k. \, Q((\text{stake } i \, \sigma @- u) | s \, m) \land (\forall p < m. \, P((\text{stake } i \, \sigma @- u) | s \, p)))\)
  unfolding defs using lel by blast
then have \(\forall i. \, k + k' \leq i \land (P \land Q) ((\text{stake } i \, \sigma @- u) \rightarrow (\exists m \leq k. \, Q((\sigma(m \rightarrow i - m) @- u) \land (\forall p < m. \, P((\sigma(p \rightarrow i - p) @- u))))\)
  unfolding defs by (metis stake-suffix add-leE nat-less-le order-trans)
then have \(\forall i. \, \exists n \geq i. \, \exists m \leq k. \, \exists u. \, (\sigma(m \rightarrow n - m) @- u) \land (\forall p < m. \, P((\sigma(p \rightarrow n - p) @- u)))\)
  using X by (metis add.commute le-add1)
then have \(\exists m \leq k. \, \forall i. \, \exists n \geq i. \, \exists u. \, (\sigma(m \rightarrow n - m) @- u) \land (\forall p < m. \, P((\sigma(p \rightarrow n - p) @- u)))\)
  by (simp add: always-eventually-pigeonhole)
with \(\langle\text{safety P}\rangle \, \langle\text{safety Q}\rangle\) show \((P \land Q) \sigma\)
  unfolding defs by (metis Nat.le-diff-conv2 add-leE safety-always-eventually)
qed
qed

lemma safety-always:
assumes safety P
shows safety \((\square P)\)
using assms by (metis norm(20) safety-def safety-unless)

lemma absolute-liveness-eventually:
shows absolute-liveness \(\longleftrightarrow (\exists \sigma. \, P \, \sigma) \land P = \diamond P\)
unfolding absolute-liveness-def defs
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 stable \(\longleftrightarrow (\exists \sigma. \, P \, \sigma) \land P = \square P\)
unfolding stable-def defs by (metis suffix-zero)

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

- it is reasonable to assume they are state formulas
- taken must be satisfiable

lemma fairness-weakly-fair:
assumes \(\exists s. \, \text{taken} \, s\)
shows fairness \((\text{weakly-fair [enabled] [taken]})\)
unfolding fairness-def stable-def absolute-liveness-def weakly-fair-def
using assms
apply auto
  apply (rule-tac \(x = \lambda - . \, \text{in exI}\))
  apply fastforce
apply (simp add: alwaysD)
apply (rule-tac \(x = \lambda - . \, \text{in exI}\))
apply fastforce
apply (metis (full-types) absolute-liveness-def absolute-liveness-eventually eventually-weakly-fair weakly-fair-def)
done
lemma fairness-strongly-fair:  
assumes \( \exists s. \text{assumed} \)
shows fairness \((\text{strongly-fair} [\text{enabled}] [\text{taken}])\)
unfolding fairness-def stable-def absolute-liveness-def strongly-fair-def 
using assms
apply auto
\quad 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

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 (§22) 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 \(\Rightarrow\) 'state set with LocalOp, and conditions of type 's \(\Rightarrow\) 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 \Rightarrow\) bool

datatype ('answer, 'location, 'question, 'state) com
\quad = Request 'location 'state \Rightarrow\) 'question 'answer \Rightarrow\) 'state \Rightarrow\) 'state set (\(\langle\langle\rangle\rangle\) Request - - [0, 70, 70] 71)
\quad | Response 'location 'question \Rightarrow\) 'state \Rightarrow\) ('state \times\) 'answer) set (\(\langle\langle\rangle\rangle\) Response - [0, 70] 71)
\quad | LocalOp 'location 'state \Rightarrow\) 'state set (\(\langle\langle\rangle\rangle\) LocalOp - [0, 70] 71)
\quad | Cond1 'location 'state bexp ('answer, 'location, 'question, 'state) com (\(\langle\langle\rangle\rangle\) IF - THEN - FI [0, 0, 0] 71)
\quad | Cond2 'location 'state bexp ('answer, 'location, 'question, 'state) com ('answer, 'location, 'question, 'state) com (\(\langle\langle\rangle\rangle\) IF - / THEN - / ELSE - / FI [0, 0, 0, 71])
\quad | Loop ('answer, 'location, 'question, 'state) com (LOOP DO - / OD [0] 71)
\quad | While 'location 'state bexp ('answer, 'location, 'question, 'state) com (\(\langle\langle\rangle\rangle\) WHILE - / DO - / OD [0, 0, 0] 71)
\quad | Seq ('answer, 'location, 'question, 'state) com ('answer, 'location, 'question, 'state) com (infixr ;; 69)
choose ('answer, 'location, 'question, 'state) com
('answer, 'location, 'question, 'state) com

\textit{(infixl} \oplus \textit{68})

We provide a one-armed conditional as it is the common form and avoids the need to discover a label for an internal \texttt{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 $\pi$-calculus of Milner (1989)) or large non-deterministic sums (ala CCS (Milner 1980, §2.8)). Intuitively the requester poses a 'question with a \texttt{Request} command, which upon rendezvous with a responder’s \texttt{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 ($\oplus$) \textit{(infix ($\oplus$))}. Internal choice can be recovered in combination with local operations (see Milner (1980, §2.3)).

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

\begin{itemize}
  \item \textbf{abbreviation} \texttt{SKIP-syn} \textit{(\texttt{[]} / \texttt{SKIP [0] 71}) where}
  \begin{align*}
  \texttt{[]} \texttt{SKIP} \equiv \texttt{[]} \texttt{LocalOp} \texttt{(}\lambda s. \{s\})
  \end{align*}
  \item \textbf{abbreviation} \textit{(input)} \texttt{DetLocalOp} :: 'location \Rightarrow ('state \Rightarrow 'state)
    \Rightarrow ('answer, 'location, 'question, 'state) com \textit{(\texttt{[]} [\_] [0, 0] 71}) where
  \begin{align*}
  \texttt{[]} \texttt{f} \equiv \texttt{[]} \texttt{LocalOp} \texttt{(}\lambda s. \{f s\})
  \end{align*}
  \item \textbf{syntax}
    \begin{align*}
    \texttt{-quote} & :: 'b \Rightarrow ('a \Rightarrow 'b) (\texttt{''} [0] 1000) \\
    \texttt{-antique} & :: ('a \Rightarrow 'b) \Rightarrow 'b (\texttt{''} [1000]) \\
    \texttt{-Assign} & :: 'location \Rightarrow \texttt{idt} \Rightarrow 'b \Rightarrow ('answer, 'location, 'question, 'state) com \textit{((\texttt{[]} [\_] [0, 0] 71)} \\
    \texttt{-NonDetAssign} & :: 'location \Rightarrow \texttt{idt} \Rightarrow 'b set \Rightarrow ('answer, 'location, 'question, 'state) com \textit{((\texttt{[]} [\_] [0, 0, 70] 71)}
    \end{align*}
  \item \textbf{abbreviation} \textit{(input)} \texttt{NonDetAssign} :: 'location \Rightarrow ('val \Rightarrow 'val) \Rightarrow 'state \Rightarrow 'state) \Rightarrow ('state \Rightarrow 'val set)
    \Rightarrow ('answer, 'location, 'question, 'state) com \texttt{where}

    \begin{align*}
    \texttt{NonDetAssign} l upd es \equiv \texttt{[]} \texttt{LocalOp} \texttt{(}\lambda s. \{ upd (e) s | e. e \in es s \})
    \end{align*}
  \item \textbf{translations}
    \begin{align*}
    \texttt{[]} \texttt{'}x := e => \texttt{CONST DetLocalOp l \texttt{''}('update-name x (\lambda.. e))'} \\
    \texttt{[]} \texttt{'}x \in es => \texttt{CONST NonDetAssign l ('update-name x) \texttt{''}es'}
    \end{align*}
  \item \textbf{parse-translation}
\end{itemize}

\begin{verbatim}
let
  fun antiquote-tr tr i (Const (@{syntax-const -antique}, -) $ t as Const (@{syntax-const -antique}, -) $ s) = skip-antiquote-tr tr i t
  | antiquote-tr tr i (Const (@{syntax-const -antique}, -) $ t) = antiquote-tr tr i t $ Bound i
  | antiquote-tr tr i (t $ u) = antiquote-tr tr i t $ antiquote-tr tr i u
  | antiquote-tr tr i (Abs (x, T, t)) = Abs (x, T, antiquote-tr tr (i + 1) t)
  | antiquote-tr tr - a = a

and skip-antiquote-tr tr i (c as Const (@{syntax-const -antique}, -) $ t) =
  c $ skip-antiquote-tr tr i t
  | skip-antiquote-tr tr i t = antiquote-tr tr i t;

  fun quote-tr [t] = Abs (s, dummyT, antiquote-tr tr 0 (Term.incr-boundvars 1 t))
  | quote-tr ts = raise TERM (quote-tr, ts);

in
[(@{syntax-const -quote}, K $quote-tr)] end
\end{verbatim}
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] 55)

where
    [ [ α = action s; s' ∈ val β s ] ] ⇒ (sl l) Request action val # cs, -, s) → sα, ββ (cs, Some l, s')
    | (s', β) ∈ action α s ⇒ (sl l) Response action # cs, -, s) → sα, ββ (cs, Some l, s')
    | s' ∈ R s ⇒ (sl l) LocalOp R # cs, -, s) → τ (cs, Some l, s')
    | b s ⇒ (sl l) IF b THEN c FI # cs, -, s) → τ (c # cs, Some l, s)
    | ¬ b s ⇒ (sl l) IF b THEN c FI # cs, -, s) → τ (cs, Some l, s)
    | b s ⇒ (sl l) IF b THEN c1 ELSE c2 FI # cs, -, s) → τ (c1 # cs, Some l, s)
    | ¬ b s ⇒ (sl l) IF b THEN c1 ELSE c2 FI # cs, -, s) → τ (c2 # cs, Some l, s)
    | (c # LOOP DO c OD # cs, s) →α (cs', s') ⇒ (LOOP DO c OD # cs, s) →α (cs', s')
    | b s ⇒ (sl l) WHILE b DO c OD # cs, -, s) → τ (c # sl l) WHILE b DO c OD # cs, Some l, s)
    | ¬ b s ⇒ (sl l) WHILE b DO c OD # cs, -, s) → τ (cs, Some l, s)
    | (c1 # c2 # cs, s) →α (cs', s') ⇒ (c1;; c2 # cs, s) →α (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 \textit{Request} action must be enabled in process \( p_1 \), and a \textit{Response} action in (distinct) process \( p_2 \), where the request/response labels \( \alpha \) and \( \beta \) (semantically) match.

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

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

\[
\text{system-step} :: \langle \text{proc set} \rangle
\Rightarrow \langle \text{proc} \rangle \\
\Rightarrow \langle \text{proc} \rangle
\Rightarrow \text{system-state} \\
\Rightarrow \text{system-state} \\
\Rightarrow \text{bool}
\]

where

- \text{LocalStep:} [ \text{GST sh} p \rightarrow_{\tau} \text{ls} ]; \text{GST sh}' = (\text{GST sh})(p := \text{ls}); \text{HST sh}' = \text{HST sh} ] \implies \text{system-step} \{ p \} \text{ sh' sh}
- \text{CommunicationStep:} [ \text{GST sh} p \rightarrow_{s, \alpha, \beta} \text{ls} ]; \text{GST sh} q \rightarrow_{s, \alpha, \beta} \text{ls}'; p \neq q; \text{GST sh}' = (\text{GST sh})(p := \text{ls}'); \text{HST sh}' = \text{HST sh} @ [\langle \alpha, \beta \rangle] \implies \text{system-step} \{ p, q \} \text{ sh' sh}

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.

\[
\text{type-synonym} \; (\langle \text{answer} , \text{location} , \text{proc} , \text{question} , \text{state} \rangle \text{ state-pred}) = (\langle \text{answer} , \text{location} , \text{proc} , \text{question} , \text{state} \rangle \text{ system-state} \Rightarrow \text{bool})
\]

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

Conversely the \textit{LSTP} operator lifts predicates over local states into predicates over \langle \text{answer} , \text{location} , \text{proc} , \text{question} , \text{state} \rangle \text{ system-state}.

Predicates that do not depend on control locations were termed \textit{universal assertions} by Levin and Gries (1981, §3.6).

\[
\text{type-synonym} \; (\langle \text{proc} , \text{state} \rangle \text{ local-state-pred}) = (\langle \text{proc} , \text{state} \rangle \text{ local-states} \Rightarrow \text{bool})
\]

\[
\text{definition} \; \text{LST} :: (\langle \text{answer} , \text{location} , \text{proc} , \text{question} , \text{state} \rangle \text{ system-state}) \Rightarrow (\langle \text{proc} , \text{state} \rangle \text{ local-states} (\downarrow [1000] 1000) \text{ where})
\]

\[
s_{\downarrow} = cLST \circ \text{GST s}
\]

\[
\text{abbreviation} \; (\text{input}) \; \text{LSTP} :: (\langle \text{proc} , \text{state} \rangle \text{ local-state-pred}) \Rightarrow (\langle \text{answer} , \text{location} , \text{proc} , \text{question} , \text{state} \rangle \text{ state-pred} \text{ where})
\]

\[
LSTP \; P \equiv \lambda s. \; P \downarrow
\]
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**

\[\text{atC :: ('}\text{answer, 'location, 'question, 'state} \Rightarrow \text{'}\text{location label}\]

\[
\text{where}
\]

\[\text{atC (}\{1\}\text{ Request action val) } = \{1\}\]
\[\text{atC (}\{1\}\text{ Response action) } = \{1\}\]
\[\text{atC (}\{1\}\text{ LocalOp f) } = \{1\}\]
\[\text{atC (}\{1\}\text{ IF - THEN - FI) } = \{1\}\]
\[\text{atC (}\{1\}\text{ IF - THEN - ELSE - FI) } = \{1\}\]
\[\text{atC (}\{1\}\text{ WHILE - DO - OD) } = \{1\}\]
\[\text{atC (LOOP DO c OD) } = \text{atC c}\]
\[\text{atC (c1 ;; c2) } = \text{atC c1} \cup \text{atC c2}\]

**primrec** atCs :: (\text{'}\text{answer, 'location, 'question, 'state} \Rightarrow \text{'}\text{location label}) \Rightarrow \text{'}\text{location label} \text{ where}

\[\text{atCs [] } = \{\}\]
\[\text{atCs (}c\#\#\) = \text{atC c}\]

We provide the following definitions to the end-user.

\(\text{AT}\) maps process names to a predicate that is true of locations where control for that process resides, and the abbreviation \(\text{at}\) provides a conventional way to use it. The constant \(\text{atS}\) 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 :: (\text{'}\text{answer, 'location, 'proc, 'question, 'state}) \Rightarrow \text{'}\text{proc} \Rightarrow \text{'}\text{location label} \text{ where}

\[\text{AT s p } = \text{atCs (}c\text{PGM (GST s p))}\]

**abbreviation** at :: \text{'}\text{proc} \Rightarrow \text{'}\text{location} \Rightarrow \text{'}\text{answer, 'location, 'proc, 'question, 'state} \Rightarrow \text{'}\text{state-pred} \text{ where}

\[\text{at p l s } = l \in \text{AT s p}\]

**definition** atS :: \text{'}\text{proc} \Rightarrow \text{'}\text{location set} \Rightarrow \text{'}\text{answer, 'location, 'proc, 'question, 'state} \Rightarrow \text{'}\text{state-pred} \text{ where}

\[\text{atS p ls s } = (\exists l \in \text{ls. at p l s})\]

**definition** atLs :: \text{'}\text{proc} \Rightarrow \text{'}\text{location label set} \Rightarrow \text{'}\text{answer, 'location, 'proc, 'question, 'state} \Rightarrow \text{'}\text{state-pred} \text{ where}

\[\text{atLs p labels s } = (\text{AT s p } \in \text{labels})\]

**abbreviation** (input) atL :: \text{'}\text{proc} \Rightarrow \text{'}\text{location label} \Rightarrow \text{'}\text{answer, 'location, 'proc, 'question, 'state} \Rightarrow \text{'}\text{state-pred} \text{ where}

\[\text{atL p label } = \text{atLs p {label}}\]

**definition** atPLs :: \text{'}\text{proc} \times \text{'}\text{location label set} \Rightarrow \text{'}\text{answer, 'location, 'proc, 'question, 'state} \Rightarrow \text{'}\text{state-pred} \text{ where}

\[\text{atPLs pls } = (\forall p \text{ label. } (p, \text{ label}) \in \text{pls} ) \rightarrow \text{atL p label}\]

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 :: \text{'}\text{proc} \Rightarrow \text{'}\text{location} \Rightarrow \text{'}\text{answer, 'location, 'proc, 'question, 'state} \Rightarrow \text{'}\text{state-pred} \text{ where}

\[\text{taken p l s } \leftarrow c\text{TKN (GST s p) = Some l}\]

---

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 ≡ all 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 o 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 ←→ (∀ 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) state-pred ⇒ bool (- |= - [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
\[(\text{sys} \models \varphi) \iff (\forall \sigma. \text{run sys} \sigma \to \varphi \sigma)\]

## 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 wrt 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{\{\} Request action val} \# cs, ls) \to_{a} s'
- (\text{\{l\} Response action} \# cs, ls) \to_{a} s'
- (\text{LocalOp R} \# cs, ls) \to_{a} s'
- (\text{IF b THEN c FI} \# cs, ls) \to_{a} s'
- (\text{IF b THEN c1 ELSE c2 FI} \# cs, ls) \to_{a} s'
- (\text{WHILE b DO c OD} \# cs, ls) \to_{a} s'
- (\text{LOOP DO c OD} \# cs, ls) \to_{a} s'

**lemma** small-step-stuck:
\[-(\text{\{\}, s}) \to_{a} e'\]
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]:
\[AT s p = AT s' p \implies atS p ls s \iff atS p ls s'\]
by (auto simp: atS-def)

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

**lemma** atS-simps:
- \[\neg atS p \{\} s\]
- \[atS p \{l\} s \iff at p l s\]
- \[at p l s; l \in ls \implies atS p ls s\]
- \[(\forall l. at p l s \to l \notin ls) \implies \neg atS p ls s\]
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:
\[atS p (l \cup l') s \iff atS p l s \lor atS p l' s\]
by (auto simp: atS-def)

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

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

lemma small-step-terminated:
\[ s →_x s' ⇒ atCs \ (fst \ s) = {} \implies atCs \ (fst \ s') = {} \]
by (induct pred: small-step) auto

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

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

lemma terminated-no-commands:
assumes terminated \ p \ sh
shows ∃ \ 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 = [])) = atC \circ \ PGMs \ sys \ ∧ \ INIT \ sys \ (((GST = s, HST = [])) \downarrow \ ∧ \ (\forall \ p \ l. \ ¬ \ 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 = [])) ⇔ l \in atC \ (PGMs \ sys \ p))
using assms unfolding initial-state-def split-def AT-def by simp
definition
\text{reachable-state} :: (\text{'answer}, \text{'location}, \text{'proc}, \text{'question}, \text{'state}, \text{'ext}) \text{ pre-system-ext}
\Rightarrow (\text{'answer}, \text{'location}, \text{'proc}, \text{'question}, \text{'state}) \text{ state-pred}

where
\text{reachable-state sys s} \leftarrow (\exists \sigma. \text{prerun sys } \sigma \land \sigma \ i = s)

lemma \text{reachable-stateE}:
\text{assumes reachable-state sys sh}
\text{assumes } \bigwedge \sigma \ i. \text{prerun sys } \sigma \Rightarrow P (\sigma \ i)
\text{shows } P \ sh
\text{using assms unfolding reachable-state-def by blast}

lemma \text{prerun-reachable-state}:
\text{assumes prerun sys } \sigma
\text{shows reachable-state sys } (\sigma \ i)
\text{using assms unfolding prerun-def LTL.defs system-step-reflclp-def reachable-state-def by auto}

lemma \text{reachable-state-induct[consumes 1, case-names init LocalStep CommunicationStep, induct set: reachable-state]}:
\text{assumes } r: \text{reachable-state sys sh}
\text{assumes } i: \bigwedge s. \text{initial-state sys } s \Rightarrow P \langle \text{GST } = s, \ HST = [] \rangle
\text{assumes } l: \bigwedge sh \ ls' p. \text{reachable-state sys sh; P sh; GST sh p } \rightarrow \tau \ ls' \Rightarrow P \langle \text{GST } = (\text{GST sh})(p := ls'), \ HST = HST sh \rangle
\text{assumes } c: \bigwedge sh \ ls1' ls2' p1 p2 \alpha \beta.
\text{reachable-state sys sh; P sh;}
\text{GST sh p1 } \rightarrow_{\alpha, \beta} \ ls1'; \text{ GST sh p2 } \rightarrow_{\alpha, \beta} \ ls2'; \ p1 \neq p2 \]
\Rightarrow P \langle \text{GST } = (\text{GST sh})(p1 := ls1', p2 := ls2'), \ HST = HST sh @ [(\alpha, \beta)] \rangle
\text{shows } P \ sh
\text{using r}
proof (rule reachable-stateE)
fix \sigma \ i assume prerun sys \sigma show P (\sigma \ i)
proof (induct \ i)
case 0 from \langle \text{prerun sys } \sigma \rangle show \ ?case
unfolding prerun-def by (metis (full-types) \ i \ old.unit.exhaust system-state.surjective)
next
case (Suc \ i) with \langle \text{prerun sys } \sigma \rangle 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) \langle \text{prerun sys } \sigma \rangle \ \ i \ old.unit.exhaust prerun-reachable-state system-state.surjective)
apply (metis (full-types) \langle \text{prerun sys } \sigma \rangle \ \ c \ old.unit.exhaust prerun-reachable-state system-state.surjective)
done
qed
qed

lemma \text{prerun-valid-TrueI}:
shows sys \models_{\text{pre}} \langle \text{True} \rangle
unfolding prerun-valid-def by simp

lemma \text{prerun-valid-conjI}:
\text{assumes } sys \models_{\text{pre}} P
\text{assumes } sys \models_{\text{pre}} Q
\text{shows } sys \models_{\text{pre}} P \land Q
\text{using assms unfolding prerun-valid-def always-def by simp}
lemma valid-prerun-lift:
  assumes sys \pre I
  shows sys \sqsubseteq [I]
using assms unfolding prerun-valid-def valid-def run-def by blast

lemma prerun-valid-induct:
  assumes \sigma. prerun sys \sigma \Rightarrow [I] \sigma
  assumes \sigma. prerun sys \sigma \Rightarrow ([I] \leftrightarrow (\bigcirc [I])) \sigma
  shows sys \pre I
unfolding prerun-valid-def using assms by (simp add: always-induct)

lemma prerun-validI:
  assumes \sigma. reachable-state sys \sigma \Rightarrow [I] \sigma
  shows sys \pre I
unfolding prerun-valid-def using assms by (simp add: alwaysI prerun-reachable-state)

lemma prerun-validE:
  assumes reachable-state sys \sigma
  assumes sys \pre I
  shows [I] \sigma
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 (§22). 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 \Rightarrow 'location set

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

definition lcond :: 'location set \Rightarrow 'location set \Rightarrow 'state bexp \Rightarrow ('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 \rightarrow Q lp) \land (\neg b s \rightarrow 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 \Rightarrow 'location set
  \Rightarrow ( ('answer, 'location, 'question, 'state) com
  \times ('answer, 'location, 'question, 'state) loc-comp ) set
where
  fragments (\{1\} IF b THEN c FI) aft
We show that taking system steps preserves fragments.

**lemma** small-step-fragmentsLS:

- *assumes* $s 	o_{a} s'$
- *shows* $\text{fragmentsLS } s' \subseteq \text{fragmentsLS } s$

**using** assms **by** induct (case-tac $[]$ cs, auto)

**lemma** reachable-state-fragmentsLS:

- *assumes* reachable-state sys sh
- *shows* $\text{fragmentsLS } (\text{GST } sh \ p) \subseteq \text{fragments } (\text{PGMs } sys \ p) \{\}$

**using** assms **by** (induct rule: reachable-state-induct)

(auto simp: initial-state-def dest: subsetD[OF small-step-fragmentsLS])

**inductive**

- basic-com :: ('answer, 'location, 'question, 'state) com ⇒ bool

**where**

- basic-com (Request action val)
- basic-com (Response action)
- basic-com (LocalOp R)
- basic-com (IF b THEN c FI)
- basic-com (IF b THEN c1 ELSE c2 FI)
- basic-com (WHILE b DO c OD)

**lemma** fragments-basic-com:

- *assumes* $(c', aft') \in \text{fragments } c \ aft$
- *shows* basic-com $c'$

**using** assms **by** (induct $c$ arbitrary: $aft$) (auto intro: basic-com.intros)

**lemma** fragmentsLS-basic-com:

- *assumes* $(c', aft') \in \text{fragmentsLS } cs$
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) com list)
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. ictxt t ⊕ c2, 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) )
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 assms by (induct c 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 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 assms by (induct c 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 assms by (induct c arbitrary: \(c' \text{ ictxt fctxt}\)) auto

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

theorem context-decompose:
\(s \to_{\alpha} s' \iff (\exists \(c, ictxt, fctxt\) \in \text{decomposeLS } s).\)
\(\begin{align*}
& \text{cPGM } s = ictxt \ c \# \ tl \ (cPGM \ s) \\
& \land (c \# fctxt \ c \oplus \ tl \ (cPGM \ s), cTKN \ s, cLST \ s) \to_{\alpha} s' \\
& \land (\forall \ l \in \text{atC } c. \ cTKN \ s' = \text{Some } l) \end{align*}\)
(is \(?\text{lhs} = ?\text{rhs}\))

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

next
assume \(?\text{rhs}\) then show \(?\text{lhs}\)
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.

fun
loc-compC :: (answer, location, question, state) com
⇒ (answer, location, question, state) com list
⇒ (answer, location, question, state) loc-comp

where
loc-compC (if IF b THEN c1 ELSE c2 FI) cs = lcond (atC c1) (atC c2) b
| loc-compC (LOOP DO c OD) cs = lconst (atC c)
| loc-compC (while WHILE b DO c OD) cs = lcond (atC c) (atC cs) b
| loc-compC c cs = lconst (atC cs)

lemma decompose-fragments:
assumes (c, icxt, fctxt) ∈ decompose-com c0
shows (c, loc-compC c (fctxt c @ cs)) ∈ fragments c0 (atCs cs)
using assms
proof (induct c0 arbitrary: c icxt fctxt cs)
case (Loop c01 c icxt fctxt cs)
from Loop.prems Loop.hyps(1)[where cs=icxt c # cs] show ?case by (auto simp: decompose-com-icxt)
next
case (Seq c01 c02 c icxt fctxt cs)
from Seq.prems Seq.hyps(1)[where cs=c02 # cs] show ?case by auto
qed auto

lemma at-decompose:
assumes (c, icxt, fctxt) ∈ decompose-com c0
shows aT c c ⊆ atC c0
using assms by (induct c0 arbitrary: c icxt fctxt; fastforce)

lemma at-decomposeLS:
assumes (c, icxt, fctxt) ∈ decomposeLS s
shows aT c c ⊆ atC (cPGM s)
using assms unfolding decomposeLS-def by (auto simp: at-decompose split: list.splits)

lemma decomposeLS-fragmentsLS:
assumes (c, icxt, fctxt) ∈ decomposeLS s
shows (c, loc-compC c (fctxt c @ tl (cPGM s))) ∈ fragmentsLS s
using assms
proof (cases cPGM s)
case (Cons d ds)
with assms decompose-fragments[where cs=ds] show ?thesis
  by (cases ds) (auto simp: decomposeLS-def)
qed (simp add: decomposeLS-def)

lemma small-step-loc-compC:
assumes basic-com c
assumes (c ≠ cs, ls) → (ls)
shows loc-compC c cs (snd ls) = atCs (cPGM ls)
using assms by (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.

theorem decompose-small-step:
assumes GST sh p → (ps)
assumes reachable-state sys sh
obtains c cs aft
where \((c, \text{aft}) \in \text{fragments} (\text{PGMs sys } p)\) \{
\hspace{1em} \text{and } \text{atC } c \subseteq \text{atCs } (c\text{PGM } (\text{GST sh } p)) \\
\hspace{1em} \text{and } \text{aft } (\text{cLST } (\text{GST sh } p)) = \text{atCs } (c\text{PGM } \text{ps'}) \\
\hspace{1em} \text{and } (c \neq \text{cs}, \text{cTKN } (\text{GST sh } p), \text{cLST } (\text{GST sh } p)) \rightarrow_{\text{\alpha}} \text{ps'} \\
\hspace{1em} \text{and } \forall l \in \text{atC } c. \text{cTKN ps'} = \text{Some } l
\}

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

Reasoning by induction over the reachable states with \textit{decompose-small-step} is quite tedious. We provide a very simple VCG that generates friendlier local proof obligations in \S5.1.

### 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 (\S5). This is somewhat in the spirit of 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 Cousot and Cousot (1989); Lamport (1980); Lamport and Schneider (1984).

Intuitively we need to discharge a proof obligation for either Requests or Responses but not both. Here we choose to focus on Requests as we expect to have more local information available about these.
We abbreviate invariance with one-sided validity syntax.

**abbreviation** valid-inv (-, - ⊢ \{l\} / \{-\}/ - \{11,0,0,0,0\} 11) where
coms, p, aft ⊢ \{l\} c ≡ coms, p, aft ⊢ \{l\} c \{l\}

**inductive-cases** vcg-inv:
coms, p, aft ⊢ \{pre\} \{l\} Request action \{post\}
coms, p, aft ⊢ \{pre\} \{l\} LocalOp f \{post\}
coms, p, aft ⊢ \{pre\} \{l\} IF b THEN t FI \{post\}
coms, p, aft ⊢ \{pre\} \{l\} WHILE b DO c OD \{post\}
coms, p, aft ⊢ \{pre\} \{l\} LOOP DO c OD \{post\}
coms, p, aft ⊢ \{pre\} \{l\} Response action \{post\}
coms, p, aft ⊢ \{pre\} c1 :: c2 \{post\}
coms, p, aft ⊢ \{pre\} Choose c1 c2 \{post\}

We tweak fragments by omitting Responses, yielding fewer obligations

**fun**
vcg-fragments' :: (answer, location, question, state) com
⇒ location set
⇒ ( ('answer', 'location', 'question', 'state') com
  × ('answer', 'location', 'question', 'state') loc-comp ) set

where

vcg-fragments' (\{l\} Response action) aft = {}
| vcg-fragments' (\{l\} IF b THEN c FI) aft
  = vcg-fragments' c aft
  ∪ ( \{l\} IF b THEN c' FI, lcond (atC c) aft b ) | c'. True }
| vcg-fragments' (\{l\} IF b THEN c1 ELSE c2 FI) aft
  = vcg-fragments' c2 aft ∪ vcg-fragments' c1 aft
  ∪ ( \{l\} IF b THEN c1' ELSE c2' FI, lcond (atC c1) (atC c2) b ) | c1' c2'. True }
| vcg-fragments' (LOOP DO c OD) aft = vcg-fragments' c (atC c)
| vcg-fragments' (c1 ++ c2) aft = vcg-fragments' c1 aft ∪ vcg-fragments' c2 aft
| vcg-fragments' c aft = {(c, lconst aft)}

abbreviation

vcg-fragments :: ('answer', 'location', 'question', 'state) com
⇒ ( ('answer', 'location', 'question', 'state) com
  × ('answer', 'location', 'question', 'state') loc-comp ) set

where

vcg-fragments c ≡ vcg-fragments' c {}

fun isResponse :: ('answer', 'location', 'question', 'state) com ⇒ bool where
  isResponse (\{l\} Response action) ←→ True
| isResponse - ←→ False

lemma fragments-vcg-fragments':
  \[(c, aft) ∈ fragments c aft' ; ¬isResponse c \] ⇒ (c, aft) ∈ vcg-fragments' c' aft'
by (induct c' arbitrary: aft') auto

lemma vcg-fragments'.fragments:
  vcg-fragments' c' aft' ⊆ fragments c' aft'
by (induct c' arbitrary: aft') (auto 10 0)

lemma VCG-step:
  assumes V: \(\forall p. \forall (c, aft) ∈ vcg-fragments (PGMs sys p). PGMs sys, p, aft ⊢ \{pre\} c \{post\}\n  assumes S: system-step p sh' sh
  assumes R: reachable-state sys sh
  assumes P: pre sh
  shows post sh'
using S

proof cases
  case LocalStep with P show ?thesis
    apply –
    apply (erule decompose-small-step[OF - R])
    apply (frule fragments-basic-com)
    apply (erule basic-com.cases)
    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)+
    done
  next
  case CommunicationStep with P show ?thesis
    apply –
    apply (erule decompose-small-step[OF - R])
    apply (erule decompose-small-step[OF - R])

38
The user sees the conclusion of $V$ for each element of $\text{vcg-fragments}$.

**lemma VCG-step-inv-stable:**

- **assumes** $V$: $\forall p. \forall (c, aft) \in \text{vcg-fragments} (\text{PGMs sys } p). \text{PGMs sys, p, aft }\vdash \{ |I| \} \ c$
- **assumes** $\text{prerun sys }\sigma$
- **shows** $(|I| \leftrightarrow \circ |I|) \ \sigma$
- **apply** (rule alwaysI)
- **apply** clarsimp
- **apply** clarsimp
- **using** assms(2) unfolding 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 pre=I and post=I] V assms(2) prerun-reachable-state
- **apply** blast
- **done**

**lemma VCG:**

- **assumes** $I$: $\forall s. \text{initial-state sys } s \rightarrow I (\{ \text{GST} = s, \text{HST} = []\})$
- **assumes** $V$: $\forall p. \forall (c, aft) \in \text{vcg-fragments} (\text{PGMs sys } p). \text{PGMs sys, p, aft }\vdash \{ |I| \} \ c$
- **shows** $\text{sys }\vdash_{\text{pre}} I$
- **apply** (rule prerun-valid-induct)
- **apply** (clarsimp simp: 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.
notation valid-syn (\{\_\} / - / \{\_\})

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 (\{\_\} / -)

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

lemma vcg-conj:
\[\{I\} c \{Q\}: \{I\} c \{R\} \implies \{I\} c \{Q \land R\}\]
by (cases c) (fastforce elim: vcg-inv intro: vcg.intros)+

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

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

lemma vcg-post-imp:
\[\forall s. Q s \implies R s; \{P\} c \{Q\} \implies \{P\} c \{R\}\]
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\}
shows \{P\} c \{R \implies 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\}
shows \{P \land P\} c \{Q \land 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\}
shows \{P \lor P\} c \{Q \lor 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\}
shows \{P \lor Q\} c \{P \implies 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 $\forall s. P s \Rightarrow \{\{=\} s\} c \{\{\}\}$
  shows $\{P\} c \{Q\}$
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg.intros)+

lemma vcg-lift-comp:
  assumes $f: \forall P. \{\forall s. P (f s :: 'a :: type)\} c$
  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 $g: eq-imp \ f \ g$
  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])

41
apply (rule-tac $x = x$ and $P = \lambda fs. \ f x s$ in $f$[rule-format])
apply simp
apply (frule eq-impD[where $f = f$, OF $g$])
apply simp
apply simp
done

lemma eq-imp-vcg-LST:
assumes $g$: eq-imp $f$ $g$
assumes $f$: $\forall x. P \circ (f x) \circ \text{LST}$ $c$
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 $'a = 'a$])
apply (rule-tac $x = x$ and $P = \lambda fs. \ f x s$ in $f$[rule-format])
apply simp
apply (frule eq-impD[where $f = f$, OF $g$])
apply simp
apply simp
done

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

lemma pres-tuple-vcg-LST:
$(\forall P. \{ P \circ (\lambda s. (f s, g s)) \circ \text{LST} \} \ c)$
$\iff ( (\forall P. \{ P \circ f \circ \text{LST} \} \ c) \land (\forall P. \{ P \circ g \circ \text{LST} \} \ c) )$
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 $P = \lambda fs. P \ \text{fs } (g s)$ for $g$ in vcg-lift-comp; simp)
done

no-notation valid-syn (\{\} / - / \{\})
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''} [λs. 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 first process has also terminated. If \( p_3 \) terminates, then it has \( p_1 \)’s value. This is stronger than Lamport and Schneider’s as we don’t ask that the first process has also terminated.

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

\[
\text{datatype } \text{ex-chname} = \xi 12 \mid \xi 23 \\
\text{type-synonym } \text{ex-val} = \text{nat} \\
\text{type-synonym } \text{ex-ch} = \text{ex-chname} \times \text{ex-val} \\
\text{datatype } \text{ex-loc} = r12 \mid r23 \mid s23 \mid s12 \\
\text{datatype } \text{ex-proc} = p1 \mid p2 \mid p3 \\
\text{type-synonym } \text{ex-pgm} = (\text{unit}, \text{ex-loc}, \text{ex-ch}, \text{ex-val}) \text{ com} \\
\text{type-synonym } \text{ex-pred} = (\text{unit}, \text{ex-loc}, \text{ex-proc}, \text{ex-ch}, \text{ex-val}) \text{ state-pred} \\
\text{type-synonym } \text{ex-state} = (\text{unit}, \text{ex-loc}, \text{ex-proc}, \text{ex-ch}, \text{ex-val}) \text{ system-state} \\
\text{type-synonym } \text{ex-sys} = (\text{unit}, \text{ex-loc}, \text{ex-proc}, \text{ex-ch}, \text{ex-val}) \text{ system} \\
\text{type-synonym } \text{ex-history} = (\text{ex-ch} \times \text{unit}) \text{ list} \\
\]

We further specialise these for our particular example.

\[
\text{primrec } \\
\text{ex-coms :: } \text{ex-proc } \Rightarrow \text{ex-pgm} \\
\text{where} \\
\text{ex-coms } p1 = \{s12\} \xi 12 \text{id} \\
| \text{ex-coms } p2 = \text{LOOP DO } \{r12\} \xi 12 \text{v} \Rightarrow \text{HST} \text{id} \Rightarrow \text{OD} \\
| \text{ex-coms } p3 = \{r23\} \xi 23 \text{v} \Rightarrow \text{HST} \\
\]

Each process starts with an arbitrary initial local state.

\[
\text{abbreviation } \text{ex-init} :: (\text{ex-proc } \Rightarrow \text{ex-val}) \Rightarrow \text{bool} \text{ where} \\
\text{ex-init } \equiv \text{(True)} \\
\]

\[
\text{abbreviation } \text{sys :: } \text{ex-sys} \text{ where} \\
\text{sys } \equiv \{\text{PGMs = ex-coms, INIT = ex-init, FAIR = (True)}\} \\
\]

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.

\[
\text{abbreviation } \text{filter-on-channel :: } \text{ex-chname } \Rightarrow \text{ex-state } \Rightarrow \text{ex-val list } ([\cdot \mid 100\mid 101) \\
\text{where} \\
|ch \equiv \text{map (snd o fst) o filter ((=) ch o fst o fst) o HST} \\
\]

\[
\text{definition } \text{IL :: } \text{ex-pred where} \\
\text{IL = pred-conjoin [ \\
| \text{at } p1 s12 \Rightarrow \text{LIST-NUL} \mid \xi 12 \\
| \text{at } p2 r12 \Rightarrow \mid \xi 12 \mid \xi 23 \\
| \text{at } p2 s23 \Rightarrow \mid \xi 12 = \mid \xi 23 @ (\text{if } \xi 12 \Rightarrow \text{HST}) \Rightarrow \text{OD} \\
| \text{at } p3 r23 \Rightarrow \text{LIST-NUL} \mid \xi 33 \\
| \text{terminated } p3 \Rightarrow \mid \xi 23 = (\text{if } \xi 12 \Rightarrow \text{HST}) \Rightarrow \text{OD} \\
| \text{terminated } p1 \Rightarrow \mid \xi 23 \\
| \text{terminated } p2 \Rightarrow \mid \xi 23 \\
| } \\
\]

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.
\textbf{definition} \textit{Etern-pred} :: \textit{ex-pred} \textbf{where} \\
\textit{Etern-pred} = (\text{terminated} \ p3 \rightarrow (\lambda s. \ s↓\ p1 = s↓\ p3))

Proofs from here down.

\textbf{lemma} \textit{correct-system}: \\
assumes \textit{IL} \ \textit{sh} \\
shows \textit{Etern-pred} \ \textit{sh} \\
using \ \textit{assms} unfolding \textit{Etern-pred-def} \ \textit{IL-def} \ \textbf{by} simp

\textbf{lemma} \textit{IL-p1}: \textit{ex-coms}, \textit{p1}, \textit{lconst} \{\} \vdash \langle IL \rangle \ \langle s12 \rangle \ \langle 12 \rangle\langle \lambda s. s \rangle
\ \textbf{apply} \ \textit{rule vcg.intros} \\
\ \textbf{apply} \ \textit{(rename-tac} \ p′) \\
\ \textbf{apply} \ \textit{(case-tac} \ p′; \ clarsimp simp: \textit{IL-def} at\textit{Ls-def}) \\
\ \textbf{done}

\textbf{lemma} \textit{IL-p2}: \textit{ex-coms}, \textit{p2}, \textit{lconst} \{r12\} \vdash \langle IL \rangle \ \langle s23 \rangle \ \langle 23 \rangle\langle \lambda s. s \rangle
\ \textbf{apply} \ \textit{rule vcg.intros} \\
\ \textbf{apply} \ \textit{(rename-tac} \ p′) \\
\ \textbf{apply} \ \textit{(case-tac} \ p′; \ clarsimp simp: \textit{IL-def}) \\
\ \textbf{done}

\textbf{lemma} \textit{IL}: \textit{sys} \models_{pre} \textit{IL} \\
\ \textbf{apply} \ \textit{rule VCG} \\
\ \textbf{apply} \ \textit{(clarsimp simp: \textit{IL-def} at\textit{Ls-def} dest!: initial-stateD)} \\
\ \textbf{apply} \ \textit{(rename-tac} \ p) \\
\ \textbf{apply} \ \textit{(case-tac} \ p; \ clarsimp simp: IL-p1 IL-p2) \\
\ \textbf{done}

\textbf{lemma} \textit{IL-valid}: \textit{sys} \models \Box [IL] \\
\ \textbf{by} \ \textit{rule valid-prerun-lift[OF IL]}

\section{Example: an unbounded buffer}

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

datatype \textit{ex-chname} = \xi12 \ | \ \xi23 \\
type-synonym \textit{ex-val} = \textit{nat} \\
type-synonym \textit{ex-ls} = \textit{ex-val list} \\
type-synonym \textit{ex-ch} = \textit{ex-chname} \times \textit{ex-val} \\
datatype \textit{ex-loc} = c1 \ | \ r12 \ | \ r23 \ | \ s23 \ | \ s12 \\
datatype \textit{ex-proc} = p1 \ | \ p2 \ | \ p3 \\
type-synonym \textit{ex-pgm} = (\textit{unit}, \textit{ex-loc}, \textit{ex-ch}, \textit{ex-ls}) \com \\
type-synonym \textit{ex-pred} = (\textit{unit}, \textit{ex-loc}, \textit{ex-proc}, \textit{ex-ch}, \textit{ex-ls}) \state-pred \\
type-synonym \textit{ex-state} = (\textit{unit}, \textit{ex-loc}, \textit{ex-proc}, \textit{ex-ch}, \textit{ex-ls}) \state-pred \\
type-synonym \textit{ex-sys} = (\textit{unit}, \textit{ex-loc}, \textit{ex-proc}, \textit{ex-ch}, \textit{ex-ls}) \sys-pred \\
type-synonym \textit{ex-history} = (\textit{ex-ch} \times \textit{unit}) \list

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

\textbf{abbreviation} \ (input) \ \textbf{snoc} :: \ 'a \Rightarrow \ 'a \ \textit{list} \Rightarrow \ 'a \ \textit{list} \ where \ \textbf{snoc} \ x \ \textit{xs} \equiv \ \textit{xs} \ @ \ [x]

\textbf{primrec} \ \textit{ex-coms} :: \ \textit{ex-proc} \Rightarrow \ \textit{ex-pgm} \ \textbf{where} \\
\ \textit{ex-coms} \ p1 = \ \text{LOOP} \ \text{DO} \ !{c1} \ \text{LocalOp} (\lambda s. \ \{\mathbf{snoc} \ x \ \mathit{xs} \ | \ x. \ \mathit{True}\}) \ ; ; \ !{s12} \ \langle 12 \rangle\langle \mathit{last, id} \rangle \ \text{OD} \\
| \ \textit{ex-coms} \ p2 = \ \text{LOOP} \ \text{DO} \ !{r12} \ \langle 12 \rangle\langle \mathit{sno}\mathit{c} \rangle \\
\ \mathbf{IF} \ (\lambda s. \mathit{length} \ s > 0) \ \text{THEN} \ !{s23} \ \langle 23 \rangle\langle \mathit{hd}, \mathit{tl} \rangle \ \text{FI} \\
\ \text{OD}
\begin{verbatim}
| ex-coms p3 = LOOP DO \{r23\} \xi23\&snoc OD

abbreviation ex-init :: (ex-proc \Rightarrow ex-ls) \Rightarrow bool where
  ex-init s \equiv \forall p. s p = []

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

abbreviation
  filter-on-channel :: ex-chname \Rightarrow ex-state \Rightarrow ex-val list (\&- [100] 101)
  where
  \{ch \equiv map (snd \circ fst) \circ filter (\{=\) ch \circ fst \circ fst) \circ HST

definition I-pred :: ex-pred where
  I-pred = pred-conjoin [  
    at p1 c1 \rightarrow \{\xi12 = (\lambda s. s↓ p1)\}  
    , at p1 s12 \rightarrow (\lambda s. length (s↓ p1) > 0 \& butlast (s↓ p1) = (\{\xi12\}) s)  
    , \xi12 \leq (\lambda s. s↓ p1)  
    , \{\xi12 = \{\xi23\& (\lambda s. s↓ p2)\}  
    , at p2 s23 \rightarrow (\lambda s. length (s↓ p2) > 0)  
    , (\lambda s. s↓ p3) = \{\xi23\}
  ]

The local state of p3 is some prefix of the local state of p1.

definition Etern-pred :: ex-pred where
  Etern-pred \equiv \lambda s. s↓ p3 \leq s↓ p1

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

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

lemma p1-s12[intro]:
  ex-coms, p1, lconst \{c1\} \vdash \{I-pred\} \{s12\} \xi12\&(last, id)
apply (rule vcg.intros)
apply (rename-tac p')
apply (case-tac p'; clarsimp)
apply (clarsimp simp: I-pred-def atS-def)
apply (metis Prefix-Order.prefix-snoc append.assoc append-butlast-last-id)
done

lemma p2-s23[intro]:
  ex-coms, p2, lconst \{c1, r12\} \vdash \{I-pred\} \{s23\} \xi12\&(hd, tl)
apply (rule vcg.intros)
apply (rename-tac p')
apply (clause-tac p'; clarsimp)
apply (clarsimp simp: I-pred-def atS-def)
done

lemma p2-pi4[intro]:
  ex-coms, p2, lconst \{s23\} \{c1, r12\} (\lambda s. s \neq []) \vdash \{I-pred\} \{c1\} IF (\lambda s. s \neq []) THEN c' FI
apply (rule vcg.intros)
apply (clarsimp simp: I-pred-def atS-def split: lcond-splits)

46
\end{verbatim}
9 Concluding remarks

Previously Nipkow and Prensa Nieto (1999); Prensa Nieto (2002, 2003) 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 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 $ISABELLE/src/HOL/Hoare_Parallel$. 

47


