CIMP

Peter Gammie

May 27, 2015

Abstract

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

Contents

1 Lifted predicates 1

2 CIMP syntax and semantics 4
   2.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   2.2 Process semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   2.3 System steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   2.4 Assertions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
       2.4.1 Control predicates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
       2.4.2 Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
       2.4.3 Relating reachable states to the initial programs . . . . . . . . . . . 13
   2.5 Simple-minded Hoare Logic/VCG for CIMP . . . . . . . . . . . . . . . . . . 15
       2.5.1 VCG rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
       2.5.2 Cheap non-interference rules . . . . . . . . . . . . . . . . . . . . . . 19

3 One-place buffer example 20

4 Prefix order on lists as order class instance 23

5 Unbounded buffer example 23

6 Concluding remarks 25

References 27

1 Lifted predicates

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

abbreviation (input)
   pred-pair :: ('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ 'a ⇒ 'b × 'c (infixr ⊗ 60) where
\[ a \otimes b \equiv \lambda s. (a, b) \]

**abbreviation (input)**
\[ \text{pred-in} :: (\forall a \Rightarrow b) \Rightarrow (\forall a \Rightarrow \text{set}) \Rightarrow \lambda a \Rightarrow \text{bool} \ (\text{infix in 50}) \text{ where} \]
\[ a \text{ in } A \equiv \lambda s. a \in A_s \]

**abbreviation (input)**
\[ \text{pred-subseteq} :: (\forall a \Rightarrow \text{set}) \Rightarrow (\forall a \Rightarrow \text{set}) \Rightarrow \lambda a \Rightarrow \text{bool} \ (\text{infix subseteq 50}) \text{ where} \]
\[ A \subseteq B \equiv \lambda s. A_s \subseteq B_s \]

**abbreviation (input)**
\[ \text{pred-union} :: (\forall a \Rightarrow \text{set}) \Rightarrow (\forall a \Rightarrow \text{set}) \Rightarrow \lambda a \Rightarrow \text{set} \ (\text{infix union 65}) \text{ where} \]
\[ a \cup b \equiv \lambda s. a_s \cup b_s \]

**abbreviation (input)**
\[ \text{pred-diff} :: (\forall a \Rightarrow \text{set}) \Rightarrow (\forall a \Rightarrow \text{set}) \Rightarrow \lambda a \Rightarrow \text{set} \ (\text{infixr diff 65}) \text{ where} \]
\[ a - b \equiv \lambda s. a_s - b_s \]

**abbreviation (input)**
\[ \text{pred-comp} :: ((\forall b \Rightarrow c) \Rightarrow (\forall a \Rightarrow d)) \Rightarrow (\forall b \Rightarrow \lambda a \Rightarrow c) \Rightarrow (\forall a \Rightarrow d) \ (\text{infixl } \circ 55) \text{ where} \]
\[ f \circ g \equiv \lambda s. f (\lambda b. g b s) \]

**abbreviation (input)**
\[ \text{pred-app} :: (\forall b \Rightarrow (\forall a \Rightarrow c)) \Rightarrow (\forall a \Rightarrow \text{set}) \Rightarrow \lambda a \Rightarrow \text{c} \ (\text{infixl } \triangledown 100) \text{ where} \]
\[ f \triangledown g \equiv \lambda s. f (g s) \]

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

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

**abbreviation (input)**
\[ \text{pred-lt} :: (\forall a \Rightarrow \text{ord}) \Rightarrow (\forall a \Rightarrow \text{b}) \Rightarrow \lambda a \Rightarrow \text{bool} \ (\text{infix lt 40}) \text{ where} \]
\[ a < b \equiv \lambda s. a_s < b_s \]

**abbreviation (input)**
\[ \text{pred-and} :: (\forall a \Rightarrow \text{bool}) \Rightarrow (\forall a \Rightarrow \text{b}) \Rightarrow \lambda a \Rightarrow \text{bool} \ (\text{infixr and 35}) \text{ where} \]
\[ a \land b \equiv \lambda s. a_s \land b_s \]

**abbreviation (input)**
\[ \text{pred-or} :: (\forall a \Rightarrow \text{bool}) \Rightarrow (\forall a \Rightarrow \text{b}) \Rightarrow \lambda a \Rightarrow \text{bool} \ (\text{infixr or 30}) \text{ where} \]
\[ a \lor b \equiv \lambda s. a_s \lor b_s \]
pred-not :: ('a ⇒ bool) ⇒ 'a ⇒ bool (not - \[40\] \[40\]) where
not a ≡ λs. ¬a s

abbreviation (input)
pred-imp :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr imp 25) where
a imp b ≡ λs. a s → b s

abbreviation (input)
pred-iff :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr iff 25) where
a iff b ≡ λs. a s ↔ b s

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

abbreviation (input)
pred-conjoin :: ('a ⇒ bool) list ⇒ 'a ⇒ bool where
pred-conjoin xs ≡ foldr (op and) xs ⟨True⟩

abbreviation (input)
pred-singleton :: 'a ⇒ 'b ⇒ 'a ⇒ 'b set where
pred-singleton x ≡ λs. {x s}

abbreviation (input)
pred-empty :: ('a ⇒ 'b set) ⇒ 'a ⇒ bool (empty - \[40\] \[40\]) where
empty a ≡ λs. a s = {}

abbreviation (input)
pred-map-empty :: ('a ⇒ ('b ⇒ 'c option)) ⇒ 'a ⇒ bool (map'-empty - \[40\] \[40\]) where
map-empty a ≡ λs. a s = Map.empty

abbreviation (input)
pred-list-null :: ('a ⇒ 'b list) ⇒ 'a ⇒ bool (list'-null - \[40\] \[40\]) where
list-null a ≡ λs. a s = []

abbreviation (input)
pred-null :: ('a ⇒ 'b option) ⇒ 'a ⇒ bool (null - \[40\] \[40\]) where
null a ≡ λs. a s = None

abbreviation (input)
pred-ex :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (binder EXS 10) where
EXS x. P x ≡ λs. ∃x. P x s

abbreviation (input)
pred-all :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (binder ALLS 10) where
ALLS x. P x ≡ λs. ∀x. P x s
abbreviation (input)

pred-If :: (′a ⇒ bool) ⇒ (′a ⇒ ′b) ⇒ (′a ⇒ ′b) ⇒ ′a ⇒ ′b ((If · Then · Else ·)) [0, 0, 10] 10

where If P Then x Else y ≡ λs. if P s then x s else y s

2 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 (§2.5), 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 (§2.2), which we give in the style of structural operational semantics (SOS) as popularised by Plotkin (2004). The semantics of a complete system (§2.3) 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.

2.1 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 (§2.2) is annotated with a ′location, which we use in our assertions (§2.4.1). These locations need not be unique, though in practice they likely will be.

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

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

type-synonym ′s bexp = ′s ⇒ bool

datatype (′answer, ′location, ′question, ′state) com

= Request ′location ′state ⇒ ′question ′answer ⇒ ′state ⇒ ′state set
(,[], Request - [0, 70, 70] 71)
| Response ′location ′question ⇒ ′state ⇒ (′state × ′answer) set
(,[], Response - [0, 70] 71)
| LocalOp ′location ′state ⇒ ′state set
(,[], LocalOp - [0, 70] 71)
| Cond1 ′location ′state bexp (′answer, ′location, ′question, ′state) com
( [], IF - THEN - FI [0, 0] 71)
| Cond2 ′location ′state bexp (′answer, ′location, ′question, ′state) com
We provide a one-armed conditional as it is the common form and avoids the need to discover
label for an internal SKIP and/or trickier proofs about the VCG.
In contrast to classical process algebras, we have local state and distinct send and receive
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 (Mil-
ner 1980, §2.8)). Intuitively the sender asks a 'question with a Request command, which
upon rendezvous with a receiver's Response command receives an 'answer. The 'question
is a deterministic function of the sender’s local state, whereas a receiver can respond non-
deterministically. 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. Internal choice can be recovered in com-
bination with local operations (see Milner (1980, §2.3)).
We abbreviate some common commands: SKIP is a local operation that does nothing, and
the floor brackets simplify deterministic LocalOps. We also adopt some syntax magic from
Makarius’s Hoare and Multiquote theories in the Isabelle/HOL distribution.

\textbf{abbreviation} \texttt{SKIP-syn \{\-\}/ \texttt{SKIP \(70\) where}
\[ \{\-\} \texttt{SKIP} \equiv \{\-\} \texttt{LocalOp (\(\lambda s. \{s\}\))} \]

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

\textbf{syntax}
\begin{itemize}
  \item \texttt{-quote \:: \'(a \Rightarrow \b) \equiv \b (\leftarrow [0] 1000)}
  \item \texttt{-antiquote \:: \'(a \Rightarrow \b) \equiv \b (\leftarrow [1000] 1000)}
  \item \texttt{-Assign \:: \texttt{location \(\Rightarrow\) idt \(\Rightarrow\) \texttt{b} \(\Rightarrow\) \texttt{answer, location, question, state} \texttt{com (\texttt{\{-\} [-)} \(\leftarrow / \)} [0, \ 0, \ 70] 71)}
  \item \texttt{-NonDetAssign \:: \texttt{location \(\Rightarrow\) idt \(\Rightarrow\) \texttt{b} \(\Rightarrow\) \texttt{answer, location, question, state} \texttt{com (\texttt{\{-\}} \(\leftarrow / \)} [0, \ 0, \ 70] 71)}
\end{itemize}

\textbf{abbreviation} \texttt{(input) NonDetAssign :: \texttt{location \(\Rightarrow\) \texttt{\{(val \(\Rightarrow\) \texttt{val} \(\Rightarrow\) \texttt{state \(\Rightarrow\) \texttt{state}} \Rightarrow \texttt{state \(\Rightarrow\) \texttt{val set)}} \Rightarrow \{\texttt{answer, location, question, state} \} \texttt{com where} \]
\[ \texttt{NonDetAssign \ l \ upd \ es \equiv \{\-\} \texttt{LocalOp (\(\lambda s. \{ \texttt{upd (e) s | e \in es s \}\})} \]

5
2.2 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.

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

= ('answer', 'location', 'question', 'state) com list × 'state

**inductive**

**small-step ::** ('answer', 'location', 'question', 'state) local-state

⇒ ('answer', 'question') seq-label

⇒ ('answer', 'location', 'question', 'state) local-state ⇒ bool (≪ - ≫ [55, 0, 56]

**where**

Request: [\(\alpha = action s\); s' ∈ val \(\beta\) s ] ⇒ (\{ l \} Request action val # cs, s) → ≪\(\alpha\), \(\beta\)≫ (cs, s')

| Response: (s', \(\beta\)) ∈ action \(\alpha\) s ⇒ (\{ l \} Response action # cs, s) → ≫\(\alpha\), ≪\(\beta\) (cs, s')

| LocalOp: s' ∈ R s ⇒ (\{ l \} LocalOp R # cs, s) → \(\tau\) (cs, s')

| Cond1True: b s ⇒ (\{ l \} IF b THEN c FI # cs, s) → \(\tau\) (c # cs, s)

| Cond1False: ¬b s ⇒ (\{ l \} IF b THEN c FI # cs, s) → \(\tau\) (cs, s)

| Cond2True: b s ⇒ (\{ l \} IF b THEN c1 ELSE c2 FI # cs, s) → \(\tau\) (c1 # cs, s)

| Cond2False: ¬b s ⇒ (\{ l \} IF b THEN c1 ELSE c2 FI # cs, s) → \(\tau\) (c2 # cs, s)

| Loop: (c # LOOP DO c OD # cs, s) → \(\alpha\) (cs', s') ⇒ (LOOP DO c OD # cs, s) → \(\alpha\) (cs', s')

| While: b s ⇒ (\{ l \} WHILE b DO c OD # cs, s) → \(\tau\) (c # \{ l \} WHILE b DO c OD # cs, s)

| WhileFalse: ¬b s ⇒ (\{ l \} WHILE b DO c OD # cs, s) → \(\tau\) (cs, s)

6
\[ | \text{Seq:} \ (c_1 \neq c_2 \neq cs, s) \rightarrow_{\alpha} (cs', s') \Rightarrow (c_1:: c_2:: cs, s) \rightarrow_{\alpha} (cs', s') \]

\[ | \text{Choose1:} \ (c_1 \neq cs, s) \rightarrow_{\alpha} (cs', s') \Rightarrow (c_1 \sqcup c_2 \neq cs, s) \rightarrow_{\alpha} (cs', s') \]

\[ | \text{Choose2:} \ (c_2 \neq cs, s) \rightarrow_{\alpha} (cs', s') \Rightarrow (c_1 \sqcup c_2 \neq cs, s) \rightarrow_{\alpha} (cs', s') \]

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

**abbreviation cPGM :: ('answer, 'location, 'question, 'state) local-state \Rightarrow ('answer, 'location, 'question, 'state) com list where**

\[ cPGM \equiv \text{fst} \]

**abbreviation cLST :: ('answer, 'location, 'question, 'state) local-state \Rightarrow 'state where**

\[ cLST s \equiv \text{snd} s(\text{proof})\langle\text{proof}\rangle \]

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 identify 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**

\[ = ('answer, 'location, 'question, 'state) com \Rightarrow ('answer, 'location, 'question, 'state) com \]

**inductive-set**

\[ \text{ctxt} ::= ( ('answer, 'location, 'question, 'state) ctxt } \times ( ('answer, 'location, 'question, 'state) com } \Rightarrow ( 'answer, 'location, 'question, 'state) com \text{ list}) ) \text{ set where} \]

\[ C\text{-Hole:} \ (\text{id}, ([)]) \in \text{ctxt} \]

\[ C\text{-Loop:} \ (E, fctxt) \in \text{ctxt} \Rightarrow (\lambda t. \text{LOOP DO E t OD}, \lambda t. fctxt t @ [\text{LOOP DO E t OD}]) \in \text{ctxt} \]

\[ C\text{-Seq:} \ (E, fctxt) \in \text{ctxt} \Rightarrow (\lambda t. E t:: u, \lambda t. fctxt t @ [u]) \in \text{ctxt} \]

\[ C\text{-Choose1:} \ (E, fctxt) \in \text{ctxt} \Rightarrow (\lambda t. E t \sqcup u, fctxt) \in \text{ctxt} \]

\[ C\text{-Choose2:} \ (E, fctxt) \in \text{ctxt} \Rightarrow (\lambda t. u \sqcup E t, fctxt) \in \text{ctxt} \]

**inductive**

\[ \text{basic-com :: ('answer, 'location, 'question, 'state) com } \Rightarrow \text{bool where} \]

\[ \text{basic-com } (\text{[]}) \text{ Request action val} \]

\[ \text{basic-com } (\text{[]}) \text{ Response action} \]

\[ \text{basic-com } (\text{[]}) \text{ LocalOp R} \]

\[ \text{basic-com } (\text{[]}) \text{ IF b THEN c FI} \]

\[ \text{basic-com } (\text{[]}) \text{ IF b THEN c1 ELSE c2 FI} \]

\[ \text{basic-com } (\text{[]}) \text{ WHILE b DO c OD} \]

We can decompose a small step into a context and a basic-com.
decompose-com :: ('answer, 'location, 'question, 'state) com
⇒ ( ('answer, 'location, 'question, 'state) com
× ( ('answer, 'location, 'question, 'state) ctx
× ( ('answer, 'location, 'question, 'state) com ⇒ ( 'answer, 'location, 'question, 'state) com list ) ) set

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

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

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

theorem context-decompose:
s →α s' ↔ (∃(c, ictxt, fctxt) ∈ decomposeLS s.
cPGM s = ictxt c # tl (cPGM s)
∧ basic-com c
∧ (c ≠ fctxt c @ tl (cPGM s), cLST s) →α s')(proof)

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.

2.3 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 (2009).

type-synonym ('answer, 'location, 'proc, 'question, 'state) global-state
= 'proc ⇒ ('answer, 'location, 'question, 'state) local-state

type-synonym ('proc, 'state) local-states
= 'proc ⇒ 'state

An execution step of the overall system is either any enabled internal τ step of any process,
or a communication rendezvous between two processes. For the latter to occur, a Request action must be enabled in process $p1$, and a Response action in (distinct) process $p2$, where the request/response labels $\alpha$ and $\beta$ (semantically) match.

We also track global communication history here to support assertional reasoning (see §2.4).

**type-synonym** $(\text{'answer}', \text{'question'})$ event $\equiv \text{'question'} \times \text{'answer'}$

**type-synonym** $(\text{'answer}', \text{'question'})$ history $\equiv (\text{'answer}', \text{'question'})$ event list

**type-synonym** $(\text{'answer}', \text{'location'}, \text{'proc'}, \text{'question'}, \text{'state'})$ system-state

$\equiv (\text{'answer}', \text{'location'}, \text{'proc'}, \text{'question'}, \text{'state'})$ global-state

$\times (\text{'answer}', \text{'question'})$ history

**inductive-set**

\[\text{system-step} :: ( (\text{'answer}', \text{'ls'}, \text{'proc'}, \text{'question'}, \text{'state'}) \text{ system-state} \\
\times (\text{'answer}', \text{'ls'}, \text{'proc'}, \text{'question'}, \text{'state'}) \text{ system-state} ) \text{ set} \]

**where**

* **LocalStep:** $[ s \xrightarrow{} \text{ls'}; s' = s(p := \text{ls'}); h' = h ] \implies (s, h, (s', h')) \in \text{system-step}$

* **CommunicationStep:** $[ s \xrightarrow{} \alpha, \beta \xrightarrow{} \text{ls'}; s \xrightarrow{} \beta, \alpha \xrightarrow{} \text{ls'}; s1 \neq s2; \\
\quad s' = s(p1 := \text{ls'}1; p2 := \text{ls'}2); h' = h @ [(\alpha, \beta)] ] \implies (s, h, (s', h')) \in \text{system-step}$

**abbreviation**

\[\text{system-step-syn} :: ( (\text{'answer}', \text{'ls'}, \text{'proc'}, \text{'question'}, \text{'state'}) \text{ system-state} \\
\Rightarrow (\text{'answer}', \text{'ls'}, \text{'proc'}, \text{'question'}, \text{'state'}) \text{ system-state} \Rightarrow \text{bool} (- s\Rightarrow - [55, 56] 55) \]

**where**

$sh \Rightarrow sh' \equiv (sh, sh') \in \text{system-step}$

**abbreviation**

\[\text{system-steps-syn} :: ( (\text{'answer}', \text{'ls'}, \text{'proc'}, \text{'question'}, \text{'state'}) \text{ system-state} \\
\Rightarrow (\text{'answer}', \text{'ls'}, \text{'proc'}, \text{'question'}, \text{'state'}) \text{ system-state} \Rightarrow \text{bool} (- s\Rightarrow^* - [55, 56] 55) \]

**where**

$sh \Rightarrow^* sh' \equiv (sh, sh') \in \text{system-step}^*$

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.

### 2.4 Assertions

We now develop a technique for showing that a CIMP system satisfies a single global invariant, following Lamport (1980); Lamport and Schneider (1984) (and the later Lamport (2002)) and closely related work by Cousot and Cousot (1980) and Levin and Gries (1981), which 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 (see his webpage bibliography). See de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001) for an extended discussion of some of this.

Achieving the right level of abstraction is a bit fiddly; we want to avoid revealing too much of the program text as it executes. Intuitively we wish to expose the processes’s present control locations and local states only. Lamport avoids these issues by only providing an axiomatic semantics for his language.

2.4.1 Control predicates

Following Lamport (1980), we define the \( \text{at} \) predicate, which holds of a process when control resides at that location. Due to non-determinism processes can be \( \text{at} \) 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.

\[
\text{primrec}
\begin{align*}
\text{atC} \::&\:: \text{('answer, 'location, 'question, 'state) com ⇒ 'location ⇒ bool} \\
\text{where} &\text{atC} (\{l\}'\text{ Request action val}) = (\lambda l. \ l = l') \\
&\text{atC} (\{l\}'\text{ Response action}) = (\lambda l. \ l = l') \\
&\text{atC} (\{l\}'\text{ LocalOp f}) = (\lambda l. \ l = l') \\
&\text{atC} (\{l\}'\text{ IF - THEN - FI}) = (\lambda l. \ l = l') \\
&\text{atC} (\{l\}'\text{ IF - THEN - ELSE - FI}) = (\lambda l. \ l = l') \\
&\text{atC} (\{l\}'\text{ WHILE - DO - OD}) = (\lambda l. \ l = l') \\
&\text{atC} (\text{LOOP DO c OD}) = \text{atC} c \\
&\text{atC} (c1;; c2) = \text{atC} c1 \\
&\text{atC} (c1 \sqcup c2) = (\text{atC} c1 \text{ or atC} c2)
\end{align*}
\]

\[
\text{primrec} \text{atL} :: \text{('answer, 'location, 'question, 'state) com list ⇒ 'location ⇒ bool} \quad \text{where}
\begin{align*}
\text{atL} [] &\triangleq (\text{False}) \\
\text{atL} (c \# -) &\triangleq \text{atC} c
\end{align*}
\]

\[
\text{abbreviation} \text{atLS} :: \text{('answer, 'location, 'question, 'state) local-state ⇒ 'location ⇒ bool} \quad \text{where}
\begin{align*}
\text{atLS} &\equiv \lambda s. \text{atL} (\text{cPGM s})(\text{proof})(\text{proof})
\end{align*}
\]

We define predicates over communication histories and a projection of global states. These are uncurried to ease composition.

\[
\text{type-synonym} ('\text{location, proc, state}) \text{ pred-local-state}
\begin{align*}
= &\ ('\text{proc} \Rightarrow (('\text{location ⇒ bool}) \times '\text{state})
\end{align*}
\]

\[
\text{record} ('\text{answer, location, proc, question, state}) \text{ pred-state} =
\]

local-states :: (’location, ’proc, ’state) pred-local-state
hist :: (’answer, ’question) history

type-synonym (’answer, ’location, ’proc, ’question, ’state) pred
   = (’answer, ’location, ’proc, ’question, ’state) pred-state ⇒ bool

definition mkP :: (’answer, ’location, ’proc, ’question, ’state) system-state ⇒ (’answer, ’location, ’proc, ’question, ’state) pred-state ⇒ bool where
   mkP ≡ λ(s, h). ⌣ local-states = λp. case s p of (cs, ps) ⇒ (atL cs, ps), hist = h ⌣{proof}

We provide the following definitions to the end-user.
AT maps process names to a predicate that is true of locations where control for that process resides. The abbreviation at shuffles its parameters; the former is simplifier-friendly and eta-reduced, while the latter is convenient for writing assertions.
definition AT :: (’answer, ’location, ’proc, ’question, ’state) pred-state ⇒ ’proc ⇒ ’location ⇒ bool where
   AT ≡ λs p l. fst (local-states s p) l

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

Often we wish to talk about control residing at one of a set of locations. This stands in for, and generalises, the in predicate of Lamport (1980).
definition atS :: ’proc ⇒ ’location set ⇒ (’answer, ’location, ’proc, ’question, ’state) pred where
   atS ≡ λp ls s. ∃l∈ls. at p l s

A process is terminated if it not at any control location.
abbreviation terminated :: ’proc ⇒ (’answer, ’location, ’proc, ’question, ’state) pred where
   terminated p s ≡ ∀l. ¬at p l s

The LST operator (written as a postfix ↓) projects the local states of the processes from a pred-state, i.e. it discards control location information.
Conversely the LSTP operator lifts predicates over local states into predicates over pred-state. Levin and Gries (1981, §3.6) call such predicates universal assertions.
type-synonym (’proc, ’state) state-pred
   = (’proc, ’state) local-states ⇒ bool

definition LST :: (’answer, ’location, ’proc, ’question, ’state) pred-state ⇒ (’proc, ’state) local-states ⇒ bool where
   s↓ ≡ snd ◦ local-states s

abbreviation (input) LSTP :: (’proc, ’state) state-pred ⇒ (’answer, ’location, ’proc, ’question, ’state) pred where
   LSTP P ≡ λs. P (LST s)
By default we ask the simplifier to rewrite $atS$ using ambient $AT$ information.

**Lemma** $atS$-state-cong[cong]:

\[
\begin{array}{l}
[ AT s p = AT s' p ] \implies atS p ls s \iff atS p ls s' \\
\end{array}
\]

(proof)

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

**Lemma** $atS$-simps:

\[
\begin{array}{l}
\neg atS p \{\} s \\
\begin{array}{l}
atS p \{l\} s \iff at p l s \\
[ at p l s; l \in ls ] \implies \begin{array}{l}
atS p ls s \iff True \\
(\forall l. at p l s \implies l \notin ls) \implies \begin{array}{l}
atS p ls s \iff False \\
\end{array}
\end{array}
\end{array}
\]

(proof)

**Lemma** $atS$-mono:

\[
\begin{array}{l}
[ atS p ls s; ls \subseteq ls' ] \implies atS p ls' s \\
\end{array}
\]

(proof)

**Lemma** $atS$-un:

\[
\begin{array}{l}
atS p (l \cup l') s \iff atS p l s \lor atS p l' s \\
\end{array}
\]

(proof)

### 2.4.2 Invariants

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 (§2.3).

**Type-Synonym** ('answer', 'location', 'proc', 'question', 'state) programs

= 'proc \Rightarrow ('answer', 'location', 'question', 'state) com

**Type-Synonym** ('answer', 'location', 'proc', 'question', 'state) system

= ('answer', 'location', 'proc', 'question', 'state) programs

\times ('proc', 'state) state-pred

**Definition**

initial-states :: ('answer', 'location', 'proc', 'question', 'state) system

\Rightarrow ('answer', 'location', 'proc', 'question', 'state) global-state set

where

initial-states sys \equiv

\{ f . (\forall p. cPGM (f p) = [fst sys p]) \land snd sys (cLST \circ f) \}

**Definition**

reachable-states :: ('answer', 'location', 'proc', 'question', 'state) system

\Rightarrow ('answer', 'location', 'proc', 'question', 'state) system-state set

where

reachable-states sys \equiv system-step* \ " (initial-states sys \times \{\} )"

The following is a slightly more convenient induction rule for the set of reachable states.
lemma reachable-states-system-step-induct[consumes 1,  
     case-names init LocalStep CommunicationStep]:
assumes r: \( (s, h) \in \text{reachable-states sys} \)
assumes i: \( \bigwedge s. s \in \text{initial-states sys} \implies P \ s \ ]
assumes l: \( \bigwedge s h \ ls'. p. \ [ (s, h) \in \text{reachable-states sys}; P \ s h; s p \to_{\tau} ls'] \]
assumes c: \( \bigwedge s h \ ls1' \ ls2' \ p1 \ p2 \ \alpha \ \beta. \ [ (s, h) \in \text{reachable-states sys}; P \ s h; \]
\[ s p1 \to_{\prec} \beta > \ ls1'; s p2 \to_{\succ} \alpha, \beta < \ ls2'; p1 \neq p2 \]
\[ \implies P (s(p1 := ls1', p2 := ls2')) (h @ [(\alpha, \beta)]) \]
shows \( P \ s h\{\text{proof}\}\{\text{proof}\}\{\text{proof}\} \)

2.4.3 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 states to the programs in the initial states. The \textit{fragments} function decomposes the program into statements that can be directly executed (§2.2). We also compute the locations we could be at after executing that statement as a function of the process’s local state.

We could support Lamport’s \textit{after} control predicate with more syntactic analysis of this kind.

\begin{verbatim}
fun extract-cond :: ('answer, 'location, 'question, 'state) com \Rightarrow 'state bexp
where
  extract-cond (\{l\} IF b THEN c FI) = b
| extract-cond (\{l\} IF b THEN c1 ELSE c2 FI) = b
| extract-cond (\{l\} WHILE b DO c OD) = b
| extract-cond c = (False)

type-synonym ('answer, 'location, 'question, 'state) loc-comp
  = ('answer, 'location, 'question, 'state) com
  \Rightarrow 'state \Rightarrow 'location \Rightarrow bool

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

definition lcond :: ('location \Rightarrow bool) \Rightarrow ('location \Rightarrow bool)
  \Rightarrow ('answer, 'location, 'question, 'state) loc-comp
where
  lcond lp \ lpc = if extract-cond c s then lp else lp\{\text{proof}\}\{\text{proof}\}

fun fragments :: ('answer, 'location, 'question, 'state) com
  \Rightarrow ('location \Rightarrow bool)
  \Rightarrow ('answer, 'location, 'question, 'state) loc-comp \times ('answer, 'location, 'question, 'state) loc-comp \ set
where
  fragments (\{l\} IF b THEN c FI) ls
    = \{ (\{l\} IF b THEN c' FI, lcond (atC c) ls | c'. True \} \)
\end{verbatim}
We show that taking system steps preserves fragments.

Eliding the bodies of $\text{IF}$ and $\text{WHILE}$ statements yields smaller (but equivalent) proof obligations.

We show that taking system steps preserves fragments.

**Lemma**  reachable-states-fragmentsLS:

**Assumes** $(s, h) \in \text{reachable-states} \: \text{sys}$

**Shows** $\text{fragmentsLS} \: (s \: p) \subseteq \text{fragments} \: (\text{fst} \: \text{sys} \: p) \: (\text{False})$.

Decomposing a compound command preserves fragments too.

**Fun**

$\text{extract-inner-locations} :: (\langle \text{answer, } \text{location, } \text{question, } \text{state} \rangle \: \text{com} \\
\Rightarrow (\langle \text{answer, } \text{location, } \text{question, } \text{state} \rangle \: \text{com list} \\
\Rightarrow (\langle \text{answer, } \text{location, } \text{question, } \text{state} \rangle \: \text{loc-comp} ) \: \text{set}$

**Where**

$\text{extract-inner-locations} \: (\text{IF} \: b \: \text{THEN} \: c1 \: \text{ELSE} \: c2 \: \text{FI}) \: \text{cs} = \text{lcond} \: (\text{atC} \: c1) \: (\text{atL} \: \text{cs})$

$\text{extract-inner-locations} \: (\text{IF} \: b \: \text{THEN} \: c1 \: \text{ELSE} \: c2 \: \text{FI}) \: \text{cs} = \text{lcond} \: (\text{atC} \: c1) \: (\text{atC} \: c2)$

$\text{extract-inner-locations} \: (\text{LOOP} \: \text{DO} \: \text{c} \: \text{OD}) \: \text{cs} = \text{lconst} \: (\text{atC} \: \text{c})$

$\text{extract-inner-locations} \: (\text{WHILE} \: b \: \text{DO} \: \text{c} \: \text{OD}) \: \text{cs} = \text{lcond} \: (\text{atC} \: \text{c}) \: (\text{atL} \: \text{cs})$

**Lemma** small-step-extract-inner-locations:
The headline lemma 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** \( s \rightarrow_{\alpha} ps' \)
- **Assumes** \((s, h) \in \text{reachable-states sys}\)
- **Obtains** \(c \ c s \ l s'\)

  where
  - \( (c, l s') \in \text{fragments (fst sys } p \langle \text{False} \rangle\)
  - and \( \text{basic-com } c\)
  - and \( \forall l. \text{atC } c l \rightarrow \text{atLS } (s p) l\)
  - and \( l s' \cdot c (c \text{LST} (s p)) = \text{atLS } ps'\)
  - and \( (c \# cs, c \text{LST} (s p)) \rightarrow_{\alpha} ps'(\text{proof})\)

Reasoning with reachable-states-system-step-induct and decompose-small-step is quite tedious. We provide a very simple VCG that generates friendlier local proof obligations.

## 2.5 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 (§2.4.2). This is somewhat in the spirit of ?.

Note that this approach is not compositional: it consults the original system to find matching communicating pairs, and 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.

### inductive

- \(\text{vcg} :: \langle \text{answer, location, proc, question, state} \rangle \text{ programs}\)
  - \( \Rightarrow \text{proc}\)
  - \( \Rightarrow \langle \text{answer, location, question, state} \rangle \text{ loc-comp}\)
  - \( \Rightarrow \langle \text{answer, location, proc, question, state} \rangle \text{ pred}\)
  - \( \Rightarrow \langle \text{answer, location, question, state} \rangle \text{ com}\)
  - \( \Rightarrow \langle \text{answer, location, proc, question, state} \rangle \text{ pred}\)
  - \( \Rightarrow \text{bool (\$, \$, \$ \| / \{\}, / \{\})}\)

**where**

- Request: \[ \forall \text{aft' action' } s \ p s' \ p s' l' \beta \ s' \ p'. \]
  - \[ \text{at' } s \cdot (\text{\$l' Request action' val}) \text{ \langle \text{False} \rangle}; \text{ \ p} \neq p'; \]
  - \[ ps' \in \text{val \ } \beta \text{ (LST } s \ p); \text{ (p's', } \beta) \in \text{action' (action (LST } s \ p) \langle \text{LST } s \ p \rangle); \]
  - \[ \text{at } p \ l s; \text{ at } p' l' s; \]
  - \[ \text{AT } s' = (\text{AT } s)(p := \text{aft \ (\$l Request action val}) \text{ (LST } s \ p),\]
We abbreviate invariance with one-sided validity syntax.

\[ \text{pgms, p, aft} \models \{\text{pre}\} \{l\} \text{ LocalOp f } \{\text{post}\} \]

\[ \text{Cond2: } L s \text{ s', } \{\text{pre}\} \{l\} \text{ IF } b \text{ THEN } t \text{ ELSE } e \text{ FI } \{\text{post}\} \]

\[ \text{While: } L s \text{ s', } \{\text{pre}\} \{l\} \text{ WHILE } b \text{ DO c OD } \{\text{post}\} \]

We abbreviate invariance with one-sided validity syntax.

**abbreviation valid-inv \((-,-,-|-{-},-)\ where**

\[ \text{pgms, p, aft} \models \{l\} \text{ c } \equiv \text{pgms, p, aft} \models \{l\} \{\text{proof}\}\{\text{proof}\} \]

We tweak fragments by omitting Responses, yielding fewer obligations.

**fun**

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

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

**where**
\[
\text{vcg-fragments}' (\emptyset) \text{ Response action) ls = } \{\}
\]
\[
\text{vcg-fragments}' (\begin{smallmatrix}
\text{IF} \ b \ \text{THEN} \ c1 \ \text{FI}
\end{smallmatrix}) \text{ ls}
\]
\[
\Rightarrow \text{vcg-fragments'} \ c \text{ ls}
\]
\[
\bigcup \{ (\begin{smallmatrix}
\text{IF} \ b \ \text{THEN} \ c' \ \text{FI}, \ \text{lcond} \ \text{atC} \ c \ \text{ls}) \ | c' \end{smallmatrix}) \ | \text{True} \}
\]
\[
\text{vcg-fragments'} (\begin{smallmatrix}
\text{IF} \ b \ \text{THEN} \ c1 \ \text{ELSE} \ c2 \ \text{FI}
\end{smallmatrix}) \text{ ls}
\]
\[
= \text{vcg-fragments'} \ c2 \text{ ls} \cup \text{vcg-fragments'} \ c1 \text{ ls}
\]
\[
\bigcup \{ (\begin{smallmatrix}
\text{IF} \ b \ \text{THEN} \ c1' \ \text{ELSE} \ c2' \ \text{FI}, \ \text{lcond} \ \text{atC} \ c1 \ (\text{atC} c2)) \ | c1' \ c2'. \ \text{True} \}
\]
\[
\text{vcg-fragments'} (\langle \begin{smallmatrix}
\text{LOOP} \ DO \ c \ \text{OD}
\end{smallmatrix} \rangle) \text{ ls} = \text{vcg-fragments'} \ c \ (\text{atC} c)
\]
\[
\text{vcg-fragments'} (\langle \begin{smallmatrix}
\text{WHILE} \ b \ \text{DO} \ c \ \text{OD}
\end{smallmatrix} \rangle) \text{ ls}
\]
\[
= \text{vcg-fragments'} \ c \ (\begin{smallmatrix}
\text{\lambda l.} \ l = l' \end{smallmatrix}) \cup \{ (\begin{smallmatrix}
\text{WHILE} \ b \ \text{DO} \ c' \ \text{OD}, \ \text{lcond} \ \text{atC} \ c \ \text{ls}) \ | c'. \ \text{True} \}
\]
\[
\text{vcg-fragments'} (\langle \begin{smallmatrix}
\text{c1} :: \text{c2}
\end{smallmatrix} \rangle) \text{ ls} = \text{vcg-fragments'} \ c2 \text{ ls} \cup \text{vcg-fragments'} \ c1 \ (\text{atC} c2)
\]
\[
\text{vcg-fragments'} (\langle \begin{smallmatrix}
\text{c1} \cup \text{c2}
\end{smallmatrix} \rangle) \text{ ls} = \text{vcg-fragments'} \ c1 \text{ ls} \cup \text{vcg-fragments'} \ c2 \text{ ls}
\]
\[
\text{vcg-fragments'} \ c \text{ ls} = \{(c, \ \text{lconst \ ls})\}
\]

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

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

**lemma VCG:**
\[
\text{assumes} \ \ R: \ s \in \text{reachable-states sys}
\]
\[
\text{assumes} \ \ I: \ \forall s \in \text{initial-states sys}. \ I (\text{mkP} \ (s, []))
\]
\[
\text{assumes} \ \ V: \ \forall p. \ \forall (c, \ \text{afts}) \in \text{vcg-fragments} \ (\text{fst sys} \ p). \ ((\text{fst sys}, p, \ \text{afts} |\rangle | l) \ c)
\]
\[
\text{shows} \ I (\text{mkP} \ s)\langle \text{proof} \rangle
\]

**2.5.1 VCG rules**

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

**context**
\[
\text{fixes pgms :: (\text{'answer}, \text{'location}, \text{'proc}, \text{'question}, \text{'state}) programs}
\]
\[
\text{fixes} \ p :: \ 'proc
\]
\[
\text{fixes afts :: (\text{'answer}, \text{'location}, \text{'question}, \text{'state}) loc-comp}
\]

**begin**

**abbreviation**
\[
\text{valid-syn} :: (\text{'answer}, \text{'location}, \text{'proc}, \text{'question}, \text{'state}) \text{ pred}
\]
\[
\Rightarrow (\text{'answer}, \text{'location}, \text{'question}, \text{'state}) \text{ com}
\]
\[
\Rightarrow (\text{'answer}, \text{'location}, \text{'proc}, \text{'question}, \text{'state}) \text{ pred } \Rightarrow \text{bool where}
\]

17
\[
\text{valid-syn} \quad P \ c \ Q \equiv \text{pgms}, \ p, \ afts \models \{P\} \ c \ \{Q\}
\]

\textbf{notation} \quad \text{valid-syn} (\{\cdot\}/\{-\}/\{-\})

\textbf{abbreviation}

\text{valid-inv-syn} :: (\'answer, \'location, \'proc, \'question, \'state) \text{ pred} \\
\Rightarrow (\'answer, \'location, \'question, \'state) \text{ com} \Rightarrow \text{bool} \text{ where}

\text{valid-inv-syn} \ P \ c \equiv \{P\} \ c \ \{P\}

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

\textbf{lemma} \quad \text{vcg-True:}
\quad \{P\} \ c \ \{(\text{True})\}
\langle \text{proof} \rangle

\textbf{lemma} \quad \text{vcg-conj:}
\quad \{I\} \ c \ \{Q\}; \ {I}\ c \ \{R\} \Rightarrow \{I\} \ c \ \{Q \ and \ R\}
\langle \text{proof} \rangle

\textbf{lemma} \quad \text{vcg-pre-imp:}
\quad \{\land s. \ P \ s \Rightarrow \ Q \ s; \ {Q}\ c \ \{R\}\} \Rightarrow \{P\} \ c \ \{R\}
\langle \text{proof} \rangle

\textbf{lemmas} \quad \text{vcg-pre} = \text{vcg-pre-imp}[\text{rotated}]

\textbf{lemma} \quad \text{vcg-post-imp:}
\quad \{\land s. \ Q \ s \Rightarrow \ R \ s; \ {P}\ c \ \{Q\}\} \Rightarrow \{P\} \ c \ \{R\}
\langle \text{proof} \rangle

\textbf{lemma} \quad \text{vcg-prop[ intro]:}
\quad \{\langle P\rangle\} \ c
\langle \text{proof} \rangle

\textbf{lemma} \quad \text{vcg-drop-imp:}
\quad \text{assumes} \ {P}\ c \ \{Q\}
\quad \text{shows} \ {P}\ c \ \{R \ \text{imp} \ Q\}
\langle \text{proof} \rangle

\textbf{lemma} \quad \text{vcg-conj-lift:}
\quad \text{assumes} \ x: \ {P}\ c \ \{Q\}
\quad \text{assumes} \ y: \ {P}\ c \ \{Q\}
\quad \text{shows} \ \{P \ and \ P'\} \ c \ \{Q \ and \ Q'\}
\langle \text{proof} \rangle

\textbf{lemma} \quad \text{vcg-disj-lift:}
\quad \text{assumes} \ x: \ {P}\ c \ \{Q\}
\quad \text{assumes} \ y: \ {P}\ c \ \{Q\}
\quad \text{shows} \ \{P \ or \ P'\} \ c \ \{Q \ or \ Q'\}
\langle \text{proof} \rangle
\textbf{lemma} \textit{vcg-imp-lift}:
\begin{itemize}
  \item assumes \{P\} c \{\text{not } P\}
  \item assumes \{Q\} c \{Q\}
  \item shows \{P or Q\} c \{P imp Q\}
\end{itemize}
\langle proof \rangle

\textbf{lemma} \textit{vcg-ex-lift}:
\begin{itemize}
  \item assumes \(\forall x. \{P x\} c \{Q x\}\)
  \item shows \(\forall s. \exists x. \{P x s\} c \{\exists s. \exists x. Q x s\}\)
\end{itemize}
\langle proof \rangle

\textbf{lemma} \textit{vcg-all-lift}:
\begin{itemize}
  \item assumes \(\forall x. \{P x\} c \{Q x\}\)
  \item shows \(\forall s. \exists x. \{P x s\} c \{\exists s. \exists x. Q x s\}\)
\end{itemize}
\langle proof \rangle

\textbf{lemma} \textit{vcg-name-pre-state}:
\begin{itemize}
  \item assumes \(\forall s. P s \Rightarrow \{\text{op } s\} c \{Q\}\)
  \item shows \(\{P\} c \{Q\}\)
\end{itemize}
\langle proof \rangle

\textbf{lemma} \textit{vcg-lift-comp}:
\begin{itemize}
  \item assumes \(f : \forall P. \{\lambda s. P (f s :: 'a :: type)\} c\)
  \item assumes \(P : \forall x. \{Q x\} c \{P x\}\)
  \item shows \(\{\lambda s. Q (f s) s\} c \{\lambda s. P (f s) s\}\)
\end{itemize}
\langle proof \rangle

\subsection{2.5.2 Cheap non-interference rules}

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

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

\textbf{definition} \textit{eq-imp} :: \((a \Rightarrow 'b ⇒ 'c) ⇒ ('b ⇒ 'e) ⇒ \text{bool where}
\begin{align*}
\text{eq-imp } f g & \equiv (\forall s s'. (\forall x. f x s = f x s') \rightarrow (g s = g s'))
\end{align*}

\textbf{lemma} \textit{eq-impD}:
\[ \llbracket\text{eq-imp } f g; \forall x. f x s = f x s' \rrbracket \Rightarrow g s = g s' \]
\langle proof \rangle

\textbf{lemma} \textit{eq-imp-vcg}:
\begin{itemize}
  \item assumes \(g : \text{eq-imp } f g\)
  \item assumes \(f : \forall x. \{P \circ (f x)\} c\)
  \item shows \(\{P \circ g\} c\)
\end{itemize}
lemma eq-imp-vcg-LST:
  assumes g: eq-imp f g
  assumes f: \( \forall x. P \circ (f x) \circ LST \) c
  shows \( P \circ g \circ LST \) c
  ⟨proof⟩

lemma eq-imp-fun-upd:
  assumes g: eq-imp f g
  assumes f: \( \forall x. f x (s(...)) = f x s \)
  shows g (s(...)) = g s
  ⟨proof⟩

lemma curry-forall-eq:
  \( (\forall f. P f) = (\forall f. P (\text{split } f)) \)
  ⟨proof⟩

lemma pres-tuple-vcg:
  \( (\forall P. \{P \circ (\lambda s. (f s, g s))\} c) \leftrightarrow ((\forall P. \{P \circ f\} c) \land (\forall P. \{P \circ g\} c)) \)
  ⟨proof⟩

lemma pres-tuple-vcg-LST:
  \( (\forall P. \{P \circ (\lambda s. (f s, g s)) \circ LST\} c) \leftrightarrow ((\forall P. \{P \circ f \circ LST\} c) \land (\forall P. \{P \circ g \circ LST\} c)) \)
  ⟨proof⟩

lemmas conj-explode = conj-imp-eq-imp-imp

end ⟨ML⟩

3 One-place buffer example

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
  Receive :: 'location \Rightarrow 'channel \Rightarrow ('val \Rightarrow 'state \Rightarrow 'state)
  \Rightarrow (\text{unit}, 'location, 'channel \times 'val, 'state) \text{ com} (\{\|-\}/ \Rightarrow-)

where
  \( \{l\} \text{ ch}\_f s \equiv \{l\} \text{ Response } (\lambda \text{quest } s. \text{ if } \text{fst } \text{quest} = \text{ch} \text{ then } \{(f \text{ snd } \text{quest}) s, ()\} \text{ else } \{\}) \)

20
abbreviation

Send :: 'location ⇒ 'channel ⇒ ('state ⇒ 'val)
       ⇒ (unit, 'location, 'channel × 'val, 'state) com (\-\- / \-\-)
where
\{l\} ch< f = \{l\} Request (\λ s. (ch, f s)) (\λ ans s. \{s\})

We further specialise these for our particular example.

abbreviation

Receive :: 'location ⇒ 'channel ⇒ (unit, 'location, 'channel × 'state, 'state) com (\-\- / \-\-)
where
\{l\} ch⇒ = \{l\} ch⇒ id

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

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

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

primrec

ex-pgms :: ex-proc ⇒ ex-pgm
where
ex-pgms p1 = \{s12\} ξ12<
| ex-pgms p2 = LOOP DO \{r12\} ξ12<;; \{s23\} ξ23< OD
| ex-pgms p3 = \{r23\} ξ23><

Each process starts with an arbitrary initial local state.

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

abbreviation ex-system :: ex-system where
ex-system = (ex-pgms, ex-init)

PeteG: I don’t understand how Lamport and Schneider justify their invariants.
The following adapts Kai Engelhardt’s, from his notes titled Proving an Asynchronous Message Passing Program Correct, 2011. The history variable tracks the causality of the system,
which I feel is missing in Lamport’s treatment. We tack on Lamport’s invariant so we can establish Etern-pred.

**abbreviation**

filter-on-channel :: ex-chname ⇒ ex-history ⇒ ex-val list

**where**

filter-on-channel ch ≡ map (snd ◦ fst) ◦ filter (op = ch ◦ fst ◦ fst)

**definition** Ip1-0 :: ex-pred where

Ip1-0 ≡ at p1 s12 imp (λs. filter-on-channel ξ12 (hist s) = [])

**definition** Ip1-1 :: ex-pred where

Ip1-1 ≡ terminated p1 imp (λs. filter-on-channel ξ12 (hist s) = [LST s p1])

**definition** Ip2-0 :: ex-pred where

Ip2-0 ≡ at p2 r12 imp (λs. filter-on-channel ξ12 (hist s) = filter-on-channel ξ23 (hist s))

**definition** Ip2-1 :: ex-pred where

Ip2-1 ≡ at p2 s23 imp (λs. filter-on-channel ξ12 (hist s) = filter-on-channel ξ23 (hist s))

@ [LST s p2] ∧ LST s p1 = LST s p2

**definition** Ip3-0 :: ex-pred where

Ip3-0 ≡ at p3 r23 imp (λs. filter-on-channel ξ23 (hist s) = [])

**definition** Ip3-1 :: ex-pred where

Ip3-1 ≡ terminated p3 imp (λs. filter-on-channel ξ23 (hist s) = [LST s p2] ∧ LST s p1 = LST s p3)

**definition** I-pred :: ex-pred where

I-pred ≡ pred-conjoin [ Ip1-0, Ip1-1, Ip2-0, Ip2-1, Ip3-0, Ip3-1 ]

**lemmas** I-defs = Ip1-0-def Ip1-1-def Ip2-0-def Ip2-1-def Ip3-0-def Ip3-1-def

If process three terminates, then it has process one’s value. This is stronger than Lamport and Schneider’s as we don’t ask that the first process has also terminated.

**definition** Etern-pred :: ex-pred where

Etern-pred ≡ terminated p3 imp (λs. LST s p1 = LST s p3)

Proofs from here down.

**lemma** correct-system:

I-pred sh ⧦ Etern-pred sh

(proof)

**lemma** p1: ex-pgms, p1, lconst (False) ⧦ [I-pred] [s12] ξ12 ◦ λs. s

(proof)

**lemma** p2-1: ex-pgms, p2, lconst (λl. l = r12) ⧦ [I-pred] [s23] ξ23 ◦ λs. s

(proof)
lemma $(s, h) \in \text{reachable-states ex-system} \implies I\text{-pred} (\text{mkP} (s, h))$

\langle \text{proof} \rangle

4 Prefix order on lists as order class instance

theory Prefix-Order
imports Sublist
begin

instantiation list :: (type) order
begin

definition $(xs :: \text{a list}) \leq ys \equiv \text{prefixeq} xs ys$
definition $(xs :: \text{a list}) < ys \equiv xs \leq ys \land \neg (ys \leq xs)$

instance \langle \text{proof} \rangle
end

lemmas prefixI [intro?] = prefixeqI [folded less-eq-list-def]
lemmas prefixE [elim?] = prefixeqE [folded less-eq-list-def]
lemmas strict-prefixI' [intro?] = prefixI' [folded less-list-def]
lemmas strict-prefixE' [elim?] = prefixE' [folded less-list-def]
lemmas strict-prefixI [intro?] = prefixI [folded less-list-def]
lemmas strict-prefixE [elim?] = prefixE [folded less-list-def]
theorems Nil-prefix [iff] = Nil-prefixeq [folded less-eq-list-def]
theorems prefix-Nil [simp] = prefixeq-Nil [folded less-eq-list-def]
lemmas prefix-snoc [simp] = prefixeq-snoc [folded less-eq-list-def]
lemmas Cons-prefix-Cons [simp] = prefixeq-Cons [folded less-eq-list-def]
lemmas same-prefix-prefix [simp] = same-prefixeq-prefixeq [folded less-eq-list-def]
lemmas same-prefix-nil [iff] = same-prefixeq-nil [folded less-eq-list-def]
lemmas prefix-prefix [simp] = prefixeq-prefixeq [folded less-eq-list-def]
theorems prefix-Cons = prefixeq-Cons [folded less-eq-list-def]
theorems prefix-length-le = prefixeq-length-le [folded less-eq-list-def]
lemmas strict-prefix-simps [simp, code] = prefix-simps [folded less-list-def]
lemmas not-prefix-induct [consumes 1, case-names Nil Neq Eq] =
not-prefixeq-induct [folded less-eq-list-def]

end
\langle \text{proof} \rangle

5 Unbounded buffer example

This is more literally Kai’s example from his notes titled Proving an Asynchronous Message Passing Program Correct, 2011.
datatype ex-chname = \xi 12 \mid \xi 23

type-synonym ex-val = nat

type-synonym ex-ls = ex-val list

type-synonym ex-ch = ex-chname \times ex-val

datatype ex-loc = \pi 4 \mid \pi 5 \mid c1 \mid r12 \mid r23 \mid s23 \mid s12

datatype ex-proc = p1 \mid p2 \mid p3

type-synonym ex-pgm = (unit, ex-loc, ex-ch, ex-ls) com

type-synonym ex-state = (unit, ex-loc, ex-proc, ex-ch, ex-ls) global-state

type-synonym ex-system = (unit, ex-loc, ex-proc, ex-ch, ex-ls) system

type-synonym ex-history = (ex-ch \times unit) list

FIXME a bit fake: the local state for the producer process contains all values produced.

primrec ex-pgms :: ex-proc \Rightarrow ex-pgm where
  ex-pgms p1 = LOOP DO \{c1\} LocalOp (\lambda s. \{xs \otimes [x] | x. True\}) ;; \{s12\} \xi 12 \text{-} \text{last OD}
  | ex-pgms p2 = LOOP DO \{r12\} \xi 12 \text{-} \lambda x. xs \otimes [x] \ |
    \sqcup \{\pi 4\} IF (\lambda s. \text{length } s > 0) THEN \{s23\} Request (\lambda s. (\xi 23, \text{hd } s))
  (\lambda s. \{t l s\}) FI
  OD
  | ex-pgms p3 = LOOP DO \{r23\} \xi 23 \text{-} (\lambda x. xs \otimes [x]) OD

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

abbreviation ex-system :: ex-system where
  ex-system \equiv (ex-pgms, ex-init)

definition filter-on-channel :: ex-chname \Rightarrow ex-history \Rightarrow ex-val list where
  filter-on-channel ch \equiv \text{map} (\text{snd} \circ \text{fst}) \circ \text{filter} (\text{op} = \text{ch} \circ \text{fst} \circ \text{fst})

lemma filter-on-channel-simps [simp]:
  filter-on-channel ch [] = []
  filter-on-channel ch (xs \otimes ys) = filter-on-channel ch xs \otimes filter-on-channel ch ys
  filter-on-channel ch (((ch', v), \text{resp} \# \text{vals}) = (if ch' = ch \then [v] \text{else []}) \otimes filter-on-channel ch vals
(proof)

definition Ip1-0 :: ex-pred where
  Ip1-0 \equiv \lambda s. \text{at } p1 \ c1 \ s \rightarrow \text{filter-on-channel } \xi 12 \text{ (hist } s) = s \downarrow p1

definition Ip1-1 :: ex-pred where
  Ip1-1 \equiv \lambda s. \text{at } p1 \ s12 \ s \rightarrow \text{length } (s \downarrow p1) > 0 \land \text{butlast } (s \downarrow p1) = \text{filter-on-channel } \xi 12 \text{ (hist } s)

definition Ip1-2 :: ex-pred where
  Ip1-2 \equiv \lambda s. \text{filter-on-channel } \xi 12 \text{ (hist } s) \leq s \downarrow p1

definition Ip2-0 :: ex-pred where
\( \text{Ip2-0} \equiv \lambda s. \text{filter-on-channel} \xi_{12} (\text{hist } s) = \text{filter-on-channel} \xi_{23} (\text{hist } s) @ s \downarrow p2 \)

definition \( \text{Ip2-1} :: \text{ex-pred} \) where
\( \text{Ip2-1} \equiv \lambda s. \text{at } p2 s23 s \rightarrow \text{length} (s \downarrow p2) > 0 \)

definition \( \text{Ip3-0} :: \text{ex-pred} \) where
\( \text{Ip3-0} \equiv \lambda s. s \downarrow p3 = \text{filter-on-channel} \xi_{23} (\text{hist } s) \)

definition \( \text{I-pred} :: \text{ex-pred} \) where
\( \text{I-pred} \equiv \text{pred-conjoin} \left[ \text{Ip1-0}, \text{Ip1-1}, \text{Ip1-2}, \text{Ip2-0}, \text{Ip2-1}, \text{Ip3-0} \right] \)

lemmas \( \text{I-defs} = \text{I-pred-def} \text{Ip1-0-def} \text{Ip1-1-def} \text{Ip1-2-def} \text{Ip2-0-def} \text{Ip2-1-def} \text{Ip3-0-def} \)

The local state of \( p3 \) is some prefix of the local state of \( p1 \).

definition \( \text{Etern-pred} :: \text{ex-pred} \) where
\( \text{Etern-pred} \equiv \lambda s. s \downarrow p3 \leq s \downarrow p1 \)

lemma correct-system:
\( \text{I-pred } s \implies \text{Etern-pred } s \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \langle \text{proof} \rangle \)

lemma \( s \in \text{reachable-states ex-system} \implies \text{I-pred } (\text{mkP } s) \langle \text{proof} \rangle \)

6 Concluding remarks

Previously Nipkow and Prensa Nieto (1999); Prensa Nieto (2002, 2003)\(^2\) have developed the classical Owicki/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); de Roever et al. (2001) 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 de Roever et al. (2001, §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 (§2.5).

References


\(^2\)The theories are in \texttt{ISABELLE/src/HOL/HoareParallel}.


26


