CIMP

Peter Gammie

October 10, 2017

Abstract

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

Contents

1 Lifted predicates

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

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

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

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

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

abbreviation (*input*)
$pred\text{-}diff :: ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow 'b\ set$ (**infixr** *diff 65*) **where**
$a\ diff\ b \equiv \lambda s.\ a\ s - b\ s$

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

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

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

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

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

abbreviation (*input*)
$pred\text{-}and :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** *and 35*) **where**
$a\ and\ b \equiv \lambda s.\ a\ s \wedge b\ s$

abbreviation (*input*)
$pred\text{-}or :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** *or 30*) **where**
$a\ or\ b \equiv \lambda s.\ a\ s \vee b\ s$

abbreviation (*input*)
$pred\text{-}not :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (*not - [40] 40*) **where**
$not\ a \equiv \lambda s.\ \neg a\ s$

abbreviation (*input*)
  $pred\text{-}imp :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** *imp 25*) **where**
  $a\ imp\ b \equiv \lambda s.\ a\ s \longrightarrow b\ s$

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

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

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

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

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

abbreviation (*input*)
  $pred\text{-}map\text{-}empty :: ('a \Rightarrow ('b \Rightarrow 'c\ option)) \Rightarrow 'a \Rightarrow bool\ (map'\text{-}empty\ \text{-}\ [40]\ 40)$ **where**
  $map\text{-}empty\ a \equiv \lambda s.\ a\ s = Map.empty$

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

abbreviation (*input*)
  $pred\text{-}null :: ('a \Rightarrow 'b\ option) \Rightarrow 'a \Rightarrow bool\ (null\ \text{-}\ [40]\ 40)$ **where**
  $null\ a \equiv \lambda s.\ a\ s = None$

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

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

abbreviation (*input*)
  $pred\text{-}If :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b\ ((If\ (\text{-})/\ Then\ (\text{-})/\ Else\ (\text{-}))$

$[0,\ 0,\ 10]\ 10)$
where *If P Then x Else y* $\equiv \lambda s.$ *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* $\Rightarrow$ *'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*
  = *Request* *'location* *'state* $\Rightarrow$ *'question* *'answer* $\Rightarrow$ *'state* $\Rightarrow$ *'state set* ({|-|} *Request* - - $[0,\ 70,\ 70]\ 71)$
  | *Response* *'location* *'question* $\Rightarrow$ *'state* $\Rightarrow$ (*'state* $\times$ *'answer*) *set* ({|-|} *Response* - $[0,\ 70]\ 71)$
  | *LocalOp* *'location* *'state* $\Rightarrow$ *'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*
  (*'answer*, *'location*, *'question*, *'state*) *com* ({|-|} *IF* -/ *THEN* -/ *ELSE* -/ *FI* $[0,\ 0,\ 0]\ 71)$

| *Loop* ('*answer*, '*location*, '*question*, '*state*) *com* (*LOOP DO -/ OD* [*0*] *71*)
| *While* '*location* '*state bexp* ('*answer*, '*location*, '*question*, '*state*) *com* ({|-|} *WHILE -/ DO -/ OD* [*0*, *0*, *0*] *71*)
| *Seq* ('*answer*, '*location*, '*question*, '*state*) *com*
('*answer*, '*location*, '*question*, '*state*) *com* (**infixr** ;; *69*)
| *Choose* ('*answer*, '*location*, '*question*, '*state*) *com*
('*answer*, '*location*, '*question*, '*state*) *com* (**infixl** ⊔ *68*)

We provide a one-armed conditional as it is the common form and avoids the need to discover a label for an internal *SKIP* and/or trickier proofs about the VCG.

In contrast to classical process algebras, we have local state and distinct send and receive actions. These provide an interface to Isabelle/HOL's datatypes that avoids the need for binding (ala the π-calculus of Milner (1989)) or large non-deterministic sums (ala CCS (Milner 1980, §2.8)). Intuitively the 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 combination with local operations (see Milner (1980, §2.3)).

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

abbreviation *SKIP-syn* ({|-|}/ *SKIP* *70*) **where**
{|*l*|} *SKIP* ≡ {|*l*|} *LocalOp* (λ*s*. {*s*})

abbreviation (*input*) *DetLocalOp* :: '*location* ⇒ ('*state* ⇒ '*state*)
⇒ ('*answer*, '*location*, '*question*, '*state*) *com* ({|-|} ⌊-⌋) **where**
{|*l*|} ⌊*f*⌋ ≡ {|*l*|} *LocalOp* (λ*s*. {*f s*})

syntax
-quote :: '*b* ⇒ ('*a* ⇒ '*b*) (≪-≫ [*0*] *1000*)
-antiquote :: ('*a* ⇒ '*b*) ⇒ '*b* (´- [*1000*] *1000*)
-Assign :: '*location* ⇒ *idt* ⇒ '*b* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com* (({|-|} ´- :=/ -) [*0*, *0*, *70*] *71*)
-NonDetAssign :: '*location* ⇒ *idt* ⇒ '*b set* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com* (({|-|} ´- :∈/ -) [*0*, *0*, *70*] *71*)

abbreviation (*input*) *NonDetAssign* :: '*location* ⇒ (('*val* ⇒ '*val*) ⇒ '*state* ⇒ '*state*) ⇒ ('*state* ⇒ '*val set*)
⇒ ('*answer*, '*location*, '*question*, '*state*) *com* **where**
NonDetAssign l upd es ≡ {|*l*|} *LocalOp* (λ*s*. { *upd* ⟨*e*⟩ *s* |*e*. *e* ∈ *es s* })

translations
{|*l*|} ´*x* := *e* => *CONST DetLocalOp l* ≪´(*-update-name x* (λ-. *e*))≫

$\{\!|l|\!\}$ ´x :∈ es => $CONST\ NonDetAssign\ l\ (\text{-}update\text{-}name\ x)\ \ll es \gg$

⟨*ML*⟩

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 *SKIP*s 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* ($\text{-} \rightarrow_{\text{-}} \text{-}$ [55, 0, 56] 55)
where
 Request: $[\![\ \alpha = action\ s;\ s' \in val\ \beta\ s\ ]\!] \Longrightarrow (\{\!|l|\!\}\ Request\ action\ val\ \#\ cs,\ s) \rightarrow_{\ll\alpha,\ \beta\gg} (cs,\ s')$
| *Response*: $(s',\ \beta) \in action\ \alpha\ s \Longrightarrow (\{\!|l|\!\}\ Response\ action\ \#\ cs,\ s) \rightarrow_{\gg\alpha,\ \beta\ll} (cs,\ s')$

| *LocalOp*: $s' \in R\ s \Longrightarrow (\{\!|l|\!\}\ LocalOp\ R\ \#\ cs,\ s) \rightarrow_{\tau} (cs,\ s')$

| *Cond1True*: $b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c\ FI\ \#\ cs,\ s) \rightarrow_{\tau} (c\ \#\ cs,\ s)$
| *Cond1False*: $\neg b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c\ FI\ \#\ cs,\ s) \rightarrow_{\tau} (cs,\ s)$

| *Cond2True*: $b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI\ \#\ cs,\ s) \rightarrow_{\tau} (c1\ \#\ cs,\ s)$
| *Cond2False*: $\neg b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI\ \#\ cs,\ s) \rightarrow_{\tau} (c2\ \#\ cs,\ s)$

| *Loop*: $(c\ \#\ LOOP\ DO\ c\ OD\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s') \Longrightarrow (LOOP\ DO\ c\ OD\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s')$

| *While*: $b\ s \Longrightarrow (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ s) \rightarrow_{\tau} (c\ \#\ \{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ s)$
| *WhileFalse*: $\neg\ b\ s \Longrightarrow (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ s) \rightarrow_{\tau} (cs,\ s)$

| *Seq*: $(c1\ \#\ c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s') \Longrightarrow (c1;;\ c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s')$

| *Choose1*: $(c1 \# cs, s) \rightarrow_\alpha (cs', s') \Longrightarrow (c1 \sqcup c2 \# cs, s) \rightarrow_\alpha (cs', s')$
| *Choose2*: $(c2 \# cs, s) \rightarrow_\alpha (cs', s') \Longrightarrow (c1 \sqcup c2 \# cs, s) \rightarrow_\alpha (cs', s')$

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* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com list* **where**
cPGM ≡ *fst*

abbreviation *cLST* :: ('*answer*, '*location*, '*question*, '*state*) *local-state* ⇒ '*state* **where**
cLST s ≡ *snd s*⟨*proof*⟩⟨*proof*⟩

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* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com*

inductive-set
ctxt :: (('*answer*, '*location*, '*question*, '*state*) *ctxt*
× (('*answer*, '*location*, '*question*, '*state*) *com* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com list*)) *set*
where
C-Hole: $(id, \langle[]\rangle) \in ctxt$
| *C-Loop*: $(E, fctxt) \in ctxt \Longrightarrow (\lambda t.\ LOOP\ DO\ E\ t\ OD, \lambda t.\ fctxt\ t\ @\ [LOOP\ DO\ E\ t\ OD]) \in ctxt$
| *C-Seq*: $(E, fctxt) \in ctxt \Longrightarrow (\lambda t.\ E\ t;;\ u, \lambda t.\ fctxt\ t\ @\ [u]) \in ctxt$
| *C-Choose1*: $(E, fctxt) \in ctxt \Longrightarrow (\lambda t.\ E\ t \sqcup u, fctxt) \in ctxt$
| *C-Choose2*: $(E, fctxt) \in ctxt \Longrightarrow (\lambda t.\ u \sqcup E\ t, fctxt) \in ctxt$

inductive
basic-com :: ('*answer*, '*location*, '*question*, '*state*) *com* ⇒ *bool*
where
basic-com (⦃*l*⦄ *Request action val*)
| *basic-com* (⦃*l*⦄ *Response action*)
| *basic-com* (⦃*l*⦄ *LocalOp R*)
| *basic-com* (⦃*l*⦄ *IF b THEN c FI*)
| *basic-com* (⦃*l*⦄ *IF b THEN c1 ELSE c2 FI*)
| *basic-com* (⦃*l*⦄ *WHILE b DO c OD*)

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

fun
decompose-com :: ('*answer*, '*location*, '*question*, '*state*) *com*
⇒ (('*answer*, '*location*, '*question*, '*state*) *com*

× ('*answer*, '*location*, '*question*, '*state*) *ctxt*
× (('*answer*, '*location*, '*question*, '*state*) *com* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com list*)) *set*
where
decompose-com (*LOOP DO c1 OD*) = { (*c*, λ*t*. *LOOP DO ictxt t OD*, λ*t*. *fctxt t* @ [*LOOP DO ictxt t OD*]) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c1* }
| *decompose-com* (*c1*;; *c2*) = { (*c*, λ*t*. *ictxt t*;; *c2*, λ*t*. *fctxt t* @ [*c2*]) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c1* }
| *decompose-com* (*c1* ⊔ *c2*) = { (*c*, λ*t*. *ictxt t* ⊔ *c2*, *fctxt*) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c1* }
∪ { (*c*, λ*t*. *c1* ⊔ *ictxt t*, *fctxt*) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c2* }
| *decompose-com c* = {(*c*, *id*, ⟨[]⟩)}

definition
decomposeLS :: ('*answer*, '*location*, '*question*, '*state*) *local-state*
⇒ (('*answer*, '*location*, '*question*, '*state*) *com*
× (('*answer*, '*location*, '*question*, '*state*) *com* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com*)
× (('*answer*, '*location*, '*question*, '*state*) *com* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com list*)) *set*
where
decomposeLS s ≡ *case cPGM s of c* # - ⇒ *decompose-com c* | - ⇒ {}
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
theorem *context-decompose*:
$s \rightarrow_\alpha s' \longleftrightarrow$ (∃(*c*, *ictxt*, *fctxt*) ∈ *decomposeLS s*.
cPGM s = *ictxt c* # *tl* (*cPGM s*)
∧ *basic-com c*
∧ (*c* # *fctxt c* @ *tl* (*cPGM s*), *cLST s*) $\rightarrow_\alpha s'$)⟨*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 α and β (semantically) match.

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

type-synonym $('answer, 'question)\ event = 'question \times 'answer$
type-synonym $('answer, 'question)\ history = ('answer, 'question)\ event\ list$

type-synonym $('answer, 'location, 'proc, 'question, 'state)\ system\text{-}state$
$= ('answer, 'location, 'proc, 'question, 'state)\ global\text{-}state$
$\times ('answer, 'question)\ history$

inductive-set
$system\text{-}step :: (\ ('answer, 'ls, 'proc, 'question, 'state)\ system\text{-}state$
$\times ('answer, 'ls, 'proc, 'question, 'state)\ system\text{-}state\ )\ set$
where
$LocalStep: [\![\ s\ p \rightarrow_{\tau} ls';\ s' = s(p := ls');\ h' = h\ ]\!] \Longrightarrow ((s, h), (s', h')) \in system\text{-}step$
$|\ CommunicationStep: [\![\ s\ p1 \rightarrow_{\ll\alpha, \beta\gg} ls1';\ s\ p2 \rightarrow_{\gg\alpha, \beta\ll} ls2';\ p1 \neq p2;$
$s' = s(p1 := ls1', p2 := ls2');\ h' = h\ @\ [(\alpha, \beta)]\ ]\!] \Longrightarrow ((s, h), (s', h')) \in$
$system\text{-}step$

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

abbreviation
$system\text{-}steps\text{-}syn :: ('answer, 'ls, 'proc, 'question, 'state)\ system\text{-}state$
$\Rightarrow ('answer, 'ls, 'proc, 'question, 'state)\ system\text{-}state \Rightarrow bool\ (\text{-}\ s{\Rightarrow}^*\ \text{-}\ [55,$
$56]\ 55)$
where
$sh\ s{\Rightarrow}^*\ sh' \equiv (sh, sh') \in system\text{-}step^*$

In classical process algebras matching communication actions yield τ 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)[1], 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.

primrec
  $atC :: ('answer, 'location, 'question, 'state)\ com \Rightarrow 'location \Rightarrow bool$
where
  $atC\ (\{\!|l'|\!\}\ Request\ action\ val) = (\lambda l.\ l = l')$
| $atC\ (\{\!|l'|\!\}\ Response\ action) = (\lambda l.\ l = l')$
| $atC\ (\{\!|l'|\!\}\ LocalOp\ f) = (\lambda l.\ l = l')$
| $atC\ (\{\!|l'|\!\}\ IF - THEN - FI) = (\lambda l.\ l = l')$
| $atC\ (\{\!|l'|\!\}\ IF - THEN - ELSE - FI) = (\lambda l.\ l = l')$
| $atC\ (\{\!|l'|\!\}\ WHILE - DO - OD) = (\lambda l.\ l = l')$
| $atC\ (LOOP\ DO\ c\ OD) = atC\ c$
| $atC\ (c1;;\ c2) = atC\ c1$
| $atC\ (c1 \sqcup c2) = (atC\ c1\ or\ atC\ c2)$

primrec $atL :: ('answer, 'location, 'question, 'state)\ com\ list \Rightarrow 'location \Rightarrow bool$ **where**
  $atL\ [] = \langle False \rangle$
| $atL\ (c\ \#\ \text{-}) = atC\ c$

abbreviation $atLS :: ('answer, 'location, 'question, 'state)\ local\text{-}state \Rightarrow 'location \Rightarrow bool$ **where**
  $atLS \equiv \lambda s.\ atL\ (cPGM\ s)\langle proof\rangle\langle proof\rangle$

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

type-synonym $('location, 'proc, 'state)\ pred\text{-}local\text{-}state$
  $= 'proc \Rightarrow (('location \Rightarrow bool) \times 'state)$

record $('answer, 'location, 'proc, 'question, 'state)\ pred\text{-}state =$
  $local\text{-}states :: ('location, 'proc, 'state)\ pred\text{-}local\text{-}state$

[1] Manna and Pnueli (1995) also develop a theory of locations. I think Lamport attributes control predicates to Owicki in her PhD thesis (under Gries). I did not find a treatment of procedures. Manna and Pnueli (1991) observe that a set notation for spreading assertions over sets of locations reduces clutter significantly.

$hist :: ('answer, 'question)\ history$

type-synonym $('answer, 'location, 'proc, 'question, 'state)\ pred$
$= ('answer, 'location, 'proc, 'question, 'state)\ pred\text{-}state \Rightarrow bool$

definition $mkP :: ('answer, 'location, 'proc, 'question, 'state)\ system\text{-}state \Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ pred\text{-}state$ **where**
$mkP \equiv \lambda(s, h).\ (\!|\ local\text{-}states = \lambda p.\ case\ s\ p\ of\ (cs, ps) \Rightarrow (atL\ cs, ps),\ hist = h\ |\!)\langle proof\rangle$

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 $AT :: ('answer, 'location, 'proc, 'question, 'state)\ pred\text{-}state \Rightarrow 'proc \Rightarrow 'location \Rightarrow bool$ **where**
$AT \equiv \lambda s\ p\ l.\ fst\ (local\text{-}states\ s\ p)\ l$

abbreviation $at :: 'proc \Rightarrow 'location \Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ pred$ **where**
$at\ p\ l\ s \equiv AT\ s\ p\ l$

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 $atS :: 'proc \Rightarrow 'location\ set \Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ pred$ **where**
$atS \equiv \lambda p\ ls\ s.\ \exists\, l{\in}ls.\ at\ p\ l\ s$

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

abbreviation $terminated :: 'proc \Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ pred$ **where**
$terminated\ p\ s \equiv \forall\, l.\ \neg at\ p\ l\ s$

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 $('proc, 'state)\ state\text{-}pred$
$= ('proc, 'state)\ local\text{-}states \Rightarrow bool$

definition $LST :: ('answer, 'location, 'proc, 'question, 'state)\ pred\text{-}state$
$\Rightarrow ('proc, 'state)\ local\text{-}states\ (\text{-}\!\downarrow\ [1000]\ 1000)$ **where**
$s\!\downarrow\ \equiv snd \circ local\text{-}states\ s$

abbreviation $(input)\ LSTP :: ('proc, 'state)\ state\text{-}pred$
$\Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ pred$ **where**
$LSTP\ P \equiv \lambda s.\ P\ (LST\ s)$

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

lemma *atS-state-cong*[*cong*]:
$\llbracket$ $AT\ s\ p = AT\ s'\ p$ $\rrbracket \Longrightarrow atS\ p\ ls\ s \longleftrightarrow atS\ p\ ls\ s'$
⟨*proof*⟩

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

lemma *atS-simps*:
$\neg atS\ p\ \{\}\ s$
$atS\ p\ \{l\}\ s \longleftrightarrow at\ p\ l\ s$
$\llbracket$ $at\ p\ l\ s$; $l \in ls$ $\rrbracket \Longrightarrow atS\ p\ ls\ s \longleftrightarrow True$
$(\forall\, l.\ at\ p\ l\ s \longrightarrow l \notin ls) \Longrightarrow atS\ p\ ls\ s \longleftrightarrow False$
⟨*proof*⟩

lemma *atS-mono*:
$\llbracket$ $atS\ p\ ls\ s$; $ls \subseteq ls'$ $\rrbracket \Longrightarrow atS\ p\ ls'\ s$
⟨*proof*⟩

lemma *atS-un*:
$atS\ p\ (l \cup l')\ s \longleftrightarrow atS\ p\ l\ s \vee atS\ p\ l'\ s$
⟨*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* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com*

type-synonym (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system*
= (*'answer*, *'location*, *'proc*, *'question*, *'state*) *programs*
× (*'proc*, *'state*) *state-pred*

definition
initial-states :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *global-state set*
where
initial-states sys ≡
$\{\ f\ .\ (\forall\, p.\ cPGM\ (f\ p) = [fst\ sys\ p]) \wedge snd\ sys\ (cLST \circ f)\ \}$

definition
reachable-states :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state set*
where
reachable-states sys ≡ $system\text{-}step^{*}$ `` (*initial-states sys* × {[]})

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: $(s,\ h) \in$ *reachable-states sys*
assumes i: $\bigwedge s.\ s \in$ *initial-states sys* $\Longrightarrow P\ s\ []$
assumes l: $\bigwedge s\ h\ ls'\ p.\ [\![\ (s,\ h) \in$ *reachable-states sys*; $P\ s\ h$; $s\ p \rightarrow_{\tau} ls'\ ]\!]$
$\Longrightarrow P\ (s(p := ls'))\ h$
assumes c: $\bigwedge s\ h\ ls1'\ ls2'\ p1\ p2\ \alpha\ \beta.$
$[\![\ (s,\ h) \in$ *reachable-states sys*; $P\ s\ h$;
$s\ p1 \rightarrow_{\ll\alpha,\ \beta\gg} ls1'$; $s\ p2 \rightarrow_{\gg\alpha,\ \beta\ll} ls2'$; $p1 \neq p2\ ]\!]$
$\Longrightarrow P\ (s(p1 := ls1',\ p2 := ls2'))\ (h\ @\ [(\alpha,\ \beta)])$
shows $P\ s\ h\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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 state s to the programs in the initial states. The *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 *after* control predicate with more syntactic analysis of this kind.

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* $c = \langle False\rangle$

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 lp' c s $\equiv$ *if extract-cond c s then lp else lp'*$\langle proof\rangle\langle proof\rangle$
fun
fragments :: $('answer,\ 'location,\ 'question,\ 'state)$ *com*
$\Rightarrow ('location \Rightarrow bool)$
$\Rightarrow (\ ('answer,\ 'location,\ 'question,\ 'state)$ *com*
$\times\ ('answer,\ 'location,\ 'question,\ 'state)$ *loc-comp*) *set*
where
fragments $(\{\!|l|\!\}$ *IF b THEN c FI*$)$ *ls*
$= \{\ (\{\!|l|\!\}$ *IF b THEN c' FI*, *lcond* (*atC c*) *ls*$)\ |c'.$ *True* $\}$
$\cup$ *fragments c ls*

| *fragments* ($\{\!|l|\!\}$ *IF b THEN c1 ELSE c2 FI*) *ls*
  = { ($\{\!|l|\!\}$ *IF b THEN c1′ ELSE c2′ FI*, *lcond* (*atC c1*) (*atC c2*)) |*c1′ c2′*. *True* }
  ∪ *fragments c1 ls* ∪ *fragments c2 ls*
| *fragments* (*LOOP DO c OD*) *ls* = *fragments c* (*atC c*)
| *fragments* ($\{\!|l'|\!\}$ *WHILE b DO c OD*) *ls*
  = { ($\{\!|l'|\!\}$ *WHILE b DO c′ OD*, *lcond* (*atC c*) *ls*) |*c′*. *True* } ∪ *fragments c* ($\lambda l.\ l = l'$)
| *fragments* (*c1*;; *c2*) *ls* = *fragments c1* (*atC c2*) ∪ *fragments c2 ls*
| *fragments* (*c1* ⊔ *c2*) *ls* = *fragments c1 ls* ∪ *fragments c2 ls*
| *fragments c ls* = { (*c*, *lconst ls*) }

fun
  fragmentsL :: (′*answer*, ′*location*, ′*question*, ′*state*) *com list*
    ⇒ ((′*answer*, ′*location*, ′*question*, ′*state*) *com*
    × (′*answer*, ′*location*, ′*question*, ′*state*) *loc-comp*) *set*
where
  fragmentsL [] = {}
| *fragmentsL* [*c*] = *fragments c* ⟨*False*⟩
| *fragmentsL* (*c* # *c′* # *cs*) = *fragments c* (*atC c′*) ∪ *fragmentsL* (*c′* # *cs*)

abbreviation
  fragmentsLS :: (′*answer*, ′*location*, ′*question*, ′*state*) *local-state*
    ⇒ ((′*answer*, ′*location*, ′*question*, ′*state*) *com*
    × (′*answer*, ′*location*, ′*question*, ′*state*) *loc-comp*) *set*
where
  fragmentsLS s ≡ *fragmentsL* (*cPGM s*)⟨*proof*⟩⟨*proof*⟩

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

We show that taking system steps preserves fragments.

lemma *reachable-states-fragmentsLS*:
  assumes (*s*, *h*) ∈ *reachable-states sys*
  shows *fragmentsLS* (*s p*) ⊆ *fragments* (*fst sys p*) ⟨*False*⟩⟨*proof*⟩

Decomposing a compound command preserves fragments too.

fun
  extract-inner-locations :: (′*answer*, ′*location*, ′*question*, ′*state*) *com*
    ⇒ (′*answer*, ′*location*, ′*question*, ′*state*) *com list*
    ⇒ (′*answer*, ′*location*, ′*question*, ′*state*) *loc-comp*
where
  extract-inner-locations ($\{\!|l|\!\}$ *IF b THEN c FI*) *cs* = *lcond* (*atC c*) (*atL cs*)
| *extract-inner-locations* ($\{\!|l|\!\}$ *IF b THEN c1 ELSE c2 FI*) *cs* = *lcond* (*atC c1*) (*atC c2*)
| *extract-inner-locations* (*LOOP DO c OD*) *cs* = *lconst* (*atC c*)
| *extract-inner-locations* ($\{\!|l|\!\}$ *WHILE b DO c OD*) *cs* = *lcond* (*atC c*) (*atL cs*)
| *extract-inner-locations c cs* = *lconst* (*atL cs*)
⟨*proof*⟩⟨*proof*⟩
lemma *small-step-extract-inner-locations*:
  assumes *basic-com c*

assumes $(c \# cs, ls) \rightarrow_\alpha ls'$
shows $\mathit{extract\text{-}inner\text{-}locations}\ c\ cs\ c\ ls = \mathit{atLS}\ ls'$
$\langle proof \rangle$

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 $s\ p \rightarrow_\alpha ps'$
assumes $(s, h) \in \mathit{reachable\text{-}states}\ sys$
obtains $c\ cs\ ls'$
where $(c, ls') \in \mathit{fragments}\ (\mathit{fst}\ sys\ p)\ \langle \mathit{False} \rangle$
and $\mathit{basic\text{-}com}\ c$
and $\forall l.\ \mathit{atC}\ c\ l \longrightarrow \mathit{atLS}\ (s\ p)\ l$
and $ls'\ c\ (\mathit{cLST}\ (s\ p)) = \mathit{atLS}\ ps'$
and $(c \# cs, \mathit{cLST}\ (s\ p)) \rightarrow_\alpha ps' \langle proof \rangle$

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 *Request*s or *Response*s but not both. Here we choose to focus on *Request*s as we expect to have more local information available about these.

inductive
$\mathit{vcg} :: ('answer, 'location, 'proc, 'question, 'state)\ \mathit{programs}$
$\Rightarrow 'proc$
$\Rightarrow ('answer, 'location, 'question, 'state)\ \mathit{loc\text{-}comp}$
$\Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ \mathit{pred}$
$\Rightarrow ('answer, 'location, 'question, 'state)\ \mathit{com}$
$\Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ \mathit{pred}$
$\Rightarrow \mathit{bool}\ (\text{-}, \text{-}, \text{-} \models / \{\!|\text{-}|\!\}/\ \text{-}/\ \{\!|\text{-}|\!\})$
where
Request: $[\![\bigwedge aft'\ action'\ s\ ps'\ p's'\ l'\ \beta\ s'\ p'.$
$[\![\mathit{pre}\ s;\ (\{\!|l'|\!\}\ \mathit{Response}\ action', aft') \in \mathit{fragments}\ (\mathit{pgms}\ p')\ \langle \mathit{False} \rangle;\ p \neq p';$
$ps' \in \mathit{val}\ \beta\ (\mathit{LST}\ s\ p);\ (p's', \beta) \in action'\ (action\ (\mathit{LST}\ s\ p))\ (\mathit{LST}\ s\ p');$
$\mathit{at}\ p\ l\ s;\ \mathit{at}\ p'\ l'\ s;$
$\mathit{AT}\ s' = (\mathit{AT}\ s)(p := \mathit{aft}\ (\{\!|l|\!\}\ \mathit{Request}\ action\ val)\ (\mathit{LST}\ s\ p),$
$p' := aft'\ (\{\!|l'|\!\}\ \mathit{Response}\ action')\ (\mathit{LST}\ s\ p'));$

$LST\ s' = (LST\ s)(p := ps',\ p' := p's');$
$hist\ s' = hist\ s\ @\ [(action\ (LST\ s\ p),\ \beta)]$
$\rrbracket \Longrightarrow post\ s'$
$\rrbracket \Longrightarrow pgms,\ p,\ aft \models \{\!|pre|\!\}\ \{\!|l|\!\}\ Request\ action\ val\ \{\!|post|\!\}$
$|\ LocalOp\colon \llbracket\ \bigwedge s\ ps'\ s'.\ \llbracket\ pre\ s;\ ps' \in f\ (LST\ s\ p);$
$at\ p\ l\ s;$
$AT\ s' = (AT\ s)(p := aft\ (\{\!|l|\!\}\ LocalOp\ f)\ (LST\ s\ p));$
$LST\ s' = (LST\ s)(p := ps');$
$hist\ s' = hist\ s$
$\rrbracket \Longrightarrow post\ s'$
$\rrbracket \Longrightarrow pgms,\ p,\ aft \models \{\!|pre|\!\}\ \{\!|l|\!\}\ LocalOp\ f\ \{\!|post|\!\}$
$|\ Cond1\colon \llbracket\ \bigwedge s\ s'.\ \llbracket\ pre\ s;$
$at\ p\ l\ s;$
$AT\ s' = (AT\ s)(p := aft\ (\{\!|l|\!\}\ IF\ b\ THEN\ t\ FI)\ (LST\ s\ p));$
$LST\ s' = LST\ s;$
$hist\ s' = hist\ s$
$\rrbracket \Longrightarrow post\ s'$
$\rrbracket \Longrightarrow pgms,\ p,\ aft \models \{\!|pre|\!\}\ \{\!|l|\!\}\ IF\ b\ THEN\ t\ FI\ \{\!|post|\!\}$
$|\ Cond2\colon \llbracket\ \bigwedge s\ s'.\ \llbracket\ pre\ s;$
$at\ p\ l\ s;$
$AT\ s' = (AT\ s)(p := aft\ (\{\!|l|\!\}\ IF\ b\ THEN\ t\ ELSE\ e\ FI)\ (LST\ s\ p));$
$LST\ s' = LST\ s;$
$hist\ s' = hist\ s$
$\rrbracket \Longrightarrow post\ s'$
$\rrbracket \Longrightarrow pgms,\ p,\ aft \models \{\!|pre|\!\}\ \{\!|l|\!\}\ IF\ b\ THEN\ t\ ELSE\ e\ FI\ \{\!|post|\!\}$
$|\ While\colon \llbracket\ \bigwedge s\ s'.\ \llbracket\ pre\ s;$
$at\ p\ l\ s;$
$AT\ s' = (AT\ s)(p := aft\ (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD)\ (LST\ s\ p));$
$LST\ s' = LST\ s;$
$hist\ s' = hist\ s$
$\rrbracket \Longrightarrow post\ s'$
$\rrbracket \Longrightarrow pgms,\ p,\ aft \models \{\!|pre|\!\}\ \{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \{\!|post|\!\}$
— There are no proof obligations for the following commands.
$|\ Response\colon pgms,\ p,\ aft \models \{\!|pre|\!\}\ \{\!|l|\!\}\ Response\ action\ \{\!|post|\!\}$
$|\ Seq\colon pgms,\ p,\ aft \models \{\!|pre|\!\}\ c1\ ;;\ c2\ \{\!|post|\!\}$
$|\ Loop\colon pgms,\ p,\ aft \models \{\!|pre|\!\}\ LOOP\ DO\ c\ OD\ \{\!|post|\!\}$
$|\ Choose\colon pgms,\ p,\ aft \models \{\!|pre|\!\}\ c1 \sqcup c2\ \{\!|post|\!\}$

We abbreviate invariance with one-sided validity syntax.

abbreviation *valid-inv* (-, -, - $\models$/ $\{\!|$-$|\!\}$/ -) **where**
$pgms,\ p,\ aft \models \{\!|I|\!\}\ c \equiv pgms,\ p,\ aft \models \{\!|I|\!\}\ c\ \{\!|I|\!\}\langle proof\rangle\langle proof\rangle$

We tweak *fragments* by omitting *Response*s, yielding fewer obligations.

fun
$vcg\text{-}fragments' :: ('answer,\ 'location,\ 'question,\ 'state)\ com$
$\Rightarrow ('location \Rightarrow bool)$
$\Rightarrow (\ ('answer,\ 'location,\ 'question,\ 'state)\ com$

$\times$ ($'answer$, $'location$, $'question$, $'state$) *loc-comp*) *set*

where

vcg-fragments' ($\{\!|l|\!\}$ *Response action*) *ls* = {}
| *vcg-fragments'* ($\{\!|l|\!\}$ *IF b THEN c FI*) *ls*
  = *vcg-fragments' c ls*
  $\cup$ { ($\{\!|l|\!\}$ *IF b THEN c' FI*, *lcond* (*atC c*) *ls*) |*c'*. *True* }
| *vcg-fragments'* ($\{\!|l|\!\}$ *IF b THEN c1 ELSE c2 FI*) *ls*
  = *vcg-fragments' c2 ls* $\cup$ *vcg-fragments' c1 ls*
  $\cup$ { ($\{\!|l|\!\}$ *IF b THEN c1' ELSE c2' FI*, *lcond* (*atC c1*) (*atC c2*)) |*c1' c2'*. *True* }
| *vcg-fragments'* (*LOOP DO c OD*) *ls* = *vcg-fragments' c* (*atC c*)
| *vcg-fragments'* ($\{\!|l'|\!\}$ *WHILE b DO c OD*) *ls*
  = *vcg-fragments' c* ($\lambda l.\ l = l'$) $\cup$ { ($\{\!|l'|\!\}$ *WHILE b DO c' OD*, *lcond* (*atC c*) *ls*) |*c'*. *True* }
| *vcg-fragments'* (*c1* ;; *c2*) *ls* = *vcg-fragments' c2 ls* $\cup$ *vcg-fragments' c1* (*atC c2*)
| *vcg-fragments'* (*c1* $\sqcup$ *c2*) *ls* = *vcg-fragments' c1 ls* $\cup$ *vcg-fragments' c2 ls*
| *vcg-fragments' c ls* = {(*c*, *lconst ls*)}

abbreviation

vcg-fragments :: ($'answer$, $'location$, $'question$, $'state$) *com*
  $\Rightarrow$ (($'answer$, $'location$, $'question$, $'state$) *com*
  $\times$ ($'answer$, $'location$, $'question$, $'state$) *loc-comp*) *set*

where

vcg-fragments c $\equiv$ *vcg-fragments' c* $\langle$*False*$\rangle\langle$*proof*$\rangle\langle$*proof*$\rangle$

The user sees the conclusion of V for each element of *vcg-fragments*.

lemma *VCG*:
  assumes *R*: $s \in$ *reachable-states sys*
  assumes *I*: $\forall s \in$ *initial-states sys*. *I* (*mkP* (*s*, []))
  assumes *V*: $\bigwedge p$. $\forall$(*c*, *afts*) $\in$ *vcg-fragments* (*fst sys p*). ((*fst sys*), *p*, *afts* $\models \{\!|I|\!\}$ *c*)
  shows *I* (*mkP s*)$\langle$*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
  fixes *pgms* :: ($'answer$, $'location$, $'proc$, $'question$, $'state$) *programs*
  fixes *p* :: $'proc$
  fixes *afts* :: ($'answer$, $'location$, $'question$, $'state$) *loc-comp*
begin

abbreviation

valid-syn :: ($'answer$, $'location$, $'proc$, $'question$, $'state$) *pred*
  $\Rightarrow$ ($'answer$, $'location$, $'question$, $'state$) *com*
  $\Rightarrow$ ($'answer$, $'location$, $'proc$, $'question$, $'state$) *pred* $\Rightarrow$ *bool* **where**

valid-syn P c Q $\equiv$ *pgms*, *p*, *afts* $\models \{\!|P|\!\}$ *c* $\{\!|Q|\!\}$

notation *valid-syn* (⦃-⦄/ -/ ⦃-⦄)

abbreviation
  valid-inv-syn :: ('*answer*, '*location*, '*proc*, '*question*, '*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*⟩⦄
⟨*proof*⟩

lemma *vcg-conj*:
  ⟦ ⦃I⦄ c ⦃Q⦄; ⦃I⦄ c ⦃R⦄ ⟧ ⟹ ⦃I⦄ c ⦃Q *and* R⦄
⟨*proof*⟩

lemma *vcg-pre-imp*:
  ⟦ ⋀s. P s ⟹ Q s; ⦃Q⦄ c ⦃R⦄ ⟧ ⟹ ⦃P⦄ c ⦃R⦄
⟨*proof*⟩

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

lemma *vcg-post-imp*:
  ⟦ ⋀s. Q s ⟹ R s; ⦃P⦄ c ⦃Q⦄ ⟧ ⟹ ⦃P⦄ c ⦃R⦄
⟨*proof*⟩

lemma *vcg-prop*[*intro*]:
  ⦃⟨P⟩⦄ c
⟨*proof*⟩

lemma *vcg-drop-imp*:
  assumes ⦃P⦄ c ⦃Q⦄
  shows ⦃P⦄ c ⦃R *imp* Q⦄
⟨*proof*⟩

lemma *vcg-conj-lift*:
  assumes x: ⦃P⦄ c ⦃Q⦄
  assumes y: ⦃P'⦄ c ⦃Q'⦄
  shows ⦃P *and* P'⦄ c ⦃Q *and* Q'⦄
⟨*proof*⟩

lemma *vcg-disj-lift*:
  assumes x: ⦃P⦄ c ⦃Q⦄
  assumes y: ⦃P'⦄ c ⦃Q'⦄
  shows ⦃P *or* P'⦄ c ⦃Q *or* Q'⦄
⟨*proof*⟩

lemma *vcg-imp-lift*:
  assumes $\{\!|P'|\!\}\ c\ \{\!|not\ P|\!\}$
  assumes $\{\!|Q'|\!\}\ c\ \{\!|Q|\!\}$
  shows $\{\!|P'\ or\ Q'|\!\}\ c\ \{\!|P\ imp\ Q|\!\}$
⟨*proof*⟩

lemma *vcg-ex-lift*:
  assumes $\bigwedge x.\ \{\!|P\ x|\!\}\ c\ \{\!|Q\ x|\!\}$
  shows $\{\!|\lambda s.\ \exists\, x.\ P\ x\ s|\!\}\ c\ \{\!|\lambda s.\ \exists\, x.\ Q\ x\ s|\!\}$
⟨*proof*⟩

lemma *vcg-all-lift*:
  assumes $\bigwedge x.\ \{\!|P\ x|\!\}\ c\ \{\!|Q\ x|\!\}$
  shows $\{\!|\lambda s.\ \forall\, x.\ P\ x\ s|\!\}\ c\ \{\!|\lambda s.\ \forall\, x.\ Q\ x\ s|\!\}$
⟨*proof*⟩

lemma *vcg-name-pre-state*:
  assumes $\bigwedge s.\ P\ s \Longrightarrow \{\!|op = s|\!\}\ c\ \{\!|Q|\!\}$
  shows $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *vcg-lift-comp*:
  assumes f: $\bigwedge P.\ \{\!|\lambda s.\ P\ (f\ s :: 'a :: type)|\!\}\ c$
  assumes P: $\bigwedge x.\ \{\!|Q\ x|\!\}\ c\ \{\!|P\ x|\!\}$
  shows $\{\!|\lambda s.\ Q\ (f\ s)\ s|\!\}\ c\ \{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$
  ⟨*proof*⟩

2.5.2 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).

definition *eq-imp* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'e) \Rightarrow bool$ **where**
  $eq\text{-}imp\ f\ g \equiv (\forall\, s\ s'.\ (\forall\, x.\ f\ x\ s = f\ x\ s') \longrightarrow (g\ s = g\ s'))$

lemma *eq-impD*:
  $[\![\ eq\text{-}imp\ f\ g;\ \forall\, x.\ f\ x\ s = f\ x\ s'\ ]\!] \Longrightarrow g\ s = g\ s'$
⟨*proof*⟩

lemma *eq-imp-vcg*:
  assumes g: *eq-imp* $f\ g$
  assumes f: $\forall\, x\ P.\ \{\!|P \circ (f\ x)|\!\}\ c$
  shows $\{\!|P \circ g|\!\}\ c$
  ⟨*proof*⟩

lemma *eq-imp-vcg-LST*:
  assumes g: *eq-imp* f g
  assumes f: $\forall\, x\ P.\ \{\!\!|P \circ (f\ x) \circ LST|\!\!\}\ c$
  shows $\{\!\!|P \circ g \circ LST|\!\!\}\ c$
  ⟨*proof*⟩

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

lemma *curry-forall-eq*:
  $(\forall\, f.\ P\ f) = (\forall\, f.\ P\ (\mathit{case\text{-}prod}\ f))$
  ⟨*proof*⟩

lemma *pres-tuple-vcg*:
  $(\forall\, P.\ \{\!\!|P \circ (\lambda s.\ (f\ s,\ g\ s))|\!\!\}\ c)$
    $\longleftrightarrow ((\forall\, P.\ \{\!\!|P \circ f|\!\!\}\ c) \wedge (\forall\, P.\ \{\!\!|P \circ g|\!\!\}\ c))$
  ⟨*proof*⟩

lemma *pres-tuple-vcg-LST*:
  $(\forall\, P.\ \{\!\!|P \circ (\lambda s.\ (f\ s,\ g\ s)) \circ LST|\!\!\}\ c)$
    $\longleftrightarrow ((\forall\, P.\ \{\!\!|P \circ f \circ LST|\!\!\}\ c) \wedge (\forall\, P.\ \{\!\!|P \circ g \circ LST|\!\!\}\ c))$
  ⟨*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 :: $'location \Rightarrow 'channel \Rightarrow ('val \Rightarrow 'state \Rightarrow 'state)$
        $\Rightarrow (unit,\ 'location,\ 'channel \times 'val,\ 'state)\ com\ (\{\!\!|\text{-}|\!\!\}/\ \text{-}\triangleright\text{-})$
where
  $\{\!\!|l|\!\!\}\ ch\triangleright f \equiv \{\!\!|l|\!\!\}\ Response\ (\lambda quest\ s.\ \mathit{if}\ \mathit{fst}\ quest = ch\ \mathit{then}\ \{(f\ (snd\ quest)\ s,\ ())\}\ \mathit{else}\ \{\})$

abbreviation

$Send :: 'location \Rightarrow 'channel \Rightarrow ('state \Rightarrow 'val)$
$\Rightarrow (unit, 'location, 'channel \times 'val, 'state)\ com\ (\{\!|\text{-}|\!\}/\ \text{-}\triangleleft\text{-})$
where
$\{\!|l|\!\}\ ch\triangleleft f \equiv \{\!|l|\!\}\ Request\ (\lambda s.\ (ch, f\ s))\ (\lambda ans\ s.\ \{s\})$

We further specialise these for our particular example.

abbreviation
$Receive' :: 'location \Rightarrow 'channel \Rightarrow (unit, 'location, 'channel \times 'state, 'state)\ com\ (\{\!|\text{-}|\!\}/\ \text{-}\triangleright)$
where
$\{\!|l|\!\}\ ch\triangleright \equiv \{\!|l|\!\}\ ch\triangleright(\lambda v\ \text{-}.\ v)$

abbreviation
$Send' :: 'location \Rightarrow 'channel \Rightarrow (unit, 'location, 'channel \times 'state, 'state)\ com\ (\{\!|\text{-}|\!\}/\ \text{-}\triangleleft)$
where
$\{\!|l|\!\}\ ch\triangleleft \equiv \{\!|l|\!\}\ ch\triangleleft id$

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

datatype *ex-chname* = $\xi 12$ | $\xi 23$
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
$ex\text{-}pgms\ p1 = \{\!|s12|\!\}\ \xi 12\triangleleft$
$|\ ex\text{-}pgms\ p2 = LOOP\ DO\ \{\!|r12|\!\}\ \xi 12\triangleright;;\ \{\!|s23|\!\}\ \xi 23\triangleleft\ OD$
$|\ ex\text{-}pgms\ p3 = \{\!|r23|\!\}\ \xi 23\triangleright$

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* ($\lambda s.$ *filter-on-channel* $\xi 12$ (*hist s*) = [])
definition *Ip1-1* :: *ex-pred* **where**
Ip1-1 ≡ *terminated p1 imp* ($\lambda s.$ *filter-on-channel* $\xi 12$ (*hist s*) = [*LST s p1*])

definition *Ip2-0* :: *ex-pred* **where**
Ip2-0 ≡ *at p2 r12 imp* ($\lambda s.$ *filter-on-channel* $\xi 12$ (*hist s*) = *filter-on-channel* $\xi 23$ (*hist s*))
definition *Ip2-1* :: *ex-pred* **where**
Ip2-1 ≡ *at p2 s23 imp* ($\lambda s.$ *filter-on-channel* $\xi 12$ (*hist s*) = *filter-on-channel* $\xi 23$ (*hist s*) @ [*LST s p2*]
∧ *LST s p1* = *LST s p2*)

definition *Ip3-0* :: *ex-pred* **where**
Ip3-0 ≡ *at p3 r23 imp* ($\lambda s.$ *filter-on-channel* $\xi 23$ (*hist s*) = [])
definition *Ip3-1* :: *ex-pred* **where**
Ip3-1 ≡ *terminated p3 imp* ($\lambda s.$ *filter-on-channel* $\xi 23$ (*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* ($\lambda s.$ *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*⦄ $\xi 12 \triangleleft \lambda s.\ s$
⟨*proof*⟩

lemma *p2-1*: *ex-pgms*, *p2*, *lconst* ($\lambda l.\ l = r12$) ⊨ ⦃*I-pred*⦄ ⦃*s23*⦄ $\xi 23 \triangleleft \lambda s.\ s$
⟨*proof*⟩

lemma $(s,\ h) \in$ *reachable-states ex-system* $\Longrightarrow$ *I-pred* (*mkP* $(s,\ h)$)
⟨*proof*⟩

⟨*proof*⟩

4 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* = ξ*12* | ξ*23*
type-synonym *ex-val* = *nat*
type-synonym *ex-ls* = *ex-val list*
type-synonym *ex-ch* = *ex-chname* × *ex-val*
datatype *ex-loc* = π*4* | π*5* | *c1* | *r12* | *r23* | *s23* | *s12*
datatype *ex-proc* = *p1* | *p2* | *p3*

type-synonym *ex-pgm* = (*unit*, *ex-loc*, *ex-ch*, *ex-ls*) *com*
type-synonym *ex-pred* = (*unit*, *ex-loc*, *ex-proc*, *ex-ch*, *ex-ls*) *pred*
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* × *unit*) *list*

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

primrec *ex-pgms* :: *ex-proc* ⇒ *ex-pgm* **where**
ex-pgms p1 = *LOOP DO* ⦃*c1*⦄ *LocalOp* (λ*xs*. { *xs* @ [*x*] |*x*. *True* }) ;; ⦃*s12*⦄ ξ*12*◁*last OD*
| *ex-pgms p2* = *LOOP DO* ⦃*r12*⦄ ξ*12*▷(λ*x xs*. *xs* @ [*x*])
⊔ ⦃π*4*⦄ *IF* (λ*s*. *length s* > *0*) *THEN* ⦃*s23*⦄ *Request* (λ*s*. (ξ*23*, *hd s*)) (λ*ans s*. {*tl s*}) *FI*
OD
| *ex-pgms p3* = *LOOP DO* ⦃*r23*⦄ ξ*23*▷(λ*x xs*. *xs* @ [*x*]) *OD*

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

abbreviation *ex-system* :: *ex-system* **where**
ex-system ≡ (*ex-pgms*, *ex-init*)

definition *filter-on-channel* :: *ex-chname* ⇒ *ex-history* ⇒ *ex-val list* **where**
filter-on-channel ch ≡ *map* (*snd* ∘ *fst*) ∘ *filter* (*op* = *ch* ∘ *fst* ∘ *fst*)

lemma *filter-on-channel-simps* [*simp*]:
filter-on-channel ch [] = []
filter-on-channel ch (*xs* @ *ys*) = *filter-on-channel ch xs* @ *filter-on-channel ch ys*
filter-on-channel ch (((*ch′*, *v*), *resp*) # *vals*) = (*if ch′* = *ch then* [*v*] *else* []) @ *filter-on-channel ch vals*
⟨*proof*⟩

definition *Ip1-0* :: *ex-pred* **where**
$Ip1\text{-}0 \equiv \lambda s.\ at\ p1\ c1\ s \longrightarrow filter\text{-}on\text{-}channel\ \xi 12\ (hist\ s) = s\!\downarrow p1$
definition *Ip1-1* :: *ex-pred* **where**
$Ip1\text{-}1 \equiv \lambda s.\ at\ p1\ s12\ s \longrightarrow length\ (s\!\downarrow p1) > 0 \land butlast\ (s\!\downarrow p1) = filter\text{-}on\text{-}channel\ \xi 12\ (hist\ s)$
definition *Ip1-2* :: *ex-pred* **where**
$Ip1\text{-}2 \equiv \lambda s.\ filter\text{-}on\text{-}channel\ \xi 12\ (hist\ s) \leq s\!\downarrow p1$

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

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

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

definition *I-pred* :: *ex-pred* **where**
$I\text{-}pred \equiv pred\text{-}conjoin\ [\ Ip1\text{-}0,\ Ip1\text{-}1,\ Ip1\text{-}2,\ Ip2\text{-}0,\ Ip2\text{-}1,\ Ip3\text{-}0\ ]$

lemmas *I-defs* = *I-pred-def Ip1-0-def Ip1-1-def Ip1-2-def Ip2-0-def Ip2-1-def Ip3-0-def*

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

definition *Etern-pred* :: *ex-pred* **where**
$Etern\text{-}pred \equiv \lambda s.\ s\!\downarrow p3 \leq s\!\downarrow p1$

lemma *correct-system*:
$I\text{-}pred\ s \Longrightarrow Etern\text{-}pred\ s\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

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

5 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.

[2]The theories are in `$ISABELLE/src/HOL/Hoare_Parallel`.

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

S. Berghofer. A solution to the PoplMark challenge using de Bruijn indices in Isabelle/HOL. *J. Autom. Reasoning*, 49(3):303–326, 2012.

P. Cousot and R. Cousot. Semantic analysis of Communicating Sequential Processes (shortened version). In J. W. de Bakker and J. van Leeuwen, editors, *ICALP*, volume 85, pages 119–133. Springer, 1980.

P. Cousot and R. Cousot. A language independent proof of the soundness and completeness of generalized Hoare logic. *Information and Computation*, 80(2):165–191, February 1989.

F. S. de Boer, W. P. de Roever, and U. Hannemann. The semantic foundations of a compositional proof method for synchronously communicating processes. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *MFCS*, volume 1672, pages 343–353. Springer, 1999.

W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

C.A.R. Hoare. *Communicating Sequential Processes*. International Series In Computer Science. Prentice-Hall, 1985. URL http://www.usingcsp.com/.

L. Lamport. The "Hoare Logic" of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

L. Lamport and F. B. Schneider. The "Hoare Logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, 1984.

G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Inf.*, 15:281–302, 1981.

Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 121–156. ACM Press and Addison-Wesley, 1991.

Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

R. Milner. *A Calculus of Communicating Systems*. 1980.

R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

T. Nipkow and G. Klein. *Concrete Semantics: A Proof Assistant Approach*. 2014. URL http://www.in.tum.de/~nipkow/Concrete-Semantics/.

T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *FASE*, volume 1577, pages 188–203. Springer, 1999.

A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395, pages 378–412. Springer-Verlag, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

L. Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618, pages 348–362, 2003.

L. Prensa Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS 2000)*, volume 1893 of *LNCS*, pages 619–628. Springer-Verlag, 2000.

N. Schirmer. A verification environment for sequential imperative programs in isabelle/hol. In F. Baader and A. Voronkov, editors, *LPAR*, volume 3452, pages 398–414. Springer, 2004.

N. Schirmer and M. Wenzel. State spaces - the locale way. *Electr. Notes Theor. Comput. Sci.*, 254:161–179, 2009.

G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1993.