Denotational Semantics for `prialt`-free Handel-C

Andrew Butterfield

December 20, 2001

Contents

1 Introduction

Handel-C [Emb] is a language developed by the Hardware Compilation Group at Oxford's Computer Laboratory. It is a hybrid of C and CSP [Hoa90], designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs) [PL91, SP93, BHP94, LKL+95]. The language can be viewed as a pointer-free subset of C, augmented with a parallel construct and channel communication, as found in CSP. The type system has been modified to explicitly refer to the number of bits required to implement any given type. The language targets synchronous hardware with a single master clock. All assignments and channel communication events take one clock cycle, with all updates synchronised with the clock edge marking the cycle end. All expression and conditional evaluation (for selection, loops) is deemed to take 'zero-time', effectively being completed before the current clock cycle ends.

In order to facilitate work on a formal semantics for Handel-C, it was decided at first to consider a subset only, namely the smallest subset that would catch all the essential and potentially difficult aspects. After discussions with Ian Page, a suitable subset of Handel-C was identified. It includes channels, assignment,

conditional, one loop, and the parallel construct as well as shared variables. Key omissions include types, macros, shared expressions, RAMs, ROMs, arrays and the bus interface material, all of which can be handled fairly straightforwardly, in semantic terms. Discussing requirements for a set of algebraic laws with Jim Woodcock, in particular with respect to normal forms, it was decided to include the 'prioritised alternatives' construct (`prialt`) in the reduced syntax. Another key motivation for having the `prialt` is that it is the only mechanism for introducing non-blocking (or try-else-skip) synchronisations into programs.

This report presents a denotational semantics of a `prialt`-free subset of the Handel-C hardware compilation language. The language is presented here with such a construct, but its semantics are not fully described at this point. This is due in the main to the lack of clear documentation on the behaviour of this construct, which will require some experimentation in order to be able to elucidate an appropriate semantics. This has been left as a future task. However, the semantics presented here has been defined with `prialt` in mind, which explains why the semantics of communication appears more complicated than it needs to be.

This document presents the semantics, but does not provide any formal or informal validation. This will be the subject of another technical report.

Here we present a denotational semantics for Handel-C programs, inspired in the main by material presented at a seminar on Concurrency, held in Carnegie-Mellon University, in 1984 [BRW84].

2 Reduced Concrete Syntax

We initially start with a reduced and simplified form of the concrete syntax of Handel-C (see [Emb, §9] for details of the full syntax). We shall then transform this language in two stages. The first stage gives the conditional and communication statements a more uniform structure. The second stage breaks all communication statements down into more fundamental components dealing explicitly with requests, waits and data transfer.

In our reduced syntax we have variables and channel identifiers, as well as expressions which we do not define further.

$v \in V$	Variables
$c \in C$	Channels
$e, b \in E$	Expressions

Communication statements are referred to as guards, because they only occur as such in prialt statments, in our reduced language.

$g \in G$	$::=$		Guards
		$!?$	Self-Synch
	$\mid$	$c!e$	Output
	$\mid$	$c?v$	Input

Handel-C statements include the usual imperative language ones, plus parallel, prialt and delay constructs. In order to be general, we allow delay statements

to specify an arbitrary number of clock cycles (including zero).

$s \in S$	$::=$		Statements
		δ_n	Delay
	$\mid$	$v := e$	Assignment
	$\mid$	$s_1 ; s_2$	Sequencing
	$\mid$	$s_1 \parallel s_2$	Parallel
	$\mid$	$b \rightarrow s_1 \ , \ s_2$	Conditional
	$\mid$	$b * s$	While
	$\mid$	$\langle g_1 : s_1, \ldots, g_n : s_n \rangle$	Prialt

We have a well-formedness constraint on prialts: all but the last guard must be either input or output. Stand-alone communications statements are modelled as singleton prialts, where we often write $c!e$ or $c?v$ instead of the strictly correct form $\langle c!e : \delta_0 \rangle$ or $\langle c?v; \delta_0 \rangle$. In pure programs, Self-Synch guards (!?) only occur last in a pri-alt, and all delays are of the form δ_1.

2.1 Language Rework Stage 1

We are going to rework the language to add some new expression and statement forms, including an n-way conditional, and labels for all communication statements. We also admit 'naked' guards directly as communication statements

We add in the notion of labels (for communication statements) and we extend expressions to contain special (meta) builtin functions (w, s, e) to handle communication events. The first two (w, s) have signature $L \rightarrow \mathbb{N}$ and are used to determine if the communication statement with the corresponding label has to wait or which sub-statement has ben selected to execute, if appropriate. The third (e) has signature $C \rightarrow Val$ and returns the value currently being transmitted on the given channel.

$\ell \in L$			Labels
$p \in P$	$::=$	$\mathbb{N}$	Priorities
$e \in E$			Expressions
$b \in E'$	$::=$	E	Extended Expr.
	$\mid$	$w(\ell)$	Wait
	$\mid$	$s(\ell)$	Select
	$\mid$	$e(c)$	Expr

For statements, we allow guards outside prialts, for convenience, we change the 2-way conditional (if-then-else) to a n-way conditional (switch), and we add in

specific statements to request and release communication requests:

$$
\begin{array}{lll}
s \in S \quad ::= & & \text{Statements} \\
& \delta_n & \text{Delay} \\
\mid & v := e & \text{Assignment} \\
\mid & g_\ell & \text{Guards (excl. !?)} \\
\mid & s_1 ; s_2 & \text{Sequencing} \\
\mid & s_1 \parallel s_2 & \text{Parallel} \\
\mid & e \ \rightarrow \ s_1 \ , \ \ldots , s_n & \text{Conditional} \\
\mid & b * s & \text{While} \\
\mid & \langle g_1 : s_1 , \ldots , g_n : s_n \rangle_\ell & \text{Prialt} \\
\mid & \Re(\ell, p, g) & \text{Request} \\
\mid & \mho(\ell) & \text{Release}
\end{array}
$$

2.2 Language Rework Stage 2

We now reduce the language to a restricted subset using the following rules, which replace all communication statements, by a sequence involving issuing a request, waiting for it to be granted, releasing any other associated requests, and finally performing the appropriate data transfer operation.

$$
\begin{array}{rcl}
g_\ell & \mapsto & \Re(\ell, 1, g);\ w(\ell) * \delta_1;\ \mho(\ell); \mathrm{Act}(g) \\
\langle g_1 : s_1 , \ldots , g_n : s_n \rangle_\ell & \mapsto & \Re(\ell, 1, g_1); \ldots ; \Re(\ell, n, g_n); \\
 & & w(\ell) * \delta_1; \\
 & & s(\ell) \ \rightarrow \ \mho(\ell); \mathrm{Act}(g_1); s_1 \ , \ \ldots , \mho(\ell); \mathrm{Act}(g_n); s_n \\
\textbf{where} & & \\
\mathrm{Act}(!?) & \widehat{=} & \delta_0 \\
\mathrm{Act}(c!e) & \widehat{=} & \delta_1 \\
\mathrm{Act}(c?v) & \widehat{=} & v := e(c)
\end{array}
$$

This results in the following reduced language, here presented in full:

$v \in V$			Variables
$c \in C$			Channels
$\ell \in L$			Labels
$p \in P$	$::=$	$\mathbb{N}$	Priorities
$e \in E$			Expressions
$b \in E'$	$::=$	E	Extended Expr.
	$\mid$	$w(\ell)$	Wait
	$\mid$	$s(\ell)$	Select
	$\mid$	$e(c)$	Expr
$g \in G$	$::=$		Guards
		$!?$	Self-Synch
	$\mid$	$c!e$	Output
	$\mid$	$c?v$	Input
$s \in S$	$::=$		Statements
		δ_n	Delay
	$\mid$	$v := e$	Assignment
	$\mid$	$s_1; s_2$	Sequencing
	$\mid$	$s_1 \parallel s_2$	Parallel
	$\mid$	$e \ \rightarrow \ s_1 \ , \ \ldots, s_n$	Conditional
	$\mid$	$b * s$	While
	$\mid$	$\Re(\ell, p, g)$	Request
	$\mid$	$\mho(\ell)$	Release

The syntax as presented is a little too liberal. The only usage of the request and release statements, and the extended expressions forms (w, s, e) is that which is generated from the stage one language using the rules given above. This language will form the basis for the denotational semantics.

3 Semantic Domains Introduction

Handel-C has parallelism, with global variables, and a very synchronous timing model, with assignments and communication events all synchronising with a master clock signal. Many semantic approaches dealing with concurrency, as exemplified by [BRW84] succeed by restricting the interplayy of the above features. The obvious approaches to a denotational semantics, i.e taking the denotational semantics of CSP [BR84] or occam [Ros84] and modifying them to suit all rapidly fail, the key problem being the presence of both global variables and concurrency. This makes the semantics of sequencing much more difficult — we can't use the rule

$$[\![s_1; s_2]\!]_\omega = [\![s_2]\!]_{\omega'} \textbf{ where } \omega' = [\![s_1]\!]_\omega$$

because a simultaneous assignments elsewhere might have changed some other part of the environment. Also, most of the material in [BRW84] assumes an interleaving semantics for concurrency, so that parallel threads are 'woven' together. In [BRW84], there is a semantics given of TCCS, which has events

occurring simultaneously, but does not admit any global variables. However, in Handel-C, we must deal with real concurrency, in the presence of a single global update clock, with global variables. An idea from [Bro84] (Brooke's description of Plotkin/Hennessey Resumption Trees) is to de-couple post- and pre-conditions. In Hoare notation, we replace

$$\{P\}S_1\{Q\}S_2\{R\}$$

by

$$\{P\}S_1\{Q\}\{Q'\}S_2\{R\}$$

The change from $\{Q\}$ to $\{Q'\}$ allows us to handle other changes made to the environment by other execution threads.

What follows is a semantics, inspired by ideas from [BRW84], which resolves all of these difficulties.

4 Basic Semantic Domains

We first present the basic domains, used as building blocks for the more difficult material.

4.1 Value Domains

Our basic domains include variable values, timestamps, and directed (communication) requests

$$\begin{array}{rcll} \mathit{Val} & \widehat{=} & \mathbb{Z} \cup \{?\} & \text{Values} \\ \mathit{Time} & \widehat{=} & \mathbb{N} & \text{Timestamps} \\ \mathit{DReq} & \widehat{=} & \textsc{In}\ V \mid \textsc{Out}\ E & \text{Directed Request} \end{array}$$

We also have channel states, which are either idle or active with an associated expression, and we have request states which are a (selector number) paired with (priority-ordered) sequences of channel/directed request pairs:

$$\begin{array}{rcll} \mathit{ChState} & \widehat{=} & & \text{Channel States} \\ & & \textsc{Idle} & \text{Idle Channel} \\ & \mid & \textsc{Act}\ E & \text{Active} \\ \\ \mathit{ReqState} & \widehat{=} & \mathbb{N} \times (C \times \mathit{DReq})^\star & \text{Communication State} \\ \mathit{inv\text{-}ReqState}(n, \varsigma) & \widehat{=} & n \in \{0 \ldots \mathtt{len}\ \varsigma\} & \end{array}$$

We shall refer to the union of all the above spaces as our *datum* space:

$$d \in \mathit{Datum} \quad \widehat{=} \quad \mathit{Val} \cup \mathit{Time} \cup \mathit{DReq} \cup \mathit{ChState} \cup \mathit{ReqState}$$

4.2 Identifier Spaces

We have an identifier space containing variable and channel names, communication statement labels and a timestamp identifier (τ). Given a Handel-C

programme, the total identifier space is fixed and statically determined:

$$
\begin{array}{rcl}
Id & \widehat{=} & V \cup \{\tau\} \cup C \cup L \quad \text{Identifier Space} \\
\text{pIds} & : & S \to SetId \\
\text{pIds}[\![P]\!] & \widehat{=} & \{\tau\} \cup \text{sIds}[\![P]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\text{sIds} & : & S \to \mathcal{P}\, Id \\
\text{sIds}[\![\delta_n]\!] & \widehat{=} & \emptyset \\
\text{sIds}[\![v := e]\!] & \widehat{=} & \{v\} \cup \text{eIds}[\![e]\!] \\
\text{sIds}[\![g_\ell]\!] & \widehat{=} & \{\ell\} \cup \text{gIds}[\![g]\!] \\
\text{sIds}[\![s_1; s_2]\!] & \widehat{=} & \text{sIds}[\![s_1]\!] \cup \text{sIds}[\![s_2]\!] \\
\text{sIds}[\![s_1 \parallel s_2]\!] & \widehat{=} & \text{sIds}[\![s_1]\!] \cup \text{sIds}[\![s_2]\!] \\
\text{sIds}[\![b \ \rightarrow \ s_1 \ , \ s_2]\!] & \widehat{=} & \text{eIds}[\![b]\!] \cup \text{sIds}[\![s_1]\!] \cup \text{sIds}[\![s_2]\!] \\
\text{sIds}[\![b * s]\!] & \widehat{=} & \text{eIds}[\![b]\!] \cup \text{sIds}[\![s]\!] \\
\text{sIds}[\![\langle g_1 : s_1, \ldots, g_n : s_n \rangle_\ell]\!] & \widehat{=} & \{\ell\} \\
 & & \quad \cup\, (\bigcup\!/ \circ (\text{gIds})^\star)\langle g_1 \ldots, g_n \rangle \\
 & & \quad \cup\, (\bigcup\!/ \circ (\text{sIds})^\star)\langle s_1, \ldots, s_n \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\text{gIds} & : & G \to \mathcal{P}\, Id \\
\text{gIds}[\![!?]\!] & \widehat{=} & \emptyset \\
\text{gIds}[\![c!e]\!] & \widehat{=} & \{c\} \cup \text{eIds}[\![e]\!] \\
\text{gIds}[\![c?v]\!] & \widehat{=} & \{c, v\}
\end{array}
$$

$$
\begin{array}{rcl}
\text{eIds} & : & E \to \mathcal{P}\, Id \\
\text{eIds}[\![e]\!] & \widehat{=} & \text{vars in expression}
\end{array}
$$

4.3 Environments, Worlds and Changes

In practice we treat the execution environment as a single mapping from a number of disjoint identifier spaces to corresponding value spaces:

$$
\begin{array}{rcl}
\rho \in Env & \widehat{=} & Id \to Datum \\
inv\text{-}Env & : & Env \to \mathbb{B} \\
inv\text{-}Env\,\rho & \widehat{=} & \forall [ok_\rho](\texttt{dom}\ \rho) \\
\textbf{where} & & \\
ok_\rho(i) & \widehat{=} & \quad (i \in V \wedge \rho(i) \in Val) \\
 & & \vee\ (i = \tau \wedge \rho(i) \in Time) \\
 & & \vee\ (i \in C \wedge \rho(i) \in ChState) \\
 & & \vee\ (i \in L \wedge \rho(i) \in ReqState)
\end{array}
$$

We view a program world map as a total environment over the program identifiers, and a change map as a partial environment over the same

$$
\begin{array}{rcl}
\omega \in World_p & \widehat{=} & \text{pIds}[\![p]\!] \to Datum \\
\delta \in Change_p & \widehat{=} & \text{pIds}[\![p]\!] \xrightarrow{m} Datum
\end{array}
$$

The initial world is one which maps all variables to ?, time to zero, all channels to idle, and labels to $(0, \Lambda)$:

$$\begin{array}{rcl} \omega_0 & : & \mathit{World}_p \\ \omega_0 & \widehat{=} & (^{\sqcup}/ \circ \mathit{ival}^\star \circ \mathrm{pIds})[\![p]\!] \\ \textbf{where} & & \\ \mathit{ival}(i) & \widehat{=} & i = \tau \ \rightarrow \ \{\tau \mapsto 0\} \\ & & i \in V \ \rightarrow \ \{i \mapsto ?\} \\ & & i \in C \ \rightarrow \ \{i \mapsto \textsc{Idle}\} \\ & & i \in L \ \rightarrow \ \{i \mapsto (0, \Lambda)\} \end{array}$$

5 Main Semantic Domains

5.1 Choices

We shall introduce the notion of choice, as a mapping from the world to changes to the world:

$$\kappa \in \mathit{Choice} \quad \widehat{=} \quad \mathit{World} \rightarrow \mathit{Change}$$

5.2 Branching Choice Sequences

We shall model semantics as a branching sequence of choices — this is inspired by the resumption semantics of Plotkin and Hennessy in LNCS 197, as described by Stephen Brookes. We define a branching sequence of As as either nil, a cons of an A and a branching sequence, or a split into several sequences. The splitting is mediated by a split parameter S which resolves to a natural number:

$$\begin{array}{rcl} \sigma \in \mathit{BrSeq}\ A\ S & \widehat{=} & \textsc{Nil} \\ & | & \textsc{Cons}\ A\ (\mathit{BrSeq}\ A\ S) \\ & | & \textsc{Split}\ S\ (\mathit{BrSeq}\ A\ S)^\star \end{array}$$

The split parameter always resolves to a number (it is total in some sense) and that number is always in the range $0 \ldots n$, where n is the number of branches. How S is instantiated will be explained shortly. We shall adopt the following shorthands in the sequel:

Longhand	Shorthand
$\textsc{Nil}$	Λ
$\textsc{Cons}\ a\ \sigma$	$a : \sigma$
$a : b : c : \Lambda$	$\langle a, b, c \rangle$
$a : \Lambda$	a— if no ambiguity arises
$\textsc{Split}\ s\ \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$	$s(\sigma_1 \vert \sigma_2 \vert \ldots \vert \sigma_n)$

In passing we note that, given a special value $\mathbf{1} \in S$ which always resolves to 1, we can assert the following equivalence between a 1-way branch and its sole constituent sequence:

$$\mathbf{1}(\sigma) \cong \sigma$$

These does admit an alternative formulation, explained later — the resulting notation is more clumsy, but the operator definitions are simpler and it might be easier to use those to prove important (meta-)properties.

5.3 Branching Sequence Operators

We define two operators on branching sequences.

The first is a generalisation of sequence concatenation ($\ddagger$):

$$\begin{array}{rcl} \ddagger & : & \mathit{BrSeq}\ A\ S \times \mathit{BrSeq}\ A\ S \to \mathit{BrSeq}\ A\ S \\ \Lambda \ddagger \sigma & \widehat{=} & \sigma \\ \sigma \ddagger \Lambda & \widehat{=} & \sigma \\ a : \sigma_1 \ddagger \sigma_2 & \widehat{=} & a : (\sigma_1 \ddagger \sigma_2) \\ s(\sigma_1 | \ldots | \sigma_n) \ddagger \sigma & \widehat{=} & s(\sigma_1 \ddagger \sigma | \ldots | \sigma_n \ddagger \sigma) \end{array}$$

The key feature to note here is that concatenation left-distributes through branching.

The second is a form of interleaving ($\between$).

5.3.1 Atomic and sub-Atomic Events

Here we are going to distinguish between two types of elements of A — *atomic* elements denoted a' (with a prime) and *sub-atomic* elements denoted simply by a (without a prime). Later, we show how we actually distinguish these events. The idea is that the sequencing captures events spaced by clock ticks, these being the atomic events. The prime signifies the 'tick' of the clock. However, there are various decision-making events that occur with a finer granularity - these are the sub-atomic events. Sub-atomic events always occupy the first part of a clock cycle and all are complete before the end of the cycle. Atomic events take all of the cycle to complete and always end at the clock edge denoting the end of the cycle. The split parameter ($s \in S$) behaves somewhat like a sub-atomic event. Our interleaving therefore gives sub-atomic events a form of priority over atomic ones:

$$\begin{array}{rcl} \between & : & \mathit{BrSeq}\ A\ S \times \mathit{BrSeq}\ A\ S \to \mathit{BrSeq}\ A\ S \\ \Lambda \between \sigma & \widehat{=} & \sigma \\ \sigma \between \Lambda & \widehat{=} & \sigma \\ a : \sigma_1 \between b : \sigma_2 & \widehat{=} & a \sqcup b : (\sigma_1 \between \sigma_2) \\ a : \sigma_1 \between \sigma_2 & \widehat{=} & a : (\sigma_1 \between \sigma_2) \\ \sigma_1 \between b : \sigma_2 & \widehat{=} & b : (\sigma_1 \between \sigma_2) \\ a' : \sigma_1 \between b' : \sigma_2 & \widehat{=} & a' \sqcup b' : (\sigma_1 \between \sigma_2) \\ a' : \sigma \between s(\sigma_1 | \ldots | \sigma_n) & \widehat{=} & s(a' : \sigma \between \sigma_1 | \ldots | a' : \sigma \between \sigma_n) \\ s(\sigma_1 | \ldots | \sigma_n) \between b' : \sigma & \widehat{=} & s(\sigma_1 \between b' : \sigma | \ldots | \sigma_n \between b' : \sigma) \\ s(\sigma_1 | \ldots | \sigma_m) \between t(\varsigma_1 | \ldots | \varsigma_n) & \widehat{=} & s \times_n t(\ \sigma_1 \between \varsigma_1 | \ldots | \sigma_1 \between \varsigma_n \\ & & \quad \ldots \\ & & \quad | \sigma_m \between \varsigma_1 | \ldots | \sigma_m \between \varsigma_n) \end{array}$$

Note that the clauses in the definition of $\between$ above must be tried in the order shown. We assume a means ($\sqcup$) of combining both sub-atomic and atomic events, which is associative. By $s \times_K t$ as a split parameter we mean one that returns the " K-index product" of what the two individual parameter s and t return. This is defined as:

$$0 \times_K j = 0 = i \times_K 0 \qquad i \times_K j = K(i-1) + j$$

This goes with an intuition that a split parameter produces a number $1 \ldots n$ which selects the split branch to follow. However we also allow for a split parameter to return 0, used to signal either error conditions, or a state in which outside intervention is required, as in external communication events.

It is worth noting that atomic events distribute through splits, whereas sub-atomic events never do so. This is because sub-atomic events preceding the split parameter (itself a sub-atomic event, remember) must continue to precede it in any interleaving. We also point out that this operator is commutative: In the context in which these branching sequences are used, we shall find that the following laws will always apply, even though they are not a consequence of the definition above:

$$\sigma_1 \between \sigma_2 = \sigma_2 \between \sigma_1$$

$$s(\sigma_1 | \ldots | \sigma_n) \between s(\varsigma_1 | \ldots | \varsigma_n) = s(\sigma_1 \between \varsigma_1 | \ldots | \sigma_n \between \varsigma_n)$$

5.4 Top Domain: Branching Sequence of Choices

We are going to model our semantics domain ($\mathcal{M}$) as a branch sequence of choices, where the split parameter is a (total) function from a world to a natural number:

$$\begin{aligned} s \in \mathit{WSplit} &\mathrel{\widehat{=}} \mathit{World} \to \mathbb{N} \\ \mathcal{M} &\mathrel{\widehat{=}} \mathit{BrSeq}\ \mathit{Choice}\ \mathit{WSplit} \end{aligned}$$

We shall always arrange things that the split parameter of an n-way split always returns a number in the range $1 \ldots n$.

5.4.1 Uniformity

We can define a choice as *uniform* if the domain of the resulting change is always the same, regardless of the world:

$$\begin{aligned} \mathrm{isUniform} &: \mathit{Choice} \to \mathbb{B} \\ \mathrm{isUniform}\ \kappa &\mathrel{\widehat{=}} \forall \omega_1, \omega_2 \in \mathit{World} \cdot \mathtt{dom}\, \kappa(\omega_1) = \mathtt{dom}\, \kappa(\omega_2) \end{aligned}$$

All choices generated by our semantics will be uniform, and uniformity will be conserved by all operators we employ that act on choices or changes, either inherently, or by the way in which we use them.

An atomic event is a uniform choice that, for all worlds, returns a change that increases the time value by one. As we only ever increase time by one, we shall

recognise this by observing if the time identifier is in the domain of the change map:

$$\begin{array}{rcl} \text{isAtomic} & : & \mathit{Choice} \to \mathbb{B} \\ \text{isAtomic}\ \kappa & \widehat{=} & \text{isUniform}\ \kappa \wedge \forall \omega \in \mathit{World} \cdot \tau \in \mathtt{dom}\ \kappa(\omega) \end{array}$$

A sub-atomic event is a uniform choice that does not change the time value in any world — we observe this by noting the absence of the time identifier in the map. Our semantics never introduces the time identifier in a choice unless there is a time increase.

$$\begin{array}{rcl} \text{isSubAtomic} & : & \mathit{Choice} \to \mathbb{B} \\ \text{isSubAtomic}\ \kappa & \widehat{=} & \text{isUniform}\ \kappa \wedge \forall \omega \in \mathit{World} \cdot \tau \notin \mathtt{dom}\ \kappa(\omega) \end{array}$$

Given an assumption of uniformity, we can come up with a simple efficient way of determining atomicity:

$$\begin{array}{rcl} \text{atomicityOf} & : & \mathit{Choice} \to \{\textsc{SubAtomic}, \textsc{Atomic}\} \\ \text{atomicityOf}\ \kappa & \widehat{=} & \tau \in \mathtt{dom}\ \kappa(\omega_0)\ \to\ \textsc{Atomic}\ ,\ \textsc{SubAtomic} \end{array}$$

We simply use the initial world as a representative input to the choice and look at the resulting change !

5.4.2 Event Combinator

We need to define the event (choice!) combining operator $\sqcup$.

$$\begin{array}{rcl} \sqcup & : & \mathit{Choice} \times \mathit{Choice} \to \mathit{Choice} \\ \kappa_1 \sqcup \kappa_2 & \widehat{=} & \lambda \omega \cdot \kappa_1(\omega) \uplus \kappa_2(\omega) \end{array}$$

We need to define the behaviour of $\uplus$, which acts on changes. We note that for disjoint portions of the domains we simply extend, but for intersecting parts of the domains we need to take special action, which may include assigning an error result:

$$\begin{array}{rcl} \uplus & : & \mathit{Change} \times \mathit{Change} \to \mathit{Change} \\ \delta_1 \uplus \delta_2 & \widehat{=} & \lhd[\delta_2]\delta_1 \sqcup (\lhd[\delta_2]\delta_1 \uplus' \lhd[\delta_1]\delta_2) \sqcup \lhd[\delta_1]\delta_2 \end{array}$$

Simultaneous assignments to program variables are errors, as are attempts to give time different values, give active channel states with different expressions, or combine request states with non-zero selector values. Request states combine by concatenating the sequences, except if the second has a null sequence, in which case it overrides the first:

$$\begin{array}{rcl} \uplus' & : & \mathit{Change} \times \mathit{Change} \to \mathit{Change} \\ \{v \mapsto e_1\} \uplus' \{v \mapsto e_2\} & \widehat{=} & \{v \mapsto \bot\} \\ \{\tau \mapsto t_1\} \uplus' \{\tau \mapsto t_2\} & \widehat{=} & \{\tau \mapsto (t_1 = t_2\ \to\ t_1\ ,\ \bot)\} \\ \{c \mapsto \textsc{Act}\ e_1\} \uplus' \{c \mapsto \textsc{Act}\ e_2\} & \widehat{=} & \{c \mapsto \bot\} \\ \{c \mapsto \textsc{Act}\ e_1\} \uplus' \{c \mapsto \textsc{Idle}\} & \widehat{=} & \{c \mapsto \textsc{Act}\ e_1\} \\ \{c \mapsto \textsc{Idle}\} \uplus' \{c \mapsto \textsc{Act}\ e_2\} & \widehat{=} & \{c \mapsto \textsc{Act}\ e_2\} \\ \{c \mapsto \textsc{Idle}\} \uplus' \{c \mapsto \textsc{Idle}\} & \widehat{=} & \{c \mapsto \textsc{Idle}\} \\ \{\ell \mapsto (n, \sigma)\} \uplus' \{\ell \mapsto (0, \Lambda)\} & \widehat{=} & \{\ell \mapsto (0, \Lambda)\} \\ \{\ell \mapsto (n_1, \sigma_1)\} \uplus' \{\ell \mapsto (n_2, \sigma_2)\} & \widehat{=} & \{\ell \mapsto (n_1 + n_2 = 0\ \to\ (n_1, \sigma_1 \frown \sigma_2)\ ,\ \bot)\} \end{array}$$

We need to show the following propositions, assuming that no errors occur:

$$\begin{aligned} \forall[\text{isUniform}]\{\kappa_1, \kappa_2\} &\Rightarrow \text{isUniform}(\kappa_1 \sqcup \kappa_2) \\ \forall[\text{isAtomic}]\{\kappa_1, \kappa_2\} &\Rightarrow \text{isAtomic}(\kappa_1 \sqcup \kappa_2) \\ \forall[\text{isSubAtomic}]\{\kappa_1, \kappa_2\} &\Rightarrow \text{isSubAtomic}(\kappa_1 \sqcup \kappa_2) \end{aligned}$$

5.4.3 Useful Shorthands

Most of our choices will be of the form:

$$\lambda\omega \cdot \{\ldots, v \mapsto [\![e]\!]_\omega, \ldots\}$$

with atomic choices containing a timestep:

$$\lambda\omega \cdot \{\ldots, \tau \mapsto \omega(\tau) + 1, \ldots\}$$

We shall adopt the following shorthands: (i) we omit the $\lambda\omega \cdot$, it being understood as always present, (ii) we denote expression evaluation by $\underline{e}$ rather than $[\![e]\!]_\omega$ or $\lambda\omega \cdot [\![e]\!]_\omega$, and (iii) we omit the timestamp 'maplet' and instead add a 'tick-mark' to the map expression. We also use θ to denote the empty choice. We can summarise the shorthands as follows:

Longhand	Shorthand
$\lambda\omega \cdot \{\ldots, v \mapsto [\![e]\!]_\omega, \ldots\}$	$\{\ldots, v \mapsto \underline{e}, \ldots\}$
$\lambda\omega \cdot \{\ldots, \tau \mapsto \omega(\tau) + 1, \ldots\}$	$\{\ldots, \ldots\}'$
$\lambda\omega \cdot \theta$	θ
$\lambda\omega \cdot \{\tau \mapsto \omega(\tau) + 1\}$	θ'
$[\![e]\!]_\omega$	$\underline{e}$
$\lambda\omega \cdot [\![e]\!]_\omega$	$\underline{e}$

We shall also use κ and κ' to denote arbitrary subatomic and atomic choices respectively.

5.5 Pruning

Given a program we will be able to compute the corresponding branch tree for that program. We can then take that sequence, and an initial world, and proceed to 'prune' the branches, to obtain a sequence of worlds, effectively being the synchronous trace of the program execution. If we have a closed program (no channels to outside), this resolves to such a sequence for the whole program. If we have an open program, we can only produce a world sequence up to the first external channel request. Then we must also return the remaining branch sequence. At this point we need to use knowledge about the world to determine what happens next. We shall also interpret the occurence of an error ($\perp$) at any

stage as a form of external event, aborting any further pruning at that point:

$$\begin{array}{rcl}
\text{prune} & : & \mathit{World} \to \mathcal{M} \to \mathit{World}^\star \times \mathcal{M} \\
\text{prune}[\omega]\Lambda & \widehat{=} & (\Lambda, \Lambda) \\
\text{prune}[\omega](\kappa : \sigma) & \widehat{=} & \textbf{let } \varpi = \omega \dagger \omega(\kappa) \textbf{ in} \\
 & & \bot \in \mathtt{rng}\ \varpi \ \to\ (\langle \varpi \rangle, \sigma)\ ,\ \text{prune}[\varpi]\sigma \\
\text{prune}[\omega](\kappa' : \sigma) & \widehat{=} & \textbf{let } \varpi = \omega \dagger \omega(\kappa) \textbf{ in} \\
 & & \textbf{let } (\Omega, \Upsilon) = \text{prune}[\varpi]\sigma \textbf{ in} \\
 & & \bot \in \mathtt{rng}\ \varpi \ \to\ (\langle \varpi \rangle, \sigma)\ ,\ (\varpi : \Omega, \Upsilon) \\
\text{prune}[\omega](s(\sigma_1 | \ldots | \sigma_n)) & \widehat{=} & \textbf{let } i = s(\omega) \textbf{ in} \\
 & & i = 0 \ \to\ (\Lambda, s(\sigma_1 | \ldots | \sigma_n))\ ,\ \text{prune}[\omega]\sigma_i
\end{array}$$

This pruning function provides a context in which some key properties can be demonstrated. For example, the following property was stated earlier

$$s(\sigma_1 | \ldots | \sigma_n) \between s(\varsigma_1 | \ldots | \varsigma_n) = s(\sigma_1 \between \varsigma_1 | \ldots | \sigma_n \between \varsigma_n)$$

By saying this is true "in a certain context", what we actually meant is that pruning either side w.r.t. an arbitrary world always gives the same result:

$\forall \omega \in \mathit{World}$
- $\text{prune}[\omega](s(\sigma_1 | \ldots | \sigma_n) \between s(\varsigma_1 | \ldots | \varsigma_n)) = \text{prune}[\omega](s(\sigma_1 \between \varsigma_1 | \ldots | \sigma_n \between \varsigma_n))$

6 The Statement Semantics

We are now finally in a position to give the denotational semantics of the statements of Handel-C. We map statements to branching sequences:

$$[\![\_]\!] \ : \ S \to \mathcal{M}$$

We now present the details for each statement.

6.1 Delay Semantics

The delay statement simply waits n clock ticks, while doing nothing else in the meantime.

$$\begin{array}{rcl}
[\![\delta_0]\!] & \widehat{=} & \Lambda \\
[\![\delta_n]\!] & \widehat{=} & \theta' : [\![\delta_{n-1}]\!]
\end{array}$$

6.2 Assignment Semantics

Assignment is very traditional and takes exactly one cycle

$$[\![v := e]\!] \ \widehat{=} \ \{v \mapsto \underline{e}\}'$$

6.3 Sequencing Semantics

Sequencing in the program becomes branching sequence concatenation:

$$[\![s_1 ; s_2]\!] \ \widehat{=} \ [\![s_1]\!] \ddagger [\![s_2]\!]$$

6.4 Parallel Semantics

Parallelism in the program becomes branching sequence interleaving:

$$[\![s_1 \parallel s_2]\!] \quad \widehat{=} \quad [\![s_1]\!] \between [\![s_2]\!]$$

6.5 Conditional Semantics

We don't try to resolve conditionals now, we simply take all possible options, i.e produce an n-way branch point.

$$[\![e \ \rightarrow \ s_1 \ , \ \ldots , s_n]\!] \quad \widehat{=} \quad \underline{e}(\ [\![s_1]\!] \mid \ldots \mid [\![s_n]\!] \ )$$

6.6 While Semantics

As is typical for imperative languages, we give the while-loop semantics using fixpoints:

$$[\![b * s]\!] \quad \widehat{=} \quad \mathtt{fix}_{\mathcal{W}}(\ \underline{b}([\![s]\!] \ddagger \mathcal{W} \mid \theta) \ )$$

This basically gives an infinite tree at this point as we have

$$\begin{array}{rcl} \mathcal{W} & = & \underline{b}(\ [\![s]\!] \ddagger \mathcal{W} \mid \theta \ ) \\ & = & \underline{b}(\ [\![s]\!] \ddagger \underline{b}(\ [\![s]\!] \ddagger \mathcal{W} \mid \theta \ ) \mid \theta \ ) \\ & = & \underline{b}(\ [\![s]\!] \ddagger \underline{b}(\ [\![s]\!] \ddagger \underline{b}(\ [\![s]\!] \ddagger \mathcal{W} \mid \theta \ ) \mid \theta \ ) \mid \theta \ ) \\ & = & \ldots \end{array}$$

Pruning will produce a finite sequence if the program actually terminates.

6.7 Request Semantics

We rely on the fact that requests for a communications statement are made in order of priority, so that we do not need to explicitly mention priority.

$$[\![\Re(\ell, p, c, d)]\!] \quad \widehat{=} \quad \{\ell \mapsto (0, \langle (c, d) \rangle)\}$$

The $\uplus'$ operator takes care of assembling the properly sequenced communication state.

6.8 Release Semantics

We release communication requests by simply returning back to the state with no requests:

$$[\![\mho(\ell)]\!] \quad \widehat{=} \quad \{\ell \mapsto (0, \Lambda)\}$$

Again the $\uplus'$ operator takes care of ensuring that a release doesn't simply append the empty request list.

The following statements are strictly speaking not in our restricted language, but they can be given immediate semantics via their translation into the reduced language:

$$
\begin{array}{rcl}
g_\ell & \mapsto & \Re(\ell,1,g);\ w(\ell)*\delta_1;\ \mho(\ell);\mathrm{Act}(g) \\
\langle g_1:s_1,\ldots,g_n:s_n\rangle_\ell & \mapsto & \Re(\ell,1,g_1);\ldots;\Re(\ell,n,g_n); \\
 & & w(\ell)*\delta_1; \\
 & & s(\ell)\ \rightarrow\ \mho(\ell);\mathrm{Act}(g_1);s_1\ ,\ \ldots,\mho(\ell);\mathrm{Act}(g_n);s_n \\
\textbf{where} & & \\
\mathrm{Act}(!?) & \widehat{=} & \delta_0 \\
\mathrm{Act}(c!e) & \widehat{=} & \delta_1 \\
\mathrm{Act}(c?v) & \widehat{=} & v:=e(c)
\end{array}
$$

6.9 Channel Write Semantics

$$
[\![c!e_\ell]\!] \ \widehat{=}\ [\![\Re(\ell,1,c,!e);\ w(\ell)*\delta_1;\ \mho(\ell);\delta_1]\!]
$$

6.10 Channel Read Semantics

$$
[\![c?v_\ell]\!] \ \widehat{=}\ [\![\Re(\ell,1,c,?v);\ w(\ell)*\delta_1;\ \mho(\ell);v:=e(c)]\!]
$$

6.11 Prialt Semantics

$$
\begin{array}{rcl}
[\![\langle g_1:s_1,\ldots,g_n:s_n\rangle_\ell]\!] & \widehat{=} & [\![\ \Re(\ell,1,g_1);\ldots;\Re(\ell,n,g_n); \\
 & & w(\ell)*\delta_1; \\
 & & s(\ell)\ \rightarrow\ \mho(\ell);\mathrm{Act}(g_1);s_1\ ,\ \ldots,\mho(\ell);\mathrm{Act}(g_n);s_n\ ]\!]
\end{array}
$$

7 The Expression Semantics

We also need to give the denotational semantics to the expressions of Handel-C. We map expressions to datum values:

$$
[\![\_]\!] \ :\ E \rightarrow \mathit{World} \rightarrow \mathit{Datum}
$$

We now present the details for key expressions:

7.1 Normal Expressions

$$
[\![e]\!]_\omega \ \widehat{=}\ \text{traditional}^\dagger\ \text{expression evaluation}
$$

†: in order to allow uniform handling of conditionals of any arity, we represent the boolean values TRUE and FALSE by the numbers 1 and 2 respectively. We do not use 1 and 0, as might be expected, for reasons which are both practical (everything works better our way) and theoretical (booleans are not numbers, even in any abstract sense).

7.2 Wait Expression

We first look at the world and resolve all the various requests for communication by determining which communications statements (as indicated by their labels) have become ready to execute:

$$\begin{array}{lcl} [\![w(\ell)]\!]_\omega & \widehat{=} & \textbf{let } \varpi = \text{resolve}(\omega) \\ & & \textbf{in } \pi_1(\varpi(\ell)) = 0 \end{array}$$

Note, as for basic expressions, if the value looked up is zero (i.e the wait condition is true) then the number returned here is 1, otherwise 2 is returned.

7.3 Select Expression

The result of this is only a valid selector if the given communication statement has been resolved as ready to execute.

$$\begin{array}{lcl} [\![s(\ell)]\!]_\omega & \widehat{=} & \textbf{let } \varpi = \text{resolve}(\omega) \\ & & \textbf{in } \pi_1(\varpi(\ell)) \end{array}$$

7.4 Channel's Active Expression

The value of this expression is only defined if the channel is in the active state, in which case the expression is evaluated and returned

$$\begin{array}{lcl} [\![e(c)]\!]_\omega & \widehat{=} & \textbf{case } \omega(c) \\ & & \quad \textsc{Idle} \ \rightarrow \ \bot \\ & & \quad \textsc{Act}\ e \ \rightarrow \ \underline{e} \end{array}$$

8 Resolving Communication Requests

The question of how communication requests get resolved is not handled here. The Handel-C language reference manual is not very clear on this point, and so the resolution of this issue has been left to another technical report. We need to capture the resolution by the following function:

$$\text{resolve} : \mathit{World} \rightarrow \mathit{World}$$

It looks at all the communication states, works out which ones should be active, and marks them so by changing the entry

$$\{\ell \mapsto (0, \sigma)\}$$

to the entry

$$\{\ell \mapsto (s, \sigma)\}$$

where $s \in \{1 \ldots n\}$ designates which request is going to be serviced. We don't bother recording the results of this resolution in the world, simply because all selected communication statements proceed to release their requests before actually doing the communication action itself.

8.1 Communication Resolution

The problem can be stated most clearly as follows (in a form which is slightly different to that used in the semantics):

Given a map of requests

$$\rho \in L \to (C \times DReq)^\star$$

return a map of selections

$$\varsigma \in L \to \mathbb{N}$$

that satisfies the following requirements:

- a selection number is returned for every label:

$$\mathtt{dom}\,\varsigma = \mathtt{dom}\,\rho$$

- For every label, selection values are either zero to signify non-selection, or else are an index into the request sequence

$$\forall \ell : \mathtt{dom}\,\varsigma \cdot \varsigma(\ell) \in \{\, 0 \ldots \mathtt{len}\,\rho(\ell)\}$$

- A label gets a non-zero selection value only if the selected request has been matched up with a complementary request associated with another label

$$\begin{array}{rcl} \varsigma(\ell) > 0 & \Rightarrow & \\ \exists \ell' : \mathtt{dom}\,\varsigma & \cdot & \ell' \neq \ell \\ & \wedge & \varsigma(\ell') > 0 \\ & \wedge & \mathrm{compReq}(\ \rho(\ell)[\varsigma(\ell)]\ ,\ \rho(\ell')[\varsigma(\ell')]\ ) \end{array}$$

  Complementary requests have the same channel and opposite direction:

$$\begin{array}{rcl} \mathrm{compReq} & : & (C \times DReq)^2 \to \mathbb{B} \\ \mathrm{compReq}((c, \textsc{In}\ v), (c, \textsc{Out}\ e)) & \widehat{=} & \textsc{True} \\ \mathrm{compReq}((c, \textsc{Out}\ e), (c, \textsc{In}\ v)) & \widehat{=} & \textsc{True} \\ \mathrm{compReq}((c_1, d_1)(c_2, d_2)) & \widehat{=} & \textsc{False} \end{array}$$

- Any non-zero selection number associated with a given label identifies a channel. Any given channel is identified in this manner either exactly twice, or not at all. If so identified twice, the two instances must be complementary

$$\begin{array}{rl} \forall c \in (\bigcup\!/ \circ \mathcal{P}(\mathtt{elems} \circ \pi_1{}^\star) \circ \mathtt{rng})\rho & \\ \quad \cdot & \neg\exists \ell \cdot \varsigma(\ell) > 0 \wedge \pi_1(\rho[\varsigma(\ell)]) = c \\ \quad \vee & \exists \ell_1, \ell_2 \cdot \\ & \quad \varsigma(\ell_1) > 0 \wedge \pi_1(\rho[\varsigma(\ell_1)]) = c \\ & \wedge\ \varsigma(\ell_2) > 0 \wedge \pi_1(\rho[\varsigma(\ell_2)]) = c \\ & \wedge\ \neg\exists \ell \cdot (\ \ell \notin \{\ell_1, \ell_2\} \wedge \varsigma(\ell) > 0 \wedge \pi_1(\rho[\varsigma(\ell)]) = c\ ) \end{array}$$

- The sequencing of requests denotes priorities — the resolution should endeavour to respect these priorities. Each prialt induces an ordering on channels. When combined for all active prialts, these should resolve into a global ordering. However, certain prialt configurations are outlawed at compile time, usually with a reference to "combinatorial cycles". These involve cyclic dependencies between the labelled requests and their priorities. For example, the following is flagged as illegal at compile-time

$$\langle a?v_1 : s_1, b!e_2 : s_2\rangle_{\ell_1} \parallel \langle b?v_3 : s_3, a!e_4 : s_4\rangle_{\ell_2}$$

which would result in the following request map

$$\{\ \ell_1 \mapsto \langle (a, \textsc{In}\ v_1), (b, \textsc{Out}\ e_2)\rangle,\ \ell_2 \mapsto \langle (b, \textsc{In}\ v_3), (a, \textsc{Out}\ e_4)\rangle\ \}$$

9 Some Examples

We present worked out semantics for some examples, some straightforward, the others pathological. In well-behaved examples, we will use a simplified version of prune (prn) which has no error handling and returns a world sequence only:

$$\begin{array}{rcl}
\mathrm{prn} & : & \mathit{World} \to \mathcal{M} \to \mathit{World}^\star \\
\mathrm{prn}[\omega]\Lambda & \widehat{=} & \Lambda \\
\mathrm{prn}[\omega](\kappa : \sigma) & \widehat{=} & \mathrm{prune}[\omega \dagger \omega(\kappa)]\sigma \\
\mathrm{prn}[\omega](\kappa' : \sigma) & \widehat{=} & \textbf{let}\ \varpi = \omega \dagger \omega(\kappa)\ \textbf{in}\ \varpi : \mathrm{prn}[\varpi]\sigma \\
\mathrm{prn}[\omega](s(\sigma_1|\ldots|\sigma_n)) & \widehat{=} & \textbf{let}\ i = s(\omega)\ \textbf{in}\ \mathrm{prune}[\omega]\sigma_i
\end{array}$$

9.1 Parallel Interference

Here we have two threads sharing global variables:

$$x := 1; x := x + y \parallel y := 2; y := y - x;$$

The initial world is easily computed to be as

$$\omega_0 = \{\tau \mapsto 0, x \mapsto ?, y \mapsto ?\}$$

To keep things compact we shall define the following shorthands:

$$a' \widehat{=} \{x \mapsto \underline{1}\}' \quad b' \widehat{=} \{y \mapsto \underline{2}\}' \quad c' \widehat{=} \{x \mapsto \underline{x + y}\}' \quad d' \widehat{=} \{y \mapsto \underline{y - x}\}'$$

Our semantics immediately gives:

$$\begin{array}{ll}
 & \mathrm{prn}[\omega_0][\![x := 1; x := x + y \parallel y := 2; y := y - x;]\!] \\
= & \quad \langle \text{parallel semantics}\rangle \\
 & \mathrm{prn}[\omega_0]([\![x := 1; x := x + y]\!] \between [\![y := 2; y := y - x;]\!]) \\
= & \quad \langle \text{sequential semantics}\rangle \\
 & \mathrm{prn}[\omega_0](([\![x := 1]\!] \ddagger [\![x := x + y]\!]) \between ([\![y := 2]\!] \ddagger [\![y := y - x;]\!])) \\
= & \quad \langle \text{assignment semantics, shorthand}\rangle \\
 & \mathrm{prn}[\omega_0](\langle a'\rangle \ddagger \langle c'\rangle) \between (\langle b'\rangle \ddagger \langle d'\rangle)) \\
= & \quad \langle \text{defn. of } \ddagger, \text{ using cons form}\rangle
\end{array}$$

$$
\begin{array}{ll}
 & \mathrm{prn}[\omega_0]((a' : c' : \Lambda) \between (b' : d' : \Lambda)) \\
= & \quad \langle \text{defn. of } \between, \text{ using cons form} \rangle \\
 & \mathrm{prn}[\omega_0](a' \sqcup b' : c' \sqcup d' : \Lambda) \\
= & \quad \langle \text{defn. of } \sqcup, \text{ longhand form} \rangle \\
 & \mathrm{prn}[\omega_0](\{x \mapsto \underline{1}, y \mapsto \underline{2}\}' : \{x \mapsto \underline{x+y}, y \mapsto \underline{y-x}\}' : \Lambda) \\
= & \quad \langle \text{defn. of prn} \rangle \\
 & \omega_1 : \mathrm{prn}[\omega_1](\{x \mapsto \underline{x+y}, y \mapsto \underline{y-x}\}' : \Lambda) \\
 & \quad \textbf{where } \omega_1 = \{\tau \mapsto 1, x \mapsto 1, y \mapsto 2\} \\
= & \quad \langle \text{defn. of prn} \rangle \\
 & \omega_1 : \omega_2 : \mathrm{prn}[\omega_1]\Lambda \\
 & \quad \textbf{where } \omega_2 = \{\tau \mapsto 2, x \mapsto 3, y \mapsto 1\} \\
= & \quad \langle \text{defn. of prn} \rangle \\
 & \langle \omega_1, \omega_2 \rangle \\
= & \quad \langle \text{expand out} \rangle \\
 & \langle \{\tau \mapsto 1, x \mapsto 1, y \mapsto 2\}, \{\tau \mapsto 2, x \mapsto 3, y \mapsto 1\} \rangle
\end{array}
$$

Which is as expected !

9.2 Delayed Communication

Here we have two threads communicating with each other, but where one side has to wait:

$$x := 5; c!x_1 \parallel c?y_2; y := y + 1$$

The initial world is

$$\{\tau \mapsto 0, x \mapsto ?, y \mapsto ?, c \mapsto \textsc{Idle}, 1 \mapsto (0, \Lambda), 2 \mapsto (0, \Lambda)\}$$

At this point, we shall introduce the following shorthands:

$$
\begin{array}{rlrl}
 & x5' \mathrel{\widehat{=}} \{x \mapsto \underline{5}\}' & & \\
 & c1 \mathrel{\widehat{=}} \{1 \mapsto (0, \langle (c, !x) \rangle)\} & & r1 \mathrel{\widehat{=}} \{1 \mapsto (0, \Lambda)\} \\
 & c2 \mathrel{\widehat{=}} \{2 \mapsto (0, \langle (c, ?y) \rangle)\} & & r2 \mathrel{\widehat{=}} \{2 \mapsto (0, \Lambda)\} \\
 & ye' \mathrel{\widehat{=}} \{y \mapsto \underline{e(c)}\}' & & y1' \mathrel{\widehat{=}} \{y \mapsto \underline{y+1}\}'
\end{array}
$$

We compute the semantics as:

$$
\begin{array}{ll}
 & \mathrm{prn}[\omega_0][\![x := 5; c!x_1 \parallel c?y_2; y := y + 1]\!] \\
= & \quad \langle \text{semantics of } \parallel \rangle \\
 & \mathrm{prn}[\omega_0]([\![x := 5; c!x_1]\!] \between [\![c?y_2; y := y + 1]\!]) \\
= & \quad \langle \text{semantics of } ; \rangle \\
 & \mathrm{prn}[\omega_0](([\![x := 5]\!] \ddagger [\![c!x_1]\!]) \between ([\![c?y_2]\!] \ddagger [\![y := y + 1]\!])) \\
= & \quad \langle \text{semantics of } := \rangle \\
 & \mathrm{prn}[\omega_0]((x5' \ddagger [\![c!x_1]\!]) \between ([\![c?y_2]\!] \ddagger y1')) \\
= & \quad \langle \text{semantics of } c!x_1 \rangle \\
 & \mathrm{prn}[\omega_0]((x5' \ddagger [\![\Re(1, 1, c, !x);\ w(1) * \delta_1;\ \mho(1); \delta_1]\!]) \between ([\![c?y_2]\!] \ddagger y1')) \\
= & \quad \langle \text{semantics of } c?y_2 \rangle \\
 & \mathrm{prn}[\omega_0]((x5' \ddagger [\![\Re(1, 1, c, !x);\ w(1) * \delta_1;\ \mho(1); \delta_1]\!]) \\
 & \qquad \between ([\![\Re(2, 1, c, ?y);\ w(2) * \delta_1;\ \mho(2); y := e(c)]\!] \ddagger y1')) \\
= & \quad \langle \text{semantics of } ;, \text{ assoc. of } \ddagger \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \mathrm{prn}[\omega_0]((x5' \ddagger [\![\Re(1,1,c,!x)]\!] \ddagger [\![w(1) * \delta_1]\!] \ddagger [\![\mho(1)]\!] \ddagger [\![\delta_1]\!]) \\
 & \qquad \between ([\![\Re(2,1,c,?y)]\!] \ddagger [\![w(2) * \delta_1]\!] \ddagger [\![\mho(2)]\!] \ddagger [\![y := e(c)]\!] \ddagger y1')) \\
= & \quad \langle \text{semantics of } \delta_1 \text{ and } :=\rangle \\
 & \mathrm{prn}[\omega_0](x5' \ddagger [\![\Re(1,1,c,!x)]\!] \ddagger [\![w(1) * \delta_1]\!] \ddagger [\![\mho(1)]\!] \ddagger \theta') \\
 & \qquad \between ([\![\Re(2,1,c,?y)]\!] \ddagger [\![w(2) * \delta_1]\!] \ddagger [\![\mho(2)]\!] \ddagger ye') \ddagger y1')) \\
= & \quad \langle \text{semantics of } \mho\rangle \\
 & \mathrm{prn}[\omega_0](x5' \ddagger [\![\Re(1,1,c,!x)]\!] \ddagger [\![w(1) * \delta_1]\!] \ddagger r1 \ddagger \theta') \\
 & \qquad \between ([\![\Re(2,1,c,?y)]\!] \ddagger [\![w(2) * \delta_1]\!] \ddagger r2 \ddagger ye' \ddagger y1')) \\
= & \quad \langle \text{semantics of } \Re\rangle \\
 & \mathrm{prn}[\omega_0]((x5' \ddagger c1 \ddagger [\![w(1) * \delta_1]\!] \ddagger r1 \ddagger \theta') \\
 & \qquad \between (c2 \ddagger [\![w(2) * \delta_1]\!] \ddagger r2 \ddagger ye' \ddagger y1')) \\
= & \quad \langle \text{semantics of } b * s \text{ and } \delta_1\rangle \\
 & \mathrm{prn}[\omega_0]((x5' \ddagger c1 \ddagger \mathtt{fix}_{\mathcal{U}}(\ \underline{w(1)}(\theta' \ddagger \mathcal{U} \mid \theta)\ ) \ddagger r1 \ddagger \theta') \\
 & \qquad \between (c2 \ddagger \mathtt{fix}_{\mathcal{V}}(\ \underline{w(2)}(\theta' \ddagger \mathcal{V} \mid \theta)\ ) \ddagger r2 \ddagger ye' \ddagger y1')) \\
= & \quad \langle \text{introduce where-clauses}\rangle \\
 & \mathrm{prn}[\omega_0]((x5' \ddagger c1 \ddagger \mathcal{U} \ddagger r1 \ddagger \theta') \ \between\ (c2 \ddagger \mathcal{V} \ddagger r2 \ddagger ye' \ddagger y1')) \\
 & \textbf{where } \mathcal{U} = \mathtt{fix}_{\mathcal{U}}(\ \underline{w(1)}(\theta' \ddagger \mathcal{U} \mid \theta)\ ) \\
 & \textbf{where } \mathcal{V} = \mathtt{fix}_{\mathcal{V}}(\ \underline{w(2)}(\theta' \ddagger \mathcal{V} \mid \theta)\ ) \\
= & \quad \langle \text{defn. of } \ddagger\rangle \\
 & \mathrm{prn}[\omega_0]((x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \ \between\ (c2 : \mathcal{V} \ddagger r2 : ye' : y1')) \\
= & \quad \langle \text{defn. of } \between\rangle \\
 & \mathrm{prn}[\omega_0](c2 : ((x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \ \between\ (\mathcal{V} \ddagger r2 : ye' : y1'))) \\
= & \quad \langle \text{unfold } \mathcal{V}\rangle \\
 & \mathrm{prn}[\omega_0](c2 : ((x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \ \between\ (\underline{w(2)}(\theta' \ddagger \mathcal{V} \mid \theta) \ddagger r2 : ye' : y1'))) \\
= & \quad \langle \text{defn of } \ddagger\rangle \\
 & \mathrm{prn}[\omega_0](c2 : ((x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \\
 & \qquad \between\ (\underline{w(2)}(\theta' : \mathcal{V} \ddagger r2 : ye' : y1' \mid \theta : r2 : ye' : y1'))) \\
= & \quad \langle \text{defn of } \between\rangle \\
 & \mathrm{prn}[\omega_0](c2 : \underline{w(2)}(\ (x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta' : \mathcal{V} \ddagger r2 : ye' : y1') \\
 & \qquad \mid (x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta : r2 : ye' : y1')\ )) \\
= & \quad \langle \text{defn of prn}\rangle \\
 & \mathrm{prn}[\omega_0^1]\underline{w(2)}(\ (x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta' : \mathcal{V} \ddagger r2 : ye' : y1') \\
 & \qquad \mid (x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta : r2 : ye' : y1')\ ) \\
 & \textbf{where } \omega_0^1 = \omega_0 \dagger \{2 \mapsto (0, \langle (c, ?y)\rangle)\} \\
= & \quad \langle \text{defn of prn}\rangle \\
 & \textbf{let } i = \underline{w(2)}\omega_0^1 \textbf{ in} \\
 & \mathrm{prn}[\omega_0^1](\langle\ (x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta' : \mathcal{V} \ddagger r2 : ye' : y1') \\
 & \qquad ,\ (x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta : r2 : ye' : y1')\ \rangle[i]) \\
= & \quad \langle \text{Channel } c \text{ is not ready, } i = 1\rangle \\
 & \mathrm{prn}[\omega_0^1]((x5' : c1 : \mathcal{U} \ddagger r1 : \theta') \between (\theta' : \mathcal{V} \ddagger r2 : ye' : y1')) \\
= & \quad \langle \text{defn of } \between\rangle \\
 & \mathrm{prn}[\omega_0^1](x5' \sqcup \theta' : ((c1 : \mathcal{U} \ddagger r1 : \theta') \between (\mathcal{V} \ddagger r2 : ye' : y1'))\ ) \\
= & \quad \langle \text{defn of } \sqcup, \uplus\rangle \\
 & \mathrm{prn}[\omega_0^1](x5' : ((c1 : \mathcal{U} \ddagger r1 : \theta') \between (\mathcal{V} \ddagger r2 : ye' : y1'))\ ) \\
= & \quad \langle \text{defn of prn}\rangle \\
 & \omega_1 : \mathrm{prn}[\omega_1]((c1 : \mathcal{U} \ddagger r1 : \theta') \between (\mathcal{V} \ddagger r2 : ye' : y1')) \\
 & \textbf{where } \omega_1 = \omega_0^1 \dagger \{\tau \mapsto 1, x \mapsto \underline{5}\} \\
= & \quad \langle \text{unfold } \mathcal{V}\rangle \\
 & \omega_1 : \mathrm{prn}[\omega_1]((c1 : \mathcal{U} \ddagger r1 : \theta') \between (\underline{w(2)}(\theta' \ddagger \mathcal{V} \mid \theta) \ddagger r2 : ye' : y1'))
\end{array}
$$

$$
\begin{array}{ll}
= & \langle \text{defn of } \ddagger \rangle \\
& \omega_1 : \text{prn}[\omega_1](\ (c1 : \mathcal{U} \ddagger r1 : \theta') \\
& \qquad \qquad \between (\underline{w(2)}(\theta' : \mathcal{V} \ddagger r2 : ye' : y1' \mid \theta : r2 : ye' : y1')) \ ) \\
= & \langle \text{defn of } \between \rangle \\
& \omega_1 : \text{prn}[\omega_1](c1 : (\ (\mathcal{U} \ddagger r1 : \theta') \\
& \qquad \qquad \between (\underline{w(2)}(\theta' : \mathcal{V} \ddagger r2 : ye' : y1' \mid \theta : r2 : ye' : y1')) \ )) \\
= & \langle \text{defn of prn} \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\ (\mathcal{U} \ddagger r1 : \theta') \\
& \qquad \qquad \between (\underline{w(2)}(\theta' : \mathcal{V} \ddagger r2 : ye' : y1' \mid \theta : r2 : ye' : y1')) \ )) \\
& \textbf{where } \omega_1^1 = \omega_1 \dagger \{1 \mapsto (0, \langle (c, !x) \rangle)\} \\
= & \langle \text{unfold } \mathcal{U} \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\ (\underline{w(1)}(\theta' \ddagger \mathcal{U} \mid \theta) \ddagger r1 : \theta') \\
& \qquad \qquad \between (\underline{w(2)}(\theta' : \mathcal{V} \ddagger r2 : ye' : y1' \mid \theta : r2 : ye' : y1')) \ )) \\
\\
= & \langle \text{defn of } \ddagger \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\ (\underline{w(1)}(\theta' \ddagger \mathcal{U} \ddagger r1 : \theta' \mid \theta : r1 : \theta')) \\
& \qquad \qquad \between (\underline{w(2)}(\theta' : \mathcal{V} \ddagger r2 : ye' : y1' \mid \theta : r2 : ye' : y1')) \ )) \\
= & \langle \text{defn of } \between \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\ \underline{w(1)} \times_2 \underline{w(2)} \ (\ \theta' \ddagger \mathcal{U} \ddagger r1 : \theta' \between \theta' : \mathcal{V} \ddagger r2 : ye' : y1' \\
& \qquad \qquad \qquad \qquad \mid \theta' \ddagger \mathcal{U} \ddagger r1 : \theta' \between \theta : r2 : ye' : y1' \\
& \qquad \qquad \qquad \qquad \mid \theta : r1 : \theta' \between \theta' : \mathcal{V} \ddagger r2 : ye' : y1' \\
& \qquad \qquad \qquad \qquad \mid \theta : r1 : \theta' \between \theta : r2 : ye' : y1' \ ) \ ) \\
= & \langle \text{communication on } c \text{ enabled } i, j = 2 \text{ and } 2 \times_2 2 = 4 \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\ \theta : r1 : \theta' \between \theta : r2 : ye' : y1' \ ) \\
= & \langle \text{defn of } \between \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\theta \sqcup \theta : (\ r1 : \theta' \between r2 : ye' : y1' \ ) \\
= & \langle \text{defn of } \sqcup, \uplus \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\theta : (\ r1 : \theta' \between r2 : ye' : y1' \ ) \ ) \\
= & \langle \text{defn of prn,} \omega \dagger \theta = \theta \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](\ r1 : \theta' \between r2 : ye' : y1' \ ) \\
= & \langle \text{defn of } \between \rangle \\
& \omega_1 : \text{prn}[\omega_1^1](r1 \sqcup r2 : (\ \theta' \between ye' : y1' \ )) \\
= & \langle \text{defn of } \sqcup \rangle \\
& \omega_1 : \text{prn}[\omega_1^2](\ \theta' \between ye' : y1' \ ) \\
& \textbf{where } \omega_1^2 = \omega_1^1 \dagger \{1 \mapsto (0, \Lambda), 1 \mapsto (0, \Lambda)\} \\
= & \langle \text{defn of } \between \rangle \\
& \omega_1 : \text{prn}[\omega_1^2](\theta' \sqcup ye' : (\Lambda \between y1' \ )) \\
= & \langle \text{defn of } \sqcup \rangle \\
& \omega_1 : \text{prn}[\omega_1^2](ye' : (\Lambda \between y1' \ )) \\
= & \langle \text{defn of prn} \rangle \\
& \omega_1 : \omega_2 : \text{prn}[\omega_2](\ \Lambda \between y1' \ ) \\
& \textbf{where } \omega_2 = \omega_1^2 \dagger \{\tau \mapsto 2, y \mapsto \underline{e(c)}\} \\
= & \langle \underline{e(c)} \text{ is the expression } y \text{ which evals to } 5. \rangle \\
& \omega_1 : \omega_2 : \text{prn}[\omega_2](\ \Lambda \between y1' \ ) \\
& \textbf{where } \omega_2 = \omega_1^2 \dagger \{y \mapsto 5\} \\
= & \langle \text{defn. of } \between \rangle \\
& \omega_1 : \omega_2 \text{prn}[\omega_2](y1') \\
= & \langle \text{defn. of prn,} \underline{y + 1} = 6 \rangle \\
& \omega_1 : \omega_2 : \omega_3 \text{prn}[\omega_3]\Lambda
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{where } \omega_3 = \omega_2 \dagger \{\tau \mapsto 3, y \mapsto 6\} \\
= & \quad \langle \text{defn. of prn} \rangle \\
 & \omega_1 : \omega_2 : \omega_3 : \Lambda \\
= & \quad \langle \text{expand out} \rangle \\
 & \langle \{\tau \mapsto 1, x \mapsto 5, y \mapsto ?, c \mapsto \textsc{Idle}, 1 \mapsto (0, \Lambda), 2 \mapsto (0, \langle (c, ?y) \rangle)\} \\
 & , \{\tau \mapsto 2, x \mapsto 5, y \mapsto 5, c \mapsto \textsc{Idle}, 1 \mapsto (0, \Lambda), 2 \mapsto (0, \Lambda\} \\
 & , \{\tau \mapsto 3, x \mapsto 5, y \mapsto 6, c \mapsto \textsc{Idle}, 1 \mapsto (0, \Lambda, 2 \mapsto (0, \Lambda\} \rangle
\end{array}
$$

We get the correct result, but we have glossed over how the communication was resolved and the channel c was mapped to idle throughout. This needs fixing.

9.3 Factorial 3

The program

$$(f := 1 \parallel x := 3); (x > 1) * (f := f * x; \parallel x := x - 1)$$

has initial world

$$\omega_0 = \{\tau \mapsto 0, f \mapsto ?, x \mapsto ?\}$$

and shorthands:

$$
\begin{array}{rlrl}
 & f1' \mathrel{\widehat{=}} \{f \mapsto \underline{1}\}' & & x3' \mathrel{\widehat{=}} \{x \mapsto \underline{3}\}' \\
 & fx' \mathrel{\widehat{=}} \{f \mapsto \underline{f * x}\}' & & xd' \mathrel{\widehat{=}} \{x \mapsto \underline{x - 1}\}' \\
 & & & fxd' \mathrel{\widehat{=}} \{f \mapsto \underline{f * x}, x \mapsto \underline{x - 1}\}'
\end{array}
$$

with semantics:

$$
\begin{array}{ll}
 & \text{prn}[\omega_0][\![(f := 1 \parallel x := 3); (x > 1) * (f := f * x; \parallel x := x - 1)]\!] \\
= & \quad \langle \text{semantics of ; and } \parallel, \text{ shorthands} \rangle \\
 & \text{prn}[\omega_0](\ (f1' \between x3') \ddagger [\![(x > 1) * (f := f * x; \parallel x := x - 1)]\!] \ ) \\
= & \quad \langle \text{defn of } \between, \sqcup \rangle \\
 & \text{prn}[\omega_0](\ (f1' \sqcup x3') : [\![(x > 1) * (f := f * x; \parallel x := x - 1)]\!] \ ) \\
= & \quad \langle \text{defn of prn}, \uplus \rangle \\
 & \omega_1 : \text{prn}[\omega_1][\![(x > 1) * (f := f * x; \parallel x := x - 1)]\!] \\
 & \textbf{where } \omega_1 = \{\tau \mapsto 1, f \mapsto 1, x \mapsto 3\} \\
= & \quad \langle \text{defn of } b * s \rangle \\
 & \omega_1 : \text{prn}[\omega_1](\ \texttt{fix}_{\mathcal{W}} \underline{(x > 1)}([\![f := f * x; \parallel x := x - 1]\!] \ddagger \mathcal{W} \mid \theta) \ ) \\
= & \quad \langle \text{use where-clause} \rangle \\
 & \omega_1 : \text{prn}[\omega_1]\mathcal{W} \\
 & \textbf{where } \mathcal{W} = \underline{(x > 1)}([\![f := f * x; \parallel x := x - 1]\!] \ddagger \mathcal{W} \mid \theta) \\
= & \quad \langle \text{semantics of } \parallel, \text{ defn of } \between \rangle \\
 & \omega_1 : \text{prn}[\omega_1]\mathcal{W} \\
 & \textbf{where } \mathcal{W} = \underline{(x > 1)}((fx' \sqcup xd') \ddagger \mathcal{W} \mid \theta) \\
= & \quad \langle \text{defn of } \sqcup, \uplus, \ddagger \rangle \\
 & \omega_1 : \text{prn}[\omega_1]\mathcal{W} \\
 & \textbf{where } \mathcal{W} = \underline{(x > 1)}(fxd' : \mathcal{W} \mid \theta) \\
= & \quad \langle \text{unfold of } \mathcal{W} \rangle \\
 & \omega_1 : \text{prn}[\omega_1](\ \underline{(x > 1)}(fxd' : \mathcal{W} \mid \theta) \ ) \\
= & \quad \langle \text{defn of prn}, \underline{x > 1} = True(1) \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \omega_1 : \mathrm{prn}[\omega_1](\ fxd' : \mathcal{W}\ ) \\
= & \quad \langle \text{defn of prn, } f * x = 1 * 3,\ x - 1 = 3 - 1 \rangle \\
 & \omega_1 : \omega_2 : \mathrm{prn}[\omega_2](\ \mathcal{W}\ ) \\
 & \textbf{where } \omega_2 = \{\tau \mapsto 2, f \mapsto 3, x \mapsto 2\} \\
= & \quad \langle \text{unfold of } \mathcal{W} \rangle \\
 & \omega_1 : \omega_2 : \mathrm{prn}[\omega_2](\ \underline{(x > 1)}(fxd' : \mathcal{W} \mid \theta)\ ) \\
= & \quad \langle \text{defn of prn, } \underline{x > 1} = True(1) \rangle \\
 & \omega_1 : \omega_2 : \mathrm{prn}[\omega_2](\ fxd' : \mathcal{W}\ ) \\
= & \quad \langle \text{defn of prn, } f * x = 3 * 2,\ x - 1 = 2 - 1 \rangle \\
 & \omega_1 : \omega_2 : \omega_3 : \mathrm{prn}[\omega_3](\ \mathcal{W}\ ) \\
 & \textbf{where } \omega_3 = \{\tau \mapsto 3, f \mapsto 6, x \mapsto 1\} \\
= & \quad \langle \text{unfold of } \mathcal{W} \rangle \\
 & \omega_1 : \omega_2 : \omega_3 \mathrm{prn}[\omega_3](\ \underline{(x > 1)}(fxd' : \mathcal{W} \mid \theta)\ ) \\
= & \quad \langle \text{defn of prn, } \underline{x > 1} = False(2) \rangle \\
 & \omega_1 : \omega_2 : \omega_3 \mathrm{prn}[\omega_3](\ \theta\ ) \\
= & \quad \langle \text{defn of prn} \rangle \\
 & \omega_1 : \omega_2 : \omega_3 \mathrm{prn}[\omega_3^1]\Lambda \\
 & \textbf{where } \omega_3^1 = \omega_3 \dagger \theta = \omega_3 \\
= & \quad \langle \text{defn of prn} \rangle \\
 & \langle \{\tau \mapsto 1, f \mapsto 1, x \mapsto 3\} \\
 & , \{\tau \mapsto 2, f \mapsto 3, x \mapsto 2\} \\
 & , \{\tau \mapsto 3, f \mapsto 6, x \mapsto 1\} \rangle
\end{array}
$$

We see factorial 3 computed as expected !

9.4 Nested While (Pathological)

Consider the program

$$\textsc{True} * (\textsc{False} * s)$$

where True and False will evaluate to 2 and 1 respectively. We obtain a very simple initial world:

$$\omega_0 = \{\tau \mapsto 0\}$$

We compute the semantics as:

$$
\begin{array}{ll}
 & \mathrm{prn}[\omega_0][\![\textsc{True} * (\textsc{False} * s)]\!] \\
= & \quad \langle \text{semantics of } b * s \rangle \\
 & \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}([\![\textsc{False} * s]\!] \ddagger \mathcal{U} \mid \theta))) \\
= & \quad \langle \text{semantics of } b * s \rangle \\
 & \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}(\mathtt{fix}_{\mathcal{V}}(\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta)) \ddagger \mathcal{U} \mid \theta))) \qquad (A) \\
= & \quad \langle \text{property of fixpoint (twice)} \rangle \\
 & \mathrm{prn}[\omega_0](\mathcal{U}) \qquad\qquad\qquad\qquad\qquad (B) \\
 & \quad \textbf{where } \mathcal{U} = \underline{\textsc{True}}(\mathcal{V} \ddagger \mathcal{U} \mid \theta) \\
 & \quad \textbf{where } \mathcal{V} = \underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta) \\
= & \quad \langle \text{fixpoint fold for } \mathcal{U} \rangle \\
 & \mathrm{prn}[\omega_0](\underline{\textsc{True}}(\mathcal{V} \ddagger \mathcal{U} \mid \theta)) \\
= & \quad \langle \text{defn. of prune} \rangle \\
 & \mathrm{prn}[\omega_0](\mathcal{V} \ddagger \mathcal{U}) \\
= & \quad \langle \text{fixpoint fold for } \mathcal{V} \rangle \\
 & \mathrm{prn}[\omega_0](\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta) \ddagger \mathcal{U})
\end{array}
$$

$$
\begin{array}{ll}
= & \langle \text{defn. of prune} \rangle \\
& \mathrm{prn}[\omega_0](\theta \ddagger \mathcal{U}) \\
= & \langle \text{defn. of } \ddagger \rangle \\
& \mathrm{prn}[\omega_0](\theta : \mathcal{U}) \\
= & \langle \text{defn. of prune} \rangle \\
& \mathrm{prn}[\omega_0](\mathcal{U}) \qquad\qquad (B')
\end{array}
$$

At this point we observe the circularity emerging $(B) \ldots (B')$.

Another development, from (A), is to exploit the fact that the selection guards are constant, to simplify the semantics and work from there. (Note that this assumes that such changes are sound w.r.t the prune operation.)

$$
\begin{array}{ll}
& \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}(\mathtt{fix}_{\mathcal{V}}(\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta)) \ddagger \mathcal{U} \mid \theta))) \qquad (A) \\
= & \langle [\![\textsc{True}]\!]{=}1 \rangle \\
& \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\mathtt{fix}_{\mathcal{V}}(\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta)) \ddagger \mathcal{U})) \\
= & \langle [\![\textsc{False}]\!]{=}2 \rangle \\
& \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\mathtt{fix}_{\mathcal{V}}(\theta) \ddagger \mathcal{U})) \\
= & \langle \text{fixpoint of constant function} \rangle \\
& \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\theta \ddagger \mathcal{U})) \\
= & \langle \text{defn. of } \ddagger \rangle \\
& \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\theta : \mathcal{U})) \\
= & \langle \text{fixpoint property} \rangle \\
& \mathrm{prn}[\omega_0]\mathcal{U} \\
& \quad \textbf{where } \mathcal{U} = \theta : \mathcal{U} \\
= & \langle \text{fixpoint fold} \rangle \\
& \mathrm{prn}[\omega_0](\theta : \mathcal{U}) \\
= & \langle \text{fixpoint fold} \rangle \\
& \mathrm{prn}[\omega_0](\theta : \theta : \mathcal{U}) \\
= & \langle \text{fixpoint fold} \rangle \\
& \mathrm{prn}[\omega_0](\theta : \theta : \theta : \mathcal{U})
\end{array}
$$

With a fixpoint expression $\mathtt{fix}_{\mathcal{W}}(E(\mathcal{W}))$ we obtain the following rule

$$\mathcal{W} = E(\mathcal{W}) = E(E(\mathcal{W})) = \ldots$$

If the expression $E(w)$ is constant, independent of w, i.e $E(w) = K$ we get

$$\mathcal{W} = E(\mathcal{W}) = K$$

We see with either approach that the recursion unfolds into an infinite series of sub-atomic null events — in effect we get *infinite sub-atomic stuttering*. The pruning operator cannot help, as it simply consumes the sub-atomic event θ at the head without ever advancing the clock.

However, the program just mentioned is not legal, is spotted by the compiler and is fixed, effectively by putting the body of the while-loop in parallel with an (atomic) delay. This ensures that any loop body,if executed, always takes one clock cycle. So the actual program is

$$\textsc{True} * ((\textsc{False} * s) \parallel \delta_1)$$

with the same initial world ($\omega_0 = \{\tau \mapsto 0\}$). Note that the inner-loops body is also similarly treated, but we can assume it is subsumed into s here. We then compute the semantics as before:

$$
\begin{array}{ll}
 & \mathrm{prn}[\omega_0][\![\textsc{True} * ((\textsc{False} * s) \parallel \delta_1)]\!] \\
= & \quad \langle \text{semantics of } b * s \rangle \\
 & \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}([\![(\textsc{False} * s) \parallel \delta_1]\!] \ddagger \mathcal{U} \mid \theta))) \\
= & \quad \langle \text{semantics of } \| \rangle \\
 & \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}(([\![(\textsc{False} * s)]\!] \between [\![\delta_1]\!]) \ddagger \mathcal{U} \mid \theta))) \\
= & \quad \langle \text{semantics of } \delta_1 \rangle \\
 & \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}(([\![(\textsc{False} * s)]\!] \between \theta') \ddagger \mathcal{U} \mid \theta))) \\
= & \quad \langle \text{semantics of } b * s \rangle \\
 & \mathrm{prn}[\omega_0](\mathtt{fix}_{\mathcal{U}}(\underline{\textsc{True}}((\mathtt{fix}_{\mathcal{V}}(\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta)) \between \theta') \ddagger \mathcal{U} \mid \theta))) \\
= & \quad \langle \text{property of fixpoint (twice)} \rangle \\
 & \mathrm{prn}[\omega_0](\mathcal{U}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (A) \\
 & \quad \textbf{where } \mathcal{U} = \underline{\textsc{True}}((\mathcal{V} \between \theta') \ddagger \mathcal{U} \mid \theta) \\
 & \quad \textbf{where } \mathcal{V} = \underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta) \\
= & \quad \langle \text{fixpoint fold for } \mathcal{U} \rangle \\
 & \mathrm{prn}[\omega_0](\underline{\textsc{True}}((\mathcal{V} \between \theta') \ddagger \mathcal{U} \mid \theta)) \\
= & \quad \langle \text{defn. of prune} \rangle \\
 & \mathrm{prn}[\omega_0]((\mathcal{V} \between \theta') \ddagger \mathcal{U}) \\
= & \quad \langle \text{fixpoint fold for } \mathcal{V} \rangle \\
 & \mathrm{prn}[\omega_0]((\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \mid \theta) \between \theta') \ddagger \mathcal{U}) \\
= & \quad \langle \text{defn. of } \between \rangle \\
 & \mathrm{prn}[\omega_0](\underline{\textsc{False}}([\![s]\!] \ddagger \mathcal{V} \between \theta' \mid \theta \between \theta') \ddagger \mathcal{U}) \\
= & \quad \langle \text{defn. of } \ddagger \rangle \\
 & \mathrm{prn}[\omega_0](\underline{\textsc{False}}(([\![s]\!] \ddagger \mathcal{V} \between \theta') \ddagger \mathcal{U} \mid (\theta \between \theta') \ddagger \mathcal{U})) \\
= & \quad \langle \text{defn. of prune} \rangle \\
 & \mathrm{prn}[\omega_0]((\theta \between \theta') \ddagger \mathcal{U}) \\
= & \quad \langle \text{defn. of } \between \rangle \\
 & \mathrm{prn}[\omega_0]((\theta : \theta') \ddagger \mathcal{U}) \\
= & \quad \langle \text{defn. of prune} \rangle \\
 & \mathrm{prn}[\omega_0](\theta' \ddagger \mathcal{U}) \\
= & \quad \langle \text{defn. of } \ddagger \text{ (several)} \rangle \\
 & \mathrm{prn}[\omega_0](\theta' : \mathcal{U}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (A') \\
= & \quad \langle \text{rewrite using } (A), (A') \rangle \\
 & \mathrm{prn}[\omega_0](\theta' : \theta' : \mathcal{U}) \\
= & \quad \langle \text{rewrite using } (A), (A') \rangle \\
 & \mathrm{prn}[\omega_0](\theta' : \theta' : \theta' : \mathcal{U})
\end{array}
$$

We can see the outcome is an infinite series of atomic delay events, each taking one clock cycle, as expected.

10 Acknowledgments

We would like to thank Jim Woodcock and Alistair McEwan for their many helpful discussions and comments, as well as Ian Page for his help in identifying the key parts of the language. We would also like to thank the Dean of Research of Trinity College Dublin for his support.

A Alternative Formulation of *BrTree*

As this branches at every level, even if there is only one branch in places, we call the a branch tree rather than sequence:

$$\begin{array}{rcl} \sigma \in \mathit{BrTree}\ A\ S & \widehat{=} & \textsc{Nil} \\ & | & \textsc{Split}\ A\ S\ (\mathit{BrTree}\ A\ S)^\star \end{array}$$

Longhand	Shorthand
$\textsc{Nil}$	Λ
$\textsc{Split}\ a\ s\ \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$	$a : s(\sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n)$

$$\begin{array}{rcl} \ddagger & : & \mathit{BrTree}\ A\ S \times \mathit{BrTree}\ A\ S \to \mathit{BrTree}\ A\ S \\ \Lambda \ddagger \sigma & \widehat{=} & \sigma \\ \sigma \ddagger \Lambda & \widehat{=} & \sigma \\ a : s(\sigma_1 \mid \ldots \mid \sigma_n) \ddagger \sigma & \widehat{=} & a : s(\sigma_1 \ddagger \sigma \mid \ldots \mid \sigma_n \ddagger \sigma) \end{array}$$

$$\begin{array}{rcl} \between & : & \mathit{BrTree}\ A\ S \times \mathit{BrTree}\ A\ S \to \mathit{BrTree}\ A\ S \\ \Lambda \between \sigma & \widehat{=} & \sigma \\ \sigma \between \Lambda & \widehat{=} & \sigma \\ a : s(\sigma_1 \mid \ldots \mid \sigma_m) \between b : t(\varsigma_1 \mid \ldots \mid \varsigma_n) & & \\ \quad \widehat{=} & (a \sqcup b) : s \times_n t(\ \sigma_1 \between \varsigma_1 \mid \ldots \mid \sigma_1 \between \varsigma_n & \\ & \qquad \ldots & \\ & \qquad \mid \sigma_m \between \varsigma_1 \mid \ldots \mid \sigma_m \between \varsigma_n) & \\ a : s(\sigma_1 \mid \ldots \mid \sigma_m) \between b' : t(\varsigma_1 \mid \ldots \mid \varsigma_n) & & \\ \quad \widehat{=} & a : \mathbf{1}(b' : s \times_n t(\ \sigma_1 \between \varsigma_1 \mid \ldots \mid \sigma_1 \between \varsigma_n & \\ & \qquad \ldots & \\ & \qquad \mid \sigma_m \between \varsigma_1 \mid \ldots \mid \sigma_m \between \varsigma_n)) & \\ a' : s(\sigma_1 \mid \ldots \mid \sigma_m) \between b : t(\varsigma_1 \mid \ldots \mid \varsigma_n) & & \\ \quad \widehat{=} & b : \mathbf{1}(a' : s \times_n t(\ \sigma_1 \between \varsigma_1 \mid \ldots \mid \sigma_1 \between \varsigma_n & \\ & \qquad \ldots & \\ & \qquad \mid \sigma_m \between \varsigma_1 \mid \ldots \mid \sigma_m \between \varsigma_n)) & \\ a' : s(\sigma_1 \mid \ldots \mid \sigma_m) \between b' : t(\varsigma_1 \mid \ldots \mid \varsigma_n) & & \\ \quad \widehat{=} & (a' \sqcup b') : s \times_n t(\ \sigma_1 \between \varsigma_1 \mid \ldots \mid \sigma_1 \between \varsigma_n & \\ & \qquad \ldots & \\ & \qquad \mid \sigma_m \between \varsigma_1 \mid \ldots \mid \sigma_m \between \varsigma_n) & \end{array}$$

References

[BHP94] J. P. Bowen, He Jifeng, and I. Page. Hardware compilation. In J. P. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*, chapter 10, pages 193–207. Elsevier, 1994.

[BR84] S. D. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Processes. In Brookes et al. [BRW84].

[Bro84] S. D. Brookes. On the Axiomatic Treatment of Concurrency. In Brookes et al. [BRW84].

[BRW84] S. D. Brookes, A. W. Roscoe, and G. Winskel, editors. *Seminar on Concurrency*, volume 197 of *LNCS*. Carnegie-Mellon University, Pittsburgh, PA, Springer-Verlag, July 1984.

[Emb] Embedded Solutions Ltd. (now Celoxica Ltd.). *Handel-C Language Reference Manual, v2.1*.

[Hoa90] C.A.R. Hoare. *Communicating Sequential Processes*. Intl. Series in Computer Science. Prentice Hall, 1990.

[LKL+95] Adrian Lawrence, Andrew Kay, Wayne Luk, Toshio Nomura, and Ian Page. Using reconfigurable hardware to speed up product development and performance. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, pages 111–119. Springer-Verlag, Berlin, August/September 1995. Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, FPL 1995. Lecture Notes in Computer Science 975.

[PL91] I. Page and W. Luk. Compiling Occam into field-programmable gate arrays. In W. Moore and W. Luk, editors, *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, pages 271–283. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK, 1991.

[Ros84] A. W. Roscoe. Denotational Semantics for *occam*. In Brookes et al. [BRW84].

[SP93] M. Spivey and I. Page. How to design hardware with Handel. Technical report, Oxford University Hardware Compilation Group, December 1993.