Interpretative 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 chan-

nel 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 an interpretive 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.

In §2 we introduce the abstract syntax, which we the augment with an explicit environment mapping in §3. Given these, in §4 we present the 'control-flow' or interpretive semantics.

2 Reduced Abstract Syntax

We present an abstract form of the reduced form of the Handel-C language, in the form of an abstract data type capturing the essential syntactical structure. This is based on chapter 9 of the Handel-C Language Reference Manual, v2.1 [Emb], as indicated in the section headings.

2.1 Program (§9.3.17, p197)

$$Prg \quad ::= \quad InitDecl^\star \times Decl^\star \times Stmnt$$

2.2 Declarations (§9.3.6, p189)

Identifiers (Id) in this subset play three rôles, naming variables (Id_v), channels (Id_c) and (builtin) operators (Id_o). Even though these three uses are syntactically distinguishable, allowing us to treat them as separate name spaces, we prefer to keep one name-space for variables and channels, to avoid having to carry distinct mappings around.

$$\mathit{InitDecl} \quad ::= \quad \mathit{Id} \times \mathbb{Z}$$

Local declarations have no initialisers.

$$\begin{array}{rcl} \mathit{Decl} & ::= & \textsc{Var}\ \mathit{Id} \\ & | & \textsc{Chan}\ \mathit{Id} \\ & | & \textsc{ChIn}\ \mathit{Id} \\ & | & \textsc{ChOut}\ \mathit{Id} \end{array}$$

2.3 Statements (§9.3.16, pp196–7)

In Handel-C we have the usual assignment (Asg), sequencing (Seq), conditional (If), looping (Whl) and jumping (Brk) constructs familiar to imperative language programmers. From CSP, we obtain the parallel construct (Par) and channel communication primitives for input (In) and output (Out). Specific to Handel-C is the delay statement (Dly) which does nothing, but takes one full clock-cycle to do it.

$$\begin{array}{rcl} \mathit{Stmnt} & ::= & \textsc{Seq}\ \mathit{Decl}^\star\ \mathit{Stmnt}^+ \\ & | & \textsc{Par}\ \mathit{Decl}^\star\ \mathit{Stmnt}^{2+} \\ & | & \textsc{Asg}\ \mathit{Id}\ \mathit{Exprn} \\ & | & \textsc{In}\ \mathit{Id}\ \mathit{Id} \\ & | & \textsc{Out}\ \mathit{Id}\ \mathit{Exprn} \\ & | & \textsc{If}\ \mathit{Exprn}\ \mathit{Stmnt}\ [\mathit{Stmnt}] \\ & | & \textsc{Whl}\ \mathit{Exprn}\ \mathit{Stmnt} \\ & | & \textsc{Pri}\ \mathit{PCse}^\star \\ & | & \textsc{Brk} \mid \textsc{Dly} \mid \mathit{Cont} \end{array}$$

The Cont (continue) statement is not in Handel-C, but is provided here as a technical device to facilitate translation into the explicit environment syntax form.

2.4 `prialt` Cases (§9.3.16, p197)

The prialt statement is a sequence of communication requests (Cin,Cout), each with an associated statement that is executed afterwards, if the corresponding communication succeeds. A prialt statement may have a final 'default' clause,

which always succeeds

$$\begin{array}{rcl} \mathit{PCse} & ::= & \textsc{Cin}\ \mathit{Id}\ \mathit{Id}\ \mathit{Stmnt} \\ & | & \textsc{Cout}\ \mathit{Id}\ \mathit{Exprn}\ \mathit{Stmnt} \\ & | & \textsc{Default}\ \mathit{Stmnt} \end{array}$$

The ordering of the prialt clauses is significant in that it assigns a priority to each request.

2.5 Expressions (§9.3.15, pp194–5)

Handel-C has the usual range of expressions, as well as ones tailored more closely to its hardware targeting rôle. However the details are not really of any immediate interest here. Instead we simply assume a simple epxression grammar built up from numbers (Num), variables (Var) and applications (App) of builtin operators and functions

$$\begin{array}{rcl} \mathit{Exprn} & ::= & \textsc{Num}\ \mathbb{Z} \\ & | & \textsc{Var}\ \mathit{Id} \\ & | & \textsc{App}\ \mathit{Id}\ \mathit{Exprn}^{+} \end{array}$$

2.6 Identifier (§9.3.1, p186)

Identifiers are just a countable type with equality defined, typically represented by character sequences

$$\mathit{Id} \quad ::= \quad \mathbb{A}^{+}$$

2.7 Abstract Syntax Summary

$$
\begin{array}{rcl}
\mathit{Prg} & ::= & \mathit{InitDecl}^\star \times \mathit{Decl}^\star \times \mathit{Stmnt} \\
\\
\mathit{InitDecl} & ::= & \mathit{Id} \times \mathbb{Z} \\
\\
\mathit{Decl} & ::= & \textsc{Var}\ \mathit{Id} \\
& | & \textsc{Chan}\ \mathit{Id} \\
& | & \textsc{ChIn}\ \mathit{Id} \\
& | & \textsc{ChOut}\ \mathit{Id} \\
\\
\mathit{Stmt} & ::= & \textsc{Seq}\ \mathit{Decl}^\star\ \mathit{Stmnt}^+ \\
& | & \textsc{Par}\ \mathit{Decl}^\star\ \mathit{Stmnt}^{2+} \\
& | & \textsc{Asg}\ \mathit{Id}\ \mathit{Exprn} \\
& | & \textsc{In}\ \mathit{Id}\ \mathit{Id} \\
& | & \textsc{Out}\ \mathit{Id}\ \mathit{Exprn} \\
& | & \textsc{If}\ \mathit{Exprn}\ \mathit{Stmnt}\ [\mathit{Stmnt}] \\
& | & \textsc{Whl}\ \mathit{Exprn}\ \mathit{Stmnt} \\
& | & \textsc{Pri}\ \mathit{PCse}^\star \\
& | & \textsc{Brk} \mid \textsc{Dly} \mid \mathit{Cont} \\
\\
\mathit{PCse} & ::= & \textsc{Cin}\ \mathit{Id}_c\ \mathit{Id}_v\ \mathit{Stmnt} \\
& | & \textsc{Cout}\ \mathit{Id}_c\ \mathit{Expr}\ \mathit{Stmnt} \\
& | & \textsc{Default}\ \mathit{Stmnt} \\
\\
\mathit{Expr} & ::= & \textsc{Num}\ \mathbb{Z} \\
& | & \textsc{Var}\ \mathit{Id} \\
& | & \textsc{App}\ \mathit{Id}\ \mathit{Expr}^+ \\
\\
\mathit{Id} & ::= & \mathbb{A}^+
\end{array}
$$

3 Explicit Environment Abstract Syntax

As all program variables are statically allocated (there are no function/procedure call frames in Handel-C), we convert all declarations into an environment mapping, which maps tagged identifiers to their values. The tags are tree cursors (natural number sequences) that identify the point in the abstract syntax tree where the declaration occurred.

3.1 Trees

3.1.1 Tree Type

We shall assume all trees are expressed in the general form

$$(a, \tau), T \in \mathit{Tree}\ A \quad \widehat{=} \quad A \times (\mathit{Tree}\ A)^\star$$

3.1.2 Tree Access Functions

We have functions to access the node ($\mathcal{N}$), determine the number of sub-trees (#) and to access the ith sub-tree ($\_.i$):

$$\begin{array}{rcl}
\mathcal{N} & : & \mathit{Tree}\ A \to A \\
\mathcal{N}(a, _) & \widehat{=} & a \\
\# & : & \mathit{Tree}\ A \to \mathbb{N} \\
\#(_, \tau) & \widehat{=} & \mathtt{len}\ \tau \\
_._ & : & (\mathit{Tree}\ A) \times \mathbb{N}_1 \to \mathit{Tree}\ A \\
\mathit{pre}\text{-}(T.i) & \widehat{=} & i \leq \#\ T \\
(_, \tau).i & \widehat{=} & \tau[i]
\end{array}$$

3.2 Tree Cursors

3.2.1 Tree Cursor Type

A tree cursor (TC) is a sequence of non-zero natural numbers:

$$\kappa \in TC \quad \widehat{=} \quad \mathbb{N}_1{}^\star$$

A tree cursor denotes the path to be followed from the root mode to the indicated node, by identifying number of each successive sub-tree to be followed. An empty sequence denotes the root, while a non-empty sequence gives the sub-tree indices *in reverse order*. For example, the sequence $\langle 3, 5, 2 \rangle$ denotes the 3rd sub-tree of the 5th sub-tree of the 2nd sub-tree of the root. This reversal is done to make the operations of moving to a parent or child simpler, in that they can be directly expressed as head and tail operations.

3.2.2 Manipulating Tree Cursors

A Tree cursor can be modified to refer to the parent or the ith child of the tree node it denotes, provided, of course this is defined.

$$\begin{array}{rcl}
_\uparrow & : & TC \to TC \\
\mathit{pre}\text{-}(\kappa\uparrow) & \widehat{=} & \kappa \neq \Lambda \\
\kappa\uparrow & \widehat{=} & \mathtt{tl}\ \kappa \\
\downarrow & TC \times \mathbb{N}_1 \to TC & \\
\kappa \downarrow i & \widehat{=} & i : \kappa
\end{array}$$

3.2.3 Tree Lookup

We can apply a cursor to a tree to find the relevant sub-tree node. This is only defined if the cursor matches the tree structure:

$$\begin{array}{rcl} \_(\_) & : & \mathit{Tree}\ A \to \mathit{Cursor} \to \mathit{Tree}\ A \\ \mathit{pre}\text{-}T(\Lambda) & \mathrel{\widehat{=}} & \textsc{True} \\ \mathit{pre}\text{-}(\_,\tau)(\kappa \frown \langle i \rangle) & \mathrel{\widehat{=}} & i \leq \mathtt{len}\ \tau\ \wedge\ \mathit{pre}\text{-}(\tau[i])(\kappa) \\ T(\Lambda) & \mathrel{\widehat{=}} & T \\ (\_,\tau)(\kappa \frown \langle i \rangle) & \mathrel{\widehat{=}} & \tau[i](\kappa) \end{array}$$

3.3 Tree Environments

3.3.1 Cursor-Tagged Identifiers

We tag identifiers by pairing them with a tree cursor denoting the tree node where the identifier was declared:

$$\mathit{CtId} \quad \mathrel{\widehat{=}} \quad \mathit{TC} \times \mathit{Id}$$

3.3.2 Tree Environment Type

A Tree Environment ($\mathit{TEnv}\ V$) is a (finite) mapping from cursor tagged identifiers to values drawn from type V:

$$\rho \in \mathit{TEnv}\ V \quad \mathrel{\widehat{=}} \quad \mathit{CtId} \to V$$

3.3.3 Tree Environment Lookup/Update

As we are recording all identifiers in the explicit syntax with cursor tags indicating their point of declaration, we can use simple map application and override to achieve lookup and update.

3.3.4 Tree Environment Key Lookup

We need to take an identifier tagged with its point of use, and look it up in the environment in order to re-tag it with its point of declaration:

$$\begin{array}{rcl} \mathrm{key} & : & \mathit{TEnv}\ V \to \mathit{CtId} \to \mathit{CtId} \\ \mathrm{key}[\rho](\kappa, i) & \mathrel{\widehat{=}} & (\kappa, i) \in \mathtt{dom}\ \rho\ \to\ (\kappa, i)\ ,\ \mathrm{key}[\rho](\kappa\uparrow, i) \end{array}$$

The use of this function in the sequel depends on variables being declared before use, in the ordering resulting from an prefix traversal of the syntax tree.

3.4 Variable Data and Channel States

We need to define the data values and the channel states that our semantics will need. Ordinary variables simply have a data value, whereas channel variables

have channel states, as well as pointers to the reading and writing statements and channel data values

$$\begin{array}{rcl} \mathit{ValState} & ::= & \textsc{Val}\ \mathit{Data} \\ & | & \textsc{Chan}\ \mathit{ChState}\ \mathit{TC}\ \mathit{TC} \\ & | & \textsc{InCh}\ \mathit{ChState}\ \mathit{TC} \\ & | & \textsc{OutCh}\ \mathit{ChState}\ \mathit{TC} \\ & | & \textsc{Stop} \mid \textsc{Go} \end{array}$$

We will use null (root) tree cursors here to denote a null value, rather than the root. This makes sense because the tree root (entire program) can never be a channel statement. A channel's state is either idle, or waiting for either input or output, or ready:

$$\mathit{ChState} \mathrel{\widehat{=}} \textsc{Idle} \mid \textsc{Wait} \mid \textsc{Input} \mid \textsc{Output} \mid \textsc{Ready} \mid \textsc{Error}$$

Data values (*Data*) are currently integers, extended with an unknown value (?):

$$\mathit{Data} \mathrel{\widehat{=}} \mathbb{Z} \cup \{?\}$$

We shall define our program environment (*Env*) to be a *ValState* tagged environment:

$$\mathit{Env} \mathrel{\widehat{=}} \mathit{TEnv}\ \mathit{ValState}$$

3.5 Explicit Environment Syntax

We are going to replace an abstract syntax based on *Prg* by one based on a new syntax tree based on a reworked definition of *Stmnt*.

3.5.1 Statements revisited

We discard any declarations (to be replaced by a single tree environment) and the sub-tree components, tag all identifiers, and introduce a continue and generalised delay statements:

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \textsc{Seq} \mid \textsc{Par} \\ & | & \textsc{Asg}\ \mathit{CtId}\ \mathit{Expr} \\ & | & \textsc{In}\ \mathit{CtId}\ \mathit{CtId} \\ & | & \textsc{Out}\ \mathit{CtId}\ \mathit{Expr} \\ & | & \textsc{If}\ \mathit{Expr} \\ & | & \textsc{Whl}\ \mathit{Expr} \\ & | & \textsc{Pri} \\ & | & \textsc{Brk} \mid \textsc{Cont} \\ & | & \textsc{Dly}\ \mathbb{N} \end{array}$$

3.5.2 `prialt` Cases revisited

The `prialt` cases now become sequential statements whose first sub-statement is the synchronisation statement.

3.5.3 Expression revisited

We redefine expressions to use *CtId*s intead of *Id*s:

$$\begin{array}{rcl} \mathit{Expr} & ::= & \textsc{Num}\ \mathbb{Z} \\ & | & \textsc{Var}\ \mathit{CtId} \\ & | & \textsc{App}\ \mathit{CtId}\ \mathit{Expr}^+ \end{array}$$

We can define the evaluation of an expression in a given environment as

$$\begin{array}{rcl} \_(\_) & : & \mathit{Expr} \to \mathit{Env} \to \mathit{Data} \\ \rho(\textsc{Num}\ n) & \widehat{=} & n \\ \rho(\textsc{Var}\ v) & \widehat{=} & n\ \textbf{where}\ (\textsc{Val}\ n) = \rho(v) \\ \rho(\textsc{App}\ f\ \sigma) & \widehat{=} & [\![f]\!] \circ \rho^\star\sigma \end{array}$$

We have overloaded the map application notation. Which is meant is determined by the type of the operand, if an expression, it is what has just been defined, if a tagged identifier, it is regular map lookup. Here $[\![f]\!]$ gives the fixed meaning of f which is builtin.

3.5.4 Abstract Syntax Tree revisited

We then define our new abstract syntax as a tree over *Stmt* with an invariant that restricts the length ($\ell = \#T$) of the sub-tree sequence:

$$\begin{array}{rcl} \mathit{StmtT} & \widehat{=} & \mathit{Tree}\ \mathit{Stmt} \\ \mathit{inv\text{-}StmtT}\ \textsc{Seq} & \widehat{=} & \ell > 0 \\ \mathit{inv\text{-}StmtT}\ \textsc{Par} & \widehat{=} & \ell > 1 \\ \mathit{inv\text{-}StmtT}\ (\textsc{If}\ \_) & \widehat{=} & \ell \in 1, 2 \\ \mathit{inv\text{-}StmtT}\ (\textsc{Whl}\ \_) & \widehat{=} & \ell = 1 \\ \mathit{inv\text{-}StmtT}\ \textsc{Pri} & \widehat{=} & \ell > 0 \\ \mathit{inv\text{-}StmtT}\ T & \widehat{=} & \ell = 0 \end{array}$$

Not shown here is that fact that the invariant is applied recursively to all sub-trees.

3.5.5 Program revisited

Our new program representation is a new abstract syntax tree coupled with a tree environment:

$$\mathit{Prog} \quad \widehat{=} \quad \mathit{StmtT} \times \mathit{Env}$$

Note that a program is now any statement with an associated environment. Note also that strictly speaking we need an invariant that states that all cursors in the domain of the environment (as tags) must be compatible with the statement tree. By compatible, we mean that they refer to actual tree nodes, and being fully pedantic, that they only indicate Seq or Par nodes, as these are the only ones to have declarations in the original Handel-C.

3.6 Build the Explicit Environment Tree

We need to convert a program from *Prg* to *Prog* form. We do this by creating an initial environment, and traversing the original tree with a cursor tracking our position, converting declarations to tagged environment entries, and building the new tree.

3.6.1 Converting *Prg*

We use the program's declarations to build an initial environment, tagged with the null (root) cursor. The program main statement becomes the sole sub-tree of a (top-level) SEQ node:

$$\begin{array}{rcl} \text{cnvPrg} & : & \mathit{Prg} \to \mathit{Prog} \\ \text{cnvPrg}(\iota, \delta, s_0) & \widehat{=} & \text{cnvStmt}(\langle 1 \rangle, \rho_0) s_0 \\ & & \textbf{where} \rho_0 = \text{cnvInitDecls}\ \iota \dagger \text{cnvDecls}[\Lambda]\delta \end{array}$$

3.6.2 Converting *InitDecl*

Initial Declarations simply become mappings from root-tagged identifiers to their initial values:

$$\begin{array}{rcl} \text{cnvInitDecl} & : & \mathit{InitDecl} \to \mathit{Env} \\ \text{cnvInitDecl}(v, i_0) & \widehat{=} & \{(\Lambda, v) \mapsto (\textsc{Val}\ i_0)\} \end{array}$$

$$\begin{array}{rcl} \text{cnvInitDecls} & : & \mathit{InitDecl}^\star \to \mathit{Env} \\ \text{cnvInitDecls}\ \Lambda & \widehat{=} & \theta \\ \text{cnvInitDecls}(i : \iota) & \widehat{=} & (\text{cnvInitDecl}\ i) \dagger (\text{cnvInitDecls}\ \iota) \end{array}$$

Note that later declarations supersede earlier ones. In Handel-C multiple declarations of the same identifier in the same scope are illegal, so this is not an issue. We simply take this approach using override to keep the semantics simple.

3.6.3 Converting *Decl*

$$\begin{array}{rcl} \text{cnvDecl} & : & \mathit{TC} \to \mathit{Decl} \to \mathit{Env} \\ \text{cnvDecl}[\kappa](\textsc{Var}'\ v) & \widehat{=} & \{(\kappa, v) \mapsto \textsc{Val}\ ?\} \\ \text{cnvDecl}[\kappa](\textsc{Chan}'\ c) & \widehat{=} & \{(\kappa, c) \mapsto \textsc{Chan Idle}\ \Lambda\ \Lambda\} \\ \text{cnvDecl}[\kappa](\textsc{ChIn}'\ c) & \widehat{=} & \{(\kappa, c) \mapsto \textsc{ChIn Idle}\ \Lambda\} \\ \text{cnvDecl}[\kappa](\textsc{ChOut}'\ c) & \widehat{=} & \{(\kappa, c) \mapsto \textsc{ChOut Idle}\ \Lambda\} \end{array}$$

$$\begin{array}{rcl} \text{cnvDecls} & : & \mathit{TC} \to \mathit{Decl}^\star \to \mathit{Env} \\ \text{cnvDecls}[\kappa]\Lambda & \widehat{=} & \theta \\ \text{cnvDecls}[\kappa](d : \delta) & \widehat{=} & (\text{cnvDecl}[\kappa]d) \dagger (\text{cnvDecls}[\kappa]\delta) \end{array}$$

3.6.4 Converting *Stmt*

We convert a statement on a case-by-case basis:

$$
\begin{array}{lcl}
\text{cnvStmt} & : & TC \times Env \to Stmnt \to StmtT \times Env \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Seq}\ \delta\ \sigma)} \\
& \widehat{=} & \textbf{let}\ \rho' = \rho \sqcup \text{cnvDecls}[\kappa]\delta \\
& & \quad \textbf{in let}\ (\tau,\rho'') = \text{cnvStmts}[\kappa,\rho']\sigma\ \textbf{in}\ ((\textsc{Seq},\tau),\rho'') \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Par}\ \delta\ \sigma)} \\
& \widehat{=} & \textbf{let}\ \rho' = \rho \sqcup \text{cnvDecls}[\kappa]\delta \sqcup \{(\kappa,\texttt{\_t}) \mapsto \textsc{Val}\ 0\} \\
& & \quad \textbf{in let}\ (\tau,\rho'') = \text{cnvStmts}[\kappa,\rho']\sigma\ \textbf{in}\ ((\textsc{Par},\tau),\rho'') \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Asg}\ v\ e)} \\
& \widehat{=} & ((\textsc{Asg}\ (\text{key}[\rho](\kappa,v))\ e,\Lambda),\rho) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{In}\ c\ v)} \\
& \widehat{=} & ((\textsc{In}\ (\text{key}[\rho](\kappa,c))\ (\text{key}[\rho](\kappa,v)),\Lambda),\rho) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Out}\ c\ e)} \\
& \widehat{=} & ((\textsc{Out}\ (\text{key}[\rho](\kappa,c))\ e,\Lambda),\rho) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{If}\ e\ s_1\ s_2)} \\
& \widehat{=} & \textbf{let}\ (T_1,\rho_1) = \text{cnvIxStmt}[\kappa,\rho](1,s_1) \\
& & \quad \textbf{in let}\ (T_2,\rho_2) = \text{cnvIxStmt}[\kappa,\rho](2,s_2)\ \textbf{in}\ ((\textsc{If}\ e,\langle T_1,T_2\rangle),\rho_1 \cup \rho_2) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{If}\ e\ s)} \\
& \widehat{=} & \textbf{let}\ (T',\rho') = \text{cnvIxStmt}[\kappa,\rho](1,s)\ \textbf{in}\ ((\textsc{If}\ e,\langle T'\rangle),\rho') \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Whl}\ e\ s)} \\
& \widehat{=} & \textbf{let}\ (T',\rho') = \text{cnvIxStmt}[\kappa,\rho](1,s)\ \textbf{in}\ ((\textsc{Whl}\ e,\langle T'\rangle),\rho') \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Pri}\ \varpi)} \\
& \widehat{=} & \textbf{let}\ (\tau',\rho') = \text{cnvStmts}[\kappa,\rho](\text{cnvPCase}[\kappa]^\star\varpi) \\
& & \quad \textbf{in}\ ((\textsc{Pri},\tau'),\rho' \sqcup \{(\kappa,\texttt{\_s}) \mapsto \textsc{Stop}\}) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Brk})} \\
& \widehat{=} & ((\textsc{Brk},\Lambda),\rho) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Dly})} \\
& \widehat{=} & ((\textsc{Dly}\ 1,\Lambda),\rho) \\
\multicolumn{3}{l}{\quad \text{cnvStmt}[\kappa,\rho](\textsc{Cont})} \\
& \widehat{=} & ((\textsc{Cont},\Lambda),\rho)
\end{array}
$$

We provide a form for converting the ith sub-statement directly — this is used above, as well as below for handling sequences of sub-statements:

$$
\begin{array}{rcl}
\text{cnvIxStmt} & : & TC \times Env \to \mathbb{N}_1 \times Stmnt \to StmtT \times Env \\
\text{cnvIxStmt}[\kappa,\rho](i,s) & \widehat{=} & \text{cnvStmt}[i:\kappa,\rho]s
\end{array}
$$

Converting statement sequences simply involves generating appropriate indices and then zipping these with the statements, mapping indexed conversion across

and unzipping to get the result:

$$
\begin{array}{rcl}
\text{cnvStmts} & : & TC \times Env \to Stmnt^+ \to StmtT^+ \times Env \\
\text{cnvStmts}[\kappa,\rho]\sigma & \widehat{=} & (\tau,\rho') \\
 & & \textbf{where} \\
 & & \quad \iota = \langle 1 \ldots \texttt{len}\ \sigma \rangle \\
 & & \quad (\tau,\varrho) = (\texttt{unzip} \circ (\text{cnvIxStmt}[\kappa,\rho]^\star \circ \texttt{zip})(\iota,\sigma) \\
 & & \quad \rho' = {}^{\cup}\!/\varrho
\end{array}
$$

3.6.5 Converting *PCase*

Converting `prialt` cases mainly involves stripping out the statements and converting the channel information to statements and returning those contained in a sequential compound statement

$$
\begin{array}{rcl}
\text{cnvPCase} & : & TC \to PCse \to Stmnt \\
\text{cnvPCase}[\kappa](\textsc{Cin}\ c\ v\ s) & \widehat{=} & \textsc{Seq}\ \Lambda\ \langle \textsc{In}\ (\text{key}[\rho](\kappa,c))\ (\text{key}[\rho](\kappa,v)), s \rangle \\
\text{cnvPCase}[\kappa](\textsc{Cout}\ c\ e\ s) & \widehat{=} & \textsc{Seq}\ \Lambda\ \langle \textsc{Out}\ (\text{key}[\rho](\kappa,c))\ e, s \rangle \\
\text{cnvPCase}[\kappa](\textsc{Default}\ s) & \widehat{=} & \textsc{Seq}\ \Lambda\ \langle \textsc{Cont}, s \rangle
\end{array}
$$

3.7 Explicit Environment Syntax Summary

$$
\begin{array}{rcl}
\mathit{Prog} & \widehat{=} & \mathit{StmtT} \times \mathit{Env} \\
\\
\mathit{StmtT} & \widehat{=} & \mathit{Tree}\ \mathit{Stmt} \\
\mathit{inv\text{-}StmtT}\ \textsc{Seq} & \widehat{=} & \ell > 0 \\
\mathit{inv\text{-}StmtT}\ \textsc{Par} & \widehat{=} & \ell > 1 \\
\mathit{inv\text{-}StmtT}\ (\textsc{If}\ \_) & \widehat{=} & \ell \in 0, 1 \\
\mathit{inv\text{-}StmtT}\ (\textsc{Whl}\ \_) & \widehat{=} & \ell = 1 \\
\mathit{inv\text{-}StmtT}\ \textsc{Pri} & \widehat{=} & \ell > 0 \\
\mathit{inv\text{-}StmtT}\ T & \widehat{=} & \ell = 0 \\
\\
\mathit{Stmt} & ::= & \textsc{Seq} \mid \textsc{Par} \\
& \mid & \textsc{Asg}\ \mathit{CtId}\ \mathit{Expr} \\
& \mid & \textsc{In}\ \mathit{CtId}\ \mathit{CtId} \\
& \mid & \textsc{Out}\ \mathit{CtId}\ \mathit{Expr} \\
& \mid & \textsc{If}\ \mathit{Expr} \\
& \mid & \textsc{Whl}\ \mathit{Expr} \\
& \mid & \textsc{Pri} \\
& \mid & \textsc{Brk} \mid \textsc{Cont} \\
& \mid & \textsc{Dly}\ \mathbb{N} \\
\\
\mathit{Expr} & ::= & \textsc{Num}\ \mathbb{Z} \\
& \mid & \textsc{Var}\ \mathit{CtId} \\
& \mid & \textsc{App}\ \mathit{CtId}\ \mathit{Expr}^{+} \\
\\
\mathit{Id} & ::= & \mathbb{A}^{+} \\
\\
\mathit{Env} & \widehat{=} & \mathit{CtId} \to \mathit{ValState} \\
\\
\mathit{ValState} & ::= & \textsc{Val}\ \mathit{Data} \\
& \mid & \textsc{Chan}\ \mathit{ChState}\ \mathit{TC}\ \mathit{TC} \\
& \mid & \textsc{InCh}\ \mathit{ChState}\ \mathit{TC} \\
& \mid & \textsc{OutCh}\ \mathit{ChState}\ \mathit{TC} \\
& \mid & \textsc{Stop} \mid \textsc{Go} \\
\\
\mathit{Data} & \widehat{=} & \mathbb{Z} \cup \{?\} \\
\\
\mathit{ChState} & \widehat{=} & \textsc{Idle} \mid \textsc{Wait} \mid \textsc{Input} \mid \textsc{Output} \mid \textsc{Ready} \mid \textsc{Error}
\end{array}
$$

4 Control Flow (Interpretative) Semantics

4.1 Introduction

We shall model control flow by the notion of "program counters" or "execution pointers" which are simply tree cursors. In general, because of the parallelism, we will have a set of active execution pointers. If this set becomes empty, then the program has terminated. Program execution involves three conceptual phases:

- *Execution Pointer Setup* The execution pointers are updated to point to the next set of atomic statements to be executed. This phase starts and ends with pointers pointing only at atomic statements. Pointers may indicate composite statements during this process. This phase is in fact split into two sub-phases. The `prialt` and communication statements require global knowledge to be fully handled, so the second phase, is used to finalise things once all the other active atomic statements have been identified.
- *Right-hand Side evaluation* The right-hand side expressions of all the selected assignments and channel statements are evaluated.
- *Left-hand Side Assignment* The left-hand variables and channel contents are then simultaneously updated.

These three phases comprise one clock cycle. The third phase coincides with the assignment clock edge.

4.2 Running the Program

We take a program, termination predicate and (initial) environment, determine the first execution pointers and then repeatedly cycle, until there are either no more execution pointers, or the termination predicate returns true. The program state is the current set of execution pointers and the environment:

$$\Sigma, (K, \rho) \in \mathit{PState} = \mathcal{P}\, \mathit{TC} \times \mathit{Env}$$

The termination predicate is over program state:

$$\mho \in \mathit{TermPred} \mathrel{\widehat{=}} \mathit{PState} \to \mathbb{B}$$

We return the program state at the time of termination.

$$\begin{array}{rcl} \texttt{cfBeh} & : & \mathit{TermPred} \to \mathit{Prog} \to \mathit{PState} \\ \texttt{cfBehStart}_{\mho}(T, \rho_0) & \mathrel{\widehat{=}} & \texttt{repeat cycle } (\texttt{fstep}_T(\rho)\Lambda)\ \mho' \\ & & \textbf{where} \\ & & \quad \mho'(K, \rho) \mathrel{\widehat{=}} \mho(K, \rho) \vee K = \emptyset \end{array}$$

The most useful purpose for the predicate is to allow either 'single-stepping' or stopping at external (observable) communications events. Such a stopped

program can be restarted by:

$$\begin{array}{rcl}
\texttt{cfBehResume} & : & \mathit{TermPred} \to \mathit{StmtT} \to \mathit{PState} \to \mathit{PState} \\
\mathit{pre}\text{-}\texttt{cfBehResume}_{\mho}[T](K,\rho) & \widehat{=} & K \neq \emptyset \\
\texttt{cfBehResume}_{\mho}[T](K,\rho) & \widehat{=} & \texttt{repeat cycle}\ (K,\rho)\ \mho' \\
 & & \textbf{where} \\
 & & \quad \mho'(K,\rho) \mathrel{\widehat{=}} \mho(K,\rho) \vee K = \emptyset
\end{array}$$

The precondition is not strictly necessary as `cycle` acts like an identity function if K is empty.

4.3 The Clock Cycle

We take the program, environment and set of execution pointers and return a modified environment and set of next execution pointers. We carry out the three phases in the order: evaluate rhs, assign lhs, and determine next execution pointers:

$$\begin{array}{rcl}
\texttt{cycle} & : & \mathit{StmtT} \to \mathit{PState} \to \mathit{PState} \\
\texttt{cycle}_T(K,\rho) & \widehat{=} & \textbf{let}\ \rho_r = \mathrm{evalRhs}_{T,\rho}[\theta]K\ \textbf{in} \\
 & & \textbf{let}\ \rho_c = \mathrm{evalComms}_{T,\rho}[\theta]\rho\ \textbf{in} \\
 & & \textbf{let}\ \rho_l = \rho \dagger (\rho_r \sqcup \rho_c)\ \textbf{in} \\
 & & \quad \mathrm{incExePtrs}_T(K,\rho_l)
\end{array}$$

The reason for this ordering is that we call `cycle` with an initial set of execution pointers.

4.3.1 Evaluating the RHS

For assignments we simply evaluate the rhs in the current environment and set the variable to that value in the new environment:

$$\begin{array}{rcl}
\mathrm{evalRhs} & : & \mathit{StmtT} \times \mathit{Env} \to \mathit{Env} \to \mathcal{P}\,\mathit{TC} \to \mathit{Env} \\
\mathrm{evalRhs}_{T,\rho}[\rho']\emptyset & \widehat{=} & \rho' \\
\mathrm{evalRhs}_{T,\rho}[\rho'](K \cup \{\kappa\}) & \widehat{=} & \mathrm{evalRhs}_{T,\rho}[\rho' \dagger \rho'']K \\
 & & \textbf{where} \\
 & & \quad \rho'' \mathrel{\widehat{=}} T(\kappa) = (\textsc{Asg}\ v\ e)\ \to\ \{v \mapsto \textsc{Val}\ \rho(e)\}\ ,\ \theta
\end{array}$$

For communication statements we simply evaluate the output rhs in the current environment and set the input variable to that value in the new environment. We work through the environment, rather than the execution pointers because

we need to pick up both statements for each channel:

$$
\begin{array}{l}
\text{evalComms} \\
\quad : \quad \mathit{StmtT} \times \mathit{Env} \to \mathit{Env} \to \mathit{Env} \to \mathit{Env} \\
\text{evalComms}_{T,\rho}[\rho']\theta \\
\quad \widehat{=} \quad \rho' \\
\text{evalComms}_{T,\rho}[\rho'](\mu \sqcup \{c \mapsto s\}) \\
\quad \widehat{=} \quad \text{evalComms}_{T,\rho}[\rho' \dagger \rho'']\mu \\
\qquad \textbf{where} \\
\qquad\quad \rho'' \mathrel{\widehat{=}} \text{stateOf}\rho(c) = \textsc{Ready} \;\to\; \text{evalComm}[T,\rho](\rho(c))\ ,\ \theta
\end{array}
$$

$$
\begin{array}{l}
\text{evalComm} \\
\quad : \quad \mathit{StmtT} \times \mathit{Env} \to \mathit{ValStateEnv} \\
\text{evalComm}[T,\rho](\textsc{Chan Ready}\ \kappa_i\ \kappa_o) \\
\quad \widehat{=} \quad \textbf{let}\ (\textsc{Out}\ \_\ e) = \pi_1(T(\kappa_o)) \\
\qquad \textbf{and}\ (\textsc{In}\ \_\ v) = \pi_1(T(\kappa_i))\ \textbf{in} \\
\qquad\quad \rho \dagger \{v \mapsto \textsc{Val}\ \rho(e)\}
\end{array}
$$

4.3.2 'Incrementing' Execution Pointers

We apply `nxtep` to all existing pointers to get new ones, factor out the communication pointers, use `resep` to resolve them, and merge the outcome back in:

$$
\begin{array}{rcl}
\text{incExePtrs} & : & \mathit{StmtT} \to \mathit{PState} \to \mathit{PState} \\
\text{incExePtrs}_T(K,\rho) & \widehat{=} & \textbf{let}\ (K',\rho') = \text{foldNxt}_T(\emptyset,\rho')K'\ \textbf{in} \\
 & & \textbf{let}\ (K_a,K_p) = (\lhd[P_T]K', \lhd[P_T]K')\ \textbf{in} \\
 & & \textbf{let}(K_c,\rho'') = \texttt{resep}_T(\rho')K_p\ \textbf{in} \\
 & & \quad (K_a \cup K_c, \rho'') \\
 & & \textbf{where} \\
 & & \quad P_T(\kappa) \mathrel{\widehat{=}} \pi_1(T(\kappa)) = \textsc{Pri} \\
 & & \qquad\quad \vee\ \pi_1(T(\kappa)) = (\textsc{In}\_) \\
 & & \qquad\quad \vee\ \pi_1(T(\kappa)) = (\textsc{Out}\_)
\end{array}
$$

This function simply applies `nxtep` to a set of execution pointers, passing the environment between them.

$$
\begin{array}{rcl}
\text{foldNxt} & : & \mathit{StmtT} \to \mathit{PState} \to \mathcal{P}\,\mathit{TC} \to \mathit{PState} \\
\text{foldNxt}_T(K,\rho)\emptyset & \widehat{=} & (K,\rho) \\
\text{foldNxt}_T(K_a,\rho)(K_p \sqcup \{\kappa\}) & \widehat{=} & \text{foldNxt}_T(K_a \cup K',\rho')K_p \\
 & & \textbf{where} \\
 & & \quad (K',\rho') = \texttt{nxtep}_T(\rho)\kappa
\end{array}
$$

This function is defined recursively over sets, so it is potentially non-deterministic. We need to show it actually is a function. The key to such a proof is that multiple execution pointers only result for Par statements, and their sole interaction

is to increment and decrement the thread count for the PAR statement (§4.5.4), in a manner that is independent of the order in which each thread is created or destroyed.

4.4 Determining Next Execution Pointers

The process of computing the next execution pointers involves the following functions:

- `fstep` (`first` execution `p`oint) — given an execution pointer pointing to a statement, this identifies the first atomic statements to be executed in that statement, relative to the current state of the environment. In general it may return zero, one, or more pointers, as well as modifying the environment (or that portion of it which deals with synchronisation). For PRI statements, it returns a pointer to the statement itself, and not an atomic sub-statement.
- `nxtep` (`next` execution `p`oint) — given an execution pointer pointing to a statement, this seeks to identify the next atomic statement to be executed, by looking at the parent of the current statement. As above, it will refer to the current environment, and it may return zero, one, or more pointers, and a modified environment.
- `resep` (`res`olve communication execution `p`oints) — given execution pointer for all active communication statements, it determines, by looking at the state of the channels involved, which are going to execute.

The function `fstep` is applied to the program (root statement) at the beginning to establish the first execution points. The first phase simply applies `nxtep` to every current execution point, then applies `resep` to all selected communication statements, and combines the results.

4.5 Definitions of `fstep` and `nxtep`

We shall start by giving a general overview of the function signatures and their first activity at the top-level. Then we present the behaviour for each statement type on a case by case basis.

Both functions take a program tree, environment, and execution pointer as arguments. Both return a set of execution pointers and an environment as results:

$$\begin{aligned} \texttt{fstep} &: \mathit{StmtT} \rightarrow \mathit{Env} \rightarrow \mathit{TC} \rightarrow \mathit{PState} \\ \texttt{nxtep} &: \mathit{StmtT} \rightarrow \mathit{Env} \rightarrow \mathit{TC} \rightarrow \mathit{PState} \end{aligned}$$

The program tree (T) is a fixed parameter, designating the entire program, and does not change.

The `fstep` function uses the execution pointer to lookup the current statement for case analysis. If the statement is atomic or a prialt then we are done, and we

return that execution pointer. With the prialt we also set it's status to STOP

$$
\begin{array}{l}
\texttt{fstep}_T(\rho)\kappa \\
\quad \widehat{=} \quad \textbf{case } T(\kappa) \\
\qquad\quad (\textsc{Asg}\ \_\ \_, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{Dly}, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{In}\ \_\ \_, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{Out}\ \_\ \_, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{Pri}, \_) \to (\{\kappa\}, \rho \dagger \{(\kappa, \texttt{\_s}) \mapsto \textsc{Stop}\}) \\
\qquad\quad \vdots
\end{array}
$$

The other statement types will be dealt with below on a case-by-case basis. We shall use the form

$$\texttt{fstep}_T(\rho)\kappa \quad \textbf{where} \quad T(\kappa) = \ldots$$

to denote the cases concerned.

The `nxtep` function uses part of the execution pointer to lookup the current statement's *parent* for case analysis, except for communications statements, discussed shortly. If the execution pointer is empty, then we are asking for the next statement after the program as a whole, so we respond with an empty pointer set, signalling program termination:

$$\texttt{nxtep}_T(\rho)\Lambda \mathrel{\widehat{=}} (\emptyset, \rho)$$

Otherwise, we determine if we have a communication statement that is blocked, in which case we return their execution pointers. Otherwise, we do case analysis on the parent, which must therefore be composite:

$$
\begin{array}{l}
\texttt{nxtep}_T(\rho)(i : \kappa) \\
\quad \widehat{=} \quad \textbf{case } T(i : \kappa) \\
\qquad\quad (\textsc{In}\ c\ v, \Lambda)\ \mid\ \text{stateOf}(\rho(i : \kappa, c)) = \textsc{Input} \\
\qquad\qquad \to (\{i : \kappa\}, \rho) \\
\qquad\quad (\textsc{Out}\ c\ e, \Lambda)\ \mid\ \text{stateOf}(\rho(i : \kappa, c)) = \textsc{Output} \\
\qquad\qquad \to (\{i : \kappa\}, \rho) \\
\qquad\quad (\textsc{Pri}, \sigma)\ \mid\ \rho(i : \kappa, \texttt{\_s}) = \textsc{Stop} \\
\qquad\qquad \to (\{i : \kappa\}, \rho) \\
\qquad\quad \textbf{otherwise} \to \textbf{case } T(\kappa) \\
\qquad\qquad\qquad\quad \vdots
\end{array}
$$

The function stateOf is :

$$
\begin{array}{rcl}
\text{stateOf} & : & \mathit{ValState} \to \mathit{ChState} \\
\text{stateOf}(\textsc{Chan}\ s\ \_) & \widehat{=} & s \\
\text{stateOf}(\textsc{InCh}\ s\ \_) & \widehat{=} & s \\
\text{stateOf}(\textsc{OutCh}\ s\ \_) & \widehat{=} & s
\end{array}
$$

The remaining statement types will be dealt with below on a case-by-case basis. We shall use the form

$$\texttt{nxtep}_T(\rho)(i:\kappa) \quad \textbf{where} \quad T(\kappa) = \ldots$$

to denote the cases concerned.

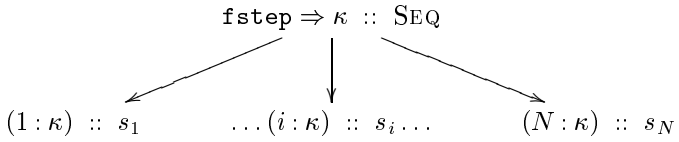
All the cases involve the composite statements. In general we assume that sub-statements are numbered 1 to N where N is the length of their sequence ($N = \texttt{len}\ \sigma$).

4.5.1 The Seq Statement

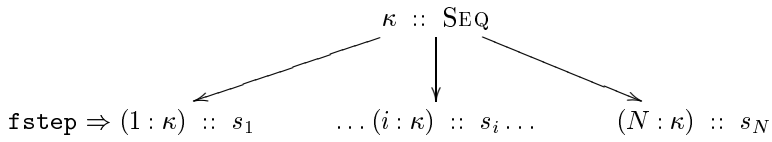
The first execution point of a sequential statement is that of its first sub-statement:

$$\begin{aligned} &\texttt{fstep}_T(\rho)\kappa \quad \textbf{where} \quad T(\kappa) = (\textsc{Seq}, \langle s_1, \ldots, s_N \rangle) \\ &\quad \widehat{=} \quad \texttt{fstep}_T(\rho)(1:\kappa) \end{aligned}$$

Pictorially, we go from



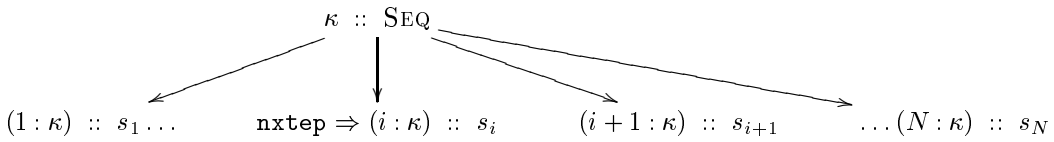
to



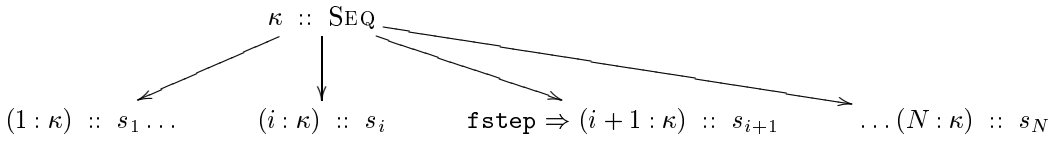
The next execution point of a child of a sequence statement is the first of the next child in sequence, if any more remain. If we are starting from the last child then we look to the sequence's own parent:

$$\begin{aligned} &\texttt{nxtep}_T(\rho)(i:\kappa) \quad \textbf{where} \quad T(\kappa) = (\textsc{Seq}, \langle s_1, \ldots, s_N \rangle) \\ &\quad \widehat{=} \quad i < N \ \rightarrow \ \texttt{fstep}_T(\rho)(i+1:\kappa) \\ &\qquad\ \ i = N \ \rightarrow \ \texttt{nxtep}_T(\rho)\kappa \end{aligned}$$

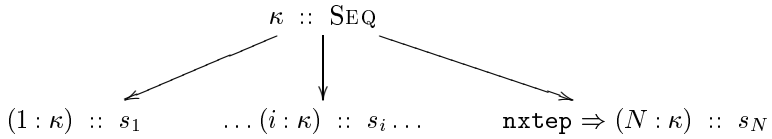
For all but the last child we go from



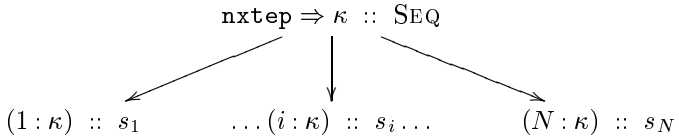
to



For the last child we go from



to

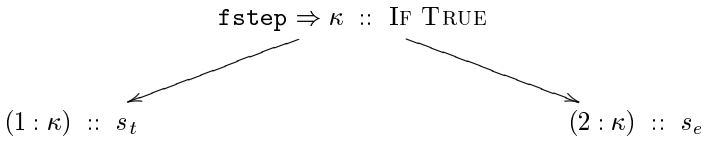


4.5.2 The If Statement

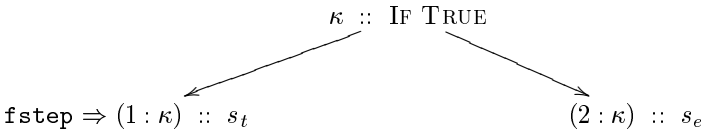
The first execution point of an if-statement depends on the current value of the condition and is the first execution point of the appropriate sub-statement:

$$\begin{array}{rl} & \texttt{fstep}_T(\rho)\kappa \quad \textbf{where} \quad T(\kappa) = (\text{IF } c, \langle s_t\ s_e\rangle) \\ \widehat{=} & \rho(c) \ \rightarrow \ \texttt{fstep}_T(\rho)(1:\kappa) \\ & \neg\rho(c) \ \rightarrow \ \texttt{fstep}_T(\rho)(2:\kappa) \end{array}$$

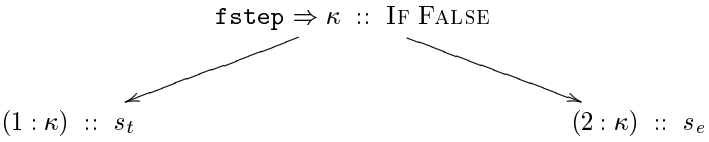
Pictorially, we go from



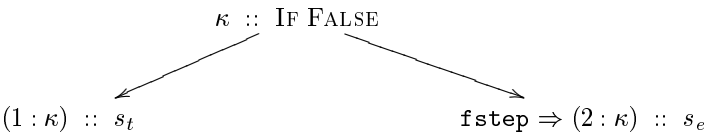
to



and from



to



The first execution point of an if-statement with no else-statement depends on the current value of the condition and is the first execution point of the then-statement if true, otherwise we look for the next statement after the if:

$$\begin{aligned} &\texttt{fstep}_T(\rho)\kappa \quad \textbf{where} \quad T(\kappa) = (\textsc{If}\ c, \langle s_t \rangle) \\ &\quad \widehat{=} \quad \rho(c) \ \rightarrow\ \texttt{fstep}_T(\rho)(1:\kappa) \\ &\qquad \neg\rho(c) \ \rightarrow\ \texttt{nxtep}_T(\rho)\kappa \end{aligned}$$

Pictorially, we go from

$$\begin{array}{c} \texttt{fstep} \Rightarrow \kappa \ ::\ \textsc{If True} \\ \downarrow \\ (1:\kappa) \ ::\ s_t \end{array}$$

to

$$\begin{array}{c} \kappa \ ::\ \textsc{If True} \\ \downarrow \\ \texttt{fstep} \Rightarrow (1:\kappa) \ ::\ s_t \end{array}$$

and from

$$\begin{array}{c} \texttt{fstep} \Rightarrow \kappa \ ::\ \textsc{If False} \\ \downarrow \\ (1:\kappa) \ ::\ s_t \end{array}$$

to

$$\begin{array}{c} \texttt{nxtep} \Rightarrow \kappa \ ::\ \textsc{If False} \\ \downarrow \\ (1:\kappa) \ ::\ s_t \end{array}$$

The next execution point of a child of an if-statement is the next point of the parent itself, (regardless of the presence/absence of the else statement):

$$\begin{aligned} &\texttt{nxtep}_T(\rho)(i:\kappa) \quad \textbf{where} \quad T(\kappa) = (\textsc{If}\ c, \sigma) \\ &\quad \widehat{=} \quad \texttt{nxtep}_T(\rho)\kappa \end{aligned}$$

We go from

$$\begin{array}{ccc} & \kappa \ ::\ \textsc{If}\ c & \\ \swarrow & & \searrow \\ (1:\kappa) \ ::\ s_t & \Leftarrow \texttt{nxtep} \Rightarrow & [(2:\kappa) \ ::\ s_e] \end{array}$$

to

$$\begin{array}{ccc} & \texttt{nxtep} \Rightarrow \kappa \ ::\ \textsc{If}\ c & \\ \swarrow & & \searrow \\ (1:\kappa) \ ::\ s_t & & [(2:\kappa) \ ::\ s_e] \end{array}$$

4.5.3 The Whl Statement

The first execution point of an while-statement depends on the current value of the condition. If true, it is the first execution point of the body. If false, it is the next execution point after the while-statement.

$$\begin{array}{l} \mathtt{fstep}_T(\rho)\kappa \quad \textbf{where} \quad T(\kappa) = (\textsc{Whl}\ c, \langle s_b \rangle) \\ \quad \widehat{=} \quad \rho(c) \ \rightarrow \ \mathtt{fstep}_T(\rho)(1:\kappa) \\ \qquad \neg\rho(c) \ \rightarrow \ \mathtt{nxtep}_T(\rho)\kappa \end{array}$$

Pictorially, we go from

$$\begin{array}{c} \mathtt{fstep} \Rightarrow \kappa \ :: \ \textsc{Whl True} \\ \Big\downarrow \\ (1:\kappa) \ :: \ s_b \end{array}$$

to

$$\begin{array}{c} \kappa \ :: \ \textsc{Whl True} \\ \Big\downarrow \\ \mathtt{fstep} \Rightarrow (1:\kappa) \ :: \ s_b \end{array}$$

and from

$$\begin{array}{c} \mathtt{fstep} \Rightarrow \kappa \ :: \ \textsc{Whl False} \\ \Big\downarrow \\ (1:\kappa) \ :: \ s_b \end{array}$$

to

$$\begin{array}{c} \mathtt{nxtep} \Rightarrow \kappa \ :: \ \textsc{Whl False} \\ \Big\downarrow \\ (1:\kappa) \ :: \ s_b \end{array}$$

The next execution point of the child of an while-statement is the first execution point of that child if the condition is true, otherwise it is the next execution point of the parent itself:

$$\begin{array}{l} \mathtt{nxtep}_T(\rho)(1:\kappa) \quad \textbf{where} \quad T(\kappa) = (\textsc{Whl}\ c, \langle s_b \rangle) \\ \quad \widehat{=} \quad \rho[c] \ \rightarrow \ \mathtt{fstep}_T(\rho)(1:\kappa) \\ \qquad \neg\rho[c] \ \rightarrow \ \mathtt{nxtep}_T(\rho)\kappa \end{array}$$

We go from

$$\begin{array}{c} \kappa \ :: \ \textsc{Whl True} \\ \Big\downarrow \\ \mathtt{nxttep} \Rightarrow (1:\kappa) \ :: \ s_b \end{array}$$

to

$$\kappa \;::\; \textsc{Whl True} \;\Big\downarrow\; \texttt{fstep} \Rightarrow (1:\kappa) \;::\; s_b$$

and from

$$\kappa \;::\; \textsc{Whl False} \;\Big\downarrow\; \texttt{nxttep} \Rightarrow (1:\kappa) \;::\; s_b$$

to

$$\texttt{nxtep} \Rightarrow \kappa \;::\; \textsc{Whl True} \;\Big\downarrow\; (1:\kappa) \;::\; s_b$$

The semantics for the WHL statement displays potentially divergent behaviour. If we have a fragment of syntax of the form (annotated with cursors):

$$\kappa :: (\textsc{Whl True}, \langle 1 : \kappa :: (\textsc{Whl False}, \sigma)\rangle)$$

and we try to compute $\texttt{ftsep}_T(\rho)\kappa$ we obtain:

$$\begin{array}{ll} & \texttt{ftsep}_T(\rho)\kappa \\ = & \quad \langle \text{Outer WHL condition is TRUE} \rangle \\ & \texttt{ftsep}_T(\rho)(1:\kappa) \\ = & \quad \langle \text{Inner WHL condition is FALSE} \rangle \\ & \texttt{nxtep}_T(\rho)(1:\kappa) \\ = & \quad \langle \text{Outer WHL condition is TRUE} \rangle \\ & \texttt{fstep}_T(\rho)(1:\kappa) \\ = & \quad \langle \text{Inner WHL condition is FALSE} \rangle \\ & \texttt{nxtep}_T(\rho)(1:\kappa) \\ & \vdots \end{array}$$

We get an infinite regress ! This is not a disaster, as we have captured an aspect of Handel-C's own behaviour. Handel-C cannot generate the combinatorial logic to compute the next statement in this case. It reports a "combinatorial cycle" at this point. Handel-C issues this as a warning, and then fixes the problem by putting a DLY statement in parallel with the inner while loop. This breaks the conbinatorial cycle, and allows the `fstep` and `nxtep` functions to terminate.

4.5.4 The Par Statement

The parallel statement is quite complex because (i) we produce multiple threads of execution, each with its own execution pointer, and (ii) we need to keep track of which threads have terminated, as the parallel construct terminates only when all the threads have done so. We use the environment, indexed by a special identifier (`_t`) tagged with the execution pointer for the parallel construct itself,

to store the number of outstanding threads. As control passes into the PAR, the relevant call to `fstep` updates the environment to reflect the fact that N threads have been started. As each thread terminates, this value is decremented by the corresponding call to `nxtep`. This is the reason that these functions need to return an environment value.

A technical difficulty could arise here. Some of the sub-statements in the parallel construct could be while-loops with false guards. These will not run and will terminate immediately, so decrementing the thread count. Indeed, if all the bodies are such while-loops, the count will be set to N, then decremented N times as the calls to `fstep` turn into calls to `nxtep`, finally resulting in `nxtep` being called on the PAR itself ! The key point is that the environment needs to be chained among the N calls to `fstep` for each thread. We cannot simply do the calls 'in parallel'.

We introduce an auxiliary function `fstpar` which applies `fstep` to a sequence of statements, and manages the environment threading. The sequence is simply denoted by the number of statements remaining to be processed:

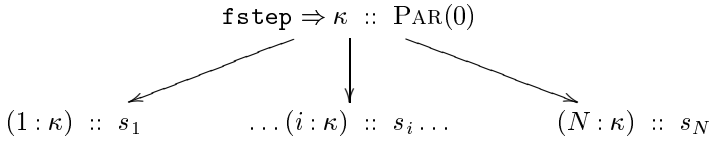
$$\begin{array}{rcl}
\texttt{fstpar} & : & \mathit{StmtT} \to \mathit{PState} \to \mathit{TC} \to \mathbb{N} \to \mathcal{P}\,\mathit{TC} \times \mathit{Env} \\
\texttt{fstpar}_T(P,\rho)\kappa\ 0 & \widehat{=} & (P,\rho) \\
\texttt{fstpar}_T(P,\rho)\kappa\ (i+1) & \widehat{=} & \textbf{let}\ (P',\rho') = \texttt{fstep}_T(\rho)(i+1:\kappa) \\
 & & \textbf{in}\ \texttt{fstpar}_T(P \cup P',\rho')\kappa\ i
\end{array}$$

A property we need to check is that the interference between the N applications of `fstep` is well-behaved.

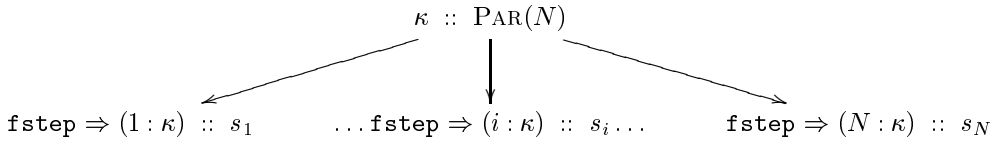
The first execution point of a parallel statement is that of all its sub-statements:

$$\begin{array}{rcl}
\texttt{fstep}_T(\rho)\kappa\quad \textbf{where}\quad T(\kappa) = (\textsc{Par}, \langle s_1, \ldots, s_N\rangle) \\
\widehat{=}\quad \texttt{fstpar}_T(\emptyset, \rho \dagger \{(\kappa, \texttt{_t}) \mapsto \textsc{Val}\ N\})\kappa\ N
\end{array}$$

Pictorially, we go from



to



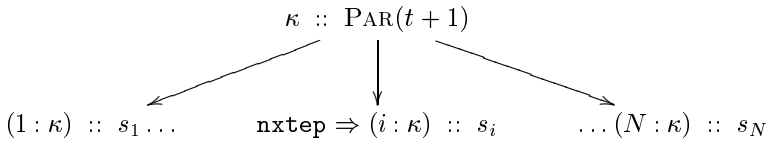
Note that `fstep` should never be applied to a PAR with outstanding threads. This denotes a serious failure of the semantics, as any given Handel-C statement only has one live instance at any point in time. In effect we have a pre-condition:

$$\begin{array}{rcl}
\texttt{pre_fstep}_T(\rho)\kappa\quad \textbf{where}\quad T(\kappa) = (\textsc{Par}, \langle s_1, \ldots, s_N\rangle) \\
\widehat{=}\quad k \in \texttt{dom}\ \rho \Rightarrow \rho(k) = \textsc{Val}\ 0 \\
\quad \textbf{where}\ k = (\kappa, \texttt{_t})
\end{array}$$

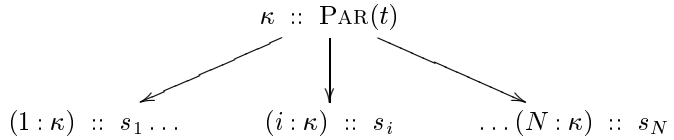
The next execution point of a child of a parallel statement depends on whether or not any other threads remain executing. The parallel construct's thread count is decremented. If there are still outstanding threads we simply discard the execution point, and return the modified environment. If that was the last thread, then we seek the next statement after the parallel construct:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\texttt{nxtep}_T(\rho)(i:\kappa) \quad \textbf{where} \quad T(\kappa) = (\textsc{Par}, \langle s_1, \ldots, s_N \rangle)} \\
\widehat{=} & \textbf{let } \rho' = \rho \ominus \{k \mapsto 1\} \textbf{ in} \\
& \quad \rho'(k) > 0 \ \rightarrow \ (\emptyset, \rho') \\
& \quad \rho'(k) = 0 \ \rightarrow \ \texttt{nxtep}_T(\rho')\kappa \\
& \quad \textbf{where } k = (\kappa, \texttt{\_t})
\end{array}
$$

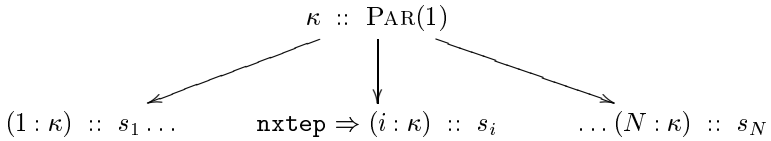
When there are other threads remaining we go from



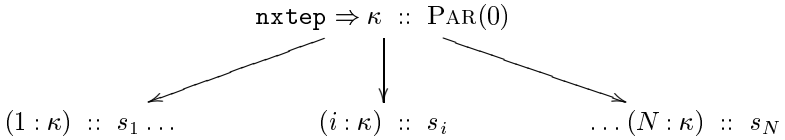
to



When there are no other threads remaining we go from



to



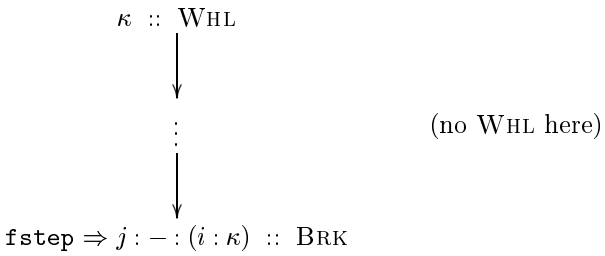
4.5.5 The Brk Statement

The Break statement simply jumps to the first statement after the most immediately enclosing while-loop, or pri-alt construct. In our reduced abstract syntax, we don't have explicit breaks in pri-alt, so we view them here as only occuring inside while-loops. While technically an atomic statement, it is treated

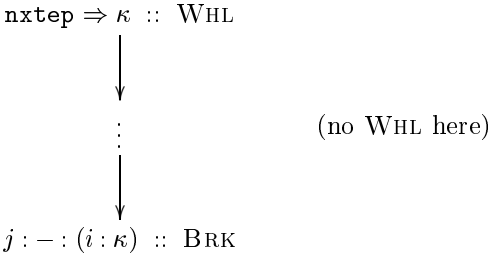
here as composite one. It is called by `fstep`, but not by `nxtep`:

$$\begin{array}{rcl} \multicolumn{3}{l}{\texttt{fstep}_T(\rho)\kappa \quad \textbf{where} \quad T(\kappa) = (\textsc{Brk}, \Lambda)} \\ & \widehat{=} & \texttt{nxtep}_T(\rho)(\texttt{brk}_T\kappa) \\ \textbf{where} & & \\ \texttt{brk} & : & TC \to TC \\ \texttt{brk}\,\Lambda & \widehat{=} & \Lambda \\ \texttt{brk}(i:\kappa) & \widehat{=} & T(i:\kappa) = (\textsc{Whl}\ldots) \to \texttt{nxtep}_T(\rho)\kappa \\ & & \qquad\qquad\qquad\qquad\quad \to \texttt{brk}\,\kappa \end{array}$$

Pictorially, we go from



to

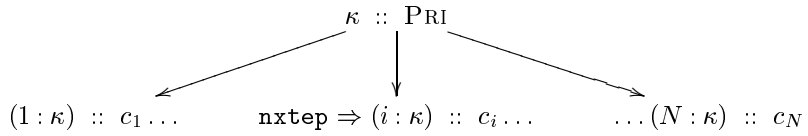


4.5.6 The Pri Statement

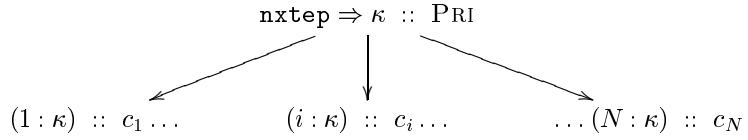
The effect of `fstep` on the prialt has already been covered. The next statement for a prialt sub-statement is simply the next statement for the prialt itself:

$$\begin{array}{rcl} \multicolumn{3}{l}{\texttt{nxtep}_T(\rho)(i:\kappa) \quad \textbf{where} \quad T(\kappa) = (\textsc{Pri}, \langle c_1, \ldots, c_N\rangle)} \\ & \widehat{=} & \texttt{nxtep}_T(\rho)\kappa \end{array}$$

Pictorially, we go from



to



4.6 Definition of `resep`

The `resep` function, handles the `prialt` and communication constructs. Unlike, the other functions, it is given a *set* of execution pointers denoting all the currently "live" `prialt`s and input and output statements.

$$
\begin{array}{rcl}
\texttt{resep} & : & \mathit{StmtT} \to \mathit{Env} \to \mathcal{P}\,\mathit{TC} \to \mathit{PState} \\
\mathit{pre}\text{-}\texttt{resep}_T(\rho)K & \mathrel{\widehat{=}} & \forall[\mathrm{isComm}_T]K \\
\mathrm{isComm} & : & \mathit{Stmt} \to \mathit{TC} \to \mathbb{B} \\
\mathrm{isComm}_T(\kappa) & \mathrel{\widehat{=}} & \textbf{case}\ T(\kappa) \\
 & & \quad (\textsc{In}\ \_, \Lambda) \to \textsc{True} \\
 & & \quad (\textsc{Out}\ \_, \Lambda) \to \textsc{True} \\
 & & \quad (\textsc{Pri}\ \_, \_) \to \textsc{True} \\
 & & \quad s \to \textsc{False}
\end{array}
$$

It determines which cases in these will become active, if any. A `prialt` without a default clause can itself be blocked, in the same way as is possible for the communications statements. If a prialt is blocked we return a pointer to the whole construct. When a prialt alternative is selected, the relevant communications statement is executed. The associated statements are executed in subsequent clock cycles.

4.6.1 Definition of `resep`, without `prialts`

We now give a definition of `resep` that does not cater for Pri. This means in effect that it changes nothing, as the only statments we expect to handle are atomic communication statements which are already handled.

$$
\begin{array}{rcl}
\texttt{resep} & : & \mathit{StmtT} \to \mathit{Env} \to \mathcal{P}\,\mathit{TC} \to \mathit{PState} \\
\texttt{resep}_T(\rho)K & \mathrel{\widehat{=}} & (K, \rho)
\end{array}
$$

4.7 Full Definitions of `fstep` and `nxtep`

The definitions of `fstep` and `nxtep` are distributed throughout the preceding text. Here we gather them together in one place.

4.7.1 Complete Definition of `fstep`

$$
\begin{array}{l}
\texttt{pre\_fstep}_T(\rho)\kappa \\
\quad \widehat{=} \quad \kappa \in \texttt{dom}\, T \\
\qquad \wedge \\
\qquad \textbf{case}\ T(\kappa) \\
\qquad\quad (\textsc{Par}, \_) \to k \in \texttt{dom}\, \rho \wedge \rho(k) = \textsc{Val}\ 0 \\
\qquad\qquad \textbf{where}\ k = (\kappa, \texttt{\_t}) \\
\qquad\quad \textbf{otherwise} \to \textsc{True}
\end{array}
$$

$$
\begin{array}{l}
\texttt{fstep}_T(\rho)\kappa \\
\quad \widehat{=} \quad \textbf{case}\ T(\kappa) \\
\qquad\quad (\textsc{Seq}, \_) \to \texttt{fstep}_T(\rho)(1:\kappa) \\
\qquad\quad (\textsc{Par}, \sigma) \to \texttt{fstpar}_T(\emptyset, \rho \dagger \{(\kappa, \texttt{\_t}) \mapsto \textsc{Val}\ n\})\ \kappa\ n \\
\qquad\qquad\qquad\quad \textbf{where}\ n = \texttt{len}\ \sigma \\
\qquad\quad (\textsc{Asg}\ \_\ \_, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{In}\ \_\ \_, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{Out}\ \_\ \_, \Lambda) \to (\{\kappa\}, \rho) \\
\qquad\quad (\textsc{If}\ c, \langle s_t, s_e \rangle) \mid \rho(c) \to \texttt{fstep}_T(\rho)(1:\kappa) \\
\qquad\quad (\textsc{If}\ c, \langle s_t, s_e \rangle) \mid \neg\rho(c) \to \texttt{fstep}_T(\rho)(2:\kappa) \\
\qquad\quad (\textsc{If}\ c, \langle s_t \rangle) \mid \rho(c) \to \texttt{fstep}_T(\rho)(1:\kappa) \\
\qquad\quad (\textsc{If}\ c, \langle s_t \rangle) \mid \neg\rho(c) \to \texttt{nxtep}_T(\rho)\kappa \\
\qquad\quad (\textsc{Whl}\ c, \_) \mid \rho(c) \to \texttt{fstep}_T(\rho)(1:\kappa) \\
\qquad\quad (\textsc{Whl}\ c, \_) \mid \neg\rho(c) \to \texttt{nxtep}_T(\rho)\kappa \\
\qquad\quad (\textsc{Pri}, \_) \to (\{\kappa\}, \rho \dagger \{(\kappa, \texttt{\_s}) \mapsto \textsc{Stop}\}) \\
\qquad\quad (\textsc{Brk}, \Lambda) \to \texttt{nxtep}_T(\rho)(\texttt{brk}_T\kappa) \\
\qquad\quad (\textsc{Dly}, \Lambda) \to (\{\kappa\}, \rho)
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{fstpar} & : & \mathit{StmtT} \to \mathit{PState} \to \mathit{TC} \to \mathbb{N} \to \mathcal{P}\,\mathit{TC} \times \mathit{Env} \\
\texttt{fstpar}_T(P, \rho)\kappa\ 0 & \widehat{=} & (P, \rho) \\
\texttt{fstpar}_T(P, \rho)\kappa\ (i+1) & \widehat{=} & \textbf{let}\ (P', \rho') = \texttt{fstep}_T(\rho)(i+1:\kappa) \\
 & & \textbf{in}\ \texttt{fstpar}_T(P \cup P', \rho')\kappa\ i
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{brk} & : & \mathit{TC} \to \mathit{TC} \\
\texttt{brk}\,\Lambda & \widehat{=} & \Lambda \\
\texttt{brk}(i:\kappa) & \widehat{=} & T(i:\kappa) = (\textsc{Whl}\ldots) \to \texttt{nxtep}_T(\rho)\kappa \\
 & & \qquad\qquad\quad \to \texttt{brk}\,\kappa
\end{array}
$$

4.7.2 Complete Definition of nxtep

$$
\begin{array}{l}
\texttt{nxtep}_T(\rho)(i:\kappa) \\
\quad \widehat{=} \quad \textbf{case } T(i:\kappa) \\
\qquad (\textsc{In}\ c\ v, \Lambda)\ \mid\ \text{stateOf}(\rho(i:\kappa, c)) = \textsc{Input} \rightarrow (\{i:\kappa\}, \rho) \\
\qquad (\textsc{Out}\ c\ e, \Lambda)\ \mid\ \text{stateOf}(\rho(i:\kappa, c)) = \textsc{Output} \rightarrow (\{i:\kappa\}, \rho) \\
\qquad (\textsc{Pri}, \sigma)\ \mid\ \rho(i:\kappa, \texttt{\_s}) = \textsc{Stop} \rightarrow (\{i:\kappa\}, \rho) \\
\quad \textbf{otherwise} \\
\qquad \textbf{case } T(\kappa) \\
\qquad\quad (\textsc{Seq}, \sigma)\ \mid\ i < \texttt{len}\,\sigma \rightarrow \texttt{fstep}_T(\rho)(i+1:\kappa) \\
\qquad\quad (\textsc{Seq}, \sigma)\ \mid\ i = \texttt{len}\,\sigma \rightarrow \texttt{nxtep}_T(\rho)\kappa \\
\qquad\quad (\textsc{If}\ \_, \sigma) \rightarrow \texttt{nxtep}_T(\rho)\kappa \\
\qquad\quad (\textsc{Whl}\ c, \_)\ \mid\ \rho(c) \rightarrow \texttt{fstep}_T(\rho)(1:\kappa) \\
\qquad\quad (\textsc{Whl}\ c, \_)\ \mid\ \neg\rho(c) \rightarrow \texttt{nxtep}_T(\rho)\kappa \\
\qquad\quad (\textsc{Par}, \langle s_1, \ldots, s_N\rangle)\ \mid\ \rho'(k) > 0 \rightarrow (\emptyset, \rho') \\
\qquad\qquad \textbf{where } k = (\kappa, \texttt{\_t}) \textbf{ and } \rho' = \rho \ominus \{k \mapsto 1\} \\
\qquad\quad (\textsc{Par}, \langle s_1, \ldots, s_N\rangle)\ \mid\ \rho'(k) = 0 \rightarrow \texttt{nxtep}_T(\rho')\kappa \\
\qquad\qquad \textbf{where } k = (\kappa, \texttt{\_t}) \textbf{ and } \rho' = \rho \ominus \{k \mapsto 1\} \\
\qquad\quad (\textsc{Pri}, \_) \rightarrow \texttt{nxtep}_T(\rho)\kappa
\end{array}
$$

4.8 Auxilliary Functions

We set channel state to waiting for output when we have a input statement using an idle channel:

$$
\begin{array}{rcl}
\text{setOutWait} & : & \mathit{TC} \rightarrow \mathit{ValState} \rightarrow \mathit{ValState} \\
\text{setOutWait}[\kappa_i](\textsc{Chan Idle}\ \Lambda\ \Lambda\ d) & \widehat{=} & (\textsc{Chan OutWait}\ \kappa_i\ \Lambda\ d) \\
\text{setOutWait}[\kappa_i](\textsc{OutCh Idle}\ \Lambda\ d) & \widehat{=} & (\textsc{OutCh OutWait}\ \kappa_i\ d)
\end{array}
$$

We set channel state to waiting for input when we have a output statement using an idle channel:

$$
\begin{array}{rcl}
\text{setInWait} & : & \mathit{TC} \rightarrow \mathit{ValState} \rightarrow \mathit{ValState} \\
\text{setInWait}[\kappa_o](\textsc{Chan Idle}\ \Lambda\ \Lambda\ d) & \widehat{=} & (\textsc{Chan InWait}\ \Lambda\ \kappa_0\ d) \\
\text{setInWait}[\kappa_o](\textsc{OutCh Idle}\ \Lambda\ d) & \widehat{=} & (\textsc{OutCh InWait}\ \kappa_o\ d)
\end{array}
$$

We set channel state to ready when we have an input statement using a channel waiting for input:

$$
\begin{array}{rcl}
\text{setInReady} & : & \mathit{TC} \rightarrow \mathit{ValState} \rightarrow \mathit{ValState} \\
\text{setInReady}[\kappa_i](\textsc{Chan InWait}\ \Lambda\ \kappa_o\ d) & \widehat{=} & (\textsc{Chan Ready}\ \kappa_i\ \kappa_o\ d)
\end{array}
$$

We also set channel state to ready when we have an output statement using a channel waiting for output:

$$
\begin{array}{rcl}
\text{setOutReady} & : & \mathit{TC} \rightarrow \mathit{ValState} \rightarrow \mathit{ValState} \\
\text{setOutReady}[\kappa_o](\textsc{Chan OutWait}\ \kappa_i\ \Lambda\ d) & \widehat{=} & (\textsc{Chan Ready}\ \kappa_i\ \kappa_o\ d)
\end{array}
$$

We also set channel state to ready when we have both input and output statements using a channel waiting for output:

$$\begin{array}{rcl} \text{setReady} & : & (TC)^2 \to \mathit{ValState} \to \mathit{ValState} \\ \text{setReady}[\kappa_i, \kappa_o](\textsc{Chan OutWait}\ \Lambda\ \Lambda\ d) & \widehat{=} & (\textsc{Chan Ready}\ \kappa_i\ \kappa_o\ d) \end{array}$$

5 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.

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.

[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.

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