An Extensible Interpreter for Value-Passing CCS[*]

Ed Harcourt
Department of Computing Science
Chalmers University of Technology
S-412 96 Gothenburg, Sweden
tel: +46 31 772 10 31
fax: +46 31 16 56 55
email: `harcourt@cs.chalmers.se`

September 8, 1995

[*]Funding from the Swedish Government agency TFR, and ESPRIT BRA CONCUR2

Abstract

We describe an interpreter for a value-passing version of CCS implemented in the lazy functional programming language Haskell. Starting from a base interpreter, we then show how to modify the interpreter for CCS extensions including, additional non-primitive combinators, new primitive operators (an interrupt operator), a time domain with a timeout operator useful in describing real-time systems, and higher order CCS where processes may be passed on channels.

Keywords: Parallel programming, process algebra, functional programming, executable semantics.

1 Introduction

Interpreters for computer languages are useful for many reasons. An interpreter may serve as an initial prototype of an implementation while more sophisticated tools are being developed. An interpreter allows a language designer to experiment with extensions or variations of a language. The ability to write and execute programs early in the language design process provides important feedback to the language designer for potential problem areas. If written in a high-level manner, an interpreter may even serve as documentation for the language's semantics.

We present an implementation of an interpreter for CCS [14], a *Calculus of Communicating Systems*, that includes value-passing. We show how to extend the interpreter with **(1)** Combinators — operators definable within CCS. **(2)** Primitive operators — operators not definable within CCS. **(3)** Time — a temporal version of CCS with a clock and a timeout operator presented in [9]. **(4)** Higher order CCS where processes can be passed on channels [19].

One of the main points of the paper is to show that by using a sufficiently expressive implementation language the interpreter can be as concise and readable as the original semantic description. As our implementation language we use the lazy functional programming language Haskell [12]. The entire interpreter is embedded in the text of this paper (until section 4.2) using Haskell's "literate comments" [1]. In this manner, the text file in which this paper is written also serves as the Haskell program file.

There are several aspects of this interpreter worth highlighting:

1. The interpreter is concise — about one page of actual Haskell code.
2. The allowable data types that can be passed on a channel are any of Haskell's data types (including functions and the data type of processes itself). However, we can easily restrict this, using Haskell's class facility, to include only those data types with a decidable equality.
3. Conditional and recursion operators (*e.g.,* **if** and **fix**) are not included in the syntax of CCS processes. The interpreter is embedded in Haskell in such a way that we borrow these from Haskell. Though subsequently we show how to implement conditionals and recursion directly in the interpreter.
4. Using a functional language allows us to make CCS specifications concise and readable. For example, the linking combinator $p\frown q$ commonly used in [14] is easily expressible, as are a variety of other useful combinators such as sequential composition. We can also represent processes parameterised on a state value.

[1] A program is written in Haskell's *literate comment* mode when comments are the default and each line of Haskell code is preceded with a ">".

5. A lazy functional language allows us to manipulate infinite objects such as infinite traces and transition trees.

For the reader, some familiarity with a process formalism such as CCS, CSP, LOTOS, or ACP is helpful though we do present all the CCS required for reading the paper. Familiarity with a functional language such as LISP, ML, or Haskell is also helpful though explanations are included of the various Haskell constructs used.

1.1 CCS Preliminaries

CCS is a concurrency language (or more specifically, a *process algebra*), where processes are constructed as terms of a simple language of process constructors. The behaviour of a process is usually given in terms of the possible transitions of the process. Transitions of the form $p \xrightarrow{\alpha} p'$ express process p's ability to perform some action α and evolve to a new Process p'. Here, p and p' are both represented *syntactically.*

The syntax of CCS processes that we implement is given by the following grammar:

$$p ::= a!v.p \;\Big|\; a?x.p \;\Big|\; p \,|\, p \;\Big|\; p + p \;\Big|\; p \backslash S \;\Big|\; p[f] \;\Big|\; \mathbf{nil} \;\Big|\; A(x) \;\Big|\; \mathbf{if}\ b\ \mathbf{then}\ p \qquad (1)$$

We omit the **else** clause on the **if**-statement as it can be encoded using summation. In the BNF, a comes from a countable set of channel names *Chan*, v from a set of values *Value*, and x from a countable set of variable names *Var*, A from a set of process variables *Pvar*, x from a possible set of states *State*, and b from a language of boolean expressions *Bexp* defined on *State* and *Value*. Let $\mathcal{P}$ be the set of processes generated by the grammar. A term $A(x)$ refers to a process constant A parameterised by x, where $A(x)$ is associated with a definition $A(x) \overset{\text{def}}{=} p$ where $A(y)$ may occur in p. To model multiple process parameters let x be a tuple. *Pure CCS* is the sub-calculus without value-passing, the conditional **if**, and process parameters.

The set of actions Act_τ is defined by expression 2.

$$(\mathit{Chan} \times \{!\} \times \mathit{Value}) \cup (\mathit{Chan} \times \{?\} \times \mathit{Var}) \cup \{\tau\} \qquad (2)$$

Some example members of Act_τ are $a?x$, $chan!5$, and τ (using syntactic sugar rather than tuple notation). The transition relation $\longrightarrow \subseteq \mathcal{P} \times Act_\tau \times \mathcal{P}$.

1.1.1 Examples

As an example, the process $a!5.\mathbf{nil}$ can speak a 5 on channel a and become the inactive process **nil** (the process that can do no actions). Two processes may communicate through a common channel. The two processes running in parallel $a!5.\mathbf{nil} \mid a?x.p$ have the following possible transition.

$$(a!5.\mathbf{nil} \mid a?x.p) \xrightarrow{\tau} (\mathbf{nil} \mid p[5/x])$$

where $p[5/x]$ means 5 is substituted for all free occurrences x in p and τ is the special silent action — the result of communication. However, the above processes are not *forced* to communicate over the channel a. There are two other possible transitions the above process can make.

$$(a!5.\mathbf{nil} \mid a?x.p) \xrightarrow{a!5} (\mathbf{nil} \mid a?x.p) \qquad\qquad (a!5.\mathbf{nil} \mid a?x.p) \xrightarrow{a?v} (a!5.\mathbf{nil} \mid p[v/x])$$

The first transition means that the left process can speak a 5 on channel a to the environment. The second transition means that the process on the right received a value v from the environment. This example shows that parallelism can introduce non-determinism into the system. We can, however, force the processes to communicate using *restriction*.

$$(a!5.\mathbf{nil} \mid a?x.p)\backslash\{a\} \xrightarrow{\tau} (\mathbf{nil} \mid p[5/x])\backslash\{a\} \tag{3}$$

Now, the above transition is the only possible transition. Other combinators include the non-deterministic choice of two process $p + q$ and the process with some channels renamed $p[f]$ where f is a *total* function, $f : \mathit{Chan} \rightarrow \mathit{Chan}$.

1.1.2 Static and Dynamic Operators

The process constructors for CCS are classified as either *static* or *dynamic* constructors. A static operator remains after a transition is taken. The parallelism, restriction, and relabelling operators are all static. A dynamic operator, however, disappears after a transition is taken. The prefixing and summation operators are dynamic. For example, in the example transition above (equation 3) notice that the parallelism and restriction operators remain after the τ transition (static) and the prefixing operators, ! and ?, have vanished (dynamic).

1.2 Early Semantics

The traditional semantics for value-passing CCS processes is usually given using *structural operational semantics* rules (SOS). An SOS rule is a syntax directed rule which describes possible transitions of a process. Figure 1 gives the operational semantics for value-passing CCS. The rules are abstract in the sense that they do not tell us how to implement them.

The rule for writing a channel, **Out**, says that the process $a!v.p$ may output the value v on channel a and become the process p. The rule for reading a channel, **In**, says that the process $a?x.p$ may receive a value v on channel a and become p where v is substituted for all free occurrences of x. Notice that v occurs free in the rule — that is, we don't actually know which value in *Value* we are supposed to receive and substitute for x. This rule non-deterministically chooses which value v is to be received and is, in this sense, *abstract*.

Said another way, if v comes from an infinite domain, the input prefixing rule describes an infinite number of possible transitions, one for each possible value v. Under this interpretation, the process $a?x.p$ is *infinitely branching*. This coincides with the usual technique of modelling value passing processes as processes in a pure, non value-passing calculus. Using this method, if v is a natural number, the process $a?v.p$ is equivalent to an infinite summation of processes

$$a?x.p \;\equiv\; a?0.p[0/x] + a?1.p[1/x] + a?2.p[2/x] + \cdots$$

usually written $\sum_{v \in \mathbb{N}} a?v.p[v/x]$.

The above parallelism rule allows two processes to "pass" a value, or communicate. This rule requires the receiver's guess match the senders transmission — again abstract.

The early semantics is un-implementable because the resulting transition relation $\longrightarrow$ is infinitely branching and, hence, not *effective* [20]. We now introduce "executable CCS" by using a variation of the operational semantics which is effective [1]. It follows, however, that the resulting calculus is less powerful but is still quite expressive.

1.3 Late Semantics

In contrast to the early semantics described in the previous section we now describe a "late" semantics due to [10] where the value being received on a channel does not need to be determined

$$\textbf{Out}\ \frac{}{\alpha.p \xrightarrow{\alpha} p}\ \alpha = \tau \text{ or } \alpha = \mathsf{ch}!v \qquad \textbf{In}\ \frac{}{a?x.p \xrightarrow{a?v} p[v/x]}$$

$$\textbf{SumL}\ \frac{p \xrightarrow{\alpha} p'}{p\ +\ q \xrightarrow{\alpha} p'\ +\ q} \qquad \textbf{SumR}\ \frac{q \xrightarrow{\alpha} q'}{p\ +\ q \xrightarrow{\alpha} p\ +\ q'}$$

$$\textbf{ParL}\ \frac{p \xrightarrow{\alpha}}{p \mid q \xrightarrow{\alpha} p' \mid q} \qquad \textbf{ParR}\ \frac{q \xrightarrow{\alpha} q'}{p \mid q \xrightarrow{\alpha} p \mid q'} \qquad \textbf{ParT}\ \frac{p \xrightarrow{a!v} p' \quad q \xrightarrow{a?v} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$$

$$\textbf{Rel}^{\dagger}\ \frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{\phi_f^{\alpha}} p'[f]} \qquad \textbf{Res}^{\ddagger}\ \frac{p \xrightarrow{\alpha} p'}{p\backslash S \xrightarrow{\alpha} p'\backslash S}\ \rho_S^{\alpha} = \textbf{true}$$

$$\textbf{Def}\ \frac{p[v/x] \xrightarrow{\alpha} p'}{P(v) \xrightarrow{\alpha} p'}\ P(x) \overset{\text{def}}{=} p \qquad \textbf{Cond}\ \frac{p \xrightarrow{\alpha} p'}{(\textbf{if } b \textbf{ then } p) \xrightarrow{\alpha} p'}\ \mathsf{eval}(b) = \textbf{true}$$

† The function f maps channels to channels, ϕ_f^{α} is defined as $\phi_f^{\tau} = \tau$, $\phi_f^{o!v} = f(o)!v$, $\phi_f^{i?v} = f(i)?v$.

‡ $\rho_S^{\tau} = \textbf{true}$, $\rho_S^{o!v} = o \in S$, $\rho_S^{i?v} = i \in S$

Figure 1: Operational semantics of CCS.

ahead of time *(early)* but is delayed until the receiver actually gets the value from a sender running in parallel *(late)*. The input prefixing rule changes so that the process $a?x.p$ evolves to a *function* which takes a value and evaluates to a process. In a sense, we identify $a?x.p$ with $a?\lambda x.p$. This kind of a transition is called *symbolic* and is important because it now yields a *finitely branching* process. The type of an input action is now $(\mathit{Chan} \times \{?\})$ and the result of an input transition is a function of type $\mathit{Value} \to \mathcal{P}$.

Figure 2 presents some of the rules for the late operational semantics of CCS. Not included are those that remain unchanged from the early semantics **SumL**, **SumR**, **Def**, and the obvious forms of **Rel** and **Res** needed for speaking a value on a channel.

2 The CCS Interpreter

Our first interpreter will consider CCS without recursive defining equations (process constants) or conditionals. However, the interpreter will still have full expressive capability as we will "borrow" Haskell's recursion and conditionals. We begin with the syntax of CCS channels, actions, and processes presented as a Haskell data type. A channel name is a string.

```
> type ChanId = String
```

$$\textbf{InPref}\quad \frac{}{a?x.p \xrightarrow{a?} \lambda x.p} \qquad\qquad \textbf{LParT}\quad \frac{p \xrightarrow{a!v} p' \quad q \xrightarrow{a?} q'}{p \mid q \xrightarrow{\tau} p' \mid (q'v)}$$

$$\textbf{LPar1}\quad \frac{q \xrightarrow{a?} q'}{p \mid q \xrightarrow{a?} \lambda x.(p \mid (q'x))} \qquad\qquad \textbf{LPar2}\quad \frac{p \xrightarrow{a?} p'}{p \mid q \xrightarrow{a?} \lambda x.((p'\ x) \mid q)}$$

$$\textbf{LRel}\quad \frac{p \xrightarrow{\alpha?} p'}{p[f] \xrightarrow{f(\alpha)?} \lambda x.(p'x)[f]} \qquad\qquad \textbf{LRes}\quad \frac{p \xrightarrow{\alpha?} p'}{p\backslash L \xrightarrow{\alpha?} \lambda x.(p'x)\backslash L}\quad (\alpha \notin L)$$

Figure 2: Late operational semantics of CCS.

Haskell note: *The Haskell keyword* `type` *introduces a* type synonym*. A declaration of the form* `type x = y` *says that* `x` *is just another name for* `y`*.*

An action is either the silent action τ or a pair consisting of a channel name and a value.

```
> data Act a = Tau | Some ChanId a
```

Haskell note: *The Haskell keyword* `data` *allows us to introduce a new inductively defined (recursive) data type. Constructors must be provided to build values in this type. For example, the type*

```
data Nat = Zero | Succ Nat
```

describes the set of natural numbers. Values in `Nat` *are* `Zero`, `Succ Zero`, `Succ (Succ Zero)`*, etc. The constructors* `Zero` *and* `Succ` *are then functions that take zero and one argument(s).* `Zero` *has type* `Nat` *and* `Succ` *has type* `Nat -> Nat` *where "->" is Haskell's function space constructor.*

The constructor `Some` is then a function of two arguments and has type `ChanId -> a -> (Act a)`. Example values in type `Act` are `Tau`, `(Some "int_chan" 5)`, and `(Some "float_chan" 3.14159)`. This type is parameterised on the type variable `a`, the type of values passed on the channels. The syntax of CCS processes is given by the following data type `(Proc a)`.

```
> data Proc a = SUM (Proc a) (Proc a)                |
>               PAR (Proc a) (Proc a)                |
>               OUT (Act a) (Proc a)                 |
>               IN  ChanId (a -> Proc a)             |
>               REL (Proc a) (ChanId -> ChanId)      |
>               RES (Proc a) [ChanId]                |
>               NIL
```

Construct name	CCS Syntax	Haskell Syntax
Inactive process	**nil**	`NIL`
Input prefixing	$a?x.p$	`IN "a" (\x -> p)`
Output prefixing	$a!v.p$	`OUT (Some "a" v) p`
Tau prefixing	$\tau.p$	`OUT Tau p`
Summation	$p+q$	`SUM p q`
Parallel Composition	$p \mid q$	`PAR p q`
Restriction	$p\backslash\{a,b\}$	`RES p ["a","b"]`
Relabelling	$p[f]$	`REL p f`

Table 1: Translating CCS to Haskell type `Proc a`.

Haskell note: *The type* `[a]` *is the type of lists with elements of type* `a`*.*

The type `(Proc a)` is an inductively defined data type and follows the structure of the grammar for CCS in equation 1. Table 1 shows the mapping from CCS terms to the appropriate value of type `(Proc a)`.

Haskell note: *The Haskell expression* `\x -> y` *corresponds to the lambda calculus term* $\lambda x.y$*.*

2.1 The Step Function

The function `step` implements the transition relation $\longrightarrow$. Since $\longrightarrow$ is a *relation*, and we are working in a functional language, the step function must return a set (list) of transitions. We first need to describe a transition of the form $p \xrightarrow{\alpha} q$, which is either an input or output transition. An input transition consists of a channel name and a function from a value to a process. An output transition consists of a complete action and the derivative process that results from executing this action.

```
> data Transition a = InT ChanId (a -> Proc a) | OutT (Act a) (Proc a)
```

An example input transition would be `InT "a" (\x -> NIL)` and an example output transition is `OutT (Some "a" 6) NIL`. An example τ transition is `OutT Tau NIL`. The step function has type:

```
> step :: Proc a -> [Transition a]
```

Haskell note: *In Haskell, the type of a function may be supplied explicitly with a* type signature*. The expression* `f::t` *says that function* `f` *is of type* `t`*.*

We will define function `step` recursively over the syntax of CCS processes using Haskell's pattern matching.

Haskell note: *Functions in Haskell can be defined by pattern matching. For example a function* `plus` *can be written using the previously defined data type* `Nat`*.*

```
> plus Zero n = n
> plus (Succ n) m = Succ (plus n m)
```

2.1.1 Nil and Prefixing

The process **nil** has no transitions and the transitions of output and input prefixing are straightforward.

```
> step NIL = []
> step (OUT act p) = [OutT act p]
> step (IN chan f) = [InT chan f]
```

2.1.2 Choice

The two rules for summation **SumL** and **SumR** specify that the transitions of $p + q$ are the transitions of p combined with the transitions of q.

```
> step (SUM p q) = (step p) ++ (step q)
```

> **Haskell note:** *The* ++ *operator concatenates two lists.*

2.1.3 Relabelling

The CCS process $p[f]$ has the same transitions as p except channel names are relabelled according to the function f. The relabelling operator is static. Consequently, in the implementation we must ensure that the relabelling function f continues to enclose the derivative processes. We discuss the implementation of the relabelling operator in detail.

```
> step (REL p f) =
>  let
>    p_steps = step p
>    outs = [OutT (Some (f chan) v) (REL p' f)  |  OutT (Some chan v) p' <- p_steps]
>    ins  = [InT (f chan) (\x -> (REL (ftop x) f))  |  InT chan ftop <- p_steps]
>    taus = [OutT Tau (REL p' f)  |  OutT Tau p' <- p_steps]
>  in
>    outs ++ ins ++ taus
```

The code uses *list comprehensions* heavily.

> **Haskell note:** *The list comprehension* `[ f x | x <- list, cond x]` *represents a list of all* `(f x)` *such that* `x` *is drawn from* `list` *with the constraint that* `(f x)` *is included in the list if* `(cond x)` *is true. For example, the list comprehension* `[(x,y) | x <- l1, y <- l2]` *builds the cross product of* `l1` *and* `l2`*. In general there can be many* list generators *of the form* `x <- list` *and many boolean valued conditions (guards).*

Combining list comprehensions with pattern matching, as in the above case, helps make the code concise and readable. The list comprehension on the second line of the let clause is

```
[OutT (Some (f chan) v) (REL p' f) | OutT (Some chan v) p' <- p_steps]
```

The right-hand-side of | says to extract all of the non-τ output transitions from the transitions of p. The left-hand-side of | rebuilds each of these output transitions with two changes; **(1)** the channel name suitably relabelled and **(2)** the relabelling operator enclosing the new process.

The list comprehension on the third line rebuilds the input transitions. Notice how the input transitions are rebuilt (the left-hand-side of |) by the expression:

```
InT (f chan) (\x -> (REL (ftop x) f))
```

Recall that the outcome of an input transition is a function from a value to a process. Here, to make relabelling static we "push" the relabelling through the lambda abstraction by applying η-conversion[2]. We can do this because relabelling changes only channel names and not values on the channels. The list comprehension on the fourth line of the let-expression handles the possible τ transitions.

2.1.4 Restriction

To determine the transitions of the process $p\backslash S$, where S is a set of channel names, we first need to determine the transitions of p and then eliminate all transitions with a channel name that appears in S.

For the side condition $\alpha \notin L$, a boolean *guard* inside a list comprehension works nicely. Again, we must be careful and make restriction a static operator. The structure of the list comprehensions are similar to the relabelling case and should be clear.

```
> step (RES p restricted) =
>    let
>       p_steps = step p
>       outs = [OutT (Some chan v) (RES p' restricted) |
>                  OutT (Some chan v) p' <- p_steps, chan `notElem` restricted]
>       ins  = [InT (chan) (\x -> (RES (ftop x) restricted)) |
>                  InT chan ftop <- p_steps, chan `notElem` restricted]
>       taus = [OutT Tau (RES p' restricted) | OutT Tau p' <- p_steps]
>    in
>       outs ++ ins ++ taus
```

2.1.5 Parallel Composition

Parallel composition is the most difficult operator to implement. To determine the transitions of $(p \mid q)$ we first need to determine the transitions of p and q separately. The transitions of $(p \mid q)$ are all of the transitions of p combined with all of the transitions of q combined with all of the possible communications of p and q.

To calculate all of the possible communications between p and q we use a list comprehension to generate the cross product of the two transition lists. Let (a, b) be a pair of actions in `(step p)` $\times$ `(step q)`. If one of a or b wishes to read a channel that the other wishes to write then these two actions can synchronise, and the late parallel rule, **Par1**, applies.

```
> step (PAR p q) =
>  let
>    ptrans = step p; qtrans = step q
```

[2]A rule in the Lambda Calculus is η-conversion which states that, if E is a function, then $\lambda x.(Ex) = E$ where x is not free in E.

Calculate the transitions of $(p \mid q)$ that result by stepping p and leaving q alone.

```
>    pOuts  = [OutT act (PAR p' q) | OutT act p' <- ptrans]
>    pIns   = [InT chan (\x -> (PAR (f x) q)) | InT chan f <- ptrans]
>    pSteps = pOuts ++ pIns
```

The symmetric case of stepping q and not p.

```
>    qOuts  = [OutT act (PAR p q') | OutT act q' <- qtrans]
>    qIns   = [InT chan (\x -> (PAR p (f x))) | InT chan f <- qtrans]
>    qSteps = qOuts ++ qIns
```

We need to determine all possible communications between p and q and combine them to form all of the possible transitions of $(p \mid q)$.

```
>    pXq   = [(p,q) | p <- ptrans, q <- qtrans]
>    taus1 = [OutT Tau (PAR (f v) q') |
>                 (InT ch f, OutT (Some ch' v) q') <- pXq, ch == ch']
>    taus2 = [OutT Tau (PAR q' (f v)) |
>                 (OutT (Some ch' v) q', InT ch f) <- pXq, ch == ch']
>  in
>    pSteps ++ qSteps ++ taus1 ++ taus2
```

This completes the `step` function which, essentially, completes the interpreter.

2.1.6 Example uses of `step`

As a simple example, consider the CCS process $a!2.\mathbf{nil} + b!3.\mathbf{nil}$. This process has two transitions

$$a!2.\mathbf{nil} + b!3.\mathbf{nil} \xrightarrow{a!2} \mathbf{nil} \qquad \text{and} \qquad a!2.\mathbf{nil} + b!3.\mathbf{nil} \xrightarrow{b!3} \mathbf{nil}$$

and in our implementation the function call:

```
step (SUM (OUT (Some "a" 2) NIL) (OUT (Some "b" 3) NIL))
```

returns the following list with two elements.

```
[OutT (Some "a" 2) NIL, OutT (Some "b" 3) NIL]
```

2.2 Running a Process

A function `runProc` repeatedly applies the function `step` and uses a random number generator to pick one of the several output transitions of a process. An infinite sequence of output actions $\langle \alpha_1, \alpha_2, \ldots, \alpha_n, \ldots \rangle$ is a trace (or run) of a process if from some initial state s

$$s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} s_n \xrightarrow{\alpha_{n+1}} \cdots$$

We consider a trace to contain only output actions because, if a process requires input, then this must be supplied explicitly by another process running in parallel.

```
> runProc :: Proc a -> Int -> [(ChanId, a)]

> runProc proc seed = run' proc (randlist seed 1 10000)
>    where run' proc (seed:seeds) =
>            let outs = [OutT x y | OutT x y <- step proc] in
>              if null outs then []
>              else case outs!!(rand seed 0 ((length outs) - 1)) of
>                     OutT Tau p'           -> run' p' seeds
>                     OutT (Some chan v) p' -> (chan,v):(run' p' seeds)
```

<u>Haskell note:</u> *The type* `(a,b)` *represents the type of pairs of* `a`*'s and* `b`*'s. The expression* `l!!n` *picks the* n^{th} *element in the list* `l` *(counting from zero). The constructor "*`:`*" is the cons operator that inserts an element to the front of a list. The predicate* `null` *tests for the empty list. The* `where`*-clause introduces a local scope much like a* `let`*-clause.*

Function `runProc` does not try to guess input — it only follows paths in the transition tree that do not require input. One could imagine a more sophisticated *default* input process that grabbed keyboard input on demand or, better yet, and more in the spirit of *reactive programming*, a system where the user could choose actions interactively.

2.3 Transition Trees

Building a structure that contains the entire transition tree of a process is straightforward. A transition tree consists of a process at the root, a list of actions that constitute the labelled transitions, and a list of child transition trees. A type `TransitionTree` is defined by

```
> data TransitionTree a = Node (Proc a) [(Act a, TransitionTree a)]
```

and a function `buildTree` takes a process to a transition tree.

```
> buildTree :: Proc a -> TransitionTree a
> buildTree proc =
>   let outs = [((Some ch v), buildTree p') | OutT (Some ch v) p' <- step proc]
>   in  Node proc outs
```

There are no input transitions on this tree. We could, of course, include the initial input transitions but not their derivatives until missing input values are resolved. Since Haskell is lazy, the transition tree is only built on demand and one can examine parts of it without requiring the other parts (possibly infinite) be constructed.

2.4 Conditionals, Recursion, and State

As we mentioned, we do not have separate conditional and recursion operators contained in the syntax of our CCS process data type `(Proc a)` and that we "borrow" them from Haskell. To see how this works, here is a one-place buffer cell that loses (filters) even integers. In CCS this is rendered as

$$Cell \overset{\text{def}}{=} \texttt{in}?x.\textbf{if } even\ x \textbf{ then } Cell \textbf{ else } \overline{\texttt{out}}!x.Cell \tag{4}$$

and in our Haskell syntax as

```
> cell = IN "in" (\x -> if even x then cell else OUT (Some "out" x) cell)
```

2.4.1 Parametric Processes

Processes are often parameterised on a value that represents the "current state" of the process. Consider a register that holds an integer where the current value in the register may be read with `read` and a new value can be written with `write` (from [14, page 172]).

$$Reg(y) = \texttt{write}?x.Reg(x) + \texttt{read}!y.Reg(y) \tag{5}$$

In pure CCS (*i.e.,* without value passing) this is described as an abbreviation for an infinite *family* of mutually recursive definitions Reg_0, Reg_1, Reg_2, *etc.*

We can implement a parameterised process as a function from the state to a process, $\mathsf{State} \rightarrow (\mathsf{Proc\ a})$. The register example is rendered in our interpreter with the following function, where the state and the process type are both integers.

```
> reg :: Int -> Proc Int
> reg y = (IN "write" (\x -> reg x)) `SUM` (OUT (Some "read" y) (reg y))
```

Haskell note: *Any binary function* `op` *can be infixed on-the-fly in Haskell by enclosing the function or constructor name in back quotes. The term* `op a b` *can be written* `a `op` b`.

2.4.2 Guarded Recursion

A process p is guarded if it occurs within a prefix $a.p$. For theoretical reasons (to guarantee unique solutions of recursive equations), all recursive process identifiers should be guarded. This prohibits us from making definitions such as $P \overset{\text{def}}{=} P$ and $P \overset{\text{def}}{=} P \mid P$. Unguarded recursion yields infinitely branching processes. The definition of `cell` above is guarded. The interpreter does not check for guardedness — and unguarded recursive definitions will type check. For example, the definition $p(k) \overset{\text{def}}{=} a!k + p(k+2)$ has an infinite number of transitions. The actions of $p(0)$ are $\{a!0, a!2, \ldots\}$. The following process `p` is infinitely branching due to the unguarded recursive call to `p`.

```
> p x = (OUT (Some "a" x) NIL) `SUM` (p (x+2))
```

We can never compute all of the transitions of an infinitely branching process. Function `step` would not terminate. Consider the process `(p 0) `SUM` (p 1)` which has an `a`-transition for each natural number. The implementation of `SUM` would attempt to append two infinite lists.

3 Combinators

A combinator is a non-primitive process constructor that can be defined by equations in terms of primitive CCS operators. We describe three example combinators: linking, sequential composition, and bounded iteration.

3.1 Linking

Consider a combinator that links two processes by connecting the output of one to the input of the other. Assume a process inputs values on a port `in` and outputs values on a port `out`. We can link two of these processes $p \frown q$ through a new port α by renaming and restricting.

$$p \frown q \overset{\text{def}}{=} (p[\alpha/\texttt{out}] \mid q[\alpha/\texttt{in}])\backslash\{\alpha\}$$

It is important that α be a new name in the sense that $\alpha \notin \texttt{sort}(p) \cup \texttt{sort}(q)$. This linking combinator is defined by the function `link`.

```
> link :: Proc a -> Proc a -> Proc a
> p `link` q =
>     let newchan = "_alpha"
>         f "out" = newchan; f x = x
>         g "in" =  newchan; g x = x
>     in
>         RES ((REL p f) `PAR` (REL q g)) [newchan]
```

Here we assume that the port `_alpha` is a fresh name. For the relabelling functions `f` and `g` notice the use of the identity functions $f(x) = x$ and $g(x) = x$. Relabelling functions should be total and not alter channel names other than those specified. We should think of all relabelling functions as the identity function perturbed on one or more values in the domain. The function `f` is the identity function perturbed such that $f(\texttt{out}) = \alpha$.

3.2 Well-Termination and Sequential Composition

Another useful combinator is sequential composition, $p; q$. Here q can begin only when p has finished. In order to determine when p has finished we introduce a special action `done` that a process uses to indicate its termination. A process that indicates its termination through the action `done` is said to be *well-terminating*. The process P *Before* Q represents the sequential composition of P and Q.

$$\begin{aligned} \mathit{Done} &\overset{\text{def}}{=} \texttt{done}!\mathbf{nil} \\ P\ \mathit{Before}\ Q &\overset{\text{def}}{=} (P[b/\texttt{done}] \mid b?Q)\backslash\{b\} \end{aligned}$$

Here b is a fresh name (not in the sort of either P or Q).

To implement this we now notice that there is more than one data type that can be transmitted on a channel. In this example the actions `done` and b are pure synchronisations which we can consider to transmit the single value in the unit data type. P and Q, however, may be transmitting some other data type so we introduce a disjoint union of possible channel types — pure synchronisations and some other type.

```
> data CCSvalue a = Value a | Sync
```

We can now give the definitions of *Done* and *Before*.

```
> doneP :: Proc (CCSvalue a)
> doneP = OUT (Some "done" Sync) NIL

> p `before` q =
>     let f "done" = "_b"; f x = x in
>        RES ((REL p f) `PAR` (IN "_b" (\Sync -> q))) ["_b"]
```

Notice in the λ-abstraction `\Sync -> q` the use of the unit value `Sync` and not a variable as the only value allowed on channel `_b` is `Sync`. If another value was passed then pattern matching would fail and we would get a runtime error. We could use a variable and do our own case analysis so we could crash more gracefully. This runtime error represents a problem with the CCS program and not the interpreter.

3.3 Bounded Iteration

We can build recursive combinators. Consider the bounded iteration combinator, *Iterate*(n, X), from chapter 9 of [14]. This combinator executes n copies of process X sequentially. The definition given in [14] is

$$\mathit{Iterate}(n, X) \overset{\text{def}}{=} \textbf{if } n = 0 \textbf{ then } \mathit{Done} \textbf{ else } X \ \mathit{Before}\ \mathit{Iterate}(n-1, X)$$

which we render as (this time using pattern matching rather than the conditional)

```
> iterateP 0 _ = doneP
> iterateP n p = p `before` (iterateP (n-1) p)
```

3.4 The Scheduler Specification

To stress that embedding CCS within Haskell gives us the freedom to express a system concisely, we now give a brief specification of the scheduler example from section 5.4 of "Communication and Concurrency" [14]. We will not go into detail how the scheduler works, just show how the specification maps nicely to our implementation.

$$(i \in X) \quad \mathit{SchedSpec}(i, X) \overset{\text{def}}{=} \sum_{j \in X} b_j.\mathit{SchedSpec}(i, X - \{j\})$$

$$(i \notin X) \quad \mathit{SchedSpec}(i, X) \overset{\text{def}}{=} a_i.\mathit{SchedSpec}(i+1, X \cup \{i\}) + \sum_{j \in X} b_j.\mathit{SchedSpec}(i, X - \{j\})$$

The above specification uses mathematical notation freely: set union, set difference, addition over natural numbers, and the set operators $\in$ and $\notin$. Using Haskell's predefined list operators `\\` (list difference), `union`, `elem`, and guarded functions we have:

```
> schedspec :: Int -> [Int] -> Proc Int
> schedspec i x | i `elem` x  =
>  foldr SUM NIL [OUT (Some "b" j) (schedspec i (x \\ [j])) | j <- x]
>
> schedspec i x | i `notElem` x  =
>  let
>    p = OUT (Some "a" (i+1)) (schedspec (i+1) (x `union` [i]))
>  in
>    p `SUM` (foldr SUM NIL [OUT (Some "b" j) (schedspec i (x \\ [j])) | j <- x])
```

Notice the use of list comprehensions and `foldr` to implement finite uses of $\sum$.

4 Experimenting with CCS Extensions

In this and the following section we implement three extensions to CCS.

1. Adding an interrupt operator to CCS — useful in real-time systems.
2. Adding time to CCS along with a new *timeout* operator.
3. A *higher order* version of CCS where channels can carry processes.

There have been many papers exploring various extensions to CCS, including interrupt and check-point operators, priority choice and temporal operators such as timeout. An interpreter can be useful for experimenting with new primitive operators [3]. When adding a primitive operator we must modify the `step` function rather than give defining equations. We consider the interrupt operator from [14]. One must be careful — if an operator can be defined by equations rather than altering the transition relation, it should be done so as altering the transition relation can change the mathematical theory.

4.1 An Interrupt Operator

Consider an interruptible process, $p\wedge q$ where q may, at any point in p's execution, interrupt it. This new operator is defined by the following transition rules.

$$\textbf{Int1}\ \frac{p \xrightarrow{\alpha} p'}{p\wedge q \xrightarrow{\alpha} p'\wedge q} \qquad\qquad \textbf{Int2}\ \frac{q \xrightarrow{\alpha} q'}{p\wedge q \xrightarrow{\alpha} q'}$$

Extending the data type `(Proc a)` to include the new constructor `INT (Proc a) (Proc a)`, we modify function `step` as:

```
step (INT p q) =
  let
    pOuts = [OutT act (INT p' q) | OutT act p' <- (step p)]
    pIns  = [InT chan (\x -> (INT (f x) q)) | InT chan f <- (step p)]
  in
    pOuts ++ pIns ++ (step q)
```

[3]An operator is primitive if it is not definable in terms of other CCS constructs.

All three lines of the let clause allow p^q to make transitions emanating from p while keeping q alive. The body of the let clause also includes the transitions of q discarding p.

The interrupt operator is both static and dynamic. It sticks around waiting to interrupt a process but goes away as soon as it does. Notice that the rules are similar to the rules for $+$ except that ^ is static in the left argument and dynamic in the right argument.

4.2 Timed CCS

In this section we outline how to extend CCS with time. The timed version of CCS we implement here is described in detail in [9].

Extend the set of actions Act_{τ} with another unique action σ resulting in a new set of actions $Act_{\tau\sigma}$. That is, $Act_{\tau\sigma} = Act \cup \{\tau, \sigma\}$. Intuitively, σ is thought of as a "clock tick". We extend the syntax of CCS operators with one new timing construct — a time-out operator $\lfloor p \rfloor q$. This process acts like p before the clock tick σ arrives. However, if p can't do anything before σ arrives then, when σ does arrive, it acts like q. The following points summarise the view of time in this calculus (restated from [9]).

1. All actions except σ are instantaneous.
2. A process will wait indefinitely until it can engage in a synchronisation.
3. Once a process can engage in a synchronisation it will do so without delay.

One additional transition rule is required to extend $\longrightarrow$ for Act_{τ} (*i.e.,* without σ). This rule captures the intuition that $\lfloor p \rfloor q$ can behave like p.

$$\frac{p \xrightarrow{\alpha} p'}{\lfloor p \rfloor q \xrightarrow{\alpha} p'}$$

The other rules extend $\longrightarrow$ for $Act_{\tau\sigma}$ and are introduced as we proceed.

Extend the type `(Proc a)` of Haskell-CCS processes with `TIMEOUT (Proc a) (Proc a)`. Also extend the type of actions `Act a` with a value `Tick`.

4.2.1 Nil and Prefixing

The process **nil**, in the un-timed calculus has no transitions. In the timed calculus, however, **nil** can wait as can any prefixed process.

$$\frac{}{\mathbf{nil} \xrightarrow{\sigma} \mathbf{nil}} \qquad \frac{}{a?x.p \xrightarrow{\sigma} a?x.p} \qquad \frac{}{a!v.p \xrightarrow{\sigma} a!v.p}$$

We replace the three lines of the earlier `step` function which deal with **nil** and prefixing with the following.

```
> step NIL          = [OutT Tick NIL]
> step (IN chan f)  = [OutT Tick (IN chan f), InT chan f]
> step (OUT Tick p) = [OutT Tick (OUT Tick p)]
> step (OUT act  p) = [OutT Tick (OUT act p), OutT act p]
```

The first line corresponds exactly with the waiting rule for **nil**. The remaining three lines corresponds with the other rules but also includes the transition from the earlier `step` function. Care is taken to make sure that the process $\sigma.p$ does not have two σ transitions, which we would get if the third line was omitted.

4.2.2 Summation

The process $p+q$ can wait only if both p and q can wait.

$$\frac{p \xrightarrow{\sigma} p' \quad q \xrightarrow{\sigma} q'}{p+q \xrightarrow{\sigma} p'+q'}$$

To implement the step function for $+$ we use the following result from [9].

Lemma 1 (Time-determinism) *If $p \xrightarrow{\sigma} q$ and $p \xrightarrow{\sigma} r$ then q is syntactically identical to r.*

The time-determinism lemma states that there is at most one (possibly zero) σ transition emanating from any process.

Rather than give the entire step function for $+$ we outline it.

1. Compute the instantaneous (*i.e.,* non-σ) transitions of p (q) in the set *pNonSigs* (*qNonSigs*).
2. Compute the σ transition of p (q) in the set *pSig* (*qSig*), if it exists.
3. If *pSig* and *qSig* are **both** non-empty then compute the σ transition for $p+q$ in the set *pqSig.*
4. The possible transitions of $p+q$ are *pNonSigs* $\cup$ *qNonSigs* $\cup$ *pqSig.*

Because we have time-determinism, the sets *pSig, qSig, pqSig* will either be empty or have one σ transition in it.

4.2.3 Relabelling and Restriction

The processes $p\backslash S$ and $p[f]$ can wait if p can wait.

$$\frac{p \xrightarrow{\sigma} p'}{p\backslash S \xrightarrow{\sigma} p'\backslash S} \qquad \frac{p \xrightarrow{\sigma} p'}{p[f] \xrightarrow{\sigma} p'[f]}$$

Extending the step function here is easy and we give it for restriction. The relabelling case holds similarly. Extend the step function above under case `(RES p restricted)` with the following lines:

```
>  let
>     -- Previous code for restriction operator goes here.
>     tick_trans = [OutT Tick (RES p' restricted) | OutT Tick p' <- p_steps]
>  in
>     outs ++ ins ++ taus ++ tick_trans
```

The first line of the let clause computes the possible σ transition of p. The subsequent lines set the list `tick_trans` with the appropriate σ transition depending on whether p had a σ transition.

4.2.4 Time-Out

The process $\lfloor p \rfloor q$, in addition to the transition rule given at the beginning of this section, has the following σ transition.

$$\frac{p \overset{\tau}{\not\longrightarrow}}{\lfloor p \rfloor q \xrightarrow{\sigma} q}$$

The negative premise ensures that q can execute (thereby discarding p) only if p cannot perform a τ.

```
> step (TIMEOUT p q) =
>   let
>     p_steps    = step p
>     outs       = [OutT (Some chan v) p' | OutT (Some chan v) p' <- p_steps]
>     taus       = [OutT Tau p'           | OutT Tau p'           <- p_steps]
>     ins        = [InT chan ftop         | InT chan ftop         <- p_steps]
>     tick_trans = if null taus then [OutT Tick q] else []
>   in
>     outs ++ ins ++ taus ++ tick_trans
```

4.2.5 Parallel Composition

The process $p \mid q$ can wait if p can wait **and** q can wait **and** $(p \mid q)$ cannot synchronise.

$$\frac{p \xrightarrow{\sigma} p' \quad q \xrightarrow{\sigma} q' \quad p \mid q \overset{\tau}{\not\longrightarrow}}{p \mid q \xrightarrow{\sigma} p' \mid q'}$$

Extend the step function under case (PAR p q) with the following.

```
>  let
>     -- Previous code for parallelism operator goes here.
>     p_ticks = [OutT Tick p' | OutT Tick p' <- ptrans]
>     q_ticks = [OutT Tick q' | OutT Tick q' <- qtrans]
>     tick_trans =
>        if not (null taus) then []
>        else case (p_ticks, q_ticks) of
>               ([],_)                        -> []
>               (_,[])                        -> []
>               ([OutT Tick p'],[OutT Tick q']) -> [OutT Tick (PAR p' q')]
>    in
>       pSteps ++ qSteps ++ taus ++ tick_trans
```

4.2.6 Delay

The process $\lfloor \mathbf{nil} \rfloor(p)$ has only one possible transition, a σ transition to p. This is, in effect, a one cycle delay. A combinator for an n-cycle delay is defined by

```
> delay n p = if n == 0 then p else TIMEOUT NIL (delay (n-1) p)
```

5 Higher Order CCS

In the current interpreter, there is no restriction on the type that processes may be parameterised by, in particular the type of channel values may itself be instantiated to a process. That is, the type `(Proc a)` can be instantiated to `(Proc (Proc a))`. A process of this type can pass other processes on channels. The resulting calculus is called *CHOCS* [19] *(Calculus of Higher Order Communicating Systems)*. This capability is powerful but we would probably also like pass values other than processes so we extend the normal type of actions to include processes.

$$(\mathit{Chan} \times \{!\} \times (\mathit{Value} \cup \mathcal{P})) \cup (\mathit{Chan} \times \{?\} \times \mathit{Var}) \cup \{\tau\}$$

Each $x \in \mathit{Var}$ can now range over both Value and $\mathcal{P}$.

In Haskell we create a new type `ChanVal` which is a disjoint union of the channel values from the original calculus and the set of processes. Also, update `(Act a)` to use `ChanVal`.

```
> data ChanVal a = P (Proc a) | V a
> data Act a = Tau | Some ChanId (ChanVal a)
```

We only need to change one line in the type of processes `(Proc a)` so that the domain of the process valued function on `IN` is now `ChanVal`. The new constructor for input prefixing is now

```
>   IN  ChanId (ChanVal a -> Proc a)
```

Now, not only is `Proc` recursively defined but is mutually recursively defined with `ChanVal` and `Act`.

There is one remaining change. The input transitions in the definition of `Transition` must use `ChanVal`.

```
> data Transition a = InT ChanId (ChanVal a -> Proc a) | OutT (Act a) (Proc a)
```

These are the only changes needed to define the higher order calculus. The `step` function remains unchanged.

As an example of passing a process on a channel, consider a process that can receive another process on a channel `prog` and evolve to that process. In CHOCS we would write $\mathsf{prog}?p.p$ which we could test with the process

$$(\mathsf{prog}!(\mathsf{out}!5.\mathbf{nil}).\mathbf{nil} \mid \mathsf{prog}?p.p)\backslash\{\mathsf{prog}\}$$

The process on the left of $\mid$ sends the process $\mathsf{out}!5.\mathbf{nil}$ on channel prog and the process on the right receives it. This process has only one transition sequence.

$$(\mathsf{prog}!(\mathsf{out}!5.\mathbf{nil}).\mathbf{nil} \mid \mathsf{prog}?p.p)\backslash\{\mathsf{prog}\} \xrightarrow{\tau} (\mathbf{nil} \mid \mathsf{out}!5.\mathbf{nil})\backslash\{\mathsf{prog}\} \xrightarrow{\mathsf{out}!5} (\mathbf{nil} \mid \mathbf{nil})\backslash\{\mathsf{prog}\}$$

In our interpreter we can write

```
> p = OUT (Some "out" (V 5)) NIL
> q = OUT (Some "prog" (P p)) NIL
> sys = RES (q `PAR` (IN "prog" (\(P p) -> p))) ["prog"]
```

and when we use `step` to examine the possible transitions we have

```
  step sys = [OutMove Tau (RES (PAR NIL (OUT (Some "out" (V 5)) NIL)) ["prog"])]
```

and running the process

```
runProc sys = [("out",5)]
```

6 On Borrowing Conditionals and Recursion from Haskell

In this interpreter we have tried to borrow as much as we can from Haskell allowing us to concentrate on the important aspects of the language — the basic operators of CCS. This has freed us from worrying about and implementing channel data types, a value language for channel values, variables, and recursion. In this section we show that this has only been for convenience and was not strictly necessary.

6.1 Adding a Conditional Construct

We can add a conditional construct to our interpreter by extending the data type of processes `Proc a` to include the new constructor `IF`.

$$\textbf{if } \textit{bool-expr} \textbf{ then } p \quad \equiv \quad \texttt{IF Bool (Proc a)}$$

An if-statement with an else-clause is encoded using summation.

$$\textbf{if } b \textbf{ then } p \textbf{ else q} \quad \equiv \quad \texttt{(IF b p) `SUM` (IF (not b) q)}$$

The transition rule for **if** (figure 1) is written in Haskell as

```
> step (IF True  p) = step p
> step (IF False _) = NIL
```

This is abstract in that we still borrow Haskell's boolean expressions and Haskell's ability to evaluate them. We could, however, implement our own syntax and semantics of boolean expressions, but this only takes us further from the essence of CCS and only makes the interpreter and programs messy without adding any insight.

6.2 Recursion

In this section we show how to represent recursive CCS processes directly without appealing to Haskell's recursion. We do this in two parts; first we add a fixed-point operator to unparameterised CCS processes. Generalising from this, we then show how to add a fixed-point operator for parametric processes.

6.2.1 Non-Parametric Processes

Add the following fixed-point operator to the syntax of CCS processes

$$\mathbf{fix}(X = P)$$

where X comes from a set of process variables and P is a process that may contain references to X. Importantly, P may contain another **fix** construct which allows the definition of mutually recursive processes. The following transition rule **Rec** describes the possible transitions of **fix**.

$$\textbf{Rec} \quad \frac{E\{\mathbf{fix}(X = E)/X\} \xrightarrow{\alpha} E'}{\mathbf{fix}(X = E) \xrightarrow{\alpha} E'}$$

The expression $E\{P/X\}$ represents substituting the process P for the variable X in E. Intuitively, the premise of the rule "unwinds" the recursion once. Using **fix**, the recursive process *Cell* (equation 4) is written as the following.

$$Cell \overset{\text{def}}{=} \mathbf{fix}(P = \mathtt{in}?x.\mathbf{if}\ even\ x\ \mathbf{then}\ P\ \mathbf{else}\ \overline{\mathtt{out}}!x.P) \tag{6}$$

For simplicity, we would like to avoid having to write our own process substitution function used in the rule **Rec**. Fortunately, we can still use Haskell's substitution to do this by using *higher-order abstract syntax* [4]. The essence of higher-order abstract syntax is that substitution is implemented using the host language's (Haskell's) substitution (*i.e.,* using λ-abstraction and function application). We have already seen an example of this with the rules for input prefixing.

In **Rec**, E is essentially a function from a process to a new process; $E : \mathcal{P} \to \mathcal{P}$ and **fix** has the usual type of a fixed-point operator; $(\mathcal{P} \to \mathcal{P}) \to \mathcal{P}$. Using function application instead of substitution, the rule **Rec** above becomes

$$\textbf{Rec'} \quad \frac{E(\mathbf{fix}\ E) \xrightarrow{\alpha} E'}{\mathbf{fix}\ E \xrightarrow{\alpha} E'}$$

Add the constructor `FIX (Proc a -> Proc a)` to `Proc` and the following line to `step`.

```
> step (FIX e)  = step (e (FIX e))
```

In our interpreter, the definition of *Cell* in equation 6, using `FIX`, becomes

```
> cell = FIX(\p -> IN "in" (\x -> if even x then p else OUT (Some "out" x) p))
```

6.2.2 Parametric Recursive Processes

Not only do we want to write recursive processes but we would like to write *parameterised* recursive processes (*e.g.,* as in the definition of the process *Reg* from section 2.4.1.) As we previously mentioned, a parametric process can be viewed as a function from a value to a process $\mathcal{S} \to \mathcal{P}$. We can represent processes that have multiple state variables (*e.g.,* the scheduler specification from section 3.4) by taking $\mathcal{S}$ to be a tuple type $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$.

We can't directly use the fixed-point operator **fix** as before as we need to ensure that the new state value gets propagated through the recursive calls. We need a new fixed-point operator **fix1** of the following type.

$$\mathbf{fix1} : ((\mathcal{S} \to \mathcal{P}) \to (\mathcal{S} \to \mathcal{P})) \to (\mathcal{S} \to \mathcal{P})$$

A similar transition rule for **fix1** is

$$\textbf{Rec1} \quad \frac{(E(\mathbf{fix1}\ E))_v \xrightarrow{\alpha} E'}{(\mathbf{fix1}\ E)_v \xrightarrow{\alpha} E'}$$

where v is the initial state of the process. The previous fixed-point construct **fix** is a special case of **fix1** where, in **fix1**, we let $\mathcal{S}$ be the unit data type and v be the singleton value in the unit data type (both are represented as "`()`" in Haskell).

To add **fix1** to our Haskell interpreter we need to parameterise the data type of processes on two type variables — for the types of states and channel values. The new type of processes is `(Proc` `a b)`, where `a` is the type of channel values and `b` is the type of state values. Now, we need to add a constructor for **fix1** to `Proc`.

```
> FIX1 ((b -> Proc a b) -> (b -> Proc a b)) b
```

We update the step function with the definition of **Rec1**.

```
> step (FIX1 e v) = step (e (FIX1 e) v)
```

As an example, consider a CCS process which models a counter

$$Counter(n) \overset{\text{def}}{=} \mathtt{out}!n.Counter(n+1)$$

which we write in our interpreter, using **fix1**, as

```
> counter :: Int -> Proc Int Int
> counter n = FIX1 (\fp state -> OUT (Some "out" state) (fp (state + 1))) n
```

The register example (equation 5) translates to

```
> register n = FIX1 (\fp state -> SUM (IN "put" (\x -> fp x))
>                                    (OUT (Some "get" state) (fp state))) n
```

The fixpoint operators do not prohibit us from writing unguarded recursive processes. We can still write expressions such as $\mathbf{fix}(X = X)$ and $\mathbf{fix}(X = X + X)$ which translate to, in our interpreter, `FIX(\x -> x)` and `FIX(\x -> x 'SUM' x)` respectively.

7 Strange Terms of Type `Proc`

In our interpreter we have been writing CCS expressions (in $\mathcal{P}$ the terms in the concrete syntax for CCS described by the BNF grammar 1) as a term in the Haskell type Proc. An interesting property is whether there is a bijective mapping between $\mathcal{P}$ and `Proc`. By an easy inductive argument it is easy to show that every term in $\mathcal{P}$ has a unique representation in `Proc`. This follows immediately from a the translation given in table 1. However, in section 5 we have already shown that the converse is not true. For example, higher-order terms are not in $\mathcal{P}$ but are expressible in `Proc` — and there are others.

Non-terminating functions. Consider a Haskell function `f` that does not terminate. The expression `IN "in" (\x -> OUT (Some "out" f(x)) NIL)` of type `(Proc a)` has no representation in $\mathcal{P}$ because in CCS actions are atomic and terminate immediately. The quick solution is to require that the programmer only use functions defined over the value and state domains that terminate (of course they can still write non-terminating processes).

Channel Passing. Channels in CCS may not be passed on other channels as in the Pi-Calculus. Consider a hypothetical CCS term such as $\mathsf{in}?x!x.$*"hello"*.**nil**, which suggests that a value is being read on channel in and is subsequently used as a channel name in which to say *"hello"*. However, in our interpreter channels are just strings and the above term does have the representation in type `(Proc String)` as `IN "in" (\x -> OUT (Some x "hello") NIL)`. This problem would still arise even if we include a separate data type of channels `Channel` there is nothing stopping the user from writing a process of type `(Proc Channel)`. We can remedy this problem by requiring that the user only use string *constants* as channel names.

Process Variables. In the version of the interpreter that uses an explicit recursion operator **fix**, process variables are integers along with a constructor `Var`. A similar problem arises as in the case for channels. The term `IN "in" (x -> Var x)` of type `(Proc Int)` has no equivalent in CCS. This situation is remedied by requiring the user use only integer *constants* as process variables.

8 Related Work

There are several implementations of combined concurrent/functional languages. The first implementation of a concurrency formalism within a functional language was PFL [11]. PFL employs many of the same ideas of using higher-order abstract syntax to model value passing and recursion and is similar to CCS except channels are first class; that is, they may be created dynamically and passed on other channels. This obviates the need for implementing the restriction and relabelling operators. To simulate parallelism, PFL interfaces with the host operating system to create processes and uses a scheduling policy to determine which process to execute. This differs from the current interpreter which is contained entirely within Haskell and directly implements the operational semantics of value-passing CCS. Consequently, in PFL, one cannot directly obtain the possible transitions of a process by applying a function such as `step`.

CML (Concurrent ML), another concurrent functional language [18], is quite different than CCS as CML includes process creation, synchronisation, and message send/receive primitives *directly* within ML by extending the language. In [17] Prasad describes a simulator for CBS, the *Calculus of Broadcasting Systems*. However, CCS is quite different from CBS as CBS does not have channels and uses a broadcasting communication model rather than a handshake one.

We can view the current paper as an exercise in implementing SOS specifications in a functional language. Watt [21] gave an example of how denotational semantic specifications can be prototyped in the functional language ML [15]. *Higher-order abstract syntax* is presented in [4]. The original work on structured operational semantics is [16] while [13, 8] are also introductions. A presentation of the implementability of structured operational semantic descriptions for process algebras is given in [20]. The Concurrency Workbench [2] is a tool for proving CCS processes equivalent, but for pure CCS (*i.e.,* non-value passing and stateless processes).

9 Summary

We have described an implementation of an interpreter for a value-passing version of the process calculus CCS. Values may come from an infinite domain and processes may be parameterised on a state value. Though the traditional semantics for CCS is given operationally (*i.e.,* that is, with respect to an abstract machine), the operational semantics is not implementable. We subsequently described an implementable operational semantics (not due to us) and its implementation in the lazy functional programming language Haskell. The implementation is interesting because it is as concise as the original operational semantics specification with the added benefit of yielding an executable interpreter. We then showed how the interpreter can be extended for various versions of CCS. The extensions include — CCS extended with an interrupt operator, a timed version of CCS, and higher-order CCS where processes may be passed on channels.

The reason the interpreter is concise is that we have borrowed as much from the host language as possible. In particular the language of data expressions is borrowed directly from Haskell allowing us to use lists, tuples, functions, user defined types, etc. without having to explicitly implement them. Also, constructs which bind identifiers in CCS (input prefixing and recursion) are implemented using Haskell's binding facility (λ-abstraction and β-reduction) freeing us from implementing our own substitution and variable renaming routines.

Acknowledgements. I would like to thank the concurrency group at Chalmers University (Pawel Pączkowski, Martin Weichart, Uno Holmer, and K.V.S. Prasad) for commenting on earlier versions of this draft and for help cleaning up and finding a few bugs in the code.

References

[1] Bard Bloom and Frits Vaandrager. SOS rule formats for state-bearing processes. Technical report, CWI, 1994. CONCUR2 1992 deliverables.

[2] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[3] Rance Cleaveland and Daniel Yankelevich. An operational framework for value-passing processes. In *POPL'94, Proceedings of the* 21^{st} *annual symposium on principles of programming languages*, 1994.

[4] C. Elliot and F. Pfenning. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 International Conference on Programming Language Desighn and Implementation*, Atlanta Georgia, June 1988. ACM, ACM Press.

[5] Ed Harcourt, Jon Mauney, and Todd Cook. Formal specification and simulation of instruction-level parallelism. In *Proceedings of the 1994 European Design Automation Conference*, pages 296–301. IEEE Computer Society Press, 1994.

[6] Ed Harcourt, Jon Mauney, and Todd Cook. From processor timing specifications to static instruction scheduling. In *Lecture Notes In Computer Science: Proceedings of the International Symposium on Static Analysis*, pages 116–130. Springer-Verlag, 1994.

[7] Ed Harcourt, Jon Mauney, and Todd Cook. Functional specification and simulation of instruction set architectures. In *Proceedings of the International Conference on Simulation and Hardware Description Languages*. SCS Press, 1994.

[8] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* John Wiley & Sons, 1990.

[9] Matthew Hennessy. On timed process algebras: a tutorial. Technical Report 2/93, School of Cognitive and Computing Sciences, University of Sussex, Brighton BN1 9QH, January 1993.

[10] Matthew Hennessy and Huimin Lin. Symbolic bisimulations. Computer Science Internal Report 1/92, University of Sussex, April 1992.

[11] Sören Holmström. PFL: A functional language for parallel programming. Technical Report 7, Dept. of Computer Sciences, Chalmers Univ. of Tech., 1983.

[12] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *Sigplan Notices*, May 1992.

[13] Gilles Kahn. Natural semantics. In *STACS 87: Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[14] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[15] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[16] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAMI FN-19, University of Aarhus, Denmark, 1981.

[17] K. V. S. Prasad. Programming with broadcasts. In *CONCUR*, August 1993. Springer Verlag LNCS 715.

[18] J. H. Reppy. A higher order concurrent language. *SIGPLAN Notices*, 26(6):294–305, 1991. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation.

[19] Bent Thomsen. A theory of higher order communicating systems. *Information and Computation*, pages 38–57, January 1995.

[20] Frits Vaandrager. Expressiveness results for process algebras. Technical Report CS-9301, CWI, University of Amsterdam, Programming Research Group, Amsterdam, The Netherlands, 1993. Appeared in Proceedings of the REX Workshop: "Semantics: Foundations and Applications". LNCS, Springer-Verlag.

[21] David A. Watt. Executable semantic descriptions. *Software—Practice and Experience*, pages 13–43, July 1986.