A semantics for ML concurrency primitives

Dave Berry, Robin Milner and David N. Turner
db@dcs.ed.ac.uk, rm@dcs.ed.ac.uk, dnt@dcs.ed.ac.uk
Laboratory for Foundations of Computer Science
University of Edinburgh. *

ACM Principles of Programming Languages, January 1992.

Abstract

We present a set of concurrency primitives for Standard ML. We define these by giving the transitional semantics of a simple language. We prove that our semantics preserves the expected behaviour of sequential programs. We also show that we can define stores as processes, such that the representation has the same behaviour as a direct definition. These proofs are the first steps towards integrating our semantics with the full definition of Standard ML.

1 Background and Motivation

There have been several attempts to add concurrency primitives to Standard ML (SML) and related languages [Hol83, Mat91, Rep91a, CM90, Ber89]. However, when we began this work none of these implementations had a published formal definition. The formal definition of SML is an integral part of the development of the language. If we are to add concurrency to the language, it is essential that we have a formal semantics for the new constructs that is compatible with the existing definition. In this paper we present our work towards such a semantics.

The first choice we faced was that of which primitives to use. The story of our choice shows an interesting convergence between theory and practice. We began by giving a semantics for the primitives used by Matthews [Mat91], because Matthews is working with us on a distributed implementation of SML. The resulting rules were similar to those that we present here, but were complicated by the addition of a mutual exclusion relation between processes. This was because Matthews' primitives allow arguments of the choice operator to be arbitrary expressions that could themselves create new sub-processes.

We then restricted the choice operator to take communications as arguments. We added the type `'a com` to enforce this restriction, following PFL [Hol83] and CML [Rep91a]. This gave us a cleaner semantics, but required the addition of an operator to coerce a value of type `'a com` to one of type `'a`. Given the need for this operator, we decided to adopt the extra functionality that Reppy gives it in CML. As a result, we now have a semantics for the basic primitives of CML. Thus for semantic reasons we have arrived at the same result as Reppy, who used purely pragmatic reasoning.

Our semantics has been influenced by Facile [GMP89] and the Chemical Abstract Machine [BB90]. However, we believe that our approach is simpler. For example, Facile can create a new process with either a behaviour expression or a `fork` function, and this results in some transitions being labelled with behaviour expressions. We avoid the need for such complex labels by eliminating behaviour expressions.

We give the semantics of a small language. This allows us to concentrate on the key features of the language, and to prove some simple properties without having to consider an unreasonable

*This work is supported by several grants from the SERC and (for Turner) support from Harlequin Ltd.

number of cases. We aim to incorporate the semantics of this language with the semantics of SML. We present two steps along this path. The first shows that our semantics preserves the expected behaviour of sequential programs, which suggests that our primitives can be added to SML without disturbing its functional features. The second shows that we can represent stores as processes with the desired behaviour. This goes a long way towards recovering the non-functional part of SML.

Reppy has independently given a semantics for CML [Rep91b]. He uses the style developed by Wright and Felleisen [WF91], but the result is very similar to our definition. We give some proofs of properties that we would like to hold for our language, which Reppy doesn't. On the other hand, Reppy gives a semantics for all the constructs in the current definition of CML, whereas we only deal with the basic operators. We don't foresee any problems in adding the other operators to our definition.

2 The Concurrency Primitives And Their Semantics

```
signature Concurrency =
sig
  type 'a channel
  val channel: unit -> '_a channel
  type 'a com
  val send:    'a channel * 'a -> 'a com
  val receive: 'a channel -> 'a com
  val choose:  'a com * 'a com -> 'a com
  val wrap:    'a com * ('a -> 'b) -> 'b com
  val noevent: 'a com
  val fork:    (unit -> 'a) -> unit
  val sync:    'a com -> 'a
end
```

Figure 1: The signature of the concurrency primitives

Figure 1 shows how our primitives might appear if they were available in full SML. The type `'a com` is the type of suspended communications. Suspended communications are actually performed by applying `sync` to them. This allows the type system to specify the possible arguments to the choice operator. It also allows a programming style based on the abstraction of synchronisation from the events being synchronised, as proposed by Reppy (who calls this type `'a event`). Furthermore, it enables us to split our semantic rules into two groups, one for evaluation and one for communication.

The *syntax* of our simple language is:

$$l = x \mid () \mid l_1\ l_2 \mid \mathtt{fn}\ x \Rightarrow l \mid \mathtt{rec}\ f(x) \Rightarrow l \mid (l_1, l_2)$$

where l, l_1 and l_2 are lexical phrases and x and f are alphabetic identifiers. The concurrent behaviour of our language is defined by the constructors and basic values shown in Figures 1 and 2. They have no special syntax beyond their existence as identifiers and values. We use parentheses to show grouping.

The *semantic objects* are summarised in Figures 2 and 3. The set of expressions is a superset of both the set of values and the set of lexical phrases. The set of identifiers includes all possible alphabetic identifiers, including the constructors and basic values.

Each singleton ProcessSet is called a *process*, and is written $[p : e]$. The notation $P[p : e]$ denotes the ProcessSet $P \cup \{[p : e]\}$. The union of two ProcessSets is only defined if their domains are disjoint. We usually omit the parentheses around configurations.

$$
\begin{array}{rcl}
k & \in & \text{ChannelId} \\
p, q & \in & \text{ProcessId} \\
x, y, z, f & \in & \text{Identifier} \\
() & \in & \text{Unit} = \{()\} \\
c & \in & \text{Constructor} = \{\texttt{send}, \texttt{receive}, \texttt{choose}, \texttt{wrap}, \texttt{noevent}\} \\
b & \in & \text{Basic Value} = \{\texttt{channel}, \texttt{fork}, \texttt{sync}\}
\end{array}
$$

Figure 2: Basic Semantic Objects

$$
\begin{array}{rcl}
K & \in & \text{ChannelIdSet} = \text{Fin(ChannelId)} \\
v & \in & \text{Value} = \text{Unit} \cup \text{ChannelId} \cup \text{Constructor} \cup \text{Constructed Value} \cup \\
 & & \text{Basic Value} \cup \text{Closure} \cup \text{ValPair} \\
\langle c, v \rangle & \in & \text{Constructed Value} = \text{Constructor} \times \text{Value} \\
(v_1, v_2) & \in & \text{ValPair} = \text{Value} \times \text{Value} \\
\texttt{fn}\ x \Rightarrow e & \in & \text{Closure} = \text{Identifier} \times \text{Expression} \\
e & \in & \text{Expression} = \text{Value} \cup \text{Application} \cup \text{Identifier} \cup \text{ExpPair} \cup \text{RecExp} \\
e_1\ e_2 & \in & \text{Application} = \text{Expression} \times \text{Expression} \\
(e_1, e_2) & \in & \text{ExpPair} = \text{Expression} \times \text{Expression} \\
\texttt{rec}\ f(x) \Rightarrow e & \in & \text{RecExp} = \text{Identifier} \times \text{Identifier} \times \text{Expression} \\
P & \in & \text{ProcessSet} = \text{ProcessId} \xmapsto{\text{fin}} \text{Expression} \\
S & \in & \text{ProcessIdSet} = \text{Fin(ProcessId)} \\
(K, P) & \in & \text{Configuration} = \text{ChannelIdSet} \times \text{ProcessSet}
\end{array}
$$

Figure 3: Compound Semantic Objects

The notation $f+g$ and $f-g$ denotes the usual operations on functions. The notation K_1-K_2 is also used to denote set difference. The notation $e\{v/x\}$ denotes the substitution of v for x in e, with the usual renaming to avoid the capture of free variables.

We divide the *semantic rules* into two groups. The rules in the first group defines the evaluation of expressions. They are given in the transitional style, in which each sentence defines one step of evaluation. The rules in the second group define communication between two processes. They are given in the relational style, in which a sentence gives the result of the communication.

The sentences in the *evaluation rules* are transitions between configurations. Each configuration specifies the channels and processes that have been created by the computation so far. A transition specifies some computation involving one or two processes, called the *selected processes*. Each transition is labelled with a set of the ProcessIds of its selected processes.

It is important to realise that these labels are not related to the communication labels of CCS or Facile. They are mainly used to state and prove properties about the language. Indeed, we often omit labels from sentences when the information is irrelevant.

The sentences in the *communication rules* are relations between a pair of suspended communications and the expressions that they return when they communicate with each other. We use a double arrow to mark the fact that the communications happen at one go, rather than as a series of transitions. There aren't any rules for `noevent`. `noevent` can never synchronise with another communication.

$$\frac{K, P[p : e_1] \xrightarrow{S} K', P'[p : e_1']}{K, P[p : e_1\ e_2] \xrightarrow{S} K', P'[p : e_1'\ e_2]} \tag{1}$$

$$\frac{K, P[p : e] \xrightarrow{S} K', P'[p : e']}{K, P[p : v\ e] \xrightarrow{S} K', P'[p : v\ e']} \tag{2}$$

$$K, P[p : (\texttt{fn}\, x \Rightarrow e)\ v] \xrightarrow{\{p\}} K, P[p : e\{v/x\}] \tag{3}$$

$$K, P[p : c\ v] \xrightarrow{\{p\}} K, P[p : \langle c, v\rangle] \tag{4}$$

$$K, P[p : \texttt{rec}\, f(x) \Longrightarrow e] \xrightarrow{\{p\}} K, P[p : \texttt{fn}\, x \Rightarrow e\{\texttt{rec}\, f(x) \Rightarrow e/f\}] \tag{5}$$

$$\frac{K, P[p : e_1] \xrightarrow{S} K', P'[p : e_1']}{K, P[p : (e_1, e_2)] \xrightarrow{S} K', P'[p : (e_1', e_2)]} \tag{6}$$

$$\frac{K, P[p : e] \xrightarrow{S} K', P'[p : e']}{K, P[p : (v, e)] \xrightarrow{S} K', P'[p : (v, e')]} \tag{7}$$

$$\frac{k \notin K}{K, P[p : \texttt{channel()}] \xrightarrow{\{p\}} K \cup \{k\}, P[p : k]} \tag{8}$$

$$\frac{q \notin dom(P) \cup \{p\}}{K, P[p : \texttt{fork}\, \texttt{fn}\, x \Rightarrow e] \xrightarrow{\{p,q\}} K, P[p : \texttt{()}][q : e]} \tag{9}$$

$$\frac{com_1, com_2 \Longrightarrow e_1, e_2}{K, P[p : \texttt{sync}(com_1)][q : \texttt{sync}(com_2)] \xrightarrow{\{p,q\}} K, P[p : e_1][q : e_2]} \tag{10}$$

Figure 4: Evaluation Rules

$$\frac{c_i, com \Longrightarrow e_1,\ e_2}{\langle \texttt{choose}, (c_1, c_2)\rangle,\ com \Longrightarrow e_1,\ e_2} \quad i \in \{1, 2\} \tag{11}$$

$$\langle \texttt{send}, (k, v)\rangle,\ \langle \texttt{receive}, k\rangle \Longrightarrow v,\ v \tag{12}$$

$$\frac{com_1, com_2 \Longrightarrow e_1,\ e_2}{\langle \texttt{wrap}, (com_1, e_3)\rangle,\ com_2 \Longrightarrow e_3\ e_1,\ e_2} \tag{13}$$

$$\frac{com_2, com_1 \Longrightarrow e_2,\ e_1}{com_1, com_2 \Longrightarrow e_1,\ e_2} \tag{14}$$

Figure 5: Communication Rules

Definition 2.1 *The set of all channels in P is denoted* chan P. *A configuration K, P is* well-formed *iff* (chan P) $\subseteq K$. *A reduction sequence is well-formed if all the configurations in the sequence are well-formed.*

Lemma 2.1 ***(Preservation Lemma)*** *In a transition $\vdash K, P \xrightarrow{S} K', P'$, the following all hold:*

1. $\mathrm{dom}(P) \subseteq \mathrm{dom}(P')$.
2. $K \subseteq K'$.
3. *If K, P is well-formed then so is K', P'.*
4. *If $k \notin K'$, then $\vdash K \cup \{k\}, P \xrightarrow{S} K' \cup \{k\}, P'$.*
5. *If $k \in K$ and $k \notin \mathrm{chan}(P)$, then* $\vdash K - \{k\}, P \xrightarrow{S} K' - \{k\}, P'$.
6. *If $p \notin dom(P')$ and* $\mathrm{chan}([p : e]) \subseteq K$, *then* $\vdash K, P[p : e] \xrightarrow{S} K', P'[p : e]$.
7. *If $P = P''[p : e]$ and $p \notin S$, then there exists P''' such that $\vdash K, P'' \xrightarrow{S} K', P'''$ and $P' = P'''[p : e]$.*

Proof: The proof is a simple induction on the depth of inference of $\vdash K, P \xrightarrow{S} K', P'$.

Corollary 2.2 *The same properties hold of a reduction sequence $\vdash K, P \longrightarrow^* K', P'$.*

3 Conservation of Sequential Behaviour

One property that we wish to hold for our language is that a sequential expression will produce the same result in our semantics as it would in the usual sequential semantics. We show that this is the case by giving the usual transitional rules for the sequential part of our language, and showing that the full language produces the same behaviour for sequential expressions. This result can be extended to a relational semantics of the sequential language using a standard proof of equivalence between the transitional and relational semantics, such as the one given by Hennessy [Hen90].

The following rules give the usual transitional semantics of the sequential part of our language.

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \tag{15}$$

$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \tag{16}$$

$$(\mathtt{fn}\ x \Rightarrow e)\ v \longrightarrow e\{v/x\} \tag{17}$$

$$c\ v \longrightarrow \langle c, v\rangle \tag{18}$$

$$\frac{e_1 \longrightarrow e_1'}{(e_1, e_2) \longrightarrow (e_1', e_2)} \tag{19}$$

$$\frac{e \longrightarrow e'}{(v, e) \longrightarrow (v, e')} \tag{20}$$

$$\mathtt{rec}\ f(x) \Rightarrow e \longrightarrow \mathtt{fn}\ x \Rightarrow e\{\mathtt{rec}\ f(x) \Rightarrow e/f\} \tag{21}$$

Lemma 3.1 *A transition* $\vdash K, P \longrightarrow K', P'$ *has either one or two selected processes.*

Definition 3.1 *A transition with two selected processes is called an* interaction.

Only the rules for `sync` and `fork` give rise to interactions. This is because they are the only operations that can transfer values between processes. `sync` transfers values explicitly. `fork` can transfer values in the body of the function passed to the process that it creates.

Definition 3.2 *Given a reduction sequence* T, int_T *is the set of the ProcessId pairs that label the interactions in* T. *We define* $\equiv_T$ *to be the transitive-reflexive closure of* int_T. *The* T-effect *of a set* S *of ProcessIds is the set of all ProcessIds that are equivalent* ($\equiv_T$) *to any member of* S. *If* P *is a ProcessSet such that* $\mathrm{dom}\ P = S$, *then the transitions in* T *that are labelled with ProcessIds in the* T*-effect of* S *are called the* P-affected *transitions of* T.

It follows that if T has the form $\vdash K, P \longrightarrow^* K', P'$ then *dom* P' is the T-effect of *dom* P.

Definition 3.3 *Let* T_1 *and* T_2 *be the well-formed reduction sequences* $\vdash K_1, P_1 \longrightarrow^* K_1', P_1'$ *and* $\vdash K_2, P_2 \longrightarrow^* K_2', P_2'$. *Let the length of* T_2 *be* n. *Then* T_2 *is a* projection *of* T_1 *if* $P_2 \subseteq P_1$, $K_2 \subseteq K_1$, T_1 *contains exactly* n P_2*-affected transitions, and for* $i \in \{1, ..., n\}$, *the selected processes in the* i^{th} *transition in* T_2 *are identical to those in the* i^{th} P_2*-affected transition in* T_1.

Definition 3.4 *Two ProcessSets* P_1 *and* P_2 *are* independent, *written* $P_1 \parallel P_2$, *iff* chan P_1 *and* chan P_2 *are disjoint. An expression is* sequential *if it doesn't contain any constructors or basic values.*

Lemma 3.2 ***(Projection Lemma)*** *Let* T *be the reduction sequence* $\vdash K, P \longrightarrow^* K', P'$, *and* $P = P_1 \cup P_2$, *where* $P_1 \parallel P_2$. *Let* $P_i' \subseteq P'$ *and* $\mathrm{dom}\ P_i'$ *be the* T*-effect of* $\mathrm{dom}\ P_i$ ($i = 1, 2$), *so that* $P' = P_1' \cup P_2'$. *Then there is a projection* T' *of* T *of the form* $\vdash K, P_1 \longrightarrow^* K'', P_1'$, *for some* K''.

Proof:

The proof uses induction on the length of T. Let n be the number of P_1-affected transitions in T. Let the first transition t in T be $\vdash K, P_1 \cup P_2 \xrightarrow{S} K''', P_1'' \cup P_2''$, where *dom* P_1'' and *dom* P_2'' are the t-effects of *dom* P_1 and *dom* P_2. Let T_1 be the rest of T. By induction, there is a projection T_1' of T_1 with the form $\vdash K''', P_1'' \longrightarrow^* K'', P_1'$.

If $S \subseteq P_1''$, then T_1 and T_1' each contain $n-1$ P_1-affected transitions. It follows that the reduction sequence formed by prefixing T_1' with $\vdash K, P_1 \xrightarrow{S} K''', P_1''$ is a projection of T, as required.

If $S \subseteq P_2''$, then T_1 and T_1' each contain n P_1-affected transitions. Furthermore $P_1'' = P_1$. If we also have $K''' = K$, then T_1' is a projection of T with the desired form. If $K''' \neq K$, then there exists k such that $K''' = K \cup \{k\}$. But $k \notin chan\, P_1$, so by the Preservation Lemma we have $\vdash K''' - \{k\}, P_1'' \longrightarrow^* K'' - \{k\}, P_1'$, which is a projection of T with the desired form.

Corollary 3.3 *If* T *is the reduction sequence* $\vdash K, P[p : e] \longrightarrow^* K', P'[p : e']$ *and* e *is sequential, then there exists a projection of* T *that has the form* $\vdash K, [p : e] \longrightarrow^* K, [p : e']$.

We can now state our first result. This theorem shows that sequential expressions produce the same result in our language as they do in a conventional sequential language. It suggests that the functional behaviour of SML is preserved when our primitives are added.

Theorem 3.4 ***(Conservative Extension Theorem)*** *If* e *is sequential, then for all* K, P *and* p *we have that* $\vdash e \longrightarrow^* v$ *iff* $\exists K', P'$ *such that* $\vdash K, P[p : e] \longrightarrow^* K', P'[p : v]$

Proof:

Only if **case**: If we can show that:

$$\text{If } \vdash e \longrightarrow e' \text{ then } \exists P' \text{ such that } \vdash K, P[p : e] \longrightarrow K, P'[p : e'].$$

then the result follows by a simple induction on the length of $\vdash e \longrightarrow^* v$. We can show that this property holds by using induction on the depth of inference of $\vdash e \longrightarrow e'$ and considering cases of e.

Case 1. $e \equiv \mathtt{fn}\, x \Rightarrow e_1\ v$.

Then $e' = e_1\{v/x\}$, from Rule 17. Rule 3 is the only rule in the parallel language specification that matches e. It yields the configuration $K, P[p : e_1\{v/x\}]$, and the result follows directly.

Case 2. $e \equiv c\ v$.

Case 3. $e \equiv \mathtt{rec}\, f(x) \Rightarrow e_1$.

These cases are similar to Case 1.

Case 4. $e \equiv e_1\ e_2 \quad (e_1 \notin \text{Value})$.

Then $\vdash e \longrightarrow e'$ must have been inferred by Rule 15. Therefore $\vdash e_1 \longrightarrow e_1'$. By induction, there exists P' such that $\vdash K, P[p : e_1] \longrightarrow K', P'[p : e_1']$. The result follows from Rule 1.

Case 5. $e \equiv v\ e_1$.

Case 6. $e \equiv (e_1, e_2) \quad (e_1 \notin \text{Value})$.

Case 7. $e \equiv (v, e_2)$.

These cases are similar to Case 4.

If **case**:

By the corollary of the Projection Lemma, there exists a projection of $\vdash K, P[p : e] \longrightarrow^* K', P'[p : v]$ with the form $\vdash K, [p : e] \longrightarrow^* K, [p : v]$. The result follows from a simple recasting of the *only if* case.

4 Modelling Stores as Processes

We also want to integrate our primitives with the non-functional features of SML. One approach is to define these features in terms of processes. In this section we show how stores can be defined in terms of our simple language. We prove that these definitions produce the desired behaviour, using similar techniques to those of the previous section.

We begin by giving a transitional semantics for a sequential language with stores. These rules require two new sets of semantic objects:

$$\begin{array}{rcl} a & \in & \text{Address} \\ s & \in & \text{Store} = \text{Address} \xrightarrow{\text{fin}} \text{Value} \end{array}$$

In addition, we extend the set of basic values with `ref`, `assign` and `deref`, and the set of values with the set of addresses. Sentences are extended with stores in the obvious way.

The first seven rules are the rules of the sequential language, extended with stores.

$$\frac{e_1,\ s_1 \longrightarrow e_1',\ s_2}{e_1\ e_2,\ s_1 \longrightarrow e_1'\ e_2,\ s_2} \tag{22}$$

$$\frac{e,\ s_1 \longrightarrow e',\ s_2}{v\ e,\ s_1 \longrightarrow v\ e',\ s_2} \tag{23}$$

$$\mathtt{fn}\, x \Rightarrow e\ v,\ s \longrightarrow e\{v/x\},\ s \tag{24}$$

$$c\ v,\ s \longrightarrow \langle c, v\rangle,\ s \tag{25}$$

$$\mathtt{rec}\, f(x) \Rightarrow e,\ s \longrightarrow \mathtt{fn}\, x \Rightarrow e\{\mathtt{rec}\, f(x) \Rightarrow e/f\},\ s \tag{26}$$

$$\frac{e_1,\ s_1 \longrightarrow e_1',\ s_2}{(e_1, e_2),\ s_1 \longrightarrow (e_1', e_2),\ s_2} \tag{27}$$

$$\frac{e,\ s_1 \longrightarrow e',\ s_2}{(v, e),\ s_1 \longrightarrow (v, e'),\ s_2} \tag{28}$$

In addition, there are three new rules that define the creation, updating and dereferencing of store cells:

$$\frac{a \notin dom\ s}{\texttt{ref}\ v,\ s \longrightarrow a,\ s{+}(a, v)} \tag{29}$$

$$\texttt{assign}\ (a, v),\ s \longrightarrow (),\ s{+}(a, v) \tag{30}$$

$$\texttt{deref}\ a,\ s \longrightarrow s(a),\ s \tag{31}$$

The following lemma will be needed in the proof of the result:

Lemma 4.1 ***(Reordering Lemma)*** *If*
$\vdash K, P[p : e] \xrightarrow{S} K', P'[p : e] \xrightarrow{\{p\}} K'', P'[p : e']$
where $p \notin S$ *and* $K, P[p : e]$ *is well-formed, then*
$\vdash K, P[p : e] \xrightarrow{\{p\}} K''', P[p : e'] \xrightarrow{S} K'', P'[p : e']$
where $K''' = K'' - (K' - K)$.

Proof:

We know that $\vdash K', P'[p : e] \xrightarrow{\{p\}} K'', P'[p : e']$ and that $K' - K$ and $chan[p : e]$ are disjoint. Therefore $\vdash K, P'[p : e] \xrightarrow{\{p\}} K''', P'[p : e']$ by Case 5 of the Preservation Lemma. Then Case 7 of that lemma shows that $\vdash K, P[p : e] \xrightarrow{\{p\}} K''', P[p : e']$.

We also know that $\vdash K, P[p : e] \xrightarrow{S} K', P'[p : e]$ and that $K''' - K$ and K' are disjoint. Therefore $\vdash K''', P[p : e] \xrightarrow{S} K'', P'[p : e]$ by Case 4 of the Preservation Lemma. Furthermore, since $p \notin S$ and $chan[p : e'] \subseteq K'''$, we can deduce that $\vdash K''', P[p : e'] \xrightarrow{S} K'', P'[p : e']$ by Cases 6 and 7 of the Preservation Lemma.

We now define a representation of a store as a set of processes. First, we define a map from basic values in the sequential language with stores to expressions in the concurrent language[1]. This map extends to values and expressions in the obvious way.

```
rep(ref) = fn x =>
    let addr = channel ()
    in let f = rec cell(y) =>
                cell (sync <choose, (
                        <receive, addr>,
                        <send, (addr, y)>
                )>)
        in fork (fn z => f x); addr
        end
    end
```

```
rep(deref) = fn a => sync <receive, a>
rep(assign) = fn x => sync <send, x>; ()
```

$rep(b) = b, b \notin \{\texttt{ref}, \texttt{deref}, \texttt{assign}\}$.

In our proof of the result we frequently make implicit use of the fact that the representation of an expression is composed of the representation of its sub-expressions. For example, $rep(e_1\ e_2) = (rep\ e_1)\ (rep\ e_2)$.

[1]For clarity, we use the derived forms '`let x = e1 in e2 end`' and '`e3; e4`' to mean '`(fn x => e2) e1`' and '`(fn y => e4) e3`' respectively (where `y` is not free in `e4`).

Next, we define the set of addresses to be a subset of the set of channels. We also introduce a subset of ProcessIds, called StoreIds, and an injection $stid : a \mapsto p_a$ from addresses to StoreIds. We define the following abbreviations:

```
cell_def(a) = rec cell(y) =>
                 cell (sync <choose, (
                          <receive, a>,
                          <send, (a, y)>
                      )>)

cell_fn(a) = fn y =>
                cell_def(a) (sync <choose, (
                              <receive, a>,
                              <send, (a, y)>
                          )>)

cell(a, v) = cell_fn(a) (sync <choose, (
                          <receive, a>,
                          <send, (a, v)>
                      )>)
```

These abbreviations record the recursive evaluation of a cell. When a cell $cell(a, v)$ communicates with another process, it evaluates to $cell_{fn}(a)\ v'$, which evaluates in two steps to $cell(a, v)$. When a cell is created, $cell_{def}(a)$ evaluates to $cell_{fn}(a)$.

Now we can define the representation of an element of a store. This definition extends to stores in the obvious way:

$$rep(a, v) = [p_a \ : \ cell(a, v)]$$

Now we can state our second result. This shows that our representation of stores produces the same results as the augmented sequential language, when only one process can access the store. The theorem has two parts, one showing that our language cannot produce results that the sequential language cannot, and one showing that our language can produce every result that the sequential language can. We are unable to state the theorem using *iff* because the side-conditions on the store are part of the implied statement in both cases, instead of being linked to one of the evaluations.

Theorem 4.2 ***(Store Theorem)***

Part A. *Let* $K, P[p : \mathrm{rep}\ e]$ *be well-formed and* $\{[p : \mathrm{rep}\ e]\} \parallel P$*. Let* s' *be a store with* $\mathrm{stid}(\mathrm{dom}\ s')$ *disjoint from* $(\mathrm{dom}\ P) \cup \{p\}$*. If* $\vdash e, \{\} \longrightarrow^* v, s'$ *then there is a reduction sequence* T *of the form:*

$$\vdash K, P[p : \mathrm{rep}\ e] \longrightarrow^* K' \cup (\mathrm{dom}\ s'), P' \cup (\mathrm{rep}\ s')[p : \mathrm{rep}\ v]$$

where $P' \parallel (\mathrm{rep}\ s')[p : \mathrm{rep}\ v]$ *and the* T*-effect of* $\{p\}$ *is* $\mathrm{stid}(\mathrm{dom}\ s') \cup \{p\}$*.*

Part B. *Let* $K, P[p : \mathrm{rep}\ e]$ *be well-formed and let* $\{[p : \mathrm{rep}\ e]\} \parallel P$*. Let* T *be a reduction sequence with the form* $\vdash K, P[p : \mathrm{rep}\ e] \longrightarrow^* K', P'[p : \mathrm{rep}\ v]$*. Then there exists a store* s' *such that:*

$$\vdash e, \{\} \longrightarrow^* v, s'$$

where $P' \parallel (\mathrm{rep}\ s')[p : \mathrm{rep}\ v]$*, the* T*-effect of* $\{p\}$ *is* $\mathrm{stid}(\mathrm{dom}\ s') \cup \{p\}$*,* $(\mathrm{dom}\ s) \subseteq K'$ *and* $\mathrm{rep}\ s \subseteq P'$*.*

Proof:

In each part we prove a stronger statement, from which the result follows trivially. We begin each evaluation with a store s (or representation thereof) instead of the empty store, and add the condition that $P \parallel (rep\ s)[p : rep\ e]$. As a result, $stid(dom\ s') \cup \{p\}$ is now required to be the T-effect of $\{p\} \cup (rep\ s)$. (This strengthening of the statement is why we use the variable s' instead of s in the original.)

Part A:

If we can show that the statement holds for a single transition $\vdash e, s \longrightarrow e', s'$, then the result follows by a simple induction on the length of $\vdash e, s \longrightarrow^* v, s'$. We can show that this property holds by using induction on the depth of inference of $\vdash e, s \longrightarrow^* v, s'$ and considering cases of e:

Case 1. $e \equiv \texttt{fn}\, x \Rightarrow e_1\ v$.

Then $e' = e_1\{v/x\}$and $s' = s$, from Rule 24. Rule 3 is the only rule in the parallel language specification that matches e. It yields the configuration $K \cup (dom\ s), P \cup (rep\ s)[p : rep(e_1\{v/x\})]$ and the result follows directly.

Case 2. $e \equiv c\ v$.

Case 3. $e \equiv \texttt{rec}\, f(x) \Rightarrow e_1$.

These cases are similar to Case 1.

Case 4. $e \equiv e_1\ e_2 \quad (e_1 \notin \text{Value})$.

Then $\vdash e, s \longrightarrow e', s'$ must have been inferred by Rule 22. Therefore $\vdash e_1, s \longrightarrow e_1', s'$. By induction, there exists a reduction sequence T' with the form:

$\vdash K \cup (dom\ s), P \cup (rep\ s)[p : rep\ e_1] \longrightarrow^* K' \cup (dom\ s'), P' \cup (rep\ s')[p : rep\ e_1']$

where $P' \parallel (rep\ s')[p : rep\ e_1']$ and the T'-effect of $stid(dom\ s) \cup \{p\}$ is $stid(dom\ s') \cup \{p\}$. From (repeated applications of) Rule 1, there is a reduction sequence T'' of the form:

$\vdash K \cup (dom\ s), P \cup (rep\ s)[p : rep\ e] \longrightarrow^* K' \cup (dom\ s'), P' \cup (rep\ s')[p : rep\ e']$

where the T''-effect of $stid(dom\ s) \cup \{p\}$ is $stid(dom\ s') \cup \{p\}$. Also, we know that $P' \parallel [p : rep\ e_2]$, and so we have $P' \parallel (rep\ s')[p : rep\ e']$ as desired.

Case 5. $e \equiv v\ e_1$.

Case 6. $e \equiv (e_1, e_2) \quad (e_1 \notin \text{Value})$.

Case 7. $e \equiv (v, e_2)$.

These cases are similar to Case 4.

Case 8. $e \equiv \texttt{ref}\ v$

Then $e' = a$, and $s' = s + (a, v)$, where $a \notin dom\ s$, by Rule 29. Also,

```
rep e = fn x => let addr = channel () in
        let f =  cell_def(addr)
        in fork (fn z => f x); addr end
        end   v
```

This can be evaluated by the reduction sequence shown in Figure 6, which produces the configuration $K \cup (dom\ s) \cup \{a\}, P \cup (rep\ s)[p : a][p_a : cell(a, v)]$ as desired.

Case 9. $e \equiv \texttt{assign}(a, v)$

Then $e' = ()$ and $s' = s + (a, v)$, from Rule 30. Also,

$rep\ e = (\texttt{fn x => sync <send, x>; ()})\ \ (a,\ v)$.

The evaluation of $rep\ e$ begins with the following transition:

$\vdash K \cup (dom\ s), P \cup (rep\ s)[p : rep\ e] \longrightarrow K \cup (dom\ s), P \cup (rep\ s)[p : \texttt{sync <send, } (a,\ v)\texttt{>; ()}]$

Let $rep(a, v) = [p_a : cell(a, v)]$. Then the evaluation continues with the transition shown in Figure 7. This produces the configuration:

$K \cup (dom\ s), P \cup rep(s - (a, v))[p_a : cell_{fn}(a)\ v][p : v; ()]$

which evaluates in two steps to:

$K \cup (dom\ s), P \cup rep(s - (a, v))[p_a : cell(a, v)][p : v; ()]$

and then to:

$K \cup (dom\ s), P \cup rep(s - (a, v))[p_a : cell(a, v)][p : ()]$

as desired.

Case 10. $e \equiv \texttt{deref}\ a$

Then $e' = s(a)$ and $s' = s$, from Rule 31. Also, $rep\ e = \texttt{fn x => sync <receive, x>}\ a$. The evaluation of $rep\ e$ begins with the following transition:

$\vdash K \cup (dom\ s), P \cup (rep\ s)[p : rep\ e] \longrightarrow K \cup (dom\ s), P \cup (rep\ s)[p : \texttt{sync <receive, } a\texttt{>}]$

$$
\begin{array}{ll}
K\cup(\mathit{dom}\ s), P\cup(\mathit{rep}\ s)[p : \mathit{rep}\ e] \longrightarrow & \\
K\cup(\mathit{dom}\ s), P\cup(\mathit{rep}\ s)[p : \texttt{let addr = channel () in ... end}] \longrightarrow & \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : \texttt{let addr = } a \texttt{ in ... end}] \longrightarrow & (a \notin K\cup(\mathit{dom}\ s)) \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : \texttt{let f => } \mathit{cell}_{\mathit{def}}(a) \texttt{ in ... end}] \longrightarrow & \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : \texttt{let f => } \mathit{cell}_{\mathit{fn}}(a) \texttt{ in ... end}] \longrightarrow & \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : \texttt{fork (fn z => } \mathit{cell}_{\mathit{fn}}(a)\ v); a] \longrightarrow & \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : (); a][p_a : \mathit{cell}_{\mathit{fn}}(a)\ v] & \\
\text{which evaluates in two steps to:} & \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : (); a][p_a : \mathit{cell}(a, v)] & \\
\text{which evaluates in one step to:} & \\
K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p : a][p_a : \mathit{cell}(a, v)] &
\end{array}
$$

Figure 6: The sequence of transitions for Case 8 of the proof of the Store Theorem (omitting premises).

$$
\begin{array}{c}
\texttt{<receive}, a\texttt{>}, \texttt{<send}, (a, v)\texttt{>} \\ \hline
\texttt{<choose, ... >}, \texttt{<send}, (a, v)\texttt{>} \\ \hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : \texttt{sync <choose, ... >}][p : \texttt{sync <send}, (a, v)\texttt{>}] \\ \hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : \texttt{sync <choose, ... >}][p : \texttt{sync <send}, (a, v)\texttt{>}; ()] \\ \hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : \mathit{cell}(a,v)][p : \texttt{sync <send}, (a, v)\texttt{>}; ()]
\end{array}
$$

$$
\longrightarrow \quad
\begin{array}{c}
v,\ v \\ \hline
v,\ v \\ \hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : v][p : v] \\ \hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : v][p : v; ()] \\ \hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : \mathit{cell}_{\mathit{fn}}(a)\ v][p : v; ()]
\end{array}
$$

Figure 7: The key transition of Case 9 of the proof of the Store Theorem.

Let $\mathit{rep}(a, v) = [p_a : \mathit{cell}(a, v)]$. Then the evaluation continues with the transition shown in Figure 8. This produces the configuration:

$K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : \mathit{cell}_{\mathit{fn}}(a)\ v][p : v]$

which evaluates in two steps to:

$K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a : \mathit{cell}(a,v)][p : v]$

as desired.

Part B:

By Corollary 3.3 there exists a projection T of the reduction sequence that we start from, with the form: $\vdash K\cup(\mathit{dom}\ s), (\mathit{rep}\ s)[p : \mathit{rep}\ e] \longrightarrow^* K'\cup(\mathit{dom}\ s'), (\mathit{rep}\ s')[p : \mathit{rep}\ v]$.

Each process in $\mathit{rep}\ s$ has the form $[p_a : \mathit{cell}(a, v)]$, where a is unique. The only rule that matches one of these processes is Rule 10. This rule requires another process to perform a `sync` operation at the same time, and the communication rules require that the two communications must share a channel. Therefore p must be a selected process of the first transition of T.

Also, each process in $\mathit{rep}\ s$ can only interact when it has the form $[p_a : \mathit{cell}(a, v)]$. After interacting and just after creation, it has a different form, and can't interact again until it returns to this form. Therefore we can use the Reordering Lemma to produce a reduction sequence T' which is identical to T except that all transitions that return a process in $\mathit{rep}\ s$ to the form

$$
\begin{array}{c}
\texttt{<send, }(a,\ v)\texttt{>, <receive, }a\texttt{>} \\
\hline
\texttt{<choose, }\ldots\texttt{>, <receive, }a\texttt{>} \\
\hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a\ :\ \texttt{sync <choose, }\ldots\texttt{>}][p\ :\ \texttt{sync <receive, }a\texttt{>}] \\
\hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a\ :\ \mathit{cell}(a,v)][p\ :\ \texttt{sync <receive, }a\texttt{>}]
\end{array}
$$

$$
\longrightarrow\quad
\begin{array}{c}
v,\ v \\
\hline
v,\ v \\
\hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a\ :\ v][p\ :\ v] \\
\hline
K\cup(\mathit{dom}\ s), P\cup \mathit{rep}(s-(a,v))[p_a\ :\ \mathit{cell}_{\mathit{fn}}(a)\ v][p\ :\ v]
\end{array}
$$

Figure 8: The key transition for Case 10 of the proof of the Store Theorem.

$[p_a\ :\ \mathit{cell}(a,v)]$ occur immediately after the relevant interaction or creation.

We proceed by induction on the length of T', considering cases of e.

Case 1. $e \equiv \texttt{fn}\ x \texttt{ => } e_1\ v$

By Rule 3, the first transition of T' is:

$\vdash K\cup(\mathit{dom}\ s), (\mathit{rep}\ s)[p\ :\ \mathit{rep}\ e] \longrightarrow K\cup(\mathit{dom}\ s), (\mathit{rep}\ s)[p\ :\ \mathit{rep}(e_1\{v/x\})]$

Now, Rule 24 is the only sequential rule that matches e, and it yields the configuration $e_1\{v/x\}, s$. By induction, there exists a reduction sequence $\vdash e_1\{v/x\}, s \longrightarrow^* v, s'$. The result follows.

Case 2. $e \equiv c\ v$.

Case 3. $e \equiv \texttt{rec}\ f(x) \texttt{=>} e_1$.

These cases are similar to Case 1.

Case 4. $e \equiv e_1\ e_2\ \ (e_1 \notin \text{Value})$.

T' must begin with a sequence of transitions which each have an instance of Rule 1 as the final inference of the transition. The premises of these instances form an evaluation of the form:

$\vdash K\cup(\mathit{dom}\ s), (\mathit{rep}\ s)[p\ :\ \mathit{rep}\ e_1] \longrightarrow K''\cup(\mathit{dom}\ s''), (\mathit{rep}\ s'')[p\ :\ \mathit{rep}\ v_1]$

Since this is a smaller evaluation than T', we can use induction to show that $\vdash e_1, s \longrightarrow^* v_1, s''$. Repeated applications of Rule 22 give us $\vdash e, s \longrightarrow^* v_1\ e_2, s''$. The remainder of T' has the form: $\vdash K''\cup(\mathit{dom}\ s''), (\mathit{rep}\ s'')[p\ :\ \mathit{rep}(v_1\ e_2)] \longrightarrow^* K'\cup(\mathit{dom}\ s'), (\mathit{rep}\ s')[p\ :\ \mathit{rep}\ v]$. By induction, $\vdash v_1\ e_2, s'' \longrightarrow^* v, s'$. The result follows directly.

Case 5. $e \equiv v\ e_1$.

Case 6. $e \equiv (e_1, e_2)\ \ (e_1 \notin \text{Value})$.

Case 7. $e \equiv (v, e_2)$.

These cases are similar to Case 4.

Case 8. $e \equiv \texttt{ref}\ v$

We know that

```
rep(e) =fn x => let addr = channel () in
        let f =  cell_def(addr)
        in fork (fn z => f x); addr
        end
        end   v
```

Therefore T' must begin with a permutation of the sequence of transitions given in Case 8 of Part A. This yields the configuration: $K\cup(\mathit{dom}\ s)\cup\{a\}, P\cup(\mathit{rep}\ s)[p\ :\ a][p_a\ :\ \mathit{cell}(a,v)]$. The only sequential rule that applies is Rule 29. This gives the configuration $a, s{+}(a,v)$ as desired.

Case 9. $e \equiv \texttt{assign}\ (a,v)$

We know that $\mathit{rep}\ e\ =\ \texttt{(fn x => sync <send, x>; ())}\ (a,v)$ Therefore T' must begin with a permutation of the sequence of transitions given in Case 9 of Part A. This yields the configuration $K\cup \mathit{dom}\ s, P\cup \mathit{rep}(s-(a,v))[p_a\ :\ \mathit{cell}(a,v)][p\ :\ \texttt{()}]$. The only sequential rule that applies is Rule 30. This gives the configuration $\texttt{()}, s{+}(a,v)$ as desired.

Case 10. $e \equiv$ `deref` a

Now, $rep\ e =$ `fn x => sync <receive, x>` a. Therefore T' must begin with the sequence of transitions given in Case 10 of Part A. This yields the configuration:

$K \cup dom\ s, P \cup rep(s-(a,v))[p_a : cell(a,v)][p : v]$

The only sequential rule that applies is Rule 31. This gives the configuration v, s as desired.

5 Further Work

There are three main avenues that we wish to follow with this work in the future. The first is to see how other communication constructs can be expressed in our semantics. Many constructs can be expressed as library functions using the primitives presented here, as Reppy has shown [Rep89]. We hope to define a wide range of features in this way, possibly including those of LINDA [CG89].

For example, we could define an asynchronous send operator as a function that forks a new process to send the value. We could use techniques similar to those used in the Store Theorem to show that our definition behaved as desired. A compiler could include an implementation of this function that was more efficient than the formal definition. This would continue an established tradition in the ML world, typified by the current definition of arrays in terms of lists [Ber91a].

Some constructs probably can't be expressed in terms of the ones given here. We hope to extend our semantics to cover these constructs explicitly. For example, in Reppy's latest paper on CML he includes some new primitive operators, `guard` and `wrapAbort` [Rep91a]. `guard` takes a function argument of type `() -> 'a com`. When `sync` is applied to it, it generates a communication value by applying the function. `wrapAbort` pairs a communication value with a function that is called if the value is part of a choice and the choice selects another communication. Reppy's semantics for CML includes definitions for these operators, and we don't foresee any problems defining them in our semantics.

The second avenue that we wish to follow is that of incorporating our semantics with the Definition of Standard ML [MTH90]. The Definition uses the relational style of operational semantics (which we also use for our communication rules). It seems to be impossible to use this style to define concurrent systems, because there is no way to specify potentially infinite interleaved evaluations.

We need a way of relating the two styles of operational semantics. One approach is to follow the ideas in Berry's thesis [Ber91b], and define a syntactic expansion of relational rules into corresponding transitional rules. Another approach, not necessarily disjoint, is to define several features of SML in terms of our primitives, and then to show an equivalence between an appropriately reduced version of the definition and our simple language. The theorems presented here are steps along this path.

The third avenue that we wish to follow is that of implementation. CML has already been implemented on uniprocessors and on the Mach operating system. Matthews has implemented his primitives on uniprocessors and on a shared memory multiprocessor (the DEC Firefly). He is working with us on an implementation in which the persistent store of Poly/ML is distributed across a network of workstations, thus allowing processes to be created remotely. It is trivial to define our primitives in terms of his, so we should soon be able to test our primitives in a true distributed implementation.

We are also working on a proof of correctness for a protocol that implements our communication operations in a distributed environment. Most concurrent languages don't allow choices between both `send` and `receive` constructs; our language does, and this complicates the implementation. Some extensions to CSP have tackled this problem [BS83], but these use process-to-process communication instead of communication via channels. The only work that we know of in this area is Knabe's implementation of Facile [Kna] and Mitchell's implementation of PFL [Mit86].

6 Conclusion

We have presented a semantics for a simple concurrent language. The features of this language are the same as those of CML, and are similar to other concurrent extensions of SML. As in CML, the functions `send`, `receive`, etc. build suspended communications of the type `'a com`. The `sync` operator must be applied to a value of this type to make the communication actually happen. We use this system because it gives a simple semantics. By contrast, Reppy uses it for its practical utility. This agreement of the theoretical and practical suggests that this set of primitives is both natural and desirable.

We have shown that our semantics preserves the desired behaviour of sequential expressions. We have also defined stores in terms of our primitives, and have shown that these behave as desired. The proofs of these theorems use some lemmas that should be useful in proofs of similar statements.

Our theorems are steps on the way to incorporating our semantics with the Definition of Standard ML. We are investigating other constructs for concurrency with respect to our semantics, and are implementing our primitives in a distributed version of SML.

References

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, 1990.

[Ber89] B. Berthomieu. Implementing CCS, the LCS experiment. Technical Report 89425, LAAS-CNRS, 1989.

[Ber91a] Dave Berry. The Edinburgh SML library. LFCS Report Series ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[Ber91b] Dave Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, University of Edinburgh, 1991.

[BS83] G.N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):234–238, Apr 1983.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CM90] Eric C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, 1990.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages.* Wiley, 1990.

[Hol83] Sören Holström. PFL: A functional language for parallel programming and its implementation. Report 83.03 R, Department of Computer Science, Chalmers University of Technology, 1983.

[Kna] F. Knabe. A distributed protocol for channel-based communication with choice. ECRC, Munich. In Preparation.

[Mat91] David Matthews. A distributed concurrent implementation of Standard ML. In *EurOpen Autumn 1991 Conference*, 1991. To appear.

[Mit86] Kevin Mitchell. *Implementations of Process Synchronisation and their Analysis*. PhD thesis, Department of Computer Science, University of Edinburgh, Jul 1986.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT, 1990.

[Rep89] J. H. Reppy. First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Dept. of Computer Science, Cornell University, 1989.

[Rep91a] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26(6)*, pages 294–305, 1991.

[Rep91b] J. H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Dept. of Computer Science, Cornell University, Aug 1991.

[WF91] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Dept. of Computer Science, Rice University, Apr 1991.