An Operational Semantics of First-class Synchronous Operations

John H. Reppy*

TR 91-1232
August 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853

*Supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

An Operational Semantics of First-class Synchronous Operations

John H. Reppy *
Cornell University
jhr@cs.cornell.edu

August 15, 1991

Abstract

First-class synchronous operations are a new approach to synchronization and communication in concurrent languages. They have been informally described in [Rep88], and [Rep91]; this paper presents an operational semantics for an untyped language with first-class synchronous operations. This language includes a large fraction of the concurrency primitives of Concurrent ML (**CML**), a concurrent extension of **SML**, and is the first step toward formalizing the definition of **CML**.

1 Introduction

First-class synchronous operations are an abstraction mechanism for communication and synchronization in concurrent languages[Rep88, Rep91]. The idea is to separate the description of synchronous communication from the act of synchronization (and communication), in much the same way that procedural abstraction separates the description of computation (i.e., λ–terms) from the actual computation (i.e., function application). The values used to represent synchronous operations are called *events* (we use this term instead of *communications* to stress the fact that synchronous operations other than channel I/O are covered by the mechanism).

First-class synchronous operations were originally developed as part of **PML**[Rep88], a concurrent **ML**–like language used in the **Pegasus** system. Later, the concurrency primitives of **PML** were implemented on top of **SML/NJ**, using first-class continuations[Rep89]. **CML**, which evolved out of this implementation, extends and refines the concurrency mechanisms of **PML**[Rep91]. Both **PML** and **CML** have been used for substantial implementations, which provide empirical evidence of the utility of first-class synchronous operations. Norman Ramsey independently developed a concurrency library on top of **SML/NJ** that

*Supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

supports a variation of **PML**'s primitives[Ram90]. Ramsey's work illustrates an interesting fact: adding **CSP**–style message passing to **SML** will invariably result in something akin to first-class synchronous operations[1]. This is because selective communication must operate on multiple communications, which requires that there be type of values representing communications. This is not surprising, since the original motive for **PML**'s primitives was to provide an abstraction mechanism for channel I/O that cohabits with selective communication.

None of these previous efforts made any attempt to formalize the semantics of first-class synchronous operations; this paper is the first step towards remedying this situation. We present the operational semantics of a simple, untyped, call-by-value λ–calculus extended with first-class synchronous operations. This calculus, called λ_{cv}, models most of the concurrency features of **CML**.

1.1 Related work

The semantics of **Facile** have been proposed as a model for **PML** in [PGM90], but no translation from **PML** to **Facile** is given. Independent work by Berry, Milner and Turner at the University of Edinburgh has resulted in an operational semantics for a small concurrent language, which includes the **PML** version of events[BMT92]. The semantics presented in this paper has some strong similarities with that of [BMT92], but λ_{cv} is a richer language; in particular, the language of [BMT92] does not include the `guard` and `wrapAbort` event value constructors or the `poll` operation of **CML**.

1.2 Organization of this paper

In the next section I present a brief overview of **CML**, including a description of its concurrency primitives. The following section presents λ_{cv}, which models the core concurrency features of **CML**. In section 4, I discuss extending λ_{cv} to support additional features of **CML**. Finally, section 5 presents a polymorphic type system for λ_{cv}, which uses **SML**'s imperative types. A more detailed discussion of these issues, including a proof of the soundness of the type system, can be found in the author's forthcoming dissertation[Rep92].

2 A brief introduction to CML

Before presenting the semantics of **CML**'s concurrency primitives, a brief review of the language is in order. A detailed overview of **CML** can be found in [Rep91]; the complete system is described in [Rep90], which includes a language tutorial.

[1] This does not apply in the case of **PML**, since special syntax is being used to support the primitives.

CML is based on the sequential language **SML**[MTH90, MT91] and inherits the good features of **SML**: functions as first-class values, strong static typing, polymorphism, datatypes and pattern matching, lexical scoping, exception handling and a state-of-the-art module facility. The sequential performance of **CML** benefits from the quality of the **SML/NJ** compiler. In addition **CML** has the following properties:

- **CML** provides a high-level model of concurrency with dynamic creation of threads and typed channels, and rendezvous-style communication. This distributed-memory model fits well with the mostly applicative style of **SML**.
- **CML** is a *higher-order* concurrent language. Just as **SML** supports functions as first-class values, **CML** supports synchronous operations as first-class values. These values, called *events*, provide the tools for building new synchronization abstractions, which are tailored to the application.
- **CML** provides integrated I/O support. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.
- **CML** provides automatic reclamation of threads and channels, once they become inaccessible. This permits a technique of speculative communication, which is not possible in other threads packages.
- **CML** uses pre-emptive scheduling. To guarantee interactive responsiveness, a single thread cannot be allowed to monopolize the processor. Pre-emption insures that a context switch will occur at regular intervals, which allows "off-the-shelf" code to be incorporated in a concurrent thread without destroying interactive responsiveness.
- **CML** is efficient. Thread creation, thread switching and message passing are very efficient (performance numbers are given in [Rep91]). Experience with **CML** has shown that it is a viable language for implementing usable interactive systems.
- **CML** is portable. It is written in **SML** and runs on essentially every system supported by **SML/NJ** (currently four different architectures and many different operating systems).

To make this more concrete, Figure 1 gives the signature of the basic **CML** concurrency operations. These are also the concurrency operations of λ_{cv}. **CML** programs consist of a collection of *threads*, which communicate via typed *channels*. Both threads and channels are created dynamically, using the functions `spawn` and `channel`, respectively. Rather than provide operations for communication, as is done in languages such as **CSP**[Hoa78],

```
type 'a chan
type 'a event

val spawn   : (unit -> unit) -> thread_id
val channel : unit -> '_a chan

val receive  : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

val choose    : 'a event list -> 'a event
val guard     : (unit -> 'a event) -> 'a event
val wrap      : ('a event * ('a -> 'b)) -> 'b event
val wrapAbort : ('a event * (unit -> unit)) -> 'a event

val sync : 'a event -> 'a
```

Figure 1: Basic **CML** Concurrency Operations

occam[Bur88] and **Amber**[Car86], **CML** provides first-class values, called events, to represent synchronous operations. For example, the functions `receive` and `transmit` build event values to describe channel I/O operations. The function `sync` is used to actually apply these operations. There are also event combinators to build more complex synchronous operations:

`choose`. This constructs an event value that represents the non-deterministic choice of its arguments (note that this choice is made when `sync` is applied). A choice may involve multiple communications (both receive and transmit) on the same channel, but a thread cannot communicate with itself.

`guard`. This constructs an event out of an event valued function. When `sync` is applied, the function is called first, and the result is used for synchronization.

`wrap`. This wraps a function around an event value. If the event is chosen in a synchronization, then the function is applied to the result of the event.

`wrapAbort`. This associates an action to be taken if an event is *not* chosen in a synchronization. A new thread is spawned to execute the action.

The power of this approach is that it allows the user to implement new communication and synchronization abstractions. For example, we have found uses for widely varying abstractions, such as remote procedure call, multicast channels and buffered channels.

To illustrate these features, consider the **CML** program in Figure 2. This program first creates two channels, bound to `ch1` and `ch2`; it then spawns two threads, which attempt to send messages on the channels. The parent thread synchronizes on the choice of receiving

```
fun simple () = let
      val ch1 = channel() and ch2 = channel()
      val pr = CIO.print
      in
        pr "hi-0\n";
        spawn (fn () => (pr "hi-1\n";  send(ch1, 17);  pr "bye-1\n"));
        spawn (fn () => (pr "hi-2\n";  send(ch2, 37);  pr "bye-2\n"));
        sync (choose [
            guard (fn () => (
              pr "guard-0.1\n";
              wrapAbort (wrap (receive ch1, fn _ => pr "bye-0.1\n"),
                fn () => pr "abort-0.1\n"))),
            guard (fn () => (
              pr "guard-0.2\n";
              wrapAbort (wrap (receive ch2, fn _ => pr "bye-0.2\n"),
                fn () => pr "abort-0.2\n")))
          ])
      end
```

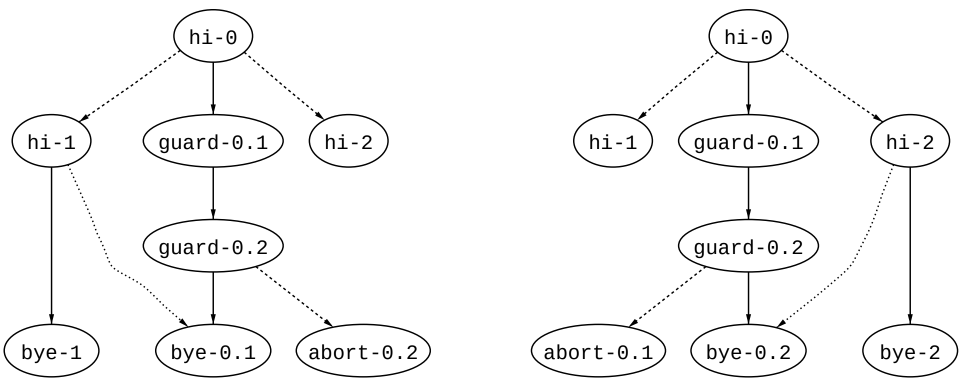
Figure 2: A simple **CML** program.

a message on either channel. Each arm of this choice has an associated wrapper and abort action; depending on which arm is selected, a different set of outputs is possible. The two graphs in Figure 3 detail the possible orderings of the output for the two choices. The dashed edges are induced by `spawn`, the dotted edges by message passing, and the solid edges by sequencing. These graphs illustrate the complementary relationship between the abort actions and wrapper functions. Note that the guards are always evaluated in left-to-right order.

Figure 4 gives the signature of some additional **CML** operations. These are discussed as extensions to λ_{cv} (see Section 4). The function `always` takes a value v builds a base event value that is always immediately available for synchronization (yielding v). The function `threadWait` takes a thread identifier, such as one returned by `spawn`, and returns a base event value for synchronizing on the thread's termination. The operations `accept` and `send` are the channel I/O operations that the event values returned by `receive` and `transmit` represent. Lastly, the operation `poll` is a non-blocking form of `sync`; in a situation in which `sync` would block, it will return `NONE` instead of blocking (the `option` datatype is part of the **SML/NJ** pervasive environment).

3 The semantics of λ_{cv}

In this section I present the dynamic semantics of λ_{cv}, a concurrent extension of Plotkin's λ_v calculus[Plo75]. Although λ_{cv} lacks many of the features of **CML**, it contains the core of



(a) receive **ch1** (b) receive **ch2**

Figure 3: Output orderings

```
val always     : 'a -> 'a event
val threadWait : thread_id -> unit event

val accept : 'a chan -> 'a
val send   : ('a * 'a chan) -> unit

datatype 'a option = NONE | SOME of 'a
val poll : 'a event -> 'a option
```

Figure 4: Other Concurrency Operations

the concurrency primitives (see Figure 1). In section 4, I describe how λ_{cv} can be extended to include many of the missing features. The two features that are not described are the real-time sensitive operations (e.g., `timeout`) and I/O operations.

The dynamic semantics of Core **CML** are defined by two evaluation relations: a sequential evaluation relation "$\longmapsto$," and a concurrent evaluation relation "$\Longrightarrow$." Concurrent evaluation is an extension of sequential evaluation to finite sets of processes.

3.1 Syntax

We start with disjoint sets of *variables*, *function constants*, *base constants* and *channel names*:

$$
\begin{array}{lcll}
x & \in & \textsc{Var} & \text{variables} \\
b & \in & \textsc{Const} = \textsc{FConst} \cup \textsc{BConst} & \text{constants} \\
 & & \textsc{FConst} = \{\texttt{+}, \texttt{-}, \texttt{fst}, \texttt{snd}, \ldots\} & \text{function constants} \\
 & & \textsc{BConst} = \{\texttt{()}, \texttt{true}, \texttt{false}, 0, 1, \ldots\} & \text{base constants} \\
\kappa & \in & \textsc{Ch} & \text{channel names}
\end{array}
$$

To support events, it is required that FCONST include the following event value constructors:

`choose`, `guard`, `never`, `receive`, `transmit`, `wrap`, `wrapAbort`

There are three syntactic classes of terms in λ_{cv}:

$$\begin{array}{rcll} e & \in & \textsc{Exp} & \text{expressions} \\ v & \in & \textsc{Val} \subset \textsc{Exp} & \text{values} \\ ev & \in & \textsc{Event} \subset \textsc{Val} & \text{event values} \end{array}$$

Values are the irreducible terms in the dynamic semantics. The terms of λ_{cv} are defined by the grammar in figure 5. Note that the syntactic class of the term $(v_1 . v_2)$ is either EXP or

$$\begin{array}{rcll}
e & ::= & v & \text{value} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & (e_1 . e_2) & \text{pair} \\
 & | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \text{let} \\
 & | & \texttt{chan}\ x\ \texttt{in}\ e & \text{channel creation} \\
 & | & \texttt{spawn}\ e & \text{process creation} \\
 & | & \texttt{sync}\ e & \text{synchronization} \\
 \\
v & ::= & b & \text{constant} \\
 & | & x & \text{variable} \\
 & | & (v_1 . v_2) & \text{pair value} \\
 & | & (\texttt{fn}\ x\ \texttt{=>}\ e) & \lambda\text{-abstraction} \\
 & | & \kappa & \text{channels} \\
 & | & ev & \text{events} \\
 & | & (\mathbf{G}\, e) & \text{guarded event function} \\
 \\
ev & ::= & \Lambda & \text{never} \\
 & | & \kappa!v & \text{channel output} \\
 & | & \kappa? & \text{channel input} \\
 & | & (ev \Rightarrow v) & \text{wrapper} \\
 & | & (ev_1 \oplus ev_2) & \text{choice} \\
 & | & (ev \mid v) & \text{abort wrapper}
\end{array}$$

Figure 5: The λ_{cv} grammar

VAL; this ambiguity is resolved in favor of VAL. The value Λ is a base event value that is never matched (equivalent to `choose[]` in **CML**). The v subterms in the terms $(ev \Rightarrow v)$ and $(ev \mid v)$ are expected to be function values (the type system in Section 5 insures this). We write $\textsc{Val}^{\circ}$ for the value terms without free variables; note, however, that closed values may contain free channel names. We write FCV(e) for the free channel names of a term e. There is no term for sequencing, but we write "$(e_1;\ e_2)$" for "`snd` $(e_1 . e_2)$," which, since the language is call-by-value, has the desired semantics. We adopt Barendregt's variable convention (p. 26 of [Bar84]) to avoid problems with capture.

Channel names and event values are not part of the concrete syntax of the language; rather, they appear as the intermediate results of evaluation. A *program* is a closed term that does not contain any subterms of the form $(\mathbf{G}\, e)$, or any subterms in EVENT or CH (in other words, programs do not have intermediate subterms).

3.2 Sequential evaluation

The meaning of the function constants is given by the partial function

$$\delta : \text{FCONST} \times \text{VAL}^{\circ} \rightharpoonup \text{VAL}^{\circ}$$

Since a closed value $v \in \text{VAL}^{\circ}$ can have free channel names, we require that for all $b \in \text{FCONST}$

$$\text{FCV}(\delta(b, v)) \subseteq \text{FCV}(v)$$

For the standard built-in function constants, the meaning of δ is the expected one. For example:

$$\begin{array}{lcl}
\delta(\texttt{+}, \texttt{(0.1)}) & = & \texttt{1} \\
\delta(\texttt{+}, \texttt{(1.1)}) & = & \texttt{2} \\
\delta(\texttt{fst}, \texttt{(}v_1\,\texttt{.}\,v_2\texttt{)}) & = & v_1 \\
\delta(\texttt{snd}, \texttt{(}v_1\,\texttt{.}\,v_2\texttt{)}) & = & v_2
\end{array}$$

With the exception of **guard**, the event value constructors are mapped directly onto their corresponding terms in EVENT:

$$\begin{array}{lcl}
\delta(\texttt{never}, \texttt{()}) & = & \Lambda \\
\delta(\texttt{transmit}, \texttt{(}\kappa\,\texttt{.}\,v\texttt{)}) & = & \kappa!v \\
\delta(\texttt{receive}, \kappa) & = & \kappa? \\
\delta(\texttt{wrap}, \texttt{(}ev\,\texttt{.}\,v\texttt{)}) & = & (ev \Rightarrow v) \\
\delta(\texttt{choose}, \texttt{(}ev_1\,\texttt{.}\,ev_2\texttt{)}) & = & (ev_1 \oplus ev_2) \\
\delta(\texttt{wrapAbort}, \texttt{(}ev\,\texttt{.}\,v\texttt{)}) & = & (ev \mid v)
\end{array}$$

The δ-rules for **guard** reflect its use as a delay operator; when another event constructor is applied to a guarded event value, then the guard operator $(\mathbf{G})$ is pulled out to delay the event construction. As is described below, the **sync** operator forces the evaluation of the guarded expression. In **Algol 60** terminology, $(\mathbf{G}\, e)$ is a *thunk*.

$$\begin{array}{lcl}
\delta(\texttt{guard}, v) & = & (\mathbf{G}\,\texttt{(}v\;\texttt{()}\texttt{)}) \\
\delta(\texttt{wrap}, \texttt{(}(\mathbf{G}\, e)\,\texttt{.}\,v\texttt{)}) & = & (\mathbf{G}\,\texttt{(wrap (}e\,\texttt{.}\,v\texttt{))}) \\
\delta(\texttt{choose}, \texttt{(}(\mathbf{G}\, e_1)\,\texttt{.}\,(\mathbf{G}\, e_2)\texttt{)}) & = & (\mathbf{G}\,\texttt{(choose (}e_1\,\texttt{.}\,e_2\texttt{))}) \\
\delta(\texttt{choose}, \texttt{(}(\mathbf{G}\, e_1)\,\texttt{.}\,ev_2\texttt{)}) & = & (\mathbf{G}\,\texttt{(choose (}e_1\,\texttt{.}\,ev_2\texttt{))}) \\
\delta(\texttt{choose}, \texttt{(}ev_1\,\texttt{.}\,(\mathbf{G}\, e_2)\texttt{)}) & = & (\mathbf{G}\,\texttt{(choose (}ev_1\,\texttt{.}\,e_2\texttt{))}) \\
\delta(\texttt{wrapAbort}, \texttt{(}(\mathbf{G}\, e)\,\texttt{.}\,v\texttt{)}) & = & (\mathbf{G}\,\texttt{(wrapAbort (}e\,\texttt{.}\,v\texttt{))})
\end{array}$$

Following [WF91], the semantics is stated using *evaluation contexts*. An evaluation context is a single hole context; the following grammar defines the evaluation contexts for

λ_{cv}:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\ e \mid v\ E \mid (E\,.\,e) \mid (v\,.\,E) \mid \\ & & \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{spawn}\ E \mid \texttt{sync}\ E \end{array}$$

These contexts force a call-by-value evaluation order. A term can be uniquely partitioned into an evaluation context and redex; we write $E[e]$ for this partition, where e is the redex. We also use $E[e]$ for the term formed by inserting e, which might not be a redex, into the hole of E. The sequential evaluation relation ($\longmapsto$) is defined in terms of the contexts.

$$\begin{array}{lcl} E[b\ v] & \longmapsto & E[\delta(b, v)] \\ E[(\texttt{fn}\ x\ \texttt{=>}\ e)\ v] & \longmapsto & E[e[x \mapsto v]] \\ E[\texttt{let}\ x = v\ \texttt{in}\ e] & \longmapsto & E[e[x \mapsto v]] \\ E[\texttt{sync}\ (\mathbf{G}\ e)] & \longmapsto & E[\texttt{sync}\ e] \end{array}$$

Note that the rule for sync forces the expression delayed by guard. We write $\longmapsto^*$ for the reflexive transitive closure of $\longmapsto$. As an example of sequential evaluation, consider the following, where $[\cdots]$ is used to mark the context/redex boundary.

$$\begin{array}{ll} \multicolumn{2}{l}{(\texttt{fn}\ x\ \texttt{=>}\ 1)\ ([(\texttt{fn}\ x\ \texttt{=>}\ x\ 10)\ (\texttt{fn}\ y\ \texttt{=>}\ y)])} \\ \longmapsto & (\texttt{fn}\ x\ \texttt{=>}\ 1)\ ([(\texttt{fn}\ y\ \texttt{=>}\ y)\ 10]) \\ \longmapsto & [(\texttt{fn}\ x\ \texttt{=>}\ 1)\ 10] \\ \longmapsto & [1] \end{array}$$

The derived syntax for sequencing leads to the following derived rule for evaluating a sequence of terms:

$$E[(v_1;\ v_2)] \longmapsto E[v_2]$$

Note that both sides of the ";" are evaluated to values before this rule is applicable.

3.3 Event value matching

The core concept in the dynamic semantics of λ_{cv} is the notion of *event matching*, which captures the semantics of rendezvous and communication. Informally, if two processes synchronize on matching events, then they can exchange values and continue evaluation. Before making this more formal, we need an auxiliary definition

Definition 3.1 The *abort action* of an event value, ev is an expression, which, when evaluated, spawns the abort wrappers of ev. The map

$$\text{AbortAct} : \text{EVENT} \to \text{EXP}$$

is defined inductively on the structure of event values:

$$\begin{array}{lcl} \text{AbortAct}(\kappa?) & = & () \\ \text{AbortAct}(\kappa!v) & = & () \\ \text{AbortAct}(ev \Rightarrow e) & = & \text{AbortAct}(ev) \\ \text{AbortAct}(ev_1 \oplus ev_2) & = & (\text{AbortAct}(ev_1)\,;\ \text{AbortAct}(ev_2)) \\ \text{AbortAct}(ev \mid v) & = & (\text{AbortAct}(ev)\,;\ \texttt{spawn}\ v) \end{array}$$

With this we can formally define the matching of event values:

Definition 3.2 (Event matching) We define a family of relations $\overset{\kappa}{\subset}$, indexed by $\textsc{Ch}$, on $(\textsc{Event} \times \textsc{Exp})$. For $\kappa \in \textsc{Ch}$, we define

$$ev_1 \overset{\kappa}{\subset} ev_2 \text{ with } (e_1,\, e_2)$$

(pronounced as "ev_1 matches ev_2 on channel κ with respective results e_1 and e_2") as the smallest relation satisfying the following rules:

(1) $\kappa!v \overset{\kappa}{\subset} \kappa?$ with $(\texttt{()},\, v)$

(2) If $ev_1 \overset{\kappa}{\subset} ev_2$ with $(e_1,\, e_2)$, then the following also hold:

- (2a) $ev_2 \overset{\kappa}{\subset} ev_1$ with $(e_2,\, e_1)$
- (2b) $ev_1 \overset{\kappa}{\subset} (ev_2 \Rightarrow v)$ with $(e_1,\, v\ e_2)$
- (2c) $ev_1 \overset{\kappa}{\subset} (ev_2 \oplus ev_3)$ with $(e_1,\, (\text{AbortAct}(ev_3);\, e_2))$
- (2d) $ev_1 \overset{\kappa}{\subset} (ev_3 \oplus ev_2)$ with $(e_1,\, (\text{AbortAct}(ev_3);\, e_2))$
- (2e) $ev_1 \overset{\kappa}{\subset} (ev_2 \mid v)$ with $(e_1,\, e_2)$

We write $ev \overset{\kappa}{\subset} ev'$ when we do not care about the results.

Informally, if two processes attempt to synchronize on matching event values, then the applications of `sync` can be replaced with the respective results. This is made more precise in the next section, where the concurrent evaluation relation is defined.

It is worth noting that even if one of the wrappers of an event value is non-terminating, the necessary abort actions for that event will be executed (assuming fair evaluation). This property is important because a common **CML** idiom is to have tail-recursive calls in wrappers. For example, the following program merges two streams of input:

```
fun loop () = sync (choose [
        wrap (input_line instrm1, fn s => (print s; loop())),
        wrap (input_line instrm2, fn s => (print s; loop()))
      ])
```

Since the `input_line` operations are implemented using `wrapAbort` (see [Rep91]) to abort the unwanted input request, it is important that the abort action be spawned, even though the wrapper never returns. Section 3.6 discusses this property in a more rigorous fashion.

3.4 Concurrent evaluation

The concurrent evaluation relation is an extension of the sequential evaluation relation ($\longmapsto$) to finite sets of processes, with the addition of rules for process creation, termination and communication. This is similar to the style of a "Chemical Abstract Machine" ([BB90]), except that there are no "cooling" or "heating" transitions (configurations can be thought of as perpetually "hot" solutions). To avoid the complications of multisets, *process identifiers* are used to uniquely tag each process. The process identifier and term comprise a *process*; a finite set of processes is called a *configuration*:

$$\begin{array}{rcll} \pi & \in & \textsc{ProcId} & \text{process identifiers} \\ p = \langle \pi;\, e \rangle & \in & \textsc{Proc} = (\textsc{ProcId} \times \textsc{Exp}) & \text{process states} \\ \mathcal{P} & \in & \text{Fin}(\textsc{Proc}) & \text{configurations} \end{array}$$

We often write a process as $\langle \pi;\, E[e] \rangle$, where the evaluation context serves the role of a program counter, marking the current state of evaluation.

Definition 3.3 A configuration $\mathcal{P}$ is *well-formed* if, for each $\langle \pi;\, e \rangle \in \mathcal{P}$, $\text{FV}(e) = \emptyset$, and there is no e' such that $\langle \pi;\, e' \rangle \in \mathcal{P}$.

It is occasionally useful to view well-formed configurations as finite maps from ProcId to Exp, so we write $\text{dom}(\mathcal{P})$ for $\{\pi \mid \langle \pi;\, e \rangle \in \mathcal{P}\}$ and $\mathcal{P}(\pi)$ for e if $\langle \pi;\, e \rangle \in \mathcal{P}$. An *initial configuration* is a singleton process set $\{\langle \pi;\, e \rangle\}$, where e is a program. An initial configuration is trivially a well-formed one.

The *concurrent evaluation relation* is defined by rules of the form

$$\mathcal{P} \Longrightarrow \mathcal{P}'$$

where $\mathcal{P}$ and $\mathcal{P}'$ are configurations. Most of these rules have enabling side conditions, so they are presented as inference rules. Each concurrent evaluation step affects one or two processes, called the *selected* processes. The left-associative operator $+$, is defined as $\mathcal{P}{+}p \equiv_{def} \mathcal{P} \cup \{p\}$, with $\pi \notin \text{dom}(\mathcal{P})$ for $p = \langle \pi;\, e \rangle$. This notation is used to distinguish the selected processes in the rules below. The first rule extends the sequential evaluation relation ($\longmapsto$) to configurations:

$$\frac{e \longmapsto e'}{\mathcal{P}{+}\langle \pi;\, e \rangle \Longrightarrow \mathcal{P}{+}\langle \pi;\, e' \rangle}$$

The creation of channels requires picking a new channel name and substituting for the variable bound to it:

$$\frac{\kappa \notin \text{FCV}(\mathcal{P}{+}\langle \pi;\, E[\texttt{chan}\ x\ \texttt{in}\ e] \rangle)}{\mathcal{P}{+}\langle \pi;\, E[\texttt{chan}\ x\ \texttt{in}\ e] \rangle \Longrightarrow \mathcal{P}{+}\langle \pi;\, E[e[x \mapsto \kappa]] \rangle}$$

Process creation requires picking a new process identifier:

$$\frac{\pi' \notin \text{dom}(\mathcal{P}){+}\pi}{\mathcal{P}{+}\langle \pi;\, E[\texttt{spawn}\ v] \rangle \Longrightarrow \mathcal{P}{+}\langle \pi;\, E[()] \rangle{+}\langle \pi';\, v\ () \rangle}$$

The most interesting rule describes communication and synchronization. If two processes are attempting synchronization on matching events, then they may rendezvous, exchange a message and continue evaluation:

$$\frac{ev \overset{\kappa}{\frown} ev' \text{ with } (e,\, e')}{\mathcal{P}{+}\langle\pi;\, E[\texttt{sync}\ ev]\rangle{+}\langle\pi';\, E'[\texttt{sync}\ ev']\rangle \Longrightarrow \mathcal{P}{+}\langle\pi;\, E[e]\rangle{+}\langle\pi';\, E'[e']\rangle}$$

We say that κ is *used* in this transition. Combined, these rules define concurrent evaluation. We write $\Longrightarrow^*$ for the reflexive transitive closure of $\Longrightarrow$.

3.5 An example

The following evaluation illustrates the concurrent evaluation relation:

$$\begin{array}{l}
\{\langle\pi_0;\, [\texttt{chan}\ k\ \texttt{in (spawn (fn}\ x\ \texttt{=> sync (transmit (}k\texttt{.5))); sync (receive}\ k\texttt{))}]\rangle\} \\
\quad\Longrightarrow\ \{\langle\pi_0;\, \texttt{(}[\texttt{spawn (fn}\ x\ \texttt{=> sync (transmit (}\kappa_0\texttt{.5)))}]\texttt{; sync (receive}\ \kappa_0\texttt{))}\rangle\} \\
\quad\Longrightarrow\ \{\langle\pi_0;\, \texttt{((); sync (}[\texttt{receive}\ \kappa_0]\texttt{))}\rangle,\, \langle\pi_1;\, [\texttt{(fn}\ x\ \texttt{=> sync (transmit (}\kappa_0\texttt{.5))) ()}]\rangle\} \\
\quad\Longrightarrow\ \{\langle\pi_0;\, \texttt{((); sync (}[\texttt{receive}\ \kappa_0]\texttt{))}\rangle,\, \langle\pi_1;\, \texttt{sync (}[\texttt{transmit (}\kappa_0\texttt{.5)}]\texttt{)}\rangle\} \\
\quad\Longrightarrow^*\ \{\langle\pi_0;\, \texttt{((); }[\texttt{sync (}\kappa_0\texttt{?)}]\texttt{)}\rangle,\, \langle\pi_1;\, [\texttt{sync (}\kappa_0\texttt{!5)}]\rangle\} \\
\quad\Longrightarrow\ \{\langle\pi_0;\, [\texttt{((); 5)}]\rangle,\, \langle\pi_1;\, [\texttt{()}]\rangle\} \\
\quad\Longrightarrow\ \{\langle\pi_0;\, [5]\rangle,\, \langle\pi_1;\, [\texttt{()}]\rangle\}
\end{array}$$

Note that this is only one of several possible evaluation sequences, although, in this example, all evaluation sequences produce the same result.

3.6 Some properties of evaluation

There are a few properties of concurrent evaluation that are worth noting.

Lemma 3.1 If $\mathcal{P}$ is a well-formed configuration and $\mathcal{P} \Longrightarrow \mathcal{P}'$, then $\mathcal{P}'$ is well-formed. Likewise for $\Longrightarrow^*$.

Proof. The proof is by examination of the rules for $\Longrightarrow$. The rule for `spawn` is the only one that introduces process identifiers, and it only introduces new ones. Examination of the rules for $\longmapsto$, shows that it preserves closed terms. Thus $\mathcal{P}'$ must be well-formed. This is extended to $\Longrightarrow^*$ by induction on the length of evaluation sequences.

Since initial configurations are well-formed, all of the interesting evaluations involve only well-formed configurations. Preservation of well-formedness is also shown in [BMT92] for their semantics. There are other properties of evaluation in [BMT92], which hold for $\Longrightarrow$.

Lemma 3.2 If $\mathcal{P}$ is well-formed and $\mathcal{P} \Longrightarrow \mathcal{P}'$, then the following hold:

- $\mathrm{dom}(\mathcal{P}) \subseteq \mathrm{dom}(\mathcal{P}')$
- If $\pi \notin \mathrm{dom}(\mathcal{P}')$, and $\mathrm{FV}(e) = \emptyset$, then $\mathcal{P} + \langle\pi;\ e\rangle \Longrightarrow \mathcal{P}' + \langle\pi;\ e\rangle$, with $\mathcal{P}' + \langle\pi;\ e\rangle$ being well-formed.
- If $\mathcal{P} = \mathcal{P}''+p$ and p is not selected in the transition to $\mathcal{P}'$, then there exists a $\mathcal{P}'''$, such that $\mathcal{P}' = \mathcal{P}'''+p$.

As discussed on page 10, an important property of the semantics is the independence of abort operations from the wrappers of the chosen event. The following lemma characterizes the structure of the terms resulting from synchronization on events:

Lemma 3.3 Let $ev_1 \overset{\kappa}{\frown} ev_2$ with $(e_1,\ e_2)$. The terms e_1 and e_2 have the form of t specified by the following grammar:

$$
\begin{array}{rcl}
t & ::= & v \\
  & | & (v\ t) \\
  & | & (a;\ t) \\
  & & \\
a & ::= & () \\
  & | & (a_1;\ a_2) \\
  & | & (a;\ \texttt{spawn}\ v)
\end{array}
$$

In this grammar, the a subterms are contributed by AbortAct.
Proof. The lemma follows from definitions 3.1 and 3.2.

This structure, coupled with the call-by-value order of sequential evaluation, insures that the abort actions are spawned prior to application of any wrapper functions.

3.7 Fairness

The semantics presented above admits unfair evaluation sequences, and thus is not adequate as an specification of **CML** implementations. It is necessary to distinguish the acceptable evaluation sequences. Informally, we require that ready processes make progress and that communication on a single channel is fair (see [Kwi89] for a survey of fairness notions). To formalize these notions of fairness, we need a few definitions.

Definition 3.4 A process $p = \langle\pi;\ e\rangle$ is in one of three *states*, depending on the form of e:

- if $e = [v]$, then p is a *zombie*,
- if $e = E[\texttt{sync}\ ev]$, then p is *blocked* on ev,
- otherwise, p is *ready*.

We define the set of ready processes in $\mathcal{P}$ by

$$\text{RDY}(\mathcal{P}) = \{\pi \mid \langle \pi;\, \mathcal{P}(\pi)\rangle \text{ is ready}\}$$

A configuration $\mathcal{P}$ is *terminal* if every $p \in \mathcal{P}$ is a zombie or is blocked (i.e., deadlock)

Definition 3.5 A process $p \in \mathcal{P}$ is *enabled* if $p \in \text{RDY}(\mathcal{P})$, or if it is blocked on ev and there is a $p' \in \mathcal{P}$ blocked on ev', with $ev \overset{\kappa}{\frown} ev'$.

We need a similar notion for channels.

Definition 3.6 A channel κ is *enabled* in a process configuration $\mathcal{P}$ if there are processes $\langle \pi;\, E[\texttt{sync}\ ev]\rangle, \langle \pi';\, E'[\texttt{sync}\ ev']\rangle \in \mathcal{P}$, with $ev \overset{\kappa}{\frown} ev'$ and $\pi \neq \pi'$.

The acceptable evaluation sequences of a program are defined in terms of fairness restrictions.

Definition 3.7 An evaluation sequence T is *acceptable* if it is finite and ends in a terminal configuration, or if T is infinite and satisfies the following fairness constraints:

(1) Any process that is enabled infinitely often is selected infinitely often.

(2) Any channel that is enabled infinitely often is used infinitely often.

In the taxonomy of [Kwi89], the first restriction is *strong process fairness* and the second is *strong event fairness*. Implementations are required to prohibit the possibility of unacceptable evaluation sequences, which, in practice, requires that implementations satisfy some stronger property on finite traces. As an example, consider the following property.

Definition 3.8 A finite evaluation sequence $\mathcal{P}_0 \Longrightarrow \mathcal{P}_1 \Longrightarrow \cdots \Longrightarrow \mathcal{P}_n$ is *k–bounded fair* (for k a fixed positive integer), if every intermediate configuration $\mathcal{P}_i$, satisfies one of the following (where $m = i + k\,|\text{RDY}(\mathcal{P}_i)|$):

- $m > n$, or
- for every $\pi \in \text{RDY}(\mathcal{P}_i)$, π is selected at least once in the evaluation subsequence $\mathcal{P}_i \Longrightarrow \cdots \Longrightarrow \mathcal{P}_m$.

An infinite evaluation sequence is *k–bounded fair*, if every finite prefix of it is k–bounded fair.

An infinite evaluation sequence that is k–bounded fair obviously satisfies restriction (1) (but not necessarily (2)). This restriction is realizable using fairly standard implementation techniques. For example, an implementation that uses fair pre-emptive scheduling[2] and FIFO queues for the process ready queue and for channel waiting queues will produce only k–bounded fair sequences, where k is determined by the length of the time-slice and speed of the processor.

4 Extending λ_{cv} to full CML

The language λ_{cv} lacks a number of features found in **CML**; in this section we examine how the semantics of λ_{cv} can be extended to model some of these features.

4.1 Syntactic sugar

There are a number of missing **CML** features that can be viewed as syntactic sugar for λ_{cv} terms. **CML** uses the function `channel` to allocate new channels and provides the more traditional synchronous operations `send` and `accept`. These functions can be used by embedding a λ_{cv} term e in the following context:

```
let channel = fn x => chan k in k in
  let send = fn x => sync (transmit x) in
    let accept = fn x => sync (receive x) in
      [e]
```

The `choose` and `select` functions of **CML** work on lists of events (instead of just pairs). Although λ_{cv} does not have **SML**'s datatypes, event lists can be implemented using the following translation:

$$\begin{aligned} [\![\texttt{nil}]\!] &= \texttt{never()} \\ [\![ev\texttt{::}r]\!] &= \texttt{choose}\ (ev\,.\,[\![r]\!]) \end{aligned}$$

4.2 Recursion

Dynamic process and channel creation is powerful enough to implement the call-by-value $\mathbf{Y}_v$ combinator, which has the following evaluation rule:

$$E[\mathbf{Y}_v\ v] \longmapsto E[v\ (\texttt{fn}\ x\ \texttt{=>}\ (\mathbf{Y}_v\ v)\ x)]$$

The following **CML** code implements $\mathbf{Y}_v$ (adopted from [GMP89]):

[2]By *fair*, I mean that a thread is guaranteed some progress before being pre-empted.

```
val Yv  = fn f => let
      val a = channel()
      val g = fn v => let val h' = accept a
                 in
                   spawn (fn () => send(a, h'));
                   f h' v
                 end
      in
        spawn (fn () => send(a, g));
        let val h = accept a
        in
          spawn (fn () => send(a, h));
          f h
        end
      end
```

This code is somewhat mysterious, but what it actually does is fairly simple. The channel a is used to cache the function g for the next iteration of f; each time g (renamed h) is read from a, a new thread is spawned to send another copy. For **CML**, which is statically typed (see section 5), this definition implements recursion at all imperative types. As an alternative, we could add the $\mathbf{Y}_v$ combinator as a built-in term constructor (as is done in [WF91]), which would provide recursion at all types.

4.3 References

It is well known that processes and channels can be used to mimic updatable references. The following **CML** code illustrates this:

```
datatype 'a ref = REF of ('a chan * 'a chan)

fun mkRef initX = let
      val inCh = channel() and outCh = channel()
      fun cell x = sync (choose [
              wrap (transmit (outCh, x), fn () => cell x),
              wrap (receive inCh, fn newX => cell newX)
            ])
      in
        spawn (fn () => cell initX);
        REF(inCh, outCh)
      end

fun assign (REF(inCh, _), x) = send (inCh, x)

fun deref (REF(_, outCh)) = accept outCh
```

One can define a formal translation from programs with references to programs using this scheme. This is done in [BMT92], and the translation is shown to be semantics preserving.

The implementation of $\mathbf{Y}_v$ described in section 4.2 is similar to the imperative $\mathbf{Y}$–combinator ($\mathbf{Y}_!$) defined by Felleisen[Fel87]. This suggests the following implementation of references, which uses channels to represent references directly and does not require explicit recursion:

```
fun mkRef initX = let val ch = channel()
      in
        spawn (fn () => send (ch, initX));
        ch
      end

fun assign (ch, x) = (accept ch; spawn (fn () => send (ch, x)))

fun deref ch = let val x = accept ch
      in
        spawn (fn () => send (ch, x));
        x
      end
```

4.4 Exceptions

CML inherits **SML**'s exception mechanism and provides an event combinator for handling exceptions that are raised during evaluation of an events wrappers (`wrapHandler`). Wright and Felleisen provide a semantics of **SML**'s exception mechanism in [WF91], but applying this technique to λ_{cv} will require some care. The problem is that the soundness of their semantics relies on limiting the scope of exception identifiers to within the scope of their binding site (the rewrite rules allow the binding sites to migrate up to the top of the term thus expanding the scope of the binding). Since processes can include exceptions in messages, a different approach is needed. The best approach seems to be to bind exception identifiers in an implicit global environment (as is done with channel names). New syntax for expressions is required:

$$
\begin{array}{rcll}
e & ::= & e\,\mathtt{handle}\,v & \text{exception handler} \\
 & | & \mathtt{raise}\,x\;e & \text{raise exception} \\
 & | & \ldots & \\
ev & ::= & (ev\,\mathbf{H}\,v) & \text{handler wrapper} \\
 & | & \ldots &
\end{array}
$$

The event constructor `wrapHandler` has the following δ–rule:

$$\delta(\mathtt{wrapHandler}, (ev\,.\,v)) \quad = \quad (ev\,\mathbf{H}\,v)$$

A new clause is needed in Definition 3.2:

(2f) $ev_1 \overset{\kappa}{\subset} (ev_2\,\mathbf{H}\,v)$ with $(e_1,\, e_2\,\mathtt{handle}\,v)$

There are also new evaluation contexts and rules; see [WF91] for details.

4.5 Process join

CML provides the event constructor `threadWait` that allows a thread to synchronize on the termination of another thread. One approach to describing this is to add a set of distinguished channel names, $\{\kappa_\pi \mid \pi \in \textsc{ProcId}\}$, to represent processes. The rule for process creation wraps the body of a process π with code to repeatedly send `()` on the channel κ_π:

$$\frac{\pi' \notin \mathrm{dom}(\mathcal{P})}{\mathcal{P}+\langle\pi;\, E[\texttt{spawn}\, v]\rangle \Longrightarrow \mathcal{P}+\langle\pi;\, E[\kappa_{\pi'}]\rangle+\langle\pi';\, \mathrm{Fork}(\pi', v)\rangle}$$

where

$$\mathrm{Fork}(\pi, v) = \texttt{(}v\ \texttt{(); }\mathbf{Y}_v\ \texttt{(fn}\ f\ \texttt{=> (send (}\kappa_\pi\texttt{.()); }f\ \texttt{())) ())}$$

Then `(threadWait` κ_π`)` is implemented by `(accept` κ_π`)`. While this is a reasonable implementation technique, it has the problem that it becomes much harder to distinguish the zombie processes in the semantics.

A better approach is to directly support `threadWait` and, while we are at it, also support the base event constructor `always`. To do so we add ProcId to the domain of values and add two new event value terms:

$$\begin{array}{rcl} v & ::= & \pi \mid \cdots \\ ev & ::= & (\mathbf{W}\,\pi) \mid \mathbf{A} \mid \cdots \end{array}$$

The implementation of the `always` function is defined by the following δ–rule:

$$\delta(\texttt{always}, v) = (\mathbf{A} \Rightarrow \texttt{(fn}\ x\ \texttt{=>}\ v\texttt{)})$$

The rule for `spawn` is slightly changed to return the identifier of the new process:

$$\frac{\pi' \notin \mathrm{dom}(\mathcal{P})}{\mathcal{P}+\langle\pi;\, E[\texttt{spawn}\, v]\rangle \Longrightarrow \mathcal{P}+\langle\pi;\, E[\pi']\rangle+\langle\pi';\, v\ \texttt{()}\rangle}$$

We then need to introduce a notion of matching an event by a set of processes:

Definition 4.1 We define a ternary relation

$$ev \supset \mathcal{P} \text{ with } e$$

(pronounced as "*ev* is *matched* by $\mathcal{P}$ with result e") as the smallest relation satisfying the following rules:

(1) $(\mathbf{W}\,\pi) \supset \mathcal{P}+\langle\pi;\, [v]\rangle$ with `()`

(2) $\mathbf{A} \supset \mathcal{P}$ with `()`

(3) If $ev \supset \mathcal{P}$ with e, then the following also hold:

(3a) $(ev \Rightarrow v) \supset \mathcal{P}$ with $(v\ e)$

(3b) $(ev \oplus ev') \supset \mathcal{P}$ with $(\text{AbortAct}(ev')\,;\ e)$

(3c) $(ev' \oplus ev) \supset \mathcal{P}$ with $(\text{AbortAct}(ev')\,;\ e)$

(3d) $(ev \mid v) \supset \mathcal{P}$ with e

And there is an additional rule for `sync`:

$$\frac{ev \supset \mathcal{P} \text{ with } e}{\mathcal{P}+\langle\pi;\ \texttt{sync}\ ev\rangle \Longrightarrow \mathcal{P}+\langle\pi;\ e\rangle}$$

4.6 Polling

Both **PML** and **CML** support *polling* of events[3]. The `poll` operation is now a non-blocking form of `sync` with the type:

```
val poll : 'a event -> 'a option
```

Applying `poll` to an event value returns `NONE` if `sync` would have blocked, and `SOME` wrapped around the synchronization result otherwise.

It is fairly straightforward to add `poll` to λ_{cv}. Since we do not have the `option` datatype in λ_{cv}, we have the `poll` operation take two arguments: an event value and a pair of functions; the first is evaluated if the event is matched and the second is evaluated otherwise, so `poll` has the following type:

```
val poll : ('a event * (('a -> 'b) * (unit -> 'b)) -> 'b
```

We extend the syntax of expression terms and the definition of evaluation contexts by

$$\begin{array}{rcl} e & ::= & \texttt{poll}\ e \mid \cdots \\ E & ::= & \texttt{poll}\ E \mid \cdots \end{array}$$

We also need to formalize the notion that an event is matched by some other offered event in a configuration. The following two definitions do this.

Definition 4.2 We say that ev is *offered* by π in a configuration $\mathcal{P}$, if $\mathcal{P}(\pi)$ is of the form $E[\texttt{sync}\ ev]$ or $E[\texttt{poll(}ev\,.v\texttt{)}]$. We define the set of offered events of $\mathcal{P}$ by

$$\text{Offered}(\mathcal{P}) = \{(\pi, ev) \mid \pi \text{ offers } ev \text{ in } \mathcal{P}\}$$

[3] In **PML** polling was supported via an awkward priority scheme on base event values. While in [Rep91] polling was presented as a event-value constructor:

```
val poll : 'a event -> 'a option event
```

In practice, however, the full generality of this mechanism was not used, so **CML** now uses the simpler mechanism described in this paper.

Definition 4.3 We say that an event ev is *matched* in a set of processes $\mathcal{P}$, written $ev \overset{\kappa}{\supset} \mathcal{P}$, if there exists $(\pi', ev') \in \text{Offered}(\mathcal{P})$ such that $ev \overset{\kappa}{\supset} ev'$.

And, we need three additional concurrent evaluation rules. The first two handle the transition in which the event is matched by some other process, the third handles the transition for when sync would have blocked:

$$\frac{ev \overset{\kappa}{\supset} ev' \text{ with } (e,\, e')}{\mathcal{P}+\langle\pi;\, E[\texttt{poll(}ev\texttt{.(}v_1\texttt{.}v_2\texttt{))}]\rangle+\langle\pi';\, E'[\texttt{sync } ev']\rangle \Longrightarrow \mathcal{P}+\langle\pi;\, E[v_1\ e]\rangle+\langle\pi';\, E'[e']\rangle}$$

$$\frac{ev \overset{\kappa}{\supset} ev' \text{ with } (e,\, e')}{\begin{array}{c}\mathcal{P}+\langle\pi;\, E[\texttt{poll(}ev\texttt{.(}v_1\texttt{.}v_2\texttt{))}]\rangle+\langle\pi';\, E'[\texttt{poll(}ev'\texttt{.(}v_1'\texttt{.}v_2'\texttt{))}]\rangle \\ \Longrightarrow \mathcal{P}+\langle\pi;\, E[v_1\ e]\rangle+\langle\pi';\, E'[v_1'\ e']\rangle\end{array}}$$

$$\frac{ev \not\supset \mathcal{P}}{\mathcal{P}+\langle\pi;\, E[\texttt{poll(}ev\texttt{.(}v_1\texttt{.}v_2\texttt{))}]\rangle \Longrightarrow \mathcal{P}+\langle\pi;\, E[v_2\ \texttt{()}]\rangle}$$

To make these rules sensible, we need an additional fairness constraint.

If $p = \langle\pi;\, E[\texttt{poll(}ev\texttt{.}v\texttt{)}]\rangle$, then a transition $\mathcal{P}+p \Longrightarrow \mathcal{P}'+p$, is acceptable if:

- $ev \supset \mathcal{P}$ and $ev \supset \mathcal{P}'$, or
- $ev \not\supset \mathcal{P}$ and $ev \not\supset \mathcal{P}'$.

This constraint captures the notion that poll is non-blocking by forcing the polling operation to complete before the state of the polled event can change.

5 Typing λ_{cv}

The fact that references can be coded up using channels makes apparent the fact that polymorphic channels incur the same typing problems as polymorphic references (see [Tof90] for a good description of these problems). It is possible, however, to soundly type λ_{cv} programs using the imperative type scheme of **SML**[MTH90, Tof90]. In this section, I present a type system for λ_{cv}programs (but not intermediate terms). I claim, without proof, that this system is sound with respect to the dynamic semantics given in section 3. The sound typing of λ_{cv} is discussed in greater detail (including a proof of the soundness of the type system) in [Rep92].

The presentation uses standard notation (e.g., see [Tof90] or [WF91]). We start with type constants (int, bool, etc.); we use ι to range over type constants. Type variables are partitioned into two sets: *imperative type variables* (ImpTyVar) and *applicative type* *variables* (AppTyVar); we use α to range over both kinds of type variables. The set of *types*, $\tau \in$ Ty, and *type schemes*, $\sigma \in$ TyScheme, are defined by

$$
\begin{array}{rcll}
\tau & ::= & \iota & \text{type constants} \\
 & | & \alpha & \text{type variables} \\
 & | & (\tau_1 \rightarrow \tau_2) & \text{function types} \\
 & | & (\tau_1 \times \tau_2) & \text{pair types} \\
 & | & \tau\ \texttt{chan} & \text{channel types} \\
 & | & \tau\ \texttt{event} & \text{event types}
\end{array}
$$

$$
\begin{array}{rcl}
\sigma & ::= & \tau \\
 & | & \forall\alpha.\sigma
\end{array}
$$

We write $\mathrm{FTV}(\tau)$ for the free type variables of τ. A type is *imperative* if its free type variables are all imperative; we use θ to range over imperative types. We write $\sigma \succ \tau$ if τ is an *instance* of σ. A *type environment* TE is a finite function from variables to type schemas. We write FTV(TE) for the free type variables of a type environment. The *closure* of a type τ with respect to the type environment TE is defined as

$$\mathrm{Clos}_{\mathrm{TE}}(\tau) = \forall\alpha_1 \cdots \alpha_n.\tau$$

where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})$, and the *applicative closure* of τ by

$$\mathrm{AppClos}_{\mathrm{TE}}(\tau) = \forall\alpha_1 \cdots \alpha_n.\tau$$

where $\{\alpha_1, \ldots, \alpha_n\} = (\mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})) \cap \mathrm{AppTyVar}$. To assign types to constants we assume the existence of the function

$$\mathrm{TypeOf} : \mathrm{Const} \longrightarrow \mathrm{TyScheme}$$

In addition to mapping `1` to `int`, etc., the TypeOf function assigns types to the various concurrency related constants; Figure 6 gives these types. Note that we treat the syntactic

$$
\begin{array}{rcl}
\texttt{never} & : & \forall\alpha.(\texttt{unit} \rightarrow \alpha\ \texttt{event}) \\
\texttt{spawn} & : & \forall\alpha.((\texttt{unit} \rightarrow \texttt{unit}) \rightarrow \texttt{unit}) \\
\texttt{sync} & : & \forall\alpha.(\alpha\ \texttt{event} \rightarrow \alpha) \\
\texttt{transmit} & : & \forall\alpha.((\alpha\ \texttt{chan} \times \alpha) \rightarrow \texttt{unit event}) \\
\texttt{receive} & : & \forall\alpha.(\alpha\ \texttt{chan} \rightarrow \alpha\ \texttt{event}) \\
\texttt{guard} & : & \forall\alpha.((\texttt{unit} \rightarrow \alpha\ \texttt{event}) \rightarrow \alpha\ \texttt{event}) \\
\texttt{wrap} & : & \forall\alpha\beta.((\alpha\ \texttt{event} \times (\alpha \rightarrow \beta)) \rightarrow \beta\ \texttt{event}) \\
\texttt{wrapAbort} & : & \forall\alpha.((\alpha\ \texttt{event} \times (\texttt{unit} \rightarrow \texttt{unit})) \rightarrow \alpha\ \texttt{event}) \\
\texttt{choose} & : & \forall\alpha.((\alpha\ \texttt{event} \times \alpha\ \texttt{event}) \rightarrow \alpha\ \texttt{event})
\end{array}
$$

Figure 6: Typing of concurrency constants

forms of sync and spawn as applications of constants in the typing system. The typing judgements are sentences of the form

$$\mathrm{TE} \vdash e : \tau$$

which are read as "e has type τ under the type assumptions of TE." The typing rules are given in Figure 7. A program e is *well-typed* if we can deduce $\{\} \vdash e : \tau$, for some τ, from

$$\frac{\mathrm{TypeOf}(b) \succ \tau}{\mathrm{TE} \vdash b : \tau} \qquad (\boldsymbol{\tau}\textbf{-const})$$

$$\frac{x \in \mathrm{dom}(\mathrm{TE}) \quad \mathrm{TE}(x) \succ \tau}{\mathrm{TE} \vdash x : \tau} \qquad (\boldsymbol{\tau}\textbf{-var})$$

$$\frac{\mathrm{TE} \vdash e_1 : (\tau' \rightarrow \tau) \quad \mathrm{TE} \vdash e_2 : \tau'}{\mathrm{TE} \vdash e_1\ e_2 : \tau} \qquad (\boldsymbol{\tau}\textbf{-app})$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \tau\} \vdash e : \tau'}{\mathrm{TE} \vdash \texttt{(fn } x \texttt{ => } e\texttt{)} : (\tau \rightarrow \tau')} \qquad (\boldsymbol{\tau}\textbf{-abs})$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau_1 \quad \mathrm{TE} \vdash e_2 : \tau_2}{\mathrm{TE} \vdash \texttt{(}e_1\texttt{.}e_2\texttt{)} : (\tau_1 \times \tau_2)} \qquad (\boldsymbol{\tau}\textbf{-pair})$$

$$\frac{\mathrm{TE} \vdash v : \tau' \quad \mathrm{TE} \pm \{x \mapsto \mathrm{C{\scriptstyle LOS}}_{\mathrm{TE}}(\tau')\} \vdash e : \tau}{\mathrm{TE} \vdash \texttt{let } x \texttt{ = } v \texttt{ in } e : \tau} \qquad (\boldsymbol{\tau}\textbf{-let-}v)$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \mathrm{A{\scriptstyle PP}C{\scriptstyle LOS}}_{\mathrm{TE}}(\tau')\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau} \qquad (\boldsymbol{\tau}\textbf{-let-}e)$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \theta\ \mathbf{chan}\} \vdash e : \tau}{\mathrm{TE} \vdash \texttt{chan } x \texttt{ in } e : \tau} \qquad (\boldsymbol{\tau}\textbf{-chan})$$

Figure 7: Typing rules for λ_{cv} programs

these rules.

The typing rules are fairly standard, but it is worth commenting on a few of them. There are two rules for let expressions: **(τ-let-v)**, which applies in the case that the variable is being bound to a *non-expansive* expression; and **(τ-let-e)**, which applies when the variable is being bound to an expression that is *expansive* (i.e., one that might introduce new channels). Expansiveness is a syntactic property; in our notation, the non-expansive expressions are exactly the value terms. The other interesting rule is **(τ-chan)**, which restricts the type of variables bound to the created channel to be imperative. These rules combine to prevent type loopholes, while allowing some polymorphic treatment of channels. For a more detailed discussion of imperative types see [Tof90].

6 Future work

This paper describes the dynamic semantics of a large fraction of the concurrency features of **CML**. There are two obvious directions to continue work: first, to show the soundness of the typing system given in section 5; and, second, to expand the semantics to cover the whole of **CML**. Work is almost complete on the first of these tasks, and will be included in [Rep92]. Another useful direction would be to build a "theory" of **CML** programs to allow reasoning about their correctness.

Acknowledgements

Bill Aitken, Bard Bloom, Dave MacQueen and Chet Murthy have all provided helpful comments on earlier drafts of this paper.

References

[Bar84] Barendregt, H. P. *The Lambda Calculus*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[BB90] Berry, G. and G. Boudol. The chemical abstract machine. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 81–94.

[BMT92] Berry, D., R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992, pp. 119–129.

[Bur88] Burns, A. *Programming in occam 2.* Addison-Wesley, Reading, Mass., 1988.

[Car86] Cardelli, L. Amber. In *Combinators and Functional Programming Languages*, vol. 242 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1986, pp. 21–47.

[Fel87] Felleisen, M. *The Calculi of Lambda-v-CS Conversion in Imperative Higher-order programming languages.* Ph.D. dissertation, Computer Science Department, Indiana University, 1987. Available as Technical Report Nr. 226.

[GMP89] Giacalone, A., P. Mishra, and S. Prasad. Facile: A symemetric integration of concurrent and functional programming. In *TAPSOFT'89 (vol. 2)*, vol. 352 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1989, pp. 184–209.

[Hoa78] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, **21**(8), August 1978, pp. 666–677.

[Kwi89] Kwiatkowska, M. Z. Survey of fairness notions. *Information and Software Technology*, **31**(7), September 1989, pp. 371–386.

[MT91] Milner, R. and M. Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Mass, 1991.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass, 1990.

[PGM90] Prasad, S., A. Giacalone, and P. Mishra. Operational and algebraic semantics for Facile: A symemetric integration of concurrent and functional programming. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, vol. 443 of *Lecture Notes in Computer Science.* Springer-Verlag, July 1990, pp. 765–780.

[Plo75] Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**, 1975, pp. 125–159.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.

[Rep88] Reppy, J. H. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 250–259.

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

[Rep90] Reppy, J. H. *Concurrent programming with events – The Concurrent ML manual.* Department of Computer Science, Cornell University, Ithaca, N.Y., November 1990. (Last revised October 1991).

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep92] Reppy, J. H. *Higher-order concurrency.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1992. Available as Technical Report TR 92-1285.

[Tof90] Tofte, M. Type inference for polymorphic references. *Information and Computation*, **89**, 1990, pp. 1–34.

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