A Concurrent ML Library in Concurrent Haskell

Avik Chaudhuri
University of Maryland, College Park
avik@cs.umd.edu

Abstract

In Concurrent ML, synchronization abstractions can be defined and passed as values, much like functions in ML. This mechanism admits a powerful, modular style of concurrent programming, called *higher-order concurrent programming*. Unfortunately, it is not clear whether this style of programming is possible in languages such as Concurrent Haskell, that support only first-order message passing. Indeed, the implementation of synchronization abstractions in Concurrent ML relies on fairly low-level, language-specific details.

In this paper we show, constructively, that synchronization abstractions can be supported in a language that supports only first-order message passing. Specifically, we implement a library that makes Concurrent ML-style programming possible in Concurrent Haskell. We begin with a core, formal implementation of synchronization abstractions in the π-calculus. Then, we extend this implementation to encode all of Concurrent ML's concurrency primitives (and more!) in Concurrent Haskell.

Our implementation is surprisingly efficient, even without possible optimizations. In several small, informal experiments, our library seems to outperform OCaml's standard library of Concurrent ML-style primitives.

At the heart of our implementation is a new distributed synchronization protocol that we prove correct. Unlike several previous translations of synchronization abstractions in concurrent languages, we remain faithful to the standard semantics for Concurrent ML's concurrency primitives. For example, we retain the symmetry of `choose`, which can express selective communication. As a corollary, we establish that implementing selective communication on distributed machines is no harder than implementing first-order message passing on such machines.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Algorithms, Languages

Keywords Concurrent ML, synchronization abstractions, distributed synchronization protocol, π-calculus, Concurrent Haskell

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

1. Introduction

As famously argued by Reppy (1999), there is a fundamental conflict between selective communication (Hoare 1978) and abstraction in concurrent programs. For example, consider a protocol run between a client and a pair of servers. In this protocol, selective communication may be necessary for liveness—if one of the servers is down, the client should be able to interact with the other. This may require some details of the protocol to be exposed. At the same time, abstraction may be necessary for safety—the client should not be able to interact with a server in an unexpected way. This may in turn require those details to be hidden.

An elegant way of resolving this conflict, proposed by Reppy (1992), is to separate the process of synchronization from the mechanism for describing synchronization protocols. More precisely, Reppy introduces a new type constructor, `event`, to type synchronous operations in much the same way as `->` ("arrow") types functional values. A synchronous operation, or *event*, describes a synchronization protocol whose execution is delayed until it is explicitly *synchronized*. Thus, roughly, an event is analogous to a function abstraction, and event synchronization is analogous to function application.

This abstraction mechanism is the essence of a powerful, modular style of concurrent programming, called *higher-order concurrent programming*. In particular, programmers can describe sophisticated synchronization protocols as `event` values, and compose them modularly. Complex `event` values can be constructed from simpler ones by applying suitable combinators. For instance, selective communication can be expressed as a *choice* among `event` values—and programmer-defined abstractions can be used in such communication without breaking those abstractions (Reppy 1992).

Reppy implements events, as well as a collection of such suitable combinators, in an extension of ML called Concurrent ML (CML) (Reppy 1999). We review these primitives informally in Section 2; their formal semantics can be found in (Reppy 1992). The implementation of these primitives in CML relies on fairly low-level, language-specific details, such as support for continuations and signals (Reppy 1999). In turn, these primitives immediately support higher-order concurrent programming in CML.

Other languages, such as Concurrent Haskell (Peyton-Jones et al. 1996), seem to be more modest in their design. Following the π-calculus (Milner et al. 1992), such languages support only first-order message passing. While functions for first-order message passing can be encoded in CML, it is unclear whether, conversely, the concurrency primitives of CML can be expressed in those languages.

Contributions In this paper, we show that CML-style concurrency primitives can in fact be implemented as a library, in a language that already supports first-order message passing. Such a library makes higher-order concurrent programming possible in a language such as Concurrent Haskell. Our implementation has further interesting consequences. For instance, the designers of Con-

current Haskell deliberately avoid a CML-style choice primitive (Peyton-Jones et al. 1996, Section 5), partly concerned that such a primitive may complicate a distributed implementation of Concurrent Haskell. By showing that such a primitive can be encoded in Concurrent Haskell itself, we eliminate that concern.

At the heart of our implementation is a new distributed protocol for synchronization of events. Our protocol is carefully designed to ensure safety, progress, and fairness. In Section 3, we formalize this protocol as an abstract state machine, and prove its correctness. Then, in Section 4, we describe a concrete implementation of this protocol in the π-calculus, and prove its correctness as well. This implementation can serve as a foundation for other implementations in related languages. Building on this implementation, in Sections 5, 6, and 7, we show how to encode all of CML's concurrency primitives, and more, in Concurrent Haskell. Our implementation is very concise, requiring less than 150 lines of code; in contrast, a related existing implementation (Russell 2001) requires more than 1300 lines of code.

In Section 8, we compare the performance of our library against OCaml's standard library of CML-style primitives, via several small, informal experiments. Our library consistently runs faster in these experiments, even without possible optimizations. While these experiments do not account for various differences between the underlying language implementations, especially those of threads, we think that these results are nevertheless encouraging.

Finally, we should point out that unlike several previous implementations of CML-style primitives in other languages, we remain faithful to the standard semantics for those primitives (Reppy 1999). For example, we retain the symmetry of `choose`, which can express selective communication. Indeed, we seem to be the first to implement a CML library that relies purely on first-order message passing, and preserves the standard semantics. We defer a more detailed discussion on related work to Section 9.

2. Overview of CML

In this section, we present a brief overview of CML's concurrency primitives. (Space constraints prevent us from motivating these primitives any further; the interested reader can find a comprehensive account of these primitives, with several programming examples, in (Reppy 1999).) We provide a small example at the end of this section.

Note that `channel` and `event` are polymorphic type constructors in CML, as follows:

- The type `channel tau` is given to channels that carry values of type `tau`.
- The type `event tau` is given to events that return values of type `tau` on synchronization (see the function `sync` below).

The combinators `receive` and `transmit` build events for synchronous communication.

```
receive : channel tau -> event tau
transmit : channel tau -> tau -> event ()
```

- `receive c` returns an event that, on synchronization, accepts a message `M` on channel `c` and returns `M`. Such an event must synchronize with `transmit c M`.
- `transmit c M` returns an event that, on synchronization, sends the message `M` on channel `c` and returns `()` (that is, "unit"). Such an event must synchronize with `receive c`.

Perhaps the most powerful of CML's concurrency primitives is the combinator `choose`; it can nondeterministically select an event from a list of events, *so that the selected event can be synchronized*. In particular, `choose` can express selective communication. Several implementations need to restrict the power of `choose` in order to tame it (Russell 2001; Reppy and Xiao 2008). Our implementation is designed to avoid such problems (see Section 9).

```
choose : [event tau] -> event tau
```

- `choose V` returns an event that, on synchronization, synchronizes one of the events in list `V` and "aborts" the other events.

Conversely, the combinator `wrapabort` can specify an action that is spawned if an event is aborted by a selection.

```
wrapabort : (() -> ()) -> event tau -> event tau
```

- `wrapabort f v` returns an event that either synchronizes the event `v`, or, if aborted, spawns a thread that runs the code `f ()`.

The combinators `guard` and `wrap` can specify pre- and post-synchronization actions.

```
guard : (() -> event tau) -> event tau
wrap : event tau -> (tau -> tau') -> event tau'
```

- `guard f` returns an event that, on synchronization, synchronizes the event returned by the code `f ()`.
- `wrap v f` returns an event that, on synchronization, synchronizes the event `v` and applies the function `f` to the result.

Finally, the function `sync` can synchronize an event and return the result.

```
sync : event tau -> tau
```

By design, an event can synchronize only at some "point", where a message is either sent or accepted on a channel. Such a point, called the *commit point*, may be selected among several other candidates at run time. Furthermore, some code may be run before, and after, synchronization—as specified by `guard` functions, by `wrap` functions that enclose the commit point, and by `wrapabort` functions that do not enclose the commit point.

For example, consider the following value of type `event ()`. (Here, `c` and `c'` are values of type `channel ()`.)

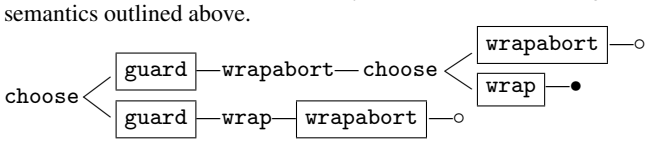
```
val v =
  choose
    [guard (fn () ->
       ...;
       wrapabort ...
         (choose [wrapabort ... (transmit c ());
                  wrap (transmit c' ()) ... ] ) );
     guard (fn () ->
       ...;
       wrap
         (wrapabort ... (receive c))
         ... ) ]
```

The event `v` describes a fairly complicated protocol that, on synchronization, selects among the communication events `transmit c ()`, `transmit c' ()`, and `receive c`, and runs some code (elided by `...`s) before and after synchronization. Now, suppose that we run the following ML program.

```
val _ =
  spawn (fn () -> sync v);
  sync (receive c')
```

This program spawns `sync v` in parallel with `sync (receive c')`. In this case, the event `transmit c' ()` is selected inside `v`, so that it synchronizes with `receive c'`. The figure below depicts `sync v` as a tree. The point marked • is the commit point; this point is selected among the other candidates, marked ○, at run time.

Furthermore, (only) code specified by the combinators marked in boxes are run before and after synchronization, following the semantics outlined above.



3. A distributed protocol for synchronization

We now present a distributed protocol for synchronizing events. We focus on events that are built with the combinators `receive`, `transmit`, and `choose`. While the other combinators are important for describing computations, they do not fundamentally affect the nature of the protocol; we consider them later, in Sections 5 and 6.

3.1 A source language

For brevity, we simplify the syntax of the source language. Let c range over channels. We use the following notations: $\overrightarrow{\varphi_\ell}$ is a sequence of the form $\varphi_1, \ldots, \varphi_n$, where $\ell \in 1..n$; furthermore, $\{\overrightarrow{\varphi_\ell}\}$ is the set $\{\varphi_1, \ldots, \varphi_n\}$, and $[\overrightarrow{\varphi_\ell}]$ is the list $[\varphi_1, \ldots, \varphi_n]$.

The syntax of the language is as follows.

- *Actions* $\alpha, \beta, \ldots$ are of the form c or $\overline{c}$ (input or output on c). Informally, actions model communication events built with `receive` and `transmit`.
- *Programs* are of the form $S_1 \mid \ldots \mid S_m$ (parallel composition of $S_1, \ldots, S_m$), where each S_k ($k \in 1..m$) is either an action α, or a selection of actions, $\mathsf{select}(\overrightarrow{\alpha_i})$. Informally, a selection of actions models the synchronization of a choice of events, following the CML function `select`.

```
select : [event tau] -> tau
select V = sync (choose V)
```

Further, we consider only the following local reduction rule:

$$\frac{c \in \{\overrightarrow{\alpha_i}\} \qquad \overline{c} \in \{\overrightarrow{\beta_j}\}}{\mathsf{select}(\overrightarrow{\alpha_i}) \mid \mathsf{select}(\overrightarrow{\beta_j}) \longrightarrow c \mid \overline{c}} \ (\textsc{Sel Comm})$$

This rule models selective communication. We also consider the usual structural rules for parallel composition. However, we ignore reduction of actions at this level of abstraction.

3.2 A distributed abstract state machine for synchronization

Our synchronization protocol is run by a distributed system of principals that include channels, *points*, and *synchronizers*. Informally, every action is associated with a point, and every select is associated with a synchronizer.

The reader may draw an analogy between our setting and one of arranging marriages, by viewing points as prospective brides and grooms, channels as matchmakers, and synchronizers as parents whose consents are necessary for marriages.

We formalize our protocol as a distributed abstract state machine that implements the rule (Sel Comm). Let σ range over states of the machine. These states are built by parallel composition $|$, inaction 0, and name creation ν (Milner et al. 1992) over various states of principals.

States of the machine σ

$\sigma ::=$	states of the machine
$\sigma \mid \sigma'$	parallel composition
0	inaction
$(\nu \overrightarrow{p_i})\ \sigma$	name creation
ς	state of principals

The various states of principals are shown below. Roughly, principals in specific states react with each other to cause transitions in the machine, following rules that appear later in the section.

States of principals ς

$\varsigma_p ::=$	states of a point
$p \mapsto \alpha$	active
$\heartsuit_p$	matched
α	released
$\varsigma_c ::=$	states of a channel
$\odot_c$	free
$\oplus_c(p, q)$	announced
$\varsigma_s ::=$	states of a synchronizer
$\boxdot_s$	open
$\boxtimes_s$	closed
$\checkmark_s(p)$	selected
$\times(p)$	refused
$\breve{\smile}_s\ (p)$	confirmed
$\breve{\frown}_s$	canceled

Let p and s range over points and synchronizers. A synchronizer can be viewed as a partial function from points to actions; we represent this function as a parallel composition of bindings of the form $p \mapsto \alpha$. Further, we require that each point is associated with a unique synchronizer, that is, for any s and s', $s \neq s' \Rightarrow \mathsf{dom}(s) \cap \mathsf{dom}(s') = \varnothing$.

The semantics of the machine is described by the local transition rules shown below, plus the usual structural rules for parallel composition, inaction, and name creation as in the π-calculus (Milner et al. 1992).

Operational semantics $\sigma \longrightarrow \sigma'$

(1) $\quad p \mapsto c \mid q \mapsto \overline{c} \mid \odot_c \longrightarrow \heartsuit_p \mid \heartsuit_q \mid \oplus_c(p, q) \mid \odot_c$

(2.i) $\quad \dfrac{p \in \mathsf{dom}(s)}{\heartsuit_p \mid \boxdot_s \longrightarrow \checkmark_s(p) \mid \boxtimes_s}$

(2.ii) $\quad \dfrac{p \in \mathsf{dom}(s)}{\heartsuit_p \mid \boxtimes_s \longrightarrow \times(p) \mid \boxtimes_s}$

(3.i) $\quad \checkmark_s(p) \mid \checkmark_{s'}(q) \mid \oplus_c(p, q) \longrightarrow \breve{\smile}_s\ (p) \mid \breve{\smile}_{s'}\ (q)$

(3.ii) $\quad \checkmark_s(p) \mid \times(q) \mid \oplus_c(p, q) \longrightarrow \breve{\frown}_s$

(3.iii) $\quad \times(p) \mid \checkmark_s(q) \mid \oplus_c(p, q) \longrightarrow \breve{\frown}_s$

(3.iv) $\quad \times(p) \mid \times(q) \mid \oplus_c(p, q) \longrightarrow 0$

(4.i) $\quad \dfrac{s(p) = \alpha}{\breve{\smile}_s\ (p) \longrightarrow \alpha}$

(4.ii) $\quad \breve{\frown}_s \triangleq (\nu \overrightarrow{p_i})\ (\boxdot_s \mid s)$ where $\mathsf{dom}(s) = \{\overrightarrow{p_i}\}$

Intuitively, these rules may be read as follows.

(1) Two points p and q, bound to complementary actions on channel c, react with c, so that p and q become matched ($\heartsuit_p$ and $\heartsuit_q$) and the channel announces their match ($\oplus_c(p, q)$).

(2.i–ii) Next, p (and likewise, q) reacts with its synchronizer s. If the synchronizer is open ($\boxdot_s$), it now becomes closed ($\boxtimes_s$), and p is declared selected by s ($\checkmark_s(p)$). If the synchronizer is already closed, then p is refused ($\times(p)$).

(3.i–iv) If both p and q are selected, c confirms the selections to both parties ($\breve{\smile}_s\ (p)$ and $\breve{\smile}_{s'}\ (q)$). If only one of them is selected, c cancels that selection ($\breve{\frown}_s$).

(4.i–ii) If the selection of p is confirmed, the action bound to p is released. Otherwise, the synchronizer "reboots" with fresh names for the points in its domain.

3.3 Compilation

Next, we show how programs in the source language are compiled on to this machine. Let Π denote indexed parallel composition; using this notation, for example, we can write a program $S_1 \mid \ldots \mid S_m$ as $\Pi_{k\in 1..m} S_k$. Suppose that the set of channels in a program $\Pi_{k\in 1..m} S_k$ is $\mathcal{C}$. We compile this program to the state $\Pi_{c\in\mathcal{C}}\ \odot_c \mid \Pi_{k\in 1..m}\ \widetilde{S_k}$, where

$$\widetilde{S} \triangleq \begin{cases} \alpha & \text{if } S = \alpha \\ (\nu \overrightarrow{p_i})\ (\boxdot_s \mid s) & \text{if } S = \mathsf{select}(\overrightarrow{\alpha_i}),\ i \in 1..n, \text{ and} \\ & s = \Pi_{i\in 1..n}\ (p_i \mapsto \alpha_i) \text{ for fresh names } \overrightarrow{p_i} \end{cases}$$

3.4 Correctness

We prove that our protocol is correct, that is, the abstract machine correctly implements (SEL COMM), by showing that the compilation from programs to states satisfies *safety*, *progress*, and *fairness*. Roughly, safety implies that any sequence of transitions in the state machine can be mapped to some sequence of reductions in the language. Furthermore, progress and fairness imply that any sequence of reductions in the language can be mapped to some sequence of transitions in the state machine. (The formal definitions of these properties are complicated because transitions in the machine have much finer granularity than reductions in the language; see below.)

Let a *denotation* be a list of actions. The denotations of programs and states are derived by the function $\ulcorner\cdot\urcorner$, as follows. (Here $\uplus$ denotes concatenation over lists.)

Denotations of programs and states $\ulcorner\cdot\urcorner$

$$\begin{array}{rcl} \ulcorner S_1 \mid \ldots \mid S_m \urcorner & = & \ulcorner S_1 \urcorner \uplus \cdots \uplus \ulcorner S_m \urcorner \\ \ulcorner \alpha \urcorner & = & [\alpha] \\ \ulcorner \mathsf{select}(\overrightarrow{\alpha_i}) \urcorner & = & [\,] \\ \hline \ulcorner \sigma \mid \sigma' \urcorner & = & \ulcorner \sigma \urcorner \uplus \ulcorner \sigma' \urcorner \\ \ulcorner 0 \urcorner & = & [\,] \\ \ulcorner (\nu \overrightarrow{p_i})\ \sigma \urcorner & = & \ulcorner \sigma \urcorner \\ \ulcorner \varsigma \urcorner & = & \begin{cases} [\alpha] & \text{if } \varsigma = \alpha \\ [\,] & \text{otherwise} \end{cases} \end{array}$$

Informally, the denotation of a program or state is the list of released actions in that program or state. Now, if a program is compiled to some state, then the denotations of the program and the state coincide. Furthermore, we expect that as intermediate programs and states are produced during execution (and other actions are released), the denotations of those intermediate programs and states remain in coincidence. Formally, we prove the following theorem (Chaudhuri 2009).

THEOREM 3.1 (Correctness of the abstract state machine). *Let $\mathcal{C}$ be the set of channels in a program $\Pi_{k\in 1..m} S_k$. Then*

$$\Pi_{k\in 1..m} S_k\ \sim\ \Pi_{c\in\mathcal{C}}\ \odot_c \mid \Pi_{k\in 1..m}\ \widetilde{S_k}$$

where $\sim$ is the largest relation such that $\mathcal{P} \sim \sigma$ iff

(Invariant) *$\sigma \longrightarrow^\star \sigma'$ for some σ' such that $\ulcorner \mathcal{P} \urcorner = \ulcorner \sigma' \urcorner$;*

(Safety) *if $\sigma \longrightarrow \sigma'$ for some σ', then $\mathcal{P} \longrightarrow^\star \mathcal{P}'$ for some $\mathcal{P}'$ such that $\mathcal{P}' \sim \sigma'$;*

(Progress) *if $\mathcal{P} \longrightarrow \_$, then $\sigma \longrightarrow^+ \sigma'$ and $\mathcal{P} \longrightarrow \mathcal{P}'$ for some σ' and $\mathcal{P}'$ such that $\mathcal{P}' \sim \sigma'$;*

(Fairness) *if $\mathcal{P} \longrightarrow \mathcal{P}'$ for some $\mathcal{P}'$, then $\sigma_0 \longrightarrow \ldots \longrightarrow \sigma_n$ for some $\sigma_0, \ldots, \sigma_n$ such that $\sigma_n = \sigma$, $\mathcal{P} \sim \sigma_i$ for all $0 \le i < n$, and $\sigma_0 \longrightarrow^+ \sigma'$ for some σ' such that $\mathcal{P}' \sim \sigma'$.*

Informally, the above theorem guarantees that any sequence of program reductions can be simulated by some sequence of state transitions, and vice versa, such that

- from any intermediate program, it is always possible to simulate any transition of a related intermediate state;
- from any intermediate state,
  - it is always possible to simulate some reduction of a related intermediate program;
  - further, by backtracking, it is always possible to simulate any reduction of that program.

3.5 Example

Consider the program

$$\mathsf{select}(\overline{x}, \overline{y}) \mid \mathsf{select}(y, z) \mid \mathsf{select}(\overline{z}) \mid \mathsf{select}(x)$$

By (SEL COMM), this program can reduce to

$$\overline{x} \mid z \mid \overline{z} \mid x$$

with denotation $[\overline{x}, z, \overline{z}, x]$, or to

$$\overline{y} \mid y \mid \mathsf{select}(\overline{z}) \mid \mathsf{select}(x)$$

with denotation $[\overline{y}, y]$.

The original program is compiled to the following state.

$$\begin{array}{l} \odot_x \mid\ \odot_y\ \mid \odot_z\ \ \mid (\nu p_{\overline{x}}, p_{\overline{y}})\ (\boxdot_{(p_{\overline{x}} \mapsto \overline{x}\ \mid\ p_{\overline{y}} \mapsto \overline{y})} \mid p_{\overline{x}} \mapsto \overline{x} \mid p_{\overline{y}} \mapsto \overline{y}) \\ \qquad\qquad\ \mid (\nu p_y, p_z)\ (\boxdot_{(p_y \mapsto y\ \mid\ p_z \mapsto z)} \mid p_y \mapsto y \mid p_z \mapsto z) \\ \qquad\qquad\ \mid (\nu p_{\overline{z}})\ (\boxdot_{(p_{\overline{z}} \mapsto \overline{z})} \mid p_{\overline{z}} \mapsto \overline{z}) \\ \qquad\qquad\ \mid (\nu p_x)\ (\boxdot_{(p_x \mapsto x)} \mid p_x \mapsto x) \end{array}$$

This state describes the states of several principals:

- channels $\odot_x$, $\odot_y$, $\odot_z$;
- points $p_{\overline{x}} \mapsto \overline{x}$, $p_{\overline{y}} \mapsto \overline{y}$, $p_y \mapsto y$, $p_z \mapsto z$, $p_{\overline{z}} \mapsto \overline{z}$, $p_x \mapsto x$;
- synchronizers $\boxdot_{(p_{\overline{x}} \mapsto \overline{x}\ \mid\ p_{\overline{y}} \mapsto \overline{y})}$, $\boxdot_{(p_y \mapsto y\ \mid\ p_z \mapsto z)}$, $\boxdot_{(p_{\overline{z}} \mapsto \overline{z})}$, $\boxdot_{(p_x \mapsto x)}$.

This state can eventually transition to

$$\odot_x \mid\ \odot_y\ \mid\ \odot_z\ \mid \overline{x} \mid z \mid \overline{z} \mid x \mid\ \sigma_{gc}$$

with denotation $[\overline{x}, z, \overline{z}, x]$, or to

$$\begin{array}{l} \odot_x \mid\ \odot_y\ \mid\ \odot_z\ \mid \overline{y} \mid y\ \mid \sigma_{gc} \\ \qquad\qquad\qquad\qquad\ \mid (\nu p_{\overline{z}})\ (\boxdot_{(p_{\overline{z}} \mapsto \overline{z})} \mid p_{\overline{z}} \mapsto \overline{z}) \\ \qquad\qquad\qquad\qquad\ \mid (\nu p_x)\ (\boxdot_{(p_x \mapsto x)} \mid p_x \mapsto x) \end{array}$$

with denotation $[\overline{y}, y]$. In these states, σ_{gc} can be garbage-collected, and is separated out for readability.

$$\begin{array}{l} \sigma_{gc} \triangleq (\nu p_{\overline{x}}, p_{\overline{y}}, p_y, p_z, p_{\overline{z}}, p_x) \\ \qquad (\boxtimes_{(p_{\overline{x}} \mapsto \overline{x}\ \mid\ p_{\overline{y}} \mapsto \overline{y})} \mid \boxtimes_{(p_y \mapsto y\ \mid\ p_z \mapsto z)} \mid \boxtimes_{(p_{\overline{z}} \mapsto \overline{z})} \mid \boxtimes_{(p_x \mapsto x)}) \end{array}$$

Let us examine the state with denotation $[\overline{y}, y]$, and trace the transitions to this state. In this state, the original synchronizers are all closed (see σ_{gc}). We can conclude that the remaining points $p_{\overline{z}} \mapsto \overline{z}$ and $p_x \mapsto x$ and their synchronizers $\boxdot_{(p_{\overline{z}} \mapsto \overline{z})}$ and $\boxdot_{(p_x \mapsto x)}$ were produced by rebooting their original synchronizers with fresh names $p_{\overline{z}}$ and p_x. Indeed, in a previous round of the protocol, the original points $p_{\overline{z}} \mapsto \overline{z}$ and $p_x \mapsto x$ were matched with the points $p_z \mapsto z$ and $p_{\overline{x}} \mapsto \overline{x}$, respectively; however, the latter points were refused by their synchronizers $\boxdot_{(p_y \mapsto y\ \mid\ p_z \mapsto z)}$ and

$\boxdot_{(p_{\bar{x}} \mapsto \overline{x} \mid p_{\bar{y}} \mapsto \overline{y})}$ (to accommodate the selected communication on y in that round); these refusals in turn necessitated the cancellations $\ddot{\frown}_{(p_{\bar{z}} \mapsto \overline{z})}$ and $\ddot{\frown}_{(p_x \mapsto x)}$.

4. Higher-order concurrency in the π-calculus

While we have an abstract state machine that correctly implements (SEL COMM), we do not yet know if the local transition rules in Section 3.2 can be implemented faithfully, say by first-order message-passing. We now show how these rules can be implemented concretely in the π-calculus (Milner et al. 1992).

The π-calculus is a minimal concurrent language that allows processes to dynamically create channels with fresh names and communicate such names over channels. This language forms the core of Concurrent Haskell. Let a, b, x range over names. The syntax of processes is as follows.

Processes π

$\pi ::=$	processes
$\pi \mid \pi'$	parallel composition
0	inaction
$(\nu a)\ \pi$	name creation
$\overline{a}\langle b \rangle .\ \pi$	output
$a(x).\ \pi$	input
$!\pi$	replication

Processes have the following informal meanings.

- $\pi \mid \pi'$ behaves as the parallel composition of π and π'.
- 0 does nothing.
- $(\nu a)\ \pi$ creates a channel with fresh name a and continues as π; the scope of a is π.
- $\overline{a}\langle b \rangle .\ \pi$ sends the name b on channel a, and continues as π.
- $a(x).\ \pi$ accepts a name on channel a, binds it to x, and continues as π; the scope of x is π.
- $!\pi$ behaves as the parallel composition of an unbounded number of copies of π; this construct, needed to model recursion, can be eliminated with recursive process definitions.

A formal operational semantics can be found in (Milner et al. 1992). Of particular interest are the following reduction rule for communication:

$$a(x).\ \pi \mid \overline{a}\langle b \rangle .\ \pi' \ \longrightarrow\ \pi\{b/x\} \mid \pi'$$

and the following structural rule for scope extrusion:

$$\frac{a \text{ is fresh in } \pi}{\pi \mid (\nu a)\ \pi' \ \equiv\ (\nu a)\ (\pi \mid \pi')}$$

The former rule models the communication of a name b on a channel a, from an output process to an input process (in parallel); b is substituted for x in the remaining input process. The latter rule models the extrusion of the scope of a fresh name a across a parallel composition. These rules allow other derivations, such as the following for communication of fresh names:

$$\frac{b \text{ is fresh in } a(x).\ \pi}{a(x).\ \pi \mid (\nu b)\ \overline{a}\langle b \rangle .\ \pi' \ \equiv\ (\nu b)\ (\pi\{b/x\} \mid \pi')}$$

4.1 A π-calculus model of the abstract state machine

We interpret states of our machine as π-calculus processes that run at points, channels, and synchronizers. These processes reduce by communication to simulate transitions in the abstract state machine. In this setting:

- Each point is identified with a fresh name p.
- Each channel c is identified with a pair of fresh names $(i^{[c]}, o^{[c]})$, on which it accepts messages from points that are bound to input or output actions on c.
- Each synchronizer is identified with a fresh name s, on which it accepts messages from points in its domain.

Informally, the following sequence of messages are exchanged in any round of the protocol.

- A point p (at state $p \mapsto c$ or $p \mapsto \overline{c}$) begins by sending a message to c on its respective input or output name $i^{[c]}$ or $o^{[c]}$; the message contains a fresh name $candidate^{[p]}$ on which p expects a reply from c.
- When c (at state $\odot_c$) gets a pair of messages on $i^{[c]}$ and $o^{[c]}$, say from p and another point q, it replies by sending messages on $candidate^{[p]}$ and $candidate^{[q]}$ (reaching state $\oplus_c(p, q) \mid \odot_c$); these messages contain fresh names $decision^{[p]}$ and $decision^{[q]}$ on which c expects replies from the synchronizers for p and q.
- On receiving a message from c on $candidate^{[p]}$, p (reaching state $\heartsuit_p$) tags the message with its name and forwards it to its synchronizer on the name s.
- If p is the first point to send such a message on s (that is, s is at state $\boxdot_s$), a pair of fresh names $(confirm^{[p]}, cancel^{[p]})$ is sent back on $decision^{[p]}$ (reaching state $\checkmark_s(p) \mid \boxtimes_s$); for each subsequent message accepted on s, say from p', a blank message is sent back on $decision^{[p']}$ (reaching state $\times(p') \mid \boxtimes_s$).
- On receiving messages from the respective synchronizers of p and q on $decision^{[p]}$ and $decision^{[q]}$, c inspects the messages and responds.
  - If both $(confirm^{[p]}, \_)$ and $(confirm^{[q]}, \_)$ have come in, signals are sent back on $confirm^{[p]}$ and $confirm^{[q]}$.
  - If only $(\_, cancel^{[p]})$ has come in (and the other message is blank), a signal is sent back on $cancel^{[p]}$; likewise, if only $(\_, cancel^{[q]})$ has come in, a signal is sent back on $cancel^{[q]}$.
- If s gets a signal on $confirm^{[p]}$ (reaching state $\breve{\smile}_s\ (p)$), it signals on p to continue. If s gets a signal on $cancel^{[p]}$ (reaching state $\ddot{\frown}_s$), it "reboots" with fresh names for the points in its domain, so that those points can begin another round.

Below we formalize this interpretation of states as (recursively defined) processes. For convenience, we let the interpreted states carry some auxiliary state variables in $[\![\ldots]\!]$; these state variables represent names that are created at run time. The state variables carried by any state are unique to that state. Thus, they do not convey any new, distinguishing information about that state.

For simplicity, we leave states of the form α uninterpreted, and consider them inactive. We define $\hat{\alpha}$ as shorthand for $i^{[c]}$ if α is of the form c, and $o^{[c]}$ if α is of the form $\overline{c}$.

Programs in the source language are now compiled to processes in the π-calculus. Suppose that the set of channels in a program $\Pi_{k \in 1..m} S_k$ is $\mathcal{C}$. We compile this program to the process $(\nu_{c \in \mathcal{C}}\ i^{[c]}, o^{[c]})\ (\Pi_{c \in \mathcal{C}}\ \odot_c \mid \Pi_{k \in 1..m}\ \tilde{\tilde{S}}_k)$, where

$$\tilde{\tilde{S}} \ \triangleq\ \begin{cases} \alpha & \text{if } S = \alpha \\ (\nu s, \overrightarrow{p_i})\ \ (\boxdot_s \mid \Pi_{i \in 1..n} (p_i \mapsto \alpha_i)[\![s, \hat{\alpha}_i]\!]) & \text{if } S = \mathsf{select}(\overrightarrow{\alpha_i}),\ i \in 1..n, \text{ and } s, \overrightarrow{p_i} \text{ are fresh names} \end{cases}$$

Interpretation of states as processes

States of a point

$$(p \mapsto c)[\![s, i^{[c]}]\!] \triangleq$$
$$(\nu\ candidate^{[p]})\ \overline{i^{[c]}}\langle candidate^{[p]}\rangle.\ candidate^{[p]}(decision^{[p]}).$$
$$\heartsuit_p[\![decision^{[p]}, s, c]\!]$$

$$(q \mapsto \overline{c})[\![s, o^{[c]}]\!] \triangleq$$
$$(\nu\ candidate^{[q]})\ \overline{o^{[c]}}\langle candidate^{[q]}\rangle.\ candidate^{[q]}(decision^{[q]}).$$
$$\heartsuit_q[\![decision^{[q]}, s, \overline{c}]\!]$$

$$\heartsuit_p[\![decision^{[p]}, s, \alpha]\!] \triangleq$$
$$\overline{s}\langle p, decision^{[p]}\rangle.\ p().$$
$$\alpha$$

States of a channel

$$\odot_c[\![i^{[c]}, o^{[c]}]\!] \triangleq$$
$$i^{[c]}(candidate^{[p]}).\ o^{[c]}(candidate^{[q]}).$$
$$((\nu\, decision^{[p]}, decision^{[q]})$$
$$\overline{candidate^{[p]}}\langle decision^{[p]}\rangle.\ \overline{candidate^{[q]}}\langle decision^{[q]}\rangle.$$
$$\oplus_c\ (p, q)[\![decision^{[p]}, decision^{[q]}]\!]$$
$$\mid\ \odot_c\ [\![i^{[c]}, o^{[c]}]\!])$$

$$\oplus_c(p, q)[\![decision^{[p]}, decision^{[q]}]\!] \triangleq$$
$$(decision^{[p]}(confirm^{[p]}, cancel^{[p]}).$$
$$(\overline{decision^{[q]}}(confirm^{[q]}, cancel^{[q]}).$$
$$\overline{confirm^{[p]}}\langle\rangle.\ \overline{confirm^{[q]}}\langle\rangle.\ 0$$
$$\mid decision^{[q]}().$$
$$\overline{cancel^{[p]}}\langle\rangle.\ 0)$$
$$\mid decision^{[p]}().$$
$$(\overline{decision^{[q]}}(confirm^{[q]}, cancel^{[q]}).$$
$$\overline{cancel^{[q]}}\langle\rangle.\ 0$$
$$\mid decision^{[q]}().$$
$$0))$$

States of a synchronizer

$$\boxdot_s \triangleq$$
$$s(p, decision^{[p]}).$$
$$(\checkmark_s(p)[\![decision^{[p]}]\!] \mid \boxtimes_s)$$

$$\boxtimes_s \triangleq$$
$$s(p, decision^{[p]}).$$
$$(\times(p)[\![decision^{[p]}]\!] \mid \boxtimes_s)$$

$$\checkmark_s(p)[\![decision^{[p]}]\!] \triangleq$$
$$(\nu\ confirm^{[p]}, cancel^{[p]})\ \overline{decision^{[p]}}\langle confirm^{[p]}, cancel^{[p]}\rangle.$$
$$(confirm^{[p]}().\ \ddot{\smile}_s\ (p)$$
$$\mid cancel^{[p]}().\ \ddot{\frown}_s)$$

$$\times_s(p)[\![decision^{[p]}]\!] \triangleq$$
$$\overline{decision^{[p]}}\langle\rangle.\ 0$$

$$\ddot{\smile}_s\ (p) \triangleq$$
$$\overline{p}\langle\rangle.\ 0$$

$$\ddot{\frown}_s \triangleq (\nu s, \overrightarrow{p_i})\ (\boxdot_s \mid \Pi_{i\in 1..n}(p_i \mapsto \alpha_i)[\![s, \hat{\alpha}_i]\!])$$
where $\mathsf{dom}(s) = \{\overrightarrow{p_i}\}$, $i \in 1..n$, and $\forall i \in 1..n.\ s(p_i) = \alpha_i$

Let $\Uparrow$ be a partial function from processes to states that, for any state σ, maps its interpretation as a process back to σ. For any process π such that $\Uparrow \pi$ is defined, we define its denotation $\ulcorner\pi\urcorner$ to be $\ulcorner\Uparrow \pi\urcorner$; the denotation of any other process is undefined. We then prove the following theorem (Chaudhuri 2009), closely following the proof of Theorem 3.1.

THEOREM 4.1 (Correctness of the π-calculus implementation). *Let $\mathcal{C}$ be the set of channels in a program $\Pi_{k\in 1..m}S_k$. Then*

$$\Pi_{k\in 1..m}S_k \approx (\nu_{c\in\mathcal{C}}\ i^{[c]}, o^{[c]})\ (\Pi_{c\in\mathcal{C}}\ \odot_c\ \mid \Pi_{k\in 1..m}\ \widetilde{\widetilde{S_k}})$$

where $\approx$ is the largest relation such that $\mathcal{P} \approx \pi$ iff

(Invariant) *$\pi \longrightarrow^\star \pi'$ for some π' such that $\ulcorner\mathcal{P}\urcorner = \ulcorner\pi'\urcorner$;*
(Safety) *if $\pi \longrightarrow \pi'$ for some π', then $\mathcal{P} \longrightarrow^\star \mathcal{P}'$ for some $\mathcal{P}'$ such that $\mathcal{P}' \approx \pi'$;*
(Progress) *if $\mathcal{P} \longrightarrow \_$, then $\pi \longrightarrow^+ \pi'$ and $\mathcal{P} \longrightarrow \mathcal{P}'$ for some π' and $\mathcal{P}'$ such that $\mathcal{P}' \approx \pi'$;*
(Fairness) *if $\mathcal{P} \longrightarrow \mathcal{P}'$ for some $\mathcal{P}'$, then $\pi_0 \longrightarrow \ldots \longrightarrow \pi_n$ for some $\pi_0, \ldots, \pi_n$ such that $\pi_n = \pi$, $\mathcal{P} \approx \pi_i$ for all $0 \le i < n$, and $\pi_0 \longrightarrow^+ \pi'$ for some π' such that $\mathcal{P}' \approx \pi'$.*

5. A CML library in Concurrent Haskell

We now proceed to code a full CML-style library for events in a fragment of Concurrent Haskell with first-order message passing (Peyton-Jones et al. 1996). This fragment is close to the π-calculus, so we can simply lift our implementation of Section 4.1. Going further, we remove the restrictions on the source language: a program can be any well-typed Haskell program. We implement not only `receive`, `transmit`, `choose`, and `sync`, but also `new`, `guard`, `wrap`, and `wrapabort`. Finally, we exploit Haskell's type system to show how events can be typed under the standard `IO` monad (Gordon 1994; Peyton-Jones and Wadler 1993).

Before we proceed, let us briefly review Concurrent Haskell's concurrency primitives. (The reader may wish to refer (Peyton-Jones et al. 1996) for details.) These primitives support concurrent I/O computations, such as forking threads and communicating on *mvars*—which are synchronized mutable variables, similar to π-calculus channels (see below).

Note that `MVar` and `IO` are polymorphic type constructors in Concurrent Haskell, as follows:

- The type `MVar tau` is given to a communication cell that carries values of type `tau`.
- The type `IO tau` is given to a computation that yields results of type `tau`, with possible side effects via communication.

We rely on the following semantics of `MVar` cells.

- A cell can carry at most one value at a time, that is, it is either empty or full.
- The function `newEmptyMVar :: IO (MVar tau)` returns a fresh cell that is empty.
- The function `takeMVar :: MVar tau -> IO tau` is used to read from a cell; `takeMVar m` blocks if the cell `m` is empty, else gets the content of `m` (thereby emptying it).
- The function `putMVar :: MVar tau -> tau -> IO ()` is used to write to a cell; `putMVar m M` blocks if the cell `m` is full, else puts the term `M` in `m` (thereby filling it).

Further, we rely on the following semantics of `IO` computations; see (Peyton-Jones and Wadler 1993) for details.

- The function `forkIO :: IO () -> IO ()` is used to spawn a concurrent computation; `forkIO f` forks a thread that runs the computation `f`.

- The function `return :: tau -> IO tau` is used to inject a value into a computation.
- Computations can be sequentially composed by "piping". We use Haskell's convenient `do {...}` notation for this purpose, instead of applying the underlying piping function

  ```
  (>>=) :: IO tau -> (tau -> IO tau') -> IO tau'
  ```

  Thus, *e.g.*, we write `do {x <- takeMVar m; putMVar m x}` instead of `takeMVar m >>= \x -> putMVar m x`.

Our library provides the following CML-style functions for programming with events in Concurrent Haskell.[1] (Observe the differences between ML and Haskell types for these functions. Since Haskell is purely functional, we must embed types for computations, with possible side-effects via communication, within the `IO` monad. Further, since evaluation in Haskell is lazy, we can discard λ-abstractions that simply "delay" eager evaluation.)

```
new :: IO (channel tau)
receive :: channel tau -> event tau
transmit :: channel tau -> tau -> event ()
guard :: IO (event tau) -> event tau
wrap :: event tau -> (tau -> IO tau') -> event tau'
choose :: [event tau] -> event tau
wrapabort :: IO () -> event tau -> event tau
sync :: event tau -> IO tau
```

In this section, we focus on events that are built without `wrapabort`; the full implementation appears in Section 6.

5.1 Type definitions

We begin by defining the types of cells on which messages are exchanged in our protocol (recall the discussion in Section 4.1).[2]

These cells are of the form `i` and `o` (on which points initially send messages to channels), `candidate` (on which channels reply back to points), `s` (on which points forward messages to synchronizers), `decision` (on which synchronizers inform channels), `confirm` and `cancel` (on which channels reply back to synchronizers), and `p` (on which synchronizers finally signal to points).

```
type In = MVar Candidate
type Out = MVar Candidate
type Candidate = MVar Decision
type Synchronizer = MVar (Point, Decision)
type Decision = MVar (Maybe (Confirm, Cancel))
type Confirm = MVar ()
type Cancel = MVar ()
type Point = MVar ()
```

Below, we use the following typings for the various cells used in our protocol: `i :: In`, `o :: Out`, `candidate :: Candidate`, `s :: Synchronizer`, `decision :: Decision`, `confirm :: Confirm`, `cancel :: Cancel`, and `p :: Point`.

We now show code run by points, channels, and synchronizers in our protocol. This code may be viewed as a typed version of the π-calculus code in Section 4.1.

[1] Instead of `wrapabort`, some implementations of CML provide the combinator `withnack`. Their expressive powers are exactly the same (Reppy 1999). Providing `withnack` is easier with an implementation strategy that relies on negative acknowledgments. Since our implementation strategy does not rely on negative acknowledgments, we stick with `wrapabort`.

[2] In Haskell, the type `Maybe tau` is given to a value that is either `Nothing`, or of the form `Just v` where `v` is of type `tau`.

5.2 Protocol code for points

The protocol code run by points abstracts on a cell `s` for the associated synchronizer, and a name `p` for the point itself. Depending on whether the point is for input or output, the code further abstracts on an input cell `i` or output cell `o`, and an associated action `alpha`.

```
AtPointI :: Synchronizer -> Point -> In ->
            IO tau -> IO tau
AtPointI s p i alpha = do {
  candidate <- newEmptyMVar;
  putMVar i candidate;
  decision <- takeMVar candidate;
  putMVar s (p,decision);
  takeMVar p;
  alpha
}

AtPointO :: Synchronizer -> Point -> Out ->
            IO () -> IO ()
AtPointO s p o alpha = do {
  candidate <- newEmptyMVar;
  putMVar o candidate;
  decision <- takeMVar candidate;
  putMVar s (p,decision);
  takeMVar p;
  alpha
}
```

We instantiate the function `AtPointI` in the code for `receive`, and the function `AtPointO` in the code for `transmit`. These associate appropriate point principals to any events constructed with `receive` and `transmit`.

5.3 Protocol code for channels

The protocol code run by channels abstracts on an input cell `i` and an output cell `o` for the channel.

```
AtChan :: In -> Out -> IO ()
AtChan i o =  do {
  candidate_i <- takeMVar i;
  candidate_o <- takeMVar o;
  forkIO (AtChan i o);
  decision_i <- newEmptyMVar;
  decision_o <- newEmptyMVar;
  putMVar candidate_i decision_i;
  putMVar candidate_o decision_o;
  x_i <- takeMVar decision_i;
  x_o <- takeMVar decision_o;
  case (x_i,x_o) of
    (Nothing, Nothing) ->
      return ()
    (Just(_,cancel_i), Nothing) ->
      putMVar cancel_i ()
    (Nothing, Just(_,cancel_o)) ->
      putMVar cancel_o ()
    (Just(confirm_i,_), Just(confirm_o,_)) -> do {
      putMVar confirm_i ();
      putMVar confirm_o ()
    }
}
```

We instantiate this function in the code for `new`. This associates an appropriate channel principal to any channel created with `new`.

5.4 Protocol code for synchronizers

The protocol code run by synchronizers abstracts on a cell `s` for that synchronizer and some "rebooting code" `reboot`, provided later.

(We encode a loop with the function `fix :: (tau -> tau) -> tau`; the term `fix f` reduces to `f (fix f)`.)

```
AtSync :: Synchronizer -> IO () -> IO ()
AtSync s reboot = do {
  (p,decision) <- takeMVar s;
  forkIO
    (fix (\iter -> do {
      (p',decision') <- takeMVar s;
      putMVar decision' Nothing;
      iter
     } ) );
  confirm <- newEmptyMVar;
  cancel <- newEmptyMVar;
  putMVar decision (Just (confirm,cancel));
  forkIO
    (do {
      takeMVar confirm;
      putMVar p ()
     } );
  takeMVar cancel;
  reboot
}
```

We instantiate this function in the code for `sync`. This associates an appropriate synchronizer principal to any application of `sync`.

5.5 Translation of types

Next, we translate types for channels and events. The Haskell types for ML `channel` and `event` values are:

```
type channel tau = (In, Out, MVar tau)
type event tau = Synchronizer -> IO tau
```

An ML `channel` is a Haskell `MVar` tagged with a pair of input and output cells. An ML `event` is a Haskell `IO` function that abstracts on a synchronizer cell.

5.6 Translation of functions

We now translate functions for programming with events. We begin by encoding the ML function for creating channels.

```
new :: IO (channel tau)
new  = do {
  i <- newEmptyMVar;
  o  <- newEmptyMVar;
  forkIO (AtChan i o);
  m  <- newEmptyMVar;
  return (i,o,m)
}
```

- The term `new` spawns an instance of `AtChan` with a fresh pair of input and output cells, and returns that pair along with a fresh `MVar` cell that carries messages for the channel.

Next, we encode the ML combinators for building communication events. Recall that a Haskell event is an `IO` function that abstracts on the cell of its synchronizer.

```
receive :: channel tau -> event tau
receive (i,o,m) = \s -> do {
  p <- newEmptyMVar;
  AtPointI s p i (takeMVar m)
}
transmit :: channel tau -> tau -> event ()
transmit (i,o,m) M = \s -> do {
  p <- newEmptyMVar;
  AtPointO s p o (putMVar m M)
}
```

- The term `receive c s` runs an instance of `AtPointI` with the synchronizer `s`, a fresh name for the point, the input cell for channel `c`, and an action that inputs on `c`.
- The term `transmit c M s` is symmetric; it runs an instance of `AtPointO` with the synchronizer `s`, a fresh name for the point, the output cell for channel `c`, and an action that outputs term `M` on `c`.

Next, we encode the ML combinators for specifying pre- and post-synchronization actions.

```
guard :: IO (event tau) -> event tau
guard f = \s -> do {
  v <- f;
  v s
}
wrap :: event tau -> (tau -> IO tau') -> event tau'
wrap v f = \s -> do {
  x <- v s;
  f x
}
```

- The term `guard f s` runs the computation `f` and passes the synchronizer `s` to the event returned by the computation.
- The term `wrap v f s` passes the synchronizer `s` to the event `v` and pipes the returned value to function `f`.

Next, we encode the ML combinator for choosing among a list of events. (We encode recursion over a list with the function `foldM :: (tau' -> tau -> IO tau') -> tau' -> [tau] -> IO tau'`. The term `foldM f x []` reduces to `return x`, and the term `foldM f x [v,V]` reduces to `do {x <- f x v; foldM f x V}`.)

```
choose :: [event tau] -> event tau
choose V = \s -> do {
  temp <- newEmptyMVar;
  foldM (\_ -> \v ->
      forkIO (do {
        x <- v s;
        putMVar temp x
      } ) )
    () V;
  takeMVar temp
}
```

- The term `choose V s` spawns a thread for each event `v` in `V`, passing the synchronizer `s` to `v`; any value returned by one of these threads is collected in a fresh cell `temp` and returned.

Finally, we encode the ML function for event synchronization.

```
sync :: event tau -> IO tau
sync v = do {
  temp <- newEmptyMVar;
  forkIO
    (fix (\iter -> do {
       s <- newEmptyMVar;
       forkIO (AtSync s iter);
       x <- v s;
       putMVar temp x
     } ) );
  takeMVar temp
}
```

- The term `sync v` recursively spawns an instance of `AtSync` with a fresh synchronizer `s` and passes `s` to the event `v`; any value returned by one of these instances is collected in a fresh cell `temp` and returned.

6. Implementation of `wrapabort`

The implementation of the previous section does not account for `wrapabort`. We now show how `wrapabort` can be handled by enriching the type for events.

Recall that abort actions are spawned only at events that do not enclose the commit point. Therefore, in an encoding of `wrapabort`, it makes sense to "name" events with the sets of points they enclose. Note that such names may not be static. In particular, for an event built with `guard`, we need to run the `guard` functions to compute the set of points that such an event encloses. Thus, we do not name events at compile time. Instead, we introduce events as principals in our protocol; each event is named *in situ* by computing the list of points it encloses at run time. This list is carried on a fresh cell `name :: Name` for that event.

```
type Name = MVar [Point]
```

Further, each synchronizer carries a fresh cell `abort :: Abort` on which it accepts `wrapabort` functions from events, tagged with the list of points they enclose.

```
type Abort = MVar ([Point], IO ())
```

The protocol code run by points and channels remains the same. We only add a handler for `wrapabort` functions to the protocol code run by synchronizers. Accordingly, that code now abstracts on an `abort` cell.

```
AtSync :: Synchronizer -> Abort -> IO () -> IO ()
AtSync s abort X = do {
  ...;
  forkIO (do {
    ...;
    fix (\iter -> do {
      (P,f) <- takeMVar abort;
      forkIO iter;
      if (elem p P) then return ()
      else f
    } )
  } );
  ...
}
```

Now, after signaling the commit point `p` to continue, the synchronizer continues to accept abort code `f` on `abort`; such code is spawned only if the list of points `P`, enclosed by the event that sends that code, does not include `p`.

The enriched Haskell type for `event` values is as follows.

```
type event tau =
       Synchronizer -> Name -> Abort -> IO tau
```

Now, an ML `event` is a Haskell `IO` function that abstracts on a synchronizer, an abort cell, and a name cell that carries the list of points the event encloses.

The Haskell function `new` does not change. We highlight minor changes in the remaining translations. We begin with the functions `receive` and `transmit`. An event built with either function is named by a singleton containing the name of the enclosed point.

```
receive (i,o,m) = \s -> \name -> \abort -> do {
  ...;
  forkIO (putMVar name [p]);
  ...
}
transmit (i,o,m) M = \s -> \name -> \abort -> do {
  ...;
  forkIO (putMVar name [p]);
  ...
}
```

In the function `choose`, a fresh `name'` cell is passed to each event in the list of choices; the names of those events are concatenated to name the `choose` event.

```
choose V = \s -> \name -> \abort -> do {
  ...;
  P <-
    foldM (\P -> \v ->
      do {
        name' <- newEmptyMVar;
        forkIO (do {
          x <- v s name' abort;
          ...
        } );
        P' <- takeMVar name';
        putMVar name' P';
        return (P' ++ P)
      } ) [] V;
  forkIO (putMVar name P);
  ...
}
```

We now encode the ML combinator for specifying abort actions.

```
wrapabort :: IO () -> event tau -> event tau
wrapabort f v = \s -> \name -> \abort -> do {
  forkIO (do {
    P <- takeMVar name;
    putMVar name P;
    putMVar abort (P,f)
  } );
  v s name abort
}
```

- The term `wrapabort f v s name abort` spawns a thread that reads the list of enclosed events `P` on the cell `name` and sends the function `f` along with `P` on the cell `abort`; the synchronizer `s` is passed to the event `v` along with `name` and `abort`.

The functions `guard` and `wrap` remain similar.

```
guard f = \s -> \name -> \abort -> do {
  v <- f;
  v s name abort
}

wrap v f = \s -> \name -> \abort -> do {
  x <- v s name abort;
  f x
}
```

Finally, in the function `sync`, a fresh `abort` cell is now passed to `AtSync`, and a fresh `name` cell is created for the event to be synchronized.

```
sync v = do {
  ...;
  forkIO (fix (\iter -> do {
    ...;
    name <- newEmptyMVar;
    abort <- newEmptyMVar;
    forkIO (AtSync s abort iter);
    x <- v s name abort;
    ...
  } ) );
  ...
}
```

7. Implementation of communication guards

Beyond the standard primitives, some implementations of CML further consider primitives for *guarded communication*. In particular, Russell (2001) implements such primitives in Concurrent Haskell, but his implementation strategy is fairly specialized—for example, it requires a notion of guarded events (see Section 9 for a discussion on this issue). We show that in contrast, our implementation strategy can accommodate such primitives with little effort.

Specifically, we wish to support the following `receive` combinator, that can carry a communication guard.

```
receive :: channel tau -> (tau -> Bool) -> event tau
```

Intuitively, `(receive c cond)` synchronizes with `(transmit c M)` only if `cond M` is true.

In our implementation, we make minor adjustments to the types of cells on which messages are exchanged between points and channels.

```
type In tau = MVar (Candidate, tau -> Bool)
type Out tau = MVar (Candidate, tau)
type Candidate = MVar (Maybe Decision)
```

Next, we adjust the protocol code run by points and channels. Input and output points bound to actions on `c` now send their conditions and messages to `c`. A pair of points is matched only if the message sent by one satisfies the condition sent by the other.

```
AtChan :: In tau -> Out tau -> IO ()
AtChan i o = do {
  (candidate_i,cond) <- takeMVar i;
  (candidate_o,M) <- takeMVar o;
  ...;
  if (cond M) then do {
    ...;
    putMVar candidate_i (Just decision_i);
    putMVar candidate_o (Just decision_o);
    ...
  } else do {
    putMVar candidate_i Nothing;
    putMVar candidate_i Nothing
  }
}

AtPointI :: Synchronizer -> Point -> In tau ->
             (tau ->  Bool) ->  IO tau ->  IO tau
AtPointI s p i cond alpha = do {
  ...;
  putMVar i (candidate,cond);
  x <- takeMVar candidate;
  case x of
    Nothing ->
      AtPointI s p i cond alpha
    Just decision -> do {
      putMVar s (p,decision);
      ...
    }
}
AtPointO :: Synchronizer -> Point -> Out tau ->
             tau ->  IO () ->  IO ()
AtPointO s p o M alpha = do {
  ...;
  putMVar o (candidate,M);
  x <- takeMVar candidate;
  case x of
    Nothing ->
      AtPointO s p o M alpha
    Just decision -> do {
      putMVar s (p,decision);
      ...
    }
}
```

Finally, we make minor adjustments to the type constructor `channel`, and the functions `receive` and `transmit`.

```
type channel tau = (In tau, Out tau, MVar tau)

receive (i,o,m) cond = \s -> \name -> \abort -> do {
  ...;
  AtPointI s p i cond (takeMVar m)
}
transmit (i,o,m) M = \s -> \name -> \abort -> do {
  ...;
  AtPointO s p o M (putMVar m M)
}
```

8. Evaluation

Our implementation is derived from a formal model, constructed for the purpose of proof (see Theorem 4.1). Not surprisingly, to simplify reasoning about the correctness of our code, we overlook several possible optimizations. For example, we heavily rely on lazy evaluation and garbage collection in the underlying language for reasonable performance of our code. It is plausible that this performance can be improved with explicit management. We also rely on fair scheduling in the underlying language to prevent starvation.

Nevertheless, preliminary experiments indicate that our code is already quite efficient. In particular, we compare the performance of our library against OCaml's `Event` module (Leroy et al. 2008). The implementation of this module is directly based on Reppy's original design of CML (Reppy 1999). Furthermore, it supports `wrapabort`, unlike recent versions of CML that favor an alternative primitive, `withnack`, which we do not support (see footnote 1, p.7). Finally, most other implementations of CML-style primitives do not reflect the standard semantics (Reppy 1999), which makes comparisons with them meaningless. Indeed, some of our benchmarks rely on the symmetry of `choose`—see, *e.g.*, the swap channel abstraction implemented below; such benchmarks cannot work correctly on a previous implementation of events in Haskell (Russell 2001).[3]

For our experiments, we use several small benchmark programs that rely heavily on higher-order concurrency. We describe these benchmarks below; their code is available online (Chaudhuri 2009). These benchmarks are duplicated in Haskell and OCaml to the extent possible. Furthermore, to minimize noise due to inherent differences in the implementations of these languages, we avoid the use of extraneous constructs in these benchmarks. Still, we cannot avoid the use of threads, and thus our results may be skewed by differences in the implementations of threads in these languages. We compile these benchmarks using `ghc` 6.8.1 and `ocamlc` 3.10.2 (using the `-vmthread` option in the latter). All these benchmarks run faster using our library than using OCaml's `Event` module.

Our benchmarks are variations of the following programs.

Extended example Recall the example of Section 3.5. This is a simple concurrent program that involves nondeterministic communication; either there is communication on channels x and z, or there is communication on channel y. To observe this nondeterminism, we add `guard`, `wrap`, and `wrapabort` functions to each communication event, which print messages such

[3] Nevertheless, we did consider comparing Russell's implementation with ours on other benchmarks, but failed to compile his implementation with recent versions of `ghc`; we also failed to find his contact information online.

as `"Trying"`, `"Succeeded"`, and `"Failed"` for that event at run time. Both the Haskell and the ML versions of the program exhibit this nondeterminism in our runs.

Primes sieve This program uses the Sieve of Eratosthenes (Wikipedia 2009) to print all prime numbers up to some $n \geq 2$. We implement two versions of this program: (I) uses `choose`, (II) does not.

(I) In this version, we create a "prime" channel and a "not prime" channel for each $i \in 2..n$, for a total of $2 * (n - 1)$ channels. Next, we spawn a thread for each $i \in 2..n$, that selects between two events: one receiving on the "prime" channel for i and printing i, the other receiving on the "not prime" channel for i and looping. Now, for each multiple $j \leq n$ of each $i \in 2..n$, we send on the "not prime" channel for j. Finally, we spawn a thread for each $i \in 2..n$, sending on the "prime" channel for i.

(II) In this version, we create a "prime/not prime" channel for each $i \in 2..n$, for a total of $n - 1$ channels. Next, we spawn a thread for each $i \in 2..n$, receiving a message on the "prime/not prime" channel for i, and printing i if the message is `true` or looping if the message is `false`. Now, for each multiple $j \leq n$ of each $i \in 2..n$, we send `false` on the "prime/not prime" channel for j. Finally, we spawn a thread for each $i \in 2..n$, sending `true` on the "prime/not prime" channel for i.

Swap channels This program implements and uses a *swap channel abstraction*, as described in (Reppy 1994). Intuitively, if `x` is a swap channel, and we run the program

```
forkIO (do {y <- sync (swap x M); ...});
do {y' <- sync (swap x M'); ...}
```

then `M'` is substituted for `y` and `M` is substituted for `y'` in the continuation code (elided by `...`s).

```
type swapChannel tau = channel (tau, channel tau)

swap :: swapChannel tau -> tau -> event tau
swap ch msgOut = guard (do {
  inCh <- new;
  choose [
    wrap (receive ch)
      (\x -> let (msgIn, outCh) = x in do {
                sync (transmit outCh msgOut);
                return msgIn
              } ),
    wrap (transmit ch (msgOut, inCh))
      (\_ -> sync (receive inCh)) ]
} )
```

Communication over a swap channel is already highly nondeterministic, since one of the ends must choose to send its message first (and accept the message from the other end later), while the other end must make exactly the opposite choice. We add further nondeterminism by spawning multiple pairs of `swap` on the same swap channel.

Buffered channels This program implements and uses a *buffered channel abstraction*, as described in (Reppy 1992). Intuitively, a buffered channel maintains a queue of messages, and chooses between receiving a message and adding it to the queue, or removing a message from the queue and sending it.

Our library performs significantly better for all except one of these benchmarks—for the swap channels benchmark, the difference is only marginal. Note that in this case, our protocol possibly wastes some rounds by matching points that have the same synchronizer (and eventually canceling these matches, since such points can never be selected together). An optimization that eliminates such matches altogether should improve the performance of our implementation.

Beyond total running times, it should also be interesting to compare performance relative to each CML-style primitive, to pinpoint other possible sources of inefficiency. We defer a more detailed investigation of these issues, as well as a more robust account of implementation differences between the underlying languages (especially those of threads), to future work.

All the code that appears in this paper is available online at:

`http://code.haskell.org/cml/`

Additional resources on this project are available at (Chaudhuri 2009; Chaudhuri and Franksen 2009).

9. Related work

We are not the first to implement CML-style concurrency primitives in another language. In particular, Russell (2001) presents an implementation of events in Concurrent Haskell. The implementation provides guarded channels, which filter communication based on conditions on message values (as in Section 7). Unfortunately, the implementation requires a rather complex Haskell type for `event` values. In particular, a value of type `event tau` needs to carry a higher-order function that manipulates a continuation of type `IO tau -> IO ()`. Further, a critical weakness of Russell's implementation is that the `choose` combinator is asymmetric. As observed in (Reppy and Xiao 2008), this restriction is necessary for the correctness of that implementation. In contrast, we implement a (more expressive) symmetric `choose` combinator, following the standard CML semantics. Finally, we should point out that Russell's CML library is more than 1300 lines of Haskell code, while ours is less than 150. Yet, guarded communication as proposed by Russell is already implemented in our setting, as shown in Section 7. In the end, we believe that this difference in complexity is due to the clean design of our synchronization protocol.

Independently of our work, Reppy and Xiao (2008) recently pursue a parallel implementation of a subset of CML, with a distributed protocol for synchronization. As in (Reppy 1999), this implementation builds on ML machinery such as continuations, and further relies on a compare-and-swap instruction. Unfortunately, their `choose` combinator cannot select among `transmit` events, that is, their subset of CML cannot express selective communication with `transmit` events. It is not clear whether their implementation can be extended to account for the full power of `choose`.

Orthogonally, Donnelly and Fluet (2006) introduce *transactional events* and implement them over the software transactional memory (STM) module in Concurrent Haskell. More recently, Effinger-Dean et al. (2008) implement transactional events in ML. Combining all-or-nothing transactions with CML-style concurrency primitives is attractive, since it recovers a monad. Unfortunately, implementing transactional events requires solving NP-hard problems (Donnelly and Fluet 2006), and these problems seem to interfere even with their implementation of the core CML-style concurrency primitives. In contrast, our implementation of those primitives remains rather lightweight.

Other related implementations of events include those of Flatt and Findler (2004) in Scheme and of Demaine (1998) in Java. Flatt and Findler provide support for *kill-safe abstractions*, extending the semantics of some of the CML-style primitives. On the other hand, Demaine focuses on efficiency by exploiting communication patterns that involve either single receivers or single transmitters. It is unclear whether Demaine's implementation of non-deterministic communication can accommodate event combinators.

Distributed protocols for implementing selective communication date back to the 1980s. The protocols of Buckley and Silberschatz (1983) and Bagrodia (1986) seem to be among the earliest in this line of work. Unfortunately, those protocols are prone to deadlock. Bornat (1986) proposes a protocol that is deadlock-free assuming communication between single receivers and single transmitters. Finally, Knabe (1992) presents the first deadlock-free protocol to implement selective communication for arbitrary channel communication. Knabe's protocol appears to be the closest to ours. Channels act as locations of control, and messages are exchanged between communication points and channels to negotiate synchronization. However, Knabe assumes a global ordering on processes and maintains queues for matching communication points; we do not require either of these facilities in our protocol. Furthermore, as in (Demaine 1998), it is unclear whether the protocol can accommodate event combinators.

Finally, our work should not be confused with Sangiorgi's translation of the higher-order π-calculus (HOπ) to the π-calculus (Sangiorgi 1993). While HOπ allows processes to be passed as values, it does not immediately support higher-order concurrency. For instance, processes cannot be modularly composed in HOπ. On the other hand, it may be possible to show alternate encodings of the process-passing primitives of HOπ in π-like languages, via an intermediate encoding with CML-style primitives.

10. Conclusion

In this paper, we show how to implement higher-order concurrency in the π-calculus, and thereby, how to encode CML's concurrency primitives in Concurrent Haskell, a language with first-order message passing. We appear to be the first to implement the standard CML semantics for event combinators in this setting.

An interesting consequence of our work is that implementing selective communication *à la* CML on distributed machines is reduced to implementing first-order message passing on such machines. This clarifies a doubt raised in (Peyton-Jones et al. 1996).

At the heart of our implementation is a new, deadlock-free protocol that is run among communication points, channels, and synchronization applications. This protocol seems to be robust enough to allow implementations of sophisticated synchronization primitives, even beyond those of CML.

Acknowledgments Thanks to Cormac Flanagan for suggesting this project for his Spring 2007 *Concurrent Programming* course at UC Santa Cruz. Thanks to Martín Abadi, Jeff Foster, and several anonymous referees of Haskell'07 and ICFP'09 for their comments on this paper. Finally, thanks to Ben Franksen for maintaining the source package for this library at HackageDB. This work was supported in part by NSF under grants CCR-0208800 and CCF-0524078, and by DARPA under grant ODOD.HR00110810073.

References

R. Bagrodia. A distributed algorithm to implement the generalized alternative command of CSP. In *ICDCS'86: International Conference on Distributed Computing Systems*, pages 422–427. IEEE, 1986.

R. Bornat. A protocol for generalized Occam. *Software Practice and Experience*, 16(9):783–799, 1986. ISSN 0038-0644.

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):223–235, 1983. ISSN 0164-0925.

A. Chaudhuri. A Concurrent ML library in Concurrent Haskell, 2009. *Links to proofs and experiments at* `http://www.cs.umd.edu/~avik/projects/cmllch/`.

Avik Chaudhuri and Benjamin Franksen. Hackagedb cml package, 2009. Available at `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/cml`.

E. D. Demaine. Protocols for non-deterministic communication over synchronous channels. In *IPPS/SPDP'98: Symposium on Parallel and Distributed Processing*, pages 24–30. IEEE, 1998.

K. Donnelly and M. Fluet. Transactional events. In *ICFP'06: International Conference on Functional Programming*, pages 124–135. ACM, 2006.

L. Effinger-Dean, M. Kehrt, and D. Grossman. Transactional events for ML. In *ICFP'08: International Conference on Functional Programming*, pages 103–114. ACM, 2008.

M. Flatt and R. B. Findler. Kill-safe synchronization abstractions. In *PLDI'04: Programming Language Design and Implementation*, pages 47–58. ACM, 2004. ISBN 1-58113-807-5.

A. D. Gordon. *Functional programming and Input/Output*. Cambridge University, 1994. ISBN 0-521-47103-6.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

F. Knabe. A distributed protocol for channel-based communication with choice. In *PARLE'92: Parallel Architectures and Languages, Europe*, pages 947–948. Springer, 1992. ISBN 3-540-55599-4.

X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system documentation: Event module, 2008. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Event.html`.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1): 1–77, 1992.

S. L. Peyton-Jones and P. Wadler. Imperative functional programming. In *POPL'93: Principles of Programming Languages*, pages 71–84. ACM, 1993.

S. L. Peyton-Jones, A. D. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96: Principles of Programming Languages*, pages 295–308. ACM, 1996.

J. H. Reppy. *Concurrent programming in ML*. Cambridge University, 1999. ISBN 0-521-48089-2.

J. H. Reppy. *Higher-order concurrency*. PhD thesis, Cornell University, 1992. Technical Report 92-1852.

J. H. Reppy. First-class synchronous operations. In *TPPP'94: Theory and Practice of Parallel Programming*. Springer, 1994.

J. H. Reppy and Y. Xiao. Towards a parallel implementation of Concurrent ML. In *DAMP'08: Declarative Aspects of Multicore Programming*. ACM, 2008.

G. Russell. Events in Haskell, and how to implement them. In *ICFP'01: International Conference on Functional Programming*, pages 157–168. ACM, 2001. ISBN 1-58113-415-0.

D. Sangiorgi. From pi-calculus to higher-order pi-calculus, and back. In *TAPSOFT'93: Theory and Practice of Software Development*, pages 151–166. Springer, 1993.

Wikipedia. Sieve of Eratosthenes, 2009. See `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.