Programming Languages
The Semantics of Concurrent ML
Robert Harper
Spring, 1999

Semantics of CML

So far our description of CML has been very informal, based on a seat-of-the-pants conception of concurrent execution and synchronization.

In this lecture we will develop a formal operational semantics for CML that makes precise the semantics of CML.

For the sake of simplicity we will neglect the `withNack` feature of CML.

Semantics of CML

We will give a **transition semantics** for CML whose main ingredients are these:

- Dynamic allocation of channels.
- Dynamic spawning of threads.
- Synchronization of threads, including non-deterministic choice.

Semantics of References

To fix ideas we first give an operational semantics of CML with reference types. The main problem is to manage the set of assignable reference cells.

We will use two main techniques:

- **Program rewriting**. A complete program will consist of the collection of allocated reference cells and an expression to be evaluated relative to those cells.
- **Scope extrusion**. We will define an instruction that "hoists" locally-allocated cells to the top level.

MinML With References

These extensions to the syntax of MinML are needed for references:

Types	τ	$::=$	$\ldots \mid \tau\,\texttt{ref}$
Expressions	e	$::=$	$\ldots \mid \texttt{letref}\,m\,\texttt{in}\,e\,\texttt{end} \mid \texttt{!}e \mid e_1\texttt{:=}e_2 \mid e_1\texttt{;}\,e_2$
Values	v	$::=$	$\ldots \mid l$
Memories	m	$::=$	$l_1\texttt{=}v_1\,\texttt{and}\,\cdots\,\texttt{and}\,l_n\texttt{=}v_n$

In the expression $\texttt{letref}\,l_1\texttt{=}v_1\,\texttt{and}\,\cdots\,\texttt{and}\,l_n\texttt{=}v_n\,\texttt{in}\,e\,\texttt{end}$, the variables $l_1,\ldots,l_n$ are **bound** in e, and hence may be renamed at will.

The `letref` Expression

We think of the expression

$$\texttt{letref}\,l_1\texttt{=}v_1\,\texttt{and}\,\cdots\,\texttt{and}\,l_n\texttt{=}v_n\,\texttt{in}\,e\,\texttt{end}$$

as allocating a set of reference cells for use in an expression e.

An ML-style `ref` expression is defined as follows:

$$\texttt{ref}\,(e) = \texttt{let}\,\tau\texttt{:}x\,\texttt{be}\,e\,\texttt{in}\,\texttt{letref}\,l\texttt{=}x\,\texttt{in}\,l\,\texttt{end}\,\texttt{end}$$

That is, we evaluate e, allocate a reference cell with its value, and return that cell.

Complete Programs

A **program** p is an expression of the form

$$\texttt{letref}\,l_1\texttt{=}v_1\,\texttt{and}\,\cdots\,\texttt{and}\,l_n\texttt{=}v_n\,\texttt{in}\,e\,\texttt{end}$$

consisting of a global memory and an expression to be evaluated.

We **identify** programs that differ only in the names of locations, since these are just bound variables of a `letref` expression.

Evaluation With References

The idea of scope extrusion is to "hoist" locally-allocated reference cells into the global memory.

Example:

```
(let r be ref 0 in r := !r + 1 ; !r end) + 1
```

Expanding `ref 0`,

```
(let r be (letref l=0 in l end) in r := !r + 1 ; !r end) + 1
```

Evaluation With References

First, lift the `letref` to the outermost level:

```
letref l=0 in let r be l in r := !r + 1 ; !r end + 1 end
```

Second, evaluate the `let` by substitution:

```
letref l=0 in (l := !l + 1 ; !l) + 1 end
```


Evaluation With References

Third, evaluate the retrieval and assignment:

```
letref l=1 in ( () ; !l ) + 1 end
```

Fourth, evaluate the sequencing and fetch:

```
letref l=1 in 1 + 1 end
```

Finally, perform the addition:

```
letref l=1 in 2 end
```


Evaluation With References

The **final value** of the program is

```
letref l=1 in 2 end
```

That is,

- The value of the expression is `2`.
- The memory has been **side-effected** to allocate a cell named `l` whose contents is `1`.

Garbage Collection

Since the result value does not mention `l` we may **garbage collect** the reference to obtain:

```
letref in 2 end
```

with an empty memory and the value `2`.

Garbage Collection

Contrast this situation:

```
let r be ref 0 in fn x => !r + 1 end
```

This evaluates to

```
letref l=0 in fn x => !l + 1 end
```

We **may not** garbage collect the memory cell `l` in this case!

Circularity

Memories can be **circular**!

```
let r be ref (fn x=>x) in r := (fn x => (!r)(x)+1) end
```

evaluates to

```
letref l = fn x => (!l)(x)+1 in () end
```

with a **self-referential** memory.

(Using this, we can implement recursion by backpatching.)

Instructions

The primitive instructions are defined as transitions between **programs**.

The first rule is called **scope extrusion**:

$$\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ \texttt{letref}\ l\texttt{=}v\ \texttt{in}\ e\ \texttt{end}\ \texttt{end}$$
$$\mapsto$$
$$\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{and}\ l\texttt{=}v\ \texttt{in}\ e\ \texttt{end}$$

where l is distinct from $l_1, \ldots, l_n$.

The scope of the locally-allocated reference cell is "extruded" to encompass the entire program.

Instructions

The rules for retrieval and assignment are as follows:

$$\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ \texttt{!}l_i\ \texttt{end}$$
$$\mapsto$$
$$\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ v_i\ \texttt{end}\quad (1 \leq i \leq n)$$

$$\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ l_i\texttt{:=}v_i'\ \texttt{end}$$
$$\mapsto$$
$$\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_{i-1}\texttt{=}v_{i-1}\ \texttt{and}\ l_i\texttt{=}v_i'\ \texttt{and}\ l_{i+1}\texttt{=}v_{i+1}\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ \texttt{()}\ \texttt{end}$$

Instructions

The pure MinML instructions carry over directly as program transitions. For example,

$$\begin{array}{c}\texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ \texttt{apply(}v\texttt{,}v'\texttt{)}\ \texttt{end}\\ \mapsto\\ \texttt{letref}\ l_1\texttt{=}v_1\ \texttt{and}\ \cdots\ \texttt{and}\ l_n\texttt{=}v_n\ \texttt{in}\ [v,v'/f,x]e\ \texttt{end}\\ (v = \texttt{fun}\ f\ (x\texttt{:}\tau_1)\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end})\end{array}$$

Search Rules

The following **search rules** determine the next instruction to execute:

$$\frac{\texttt{letref}\ m\ \texttt{in}\ e_1\ \texttt{end} \mapsto \texttt{letref}\ m'\ \texttt{in}\ e_1'\ \texttt{end}}{\texttt{letref}\ m\ \texttt{in}\ \texttt{letref}\ l\texttt{=}e_1\ \texttt{in}\ e_2\ \texttt{end}\ \texttt{end} \mapsto \texttt{letref}\ m'\ \texttt{in}\ \texttt{letref}\ l\texttt{=}e_1'\ \texttt{in}\ e_2\ \texttt{end}\ \texttt{end}}$$

$$\frac{\texttt{letref}\ m\ \texttt{in}\ e\ \texttt{end} \mapsto \texttt{letref}\ m'\ \texttt{in}\ e'\ \texttt{end}}{\texttt{letref}\ m\ \texttt{in}\ \texttt{!}e\ \texttt{end} \mapsto \texttt{letref}\ m'\ \texttt{in}\ \texttt{!}e'\ \texttt{end}}$$

and similarly for the other constructs of the language.

NB: evaluation of the sub-expression can change the memory!

Semantics of CML

A CML **program** p has the form

$$\texttt{letchan}\ k_1\texttt{,}\ \ldots\ \texttt{,}k_m\ \texttt{in}\ l_1\texttt{:}e_1 \parallel \cdots \parallel l_n\texttt{:}e_n\ \texttt{end}$$

The pairwise distinct variables $k_1, \ldots, k_m$ are the **active channels**.

The pairs $l_i\texttt{:}e_i$ are the **active threads** with pairwise distinct **thread identifiers** $l_1, \ldots, l_n$.

Structural Equivalence of Programs

We **identify** programs that differ only in

- The names of the channels or the thread id's.
- The ordering of threads in the program.

This means we can think of a program as consisting of a **set** of channels governing a **set** of threads.

Channel Allocation

New channels are allocated by scope extrusion:

$$\texttt{letchan}\, k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:channel()} \parallel l_2\texttt{:}e_2 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_0\texttt{,}k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:}k_0 \parallel l_2\texttt{:}e_2 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$

where k_0 is distinct from $k_1, \dots, k_m$.

Channel Collection

Unused channels may be garbage collected:

$$\texttt{letchan}\, k_0\texttt{,}k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:}e_1 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:}e_1 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$

provided that k_0 does not occur free in any e_i.

Spawning Threads

Threads are spawned as follows:

$$\texttt{letchan}\, k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:spawn(}v\texttt{)} \parallel l_2\texttt{:}e_2 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_0\texttt{:}v\texttt{()} \parallel l_1\texttt{:}l_0 \parallel l_2\texttt{:}e_2 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$

where l_0 is distinct from $l_1, \dots, l_n$.

Waiting for Threads

A thread wait completes when the thread completes:

$$\texttt{letchan}\, k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:wait(}l_2\texttt{)} \parallel l_2\texttt{:exit()} \parallel l_3\texttt{:}e_3 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1\texttt{,}\dots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:()} \parallel l_2\texttt{:exit()} \parallel l_3\texttt{:}e_3 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$

Garbage Collecting Threads

Completed threads may be collected:

$$\texttt{letchan}\, k_1\texttt{,}\ldots\texttt{,}k_m\, \texttt{in}\, l_0\texttt{:exit()} \parallel l_1\texttt{:}e_1 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1\texttt{,}\ldots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:}e_1 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$

provided that l_0 does not occur free in any e_i.

Synchronization of Events

Events are **activated** by synchronizing on them using `sync`.

Two threads **synchronize** on a channel when

- One has an active receive event pending on that channel.
- The other has an active send event pending on that channel.

Guards, wrappers, and choice are handled on a per-thread basis. Choice is fully non-deterministic.

Synchronization of Events

Rendezvous is achieved by matching send and receive events on a channel:

$$\texttt{letchan}\, k_1\texttt{,}\ldots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:sync(recv(}k\texttt{))} \parallel l_2\texttt{:sync(send(}k\texttt{,}v\texttt{))} \parallel \cdots\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1\texttt{,}\ldots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:}v \parallel l_2\texttt{:()} \parallel l_1\texttt{:}e_1 \parallel \cdots \parallel l_n\texttt{:}e_n\, \texttt{end}$$

Synchronization of Events

Wrappers are evaluated after synchronization:

$$\texttt{letchan}\, k_1\texttt{,}\ldots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:sync(wrap(}v_1\texttt{,}v_2\texttt{))} \parallel \cdots\, \texttt{end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1\texttt{,}\ldots\texttt{,}k_m\, \texttt{in}\, l_1\texttt{:}v_2\texttt{(sync(}v_1\texttt{))} \parallel \cdots\, \texttt{end}$$

Synchronization of Events

Guards are applied when synchronizing on them:

$$\texttt{letchan}\, k_1, \ldots, k_m \,\texttt{in}\, l_1\texttt{:sync(guard(}v\texttt{))} \parallel \cdots \texttt{ end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1, \ldots, k_m \,\texttt{in}\, l_1\texttt{:sync(}v\texttt{())} \parallel \cdots \texttt{ end}$$

Synchronizaton of Events

Choice is resolved non-deterministically:

$$\texttt{letchan}\, k_1, \ldots, k_m \,\texttt{in}\, l_1\texttt{:sync(choose(}v_1\texttt{,}v_2\texttt{))} \parallel \cdots \texttt{ end}$$
$$\mapsto$$
$$\texttt{letchan}\, k_1, \ldots, k_m \,\texttt{in}\, l_1\texttt{:sync(}v_i\texttt{)} \parallel \cdots \texttt{ end} \quad (i = 1, 2)$$

Sychronization of Events

The always-enabled event may always be synchronized:

$$\texttt{letchan}\, k_1, \ldots, k_m \,\texttt{in}\, l_1\texttt{:sync(always)} \parallel \cdots \texttt{ end} \mapsto$$
$$\texttt{letchan}\, k_1, \ldots, k_m \,\texttt{in}\, l_1\texttt{:()} \parallel \cdots \texttt{ end}$$

The never-enabled event never synchronizes: `sync(never)` is "stuck".

Search Rules

In addition to these primitive instructions we also require search rules for evaluating sub-expressions.

For example,

$$\frac{\texttt{letchan}\, k_1, \ldots \,\texttt{in}\, l_1\texttt{:}e \parallel \cdots \texttt{ end} \mapsto \texttt{letchan}\, k_1, \ldots, k_{m+1}, \ldots \,\texttt{in}\, l_1\texttt{:}e' \parallel \cdots \texttt{ end}}{\texttt{letchan}\, k_1, \ldots \,\texttt{in}\, l_1\texttt{:recv(}e\texttt{)} \parallel \cdots \texttt{ end} \mapsto \texttt{letchan}\, k_1, \ldots, k_{m+1}, \ldots \,\texttt{in}\, l_1\texttt{:recv(}e'\texttt{)} \parallel \cdots \texttt{ end}}$$

NB: evaluation of the subexpression can allocate channels!

Handling Fairness

The semantics outlined above admits choices that lead to a dead end. For example, we may step from `choose(never,always)` to `never`, blocking further progress.

Should this much choice be admitted in the semantics?

- Yes, to account for all possible schedulers.
- No, should restrict choices to eliminate pointless transitions.

Some aspect of fairness must always be left implicit, *e.g.*which thread to choose next.

Handling Fairness

The goal is to resolve non-deterministic choice only in favor of an enabled event.

This can be achieved by defining a **normal form** for events according to the following rules (where α is an atomic event):

$$\begin{array}{rcl}
\alpha & \equiv & \texttt{wrap}(\alpha\texttt{,id}) \\
\texttt{wrap(wrap}(v\texttt{,}v_1)\texttt{,}v_2) & \equiv & \texttt{wrap}(v\texttt{,}v_2 \circ v_1) \\
\texttt{wrap(choose}(v_1\texttt{,}v_2)\texttt{,}v) & \equiv & \texttt{choose(wrap}(v_1\texttt{,}v)\texttt{,wrap}(v_2\texttt{,}v)) \\
\texttt{choose(guard}(v_1)\texttt{,}v_2) & \equiv & \texttt{guard(fn () in choose}(v_1\texttt{()},v_2)\texttt{ end)} \\
\texttt{choose(never,}v) & \equiv & v \\
\texttt{choose}(v_1\texttt{,}v_2) & \equiv & \texttt{choose}(v_2\texttt{,}v_1) \\
\texttt{choose(choose}(v_{1,1}\texttt{,}v_{1,2})\texttt{,}v_2) & \equiv & \texttt{choose}(v_{1,1}\texttt{,choose}(v_{1,2}\texttt{,}v_2))
\end{array}$$

Handling Fairness

Up to this equivalence relation, events have the form

1. $\texttt{guard}(v)$, where v is a value.
2. $\texttt{choose(wrap}(\alpha_1\texttt{,}v_1)\texttt{,wrap}(\alpha_2\texttt{,}v_2))$, where α_1 and α_2 are atomic events.

Handling Fairness

We can then define synchronization between choices to simultaneously

- Make a choice of matching wrapped, atomic events.
- Synchronize and apply the wrapper.

Handling Fairness

This approach rules out some obviously bad choices, but leaves unresolved

- Choosing between `always` and another event.
- Choosing which thread to execute next.

Experience indicates that it is best to leave some aspect of fairness implicit to allow implementations to make expedient choices.

Further Issues

This semantics can be extended to handle abort events, with some technical difficulty.

- Must introduce "anonymous" abort events corresponding to **non-selection** of an event.
- Must allow synchronization on abort events to occur once selection is made.

Further Issues

We did not discuss the static semantics and type safety of references or threads.

- Interesting problems arise when giving a type to `letref` due to circularities in the memory.
- Progress and preservation are more delicate in the non-deterministic (concurrent) setting.