Hot-Sliding in ML

Andrew W. Appel
Princeton University
appel@princeton.edu

December 1994

Abstract

To upgrade a embedded program *while it is running* is a daunting task. But the Standard ML of New Jersey system has several features that enable a clean solution: "applicative" module linking, interface inheritance, concurrency, and garbage collection of machine code.

1 Replacing running modules

A long-running embedded program, such as the software running a telephone switch, must sometimes be upgraded in the field. Bugs must be fixed, or new features added. Furthermore, the upgrade must be done while the program is still running: the downtime required to boot up a new version of the software is unacceptable, and there are certain long-running transactions (such as telephone calls) that cannot be interrupted. How can a software module be replaced while it is running, without interrupting the work it is doing?

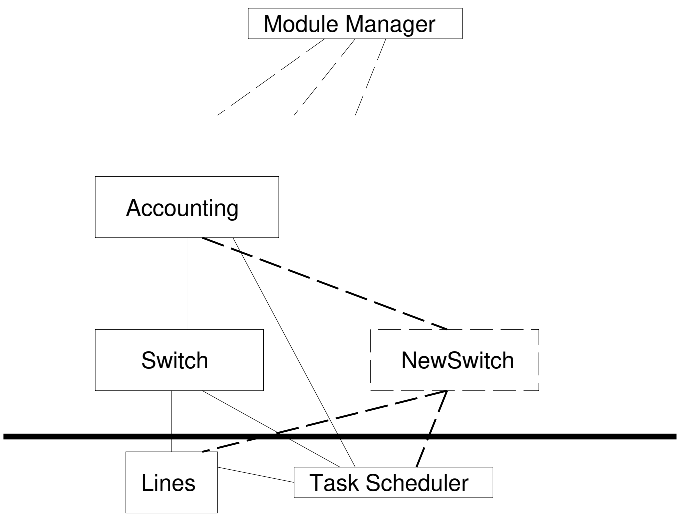

Figure 1: Installing a new module

Consider the example diagrammed in figure 1. The *Lines* module interfaces to the hardware, connecting telephone lines to each other and disconnecting them. The *Switch* module scans for requests (when users pick up the phone and dial), instructs *Lines* what connections to make, and remembers the duration of each call. At the completion of each call, *Switch* sends *Accounting* a billing record. This is in a concurrent programming language, let us suppose, so that *Lines, Switch* and *Accounting* can each have several lightweight tasks performing various operations. The are in a shared address space, so that they can communicate without crossing expensive hardware protection barriers; and the program is structured as a set of modules communicating with each other by ordinary procedure calls.

After several phone calls have been initiated, we wish to install a new version of the *Switch* module that uses different internal algorithms and data structures, and perhaps even supports new features. But the old *Switch* module is still handling some active phone calls; these might be modem connections that don't want to disconnect for weeks to come.

How can *NewSwitch* be linked to *Accounting* and *Lines* while *Switch* is still linked? How can *Switch* continue to run smoothly while *NewSwitch* also runs?

A conventional program linker will edit the machine code of the *Switch* module so that procedure calls in *Switch* can jump directly into the *Lines* module and *Accounting* modules; and will edit the machine code of *Accounting* so that *Accounting* can jump directly into *Switch*. When it is time to install *NewSwitch*, it is easy enough to edit *NewSwitch's* code for calls into *Lines* and *Accounting*, but how can *Accounting* be modified to jump to *NewSwitch* while *Switch* is still running and current connections still require *Accounting*? Finally, when all the calls handled by *Switch* eventually complete, how can the code and data for that module be removed from the system?

I will outline a solution to these problems, using the *Standard ML* language and the *Standard ML of New Jersey* compiler and runtime system for that language.

Acyclic module structure Standard ML has an acyclic module structure. That is, the import/export graph of module dependencies has no loops. If module *Accounting* imports (makes use of) functions and values from module *Switch*, then *Switch* does not import components of *Accounting*. In cases where two-way communication is necessary, *Accounting* can import a function *run* from *Switch*, and then pass a function to

Switch.run:

```
structure Switch . . .

   fun  run(tell) =
        connect calls; disconnect calls;
        tell(this_call) . . .
end


structure Accounting . . .
    fun record(call) =
            write call to accounting file
         ⋮
    Switch.run(record)
end
```

Applicative module linking Pascal or Scheme can be compiled re-entrant, so that one activation of a function f does not interfere with another activation of the same function because the two activations keep their data in different places. Each activation access its data through a *stack pointer* register instead of through hard-coded addresses.

Standard ML of New Jersey extends the principle of re-entrance even to module linking; it does not link modules together by editing their machine code.

The principle of "free variables" is used. If a module *Accounting* imports (fields of) two other modules *Switch* and *Scheduler*, then these other modules are free variables of the *Accounting* module. Many programming languages, such as Algol and Pascal, allow access to nonlocal (but not quite global) variables, usually implementing this access through the technique of *static links* (also called *access links*). Languages such as Scheme and ML that support higher-order functions with nested scope use *closures*, which are quite similar.

Whenever any function within *Accounting* runs, it can expect that a designated machine register points to a closure record pointing to *Switch* and *Scheduler*. In fact, this closure points to *the closures of Switch and Scheduler*. For *Accounting* to call a function f in *Switch*, it must fetch *Switch*'s closure from its own closure record. *Switch*'s closure contains the machine-code entry address for f, as well as the closures for modules that *Switch* imports.

This is illustrated in figure 2. The machine code segments do not point to each other. Instead, each procedure call in the code knows at what offset in the closure to find its target address. The extra indirection at every external procedure call is not too costly.

This module-linking technology has been used successfully for several years in the SML/NJ system [1], and for many years before that in other systems whose implementers were too lazy to implement "real" link-editing.

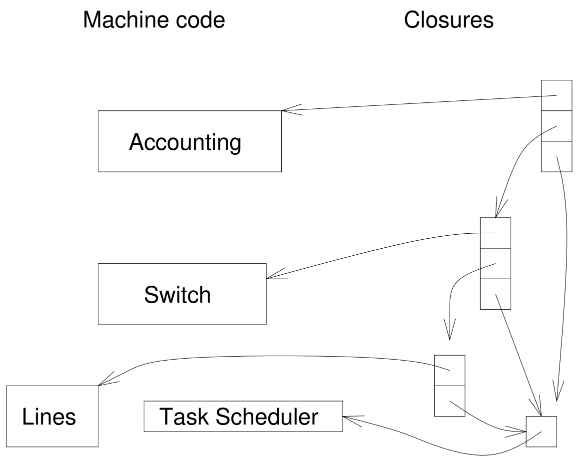

Figure 2: Closures for module linking. The named boxes are machine code, which contain no outgoing pointers; the pointer-containing boxes are closures. When a function in *Switch* is executing, a closure-pointer in registers gives access to any other modules it might want to call.

Now suppose we want to install *NewSwitch*. The structure of closures is shown in figure 3. The new closures are installed without disturbing the old closures at all. The module-linking of *Accounting* is re-entrant: there are two versions now running, sharing machine code but pointing to two different closures for the *Switch* modules.

Inheritance Suppose that the *Switch* module is improved to export a richer interface, with the intention of implementing a new *Accounting* module at a later date that can take advantage of this. The signature thinning (interface inheritance) feature of the ML module system makes this easy. As long as *Switch* exports a superset of what *Accounting* imports, everything works very straightforwardly.

Concurrent programming How can two different *Switch* modules, or two similar *Accounting* modules, run at the same time? The most straightforward approach is to have them run as coroutines or lightweight tasks. Several concurrent extensions to ML have been proposed and implemented, and almost any of them will suffice. They have in common a *ready queue* of runnable threads, with each thread represented as a closure containing machine-code resumption point and various other saved register values. Any realistic concurrent programming system will have a mechanism for synchronization between threads, with associated queues of threads waiting to synchronize, but for the purposes of this *Switch* example I will not need this level of detail.

Let us assume that the *Scheduler* module exports three functions:

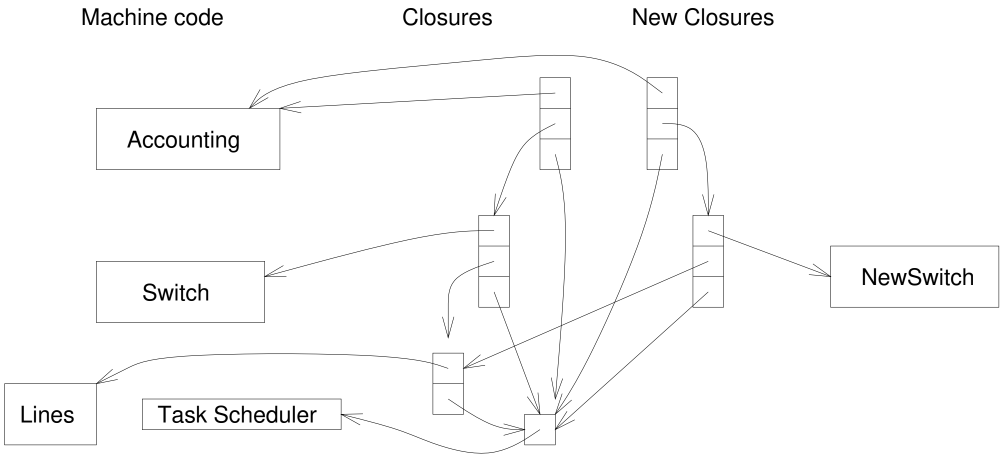

Figure 3: Replacing a module. The new closures at right allow a new version of *Switch*, and another invocation of the existing *Accounting*, to run without destroying the old *Switch* and *Accounting*.

```
structure Scheduler =  . . .
   val spawn : (unit->unit) -> unit
   val yield : unit -> unit
   val exit: unit -> 'a
```

where *Scheduler.spawn(f)* forks a new thread to run the function f, and returns immediately, not waiting for f to finish; *Scheduler.yield()* allows other tasks to make progress; and *Scheduler.exit()* terminates the thread that calls it.

When the *Switch* and its *Accounting* module are running, there may be several threads executing the machine code of these modules. When these threads are on the ready queue, their "saved state" closures will naturally point to closures for *Switch* and *Accounting* (among other things).

When *NewSwitch* is installed and a new closure for *Accounting* is built, a new thread is spawned to run it. This in turn may spawn other threads; the new threads will share the same ready queue with the older threads of the *Switch* module. (We dare not replace the *Scheduler* itself!)

Garbage collection of code The *Switch* module must stop handling new calls, and it should gradually release all of its old calls and associate resources so that *NewSwitch* can take over. *Switch* must be programmed so that we can send it a message, or call a function, or set a variable so that it knows that it must stop. At this point, it stops handling new calls; eventually all the old calls terminate, and it can tell *Accounting* that it's done. At this point, the (old invocation of) *Accounting* can also terminate.

When *Switch* terminates, its machine code should be deleted from the system. How is this done? The ML system has automatic garbage collection of any unreachable data, including machine-code segments. Only those data reachable from registers are kept; everything else is reclaimed. So perhaps the better question is, what keeps the *Switch* code alive?

First, there is a module manager that handles the installation of new or replacement modules, such as *NewSwitch*. This manager points to the closures for modules such as *Accounting* or *NewSwitch* for any of two reasons: (1) the pointer to any module (e.g. *NewSwitch*) imported by another (e.g. *Accounting*) if a replacement module *(NewAccounting)* must be installed; (2) any replaceable module *(Accounting, NewSwitch)* has a *stop* function that may need to be invoked later, to tell it to begin releasing its resources.

But after the installation of *NewSwitch*, the module manager does not need a pointer to *Switch*, so this is not what is keeping *Switch*'s machine code alive.

Runnable threads on the *Scheduler*'s ready queue contain pointers to the machine code in which they will resume (see figure 4). If *Switch* has not yet finished its shut-down actions (because certain long calls have not completed), then there will be a closure on the ready queue (or on some other synchronization queue, or actually executing in registers) pointing at the machine code for *Switch*. Or, if that thread is executing a subroutine in some other module, then the code for *Switch* is reachable by some chain of return-address pointers.

When the *Switch* code eventually notices that its last call has completed, and it has released all of its phone lines and other resources, it calls *Scheduler.exit()*, which removes it from any queues. The scheduler invokes some other ready thread, and the machine code of the *Switch* module can finally be garbage-collected.

2 A worked example

To demonstrate these ideas, I have prepared a complete, working example.

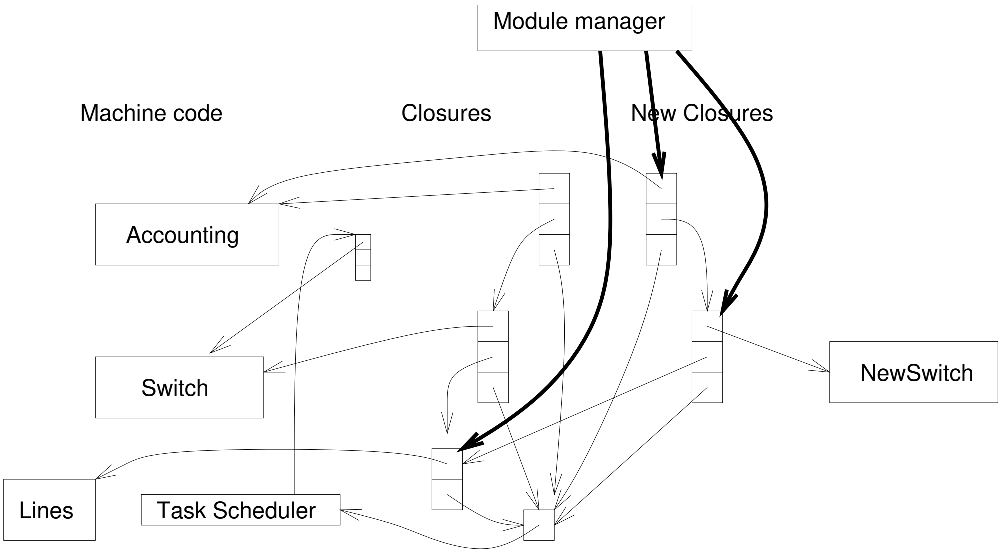

Figure 4: Scheduler keeps code live. The old *Switch* is not garbage collected, because a runnable task points to its closure.

2.1 Permanent modules

The infrastructure of permanent modules cannot be replaced with new versions.

Scheduler The *Scheduler* is the smallest possible abstraction of Concurrent ML[3] or ML-Threads[2]. It lacks synchronization and communications primitives, which are not needed for this simple example. The use of first-class continuations to implement co-routines is well known [4].

```
signature SCHEDULER =
 sig
   val spawn : (unit -> unit) -> unit
   val yield : unit -> unit
   val exit: unit -> 'a
 end
```

```
structure Scheduler :  SCHEDULER =
struct

 val rdyQ = ref (nil: unit cont list)
 fun put k = rdyQ := !rdyQ @ [k]
 fun get () = let val k::rest = !rdyQ
                in rdyQ := rest; k
               end

 fun exit () = throw (get()) ()

 fun yield() = callcc(fn k =>
                     (put k; exit()))

 fun spawn f =
         (callcc (fn k =>
             (put k; f (); exit()));
          ())
end
```

Lines We assume a simple interface to the telephone switch hardware. There are N phone lines; any one of them can be “on hook” (inactive) or “off hook” (active). The *Lines* module exports a list of the phone *lines* it handles; a special line *busysignal* is connected to the appropriate tone generator. The *get_dial* function reports what number the user has dialed as he picked up the phone. Finally, any two lines may be *connect*ed or *disconnect*ed.

The module *Lines* is abstract, so users cannot manipulate the data structures directly. But there is a “back door:” a non-abstract interface *Simulate_Lines* so that we can simulate users making telephone calls.

This interface is a very simplified view of how telephones work; for example, it assumes that people pick up the receiver and dial a number in one atomic operation; and that instead of ringing, phones have a little device to push the receiver off the hook automatically!

```
signature LINES =
sig
  type line
  val lines : line list
  val lookup: int -> line
  val number: line -> int
  val busysignal : line
  val offhook : line -> bool
  val connected: line -> bool
  val get_dial : line -> int
  val connect: line * line -> unit
  val disconnect: line * line -> unit
end
```

```
structure Simulate_Lines =
struct
 val N = 10
 type line = int

 val busysignal = 1

 val lines = let fun f(i) =
                   if i=N then nil
                   else i :: f(i+1)
              in f 2
             end

 fun lookup(i) = i
 fun number(i) = i

 val offhook' = Array.array(N,false)
 fun offhook i = Array.sub(offhook',i)

 val dial = Array.array(N,0)
 fun get_dial i =
       let val d = Array.sub(dial,i)
        in Array.update(dial,i,0);
           d
       end

 val connections = Array.array(N,0)

 fun connected(i) =
          Array.sub(connections,i) > 0

 fun connect(i,j) =
     (Array.update(connections,i,j);
      Array.update(connections,j,i);
      Array.update(offhook',j,true))

 fun disconnect(i,j) =
     (Array.update(connections,i,0);
      Array.update(connections,j,0))
end
```

```
abstraction Lines : LINES =
                     Simulate_Lines
```

Simulate_Input This module generates phone calling activity in the *Lines* module.

```
structure Simulate_Input =
struct

 structure L = Simulate_Lines

 val rand = Rand.mkRandom 1.498572
 fun random(n) =
        Rand.range(0,n-1) (rand())

 fun run() =
  let val c1 = 2+random(L.N-2)
      and c2 = 2+random(L.N-2)
      and any  = random(5)
   in if any>0 then ()
      else if L.offhook(c1)
        then Array.update(
                 L.offhook',c1,false)
      else (Array.update(
                 L.dial,c1,c2);
            Array.update(
                 L.offhook',c1,true));
      Scheduler.yield();
      run()
  end

end
```

Manager The module manager installs new and replacement versions of the telephone switch software. In this simple example, a new software module is installed by compiling and evaluating it; a more realistic example would install a previously compiled module.

The manager reads one line from an input stream; if the line is N dots, then the manager waits N time steps (by yielding control N times to the round-robin scheduler). If the line is a piece of ML code, the manager calls on the compiler (with *use_stream*) to compile and evaluate it.

```
structure Manager =
struct

  val doit = ref (fn()=>())

  fun invoke f = doit := f

  fun wait 0 = ()
    | wait n = (Scheduler.yield();
                wait(n-1))

  fun run infile =
   let val s = input_line infile
    in if size s = 0
       then ()
       else if substring(s,0,1) = "."
        then (wait(size s); run infile)
        else (use_stream(open_string s);
              let val f = !doit
               in doit:=(fn()=>());
                  Scheduler.spawn f
              end;
              Scheduler.yield();
              run infile)
   end

end
```

2.2 Replaceable modules

Switch The *Switch* module exports two interface functions, *run* and *stop*. Invoking *run(account,done)* starts the switch software with two arguments, each of which is a function. Invoking *stop()* tells the switch software not to handle any new calls, but to continue handling old ones.

account is called to generate billing information at the completion of each call; *done* is called when the last old call terminates. Presumably these two functions will be passed to *run* by the *Accounting* module.

Switch.run loops, calling in turn two functions, *newCalls* to initiate new calls (when it notices that phone lines have just gone off hook), and *hangups* to disconnect calls (when it notices that phone lines have gone back on hook). But if the *live* variable is false, then *newCalls* will be skipped. All the *stop* function has to do is set *live* to false.

When *live* is false and the last active call ends, *done* is called and then *run* exits.

```
signature SWITCH =
sig
  val run: (int*int->unit) * (unit->unit)
              -> unit
  val stop: unit -> unit
end
```

```
functor Switch(structure Lines: LINES)
                                 : SWITCH =
struct

 val live = ref false

 fun newCalls (line::lines,calls) =
    if Lines.offhook(line) andalso
        not(Lines.connected(line))
      then
       let val number = Lines.get_dial(line)
           val try = Lines.lookup(number)
           val other = if Lines.offhook(try)
                       then Lines.busysignal
                       else try
        in Lines.connect(line,other);
           (line,other) ::
                    newCalls(lines,calls)
       end
      else newCalls(lines,calls)
   | newCalls (nil,calls) = calls


 fun hangups(accounting,(c1,c2)::calls) =
    if not (Lines.offhook(c1))
       orelse not (Lines.offhook(c2))
       then (Lines.disconnect(c1,c2);
             if Lines.number c2 <>
                 Lines.number Lines.busysignal
             then accounting(Lines.number c1,
                             Lines.number c2)
             else ();
             hangups(accounting,calls))
       else (c1,c2) ::
               (hangups(accounting,calls))
   | hangups(_,nil) = nil


 fun run(accounting,done) =
  let fun loop calls =
        (Scheduler.yield();
         if !live
         then loop(
               hangups(
                accounting,
                newCalls(Lines.lines,calls)))
         else case hangups(accounting,calls)
               of nil => ()
                | calls' => loop calls')
   in live := true; loop nil; done()
  end

 fun stop() =  live := false

end
```

What happens if there are two instances of *Switch* running simultaneously? Each one "owns" a subset of the connected lines; when a line goes "off hook," *newCalls* can grab it. It's important that the body of *newCalls*, from the *offhook* test to the *connect* operation, be atomic, but in the simple co-routine version shown here that's accomplished just by omitting any *yield* call.

Ordinarily, only one *Switch* is running; and when two are more are running, only one will be *live*. The non-*live* switch modules will not be aquiring new resources (connected lines), and will be gradually releasing them.

Accounting The *Accounting* module just writes the billing information to a file. The *done* function (passed to *Switch.run*) writes to the file the information that a switch has terminated. The *id* parameter just identifies the invocation of *Accounting* that writes each entry in the file.

```
signature STOPPABLE =
sig
    val start: unit -> unit
    val stop : unit -> unit
end

functor Account(
         structure Switch: SWITCH
         val id : string) : STOPPABLE =
struct

 val file = open_append "Accounting"

 fun accounting (c1:int, c2:int) =
     let val s = implode[id, ": ",
          makestring c1, " called ",
          makestring c2, "\n"]
      in output(file,s);
         output(std_out,s)
     end

 fun done() = (print id;
              print ": exiting\n";
              close_out file)

 fun start() = Scheduler.spawn(
       fn()=>Switch.run(accounting,done))

 fun stop() = Switch.stop()

end
```

Test input

After loading all these modules, we invoke *Manager.run* on a file containing:

```
Manager.invoke Simulate_Input.run;
structure S1 = Switch(structure Lines = Lines);
structure A1 = Account(structure Switch = S1
                       val id = "A1");
Manager.invoke A1.start;
........................................
use "switch2.sml";
.............................
structure S2 = Switch(structure Lines = Lines);
..................................
structure A2 = Account(structure Switch = S2
                       val id = "A2");
.......................
A1.stop();
Manager.invoke A2.start;
.......................................
.......................................
.......................................
.......................................
.......................................
.......................................
....................................
```

This starts the phone-activity simulator; creates a closure *S1* for the *Switch* module; creates a closure *A1* for the *Accounting* module; and tells *A1* to start.

Then it waits 40 time steps while phone calls are initiated. Next it compiles a new version of the *Switch* module that uses different internal data structures; creates a closure *S2* for the new switch module; and waits some more.

Then it creates a new accounting closure *A2* using the new *Switch* module, but doesn't start it running; and waits.

Now it tells *A1* to stop, and *A2* to start; and waits another 200 time steps.

The output looks like this:

invoke Simulate˙Input.run
```
structure S1 : SWITCH
structure A1 : STOPPABLE
```
invoke A1.start
```
A1: 5 called 3
A1: 6 called 8
functor Switch
```
(compile switch2.sml)
```
A1: 3 called 0
structure S2 : SWITCH
A1: 6 called 3
structure A2 : STOPPABLE
A1: 2 called 8
```
stop A1

start A2
```
A1: 3 called 9
A2: 4 called 5
A2: 8 called 7
A2: 3 called 4
A2: 4 called 5
A1: exiting
A2: 8 called 9
A2: 3 called 2
A2: 5 called 8
```

3 Conclusion

This entire example was done as an ordinary user program in Standard ML of New Jersey, without any special access to the internals of compiler or runtime system. Features of ML and SML/NJ that make this possible are:

Re-entrant module linking. Allows two activations of the same module to run simultaneously.

Acyclic module system. When a new *Switch* is installed, re-entrant invocations of any modules above it in the hierarchy must be created. This would be a problem in the presence of cycles.

Concurrency or continuations. Multithreading is essential to allow the old and new code to run simultaneously.

Interface thinning (inheritance). Allows piecemeal enriching of module interfaces. Upgrading implementations sometimes means upgrading interfaces, too.

Garbage collection. SML/NJ's garbage collector can reclaim code space as well as data space.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–24, New York, 1987. Springer-Verlag.

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

[3] John H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[4] Mitchell Wand. Continuation-based multiprocessing. In *Conf. Record of the 1980 Lisp Conf.*, pages 19–28, New York, August 1980. ACM Press.