Thimble — Threads for MLj

Ian Stark

Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh

Ian.Stark@ed.ac.uk

Abstract. Thimble is a concurrency library for MLj, the open source ML-to-Java bytecode compiler. Thimble allows MLj programs to run multiple concurrent threads that safely communicate through typed channels, without having to reach down for the underlying Java `synchronize`, `wait` and `notify` actions. In addition Thimble *events* provide efficient multiway synchronization, so that a thread can choose between inputs from several different channels without the need for busy waiting. Both channels and events are first-class values, and Thimble provides a range of operations for users to roll their own varieties while maintaining data integrity in a multithreaded environment. Thimble itself is written entirely in MLj, building these high-level features on to the basic concurrency facilities guaranteed by any Java virtual machine.
This paper presents the Thimble library; some examples of how it can be used; a review of its implementation; and a comparison with related systems like Concurrent ML, Concurrent Haskell and indeed Java itself.

1 Introduction

The reasons for the current popularity of Java are many and varied, but they include its extensive standard libraries and easy portability. The MLj compiler aims to bring these advantages directly into Standard ML, by compiling functional programs into Java bytecodes. Of course in doing so it complements Java's features with the strengths of ML: polymorphism, type inference, first class functions, and so on.

Thimble is a library for MLj that does the same for concurrency. All Java virtual machines support threads, monitors and semaphores according to a fixed specification, independent of the underlying architecture. This provides the raw material for concurrent execution: Thimble then lifts it to a collection of safe high-level operations in the style of process algebra.

The resulting channels and events are familiar from Concurrent ML and the like [12]. Where Thimble differs from previous systems is that it does not itself contain any code for context switching or thread scheduling. All this is delegated to the Java virtual machine (the *JVM*). As ever, following a standard cuts both ways: we get portable multithreaded code for comparatively little effort; but the actual details of concurrent execution are out of our control. This seems to be a trade-off well worth making, especially as it allows Thimble to concentrate on concurrent programming at the language rather than the machine level.

In particular, the JVM specification is intentionally loose in several areas. This is good for building efficient virtual machines on a variety of systems, but programmers then face some uncertainty if they want to write safe, portable code. Thimble can factor out this uncertainty. For example, in Java threads communicate by shared memory, and if the programmer forgets a lock then data may be corrupted. Under Thimble, ML threads communicate over typed channels with no need for locks — when you have the data, it's yours.

1.1 Plain MLj

Figure 1 shows a complete MLj "Hello World" program. This compiles into just under 3K of bytecode; notice that Java itself is completely bypassed. The only remaining hint of an object is

```
structure HelloWorld  :>  sig  val main : string option array option -> unit  end  =
struct

  fun main args = print "Hello World\n"

end
```

To compile this program under MLj, place in the file `HelloWorld.sml` and type `make HelloWorld` at the MLj command line. To execute, type `run`. The MLj compiler is available from `http://www.dcs.ed.ac.uk/home/mlj/`. This site carries executable binaries for Linux, Solaris and Windows, together with complete source code ready for compilation onto other systems as required.

Fig. 1. A minimal stand-alone MLj program.

in the type of `main`: any JVM pointer can be `NULL`, so the command-line arguments are passed as a (possibly null) array of (possibly null) strings.

1.2 Threads

MLj programs can dynamically create any number of concurrently executing threads, where the precise degree of concurrency depends on the Java virtual machine where the code is eventually run. In particular this may involve free pre-emptive thread switching: Thimble remains safe whatever.

As an example, the program of Figure 2 spins off two threads, which each present their own greeting. The order in which the messages are printed is arbitrary, and may vary from one run to another. Within the program a pair of functions `greeterA` and `B` take a thread identifier `t`

```
(*-----------------------------------------------------------------------------------------
   greeterA : thread -> unit    thread : (thread -> unit) -> thread gen
   greeterB : thread -> unit       new : 'a gen -> 'a
 -----------------------------------------------------------------------------------------*)
open Thread Generator

fun main args = let fun greeterA t = print "Hello from thread A\n"
                    fun greeterB t = print "Hello from thread B\n"
                in
                    new (thread greeterA); new (thread greeterB);
                    ()
                end
```

Fig. 2. Multithreaded MLj.

and then execute some code. This identifier is needed for certain control operations: its real role here is as something to abstract over to delay evaluation. The function `thread` wraps up the desired computation and gives a *thread generator*. Finally, applying `new` to this actually forks off a separate thread of execution to do the work.

Unsurprisingly, the `Thread` structure also contains a variety of other functions for manipulating threads, such as raising or lowering priorities. The `Generator` structure however needs a little more explanation. Several notions in Thimble are *generative*, in that we can request a succession of “fresh” instances. The two most obvious are threads and channels: when we ask for two fresh channels, we expect them to be distinct, but behave identically in all respects. For a formal analysis of generativity, see Stark [14]. To highlight this property we introduce the type `'a gen` for something that generates fresh `'a`'s. The operation `new` then triggers a generator to do its thing. Of course inside an `'a gen` is just a function of type `(unit->'a)`, but by marking it as a generator we make some informal claims for it; principally that each application will terminate, and produce a fresh item.

1.3 Channels

Where the threads of Figure 2 lead independent lives, Figure 3 adds communication, with one thread passing a message to the other over a *channel*. The `Channel` structure provides the type constructor `'a chan` for channels that carry values of type `'a`. This is matched by a generator `chan : ('a chan) gen` that creates fresh channels of every type on demand.

Execution of the program runs like this: `new chan` creates a fresh channel `c` for carrying strings; a `sender` and `receiver` thread are declared and started up; the first sends a greeting on `c`; the second collects it and prints it out. The `get` operation blocks until a value is ready; there is also a non-blocking `poll : 'a chan -> 'a option`.

```
(*-----------------------------------------------------------------------------------------
    chan : 'a chan gen     sender : thread -> unit    write : 'a chan -> 'a -> unit
       c : string chan   receiver : thread -> unit      get : 'a chan -> 'a
 -----------------------------------------------------------------------------------------*)
open Thread Channel Generator

fun main args = let val c = new chan
                    fun sender t = write c "Hello World\n"
                    fun receiver t = print (get c)
                in
                    new (thread sender); new (thread receiver);
                    ()
                end
```

Fig. 3. Communication between threads.

Thimble channels are by default asynchronous: once a message is sent, the sending thread may continue without waiting for collection. All messages will get through without duplication, but there are no guarantees about their order of arrival. Other kinds of channel are also available.

```
AckChannel.ackChan : ('a chan) gen
Padding.paddedChan : ('a chan) gen
  Pacing.pacedChan : ('a chan) gen
```

The first of these, *acknowledged* channels, capture synchronous communication: the sender cannot proceed until the receiver has collected the message. The other two are useful for matching processes working at different rates. A *padded* channel guarantees not to lose values, but may duplicate them; reading from a padded channel always succeeds at once. A *paced* channel on the other hand will never duplicate, but will drop values if supply outstrips demand (compare the *skip channels* of Concurrent Haskell [11, §3.3]).

All of these create values of the standard `'a chan` type, and there are facilities for the user to create their own kinds of channel by tying together a *sink* and a *source*.

As well as channels, it is possible to use shared memory for communication in Thimble, but care is required. Ordinary ML references are fine within a single thread of control, but with multiple threads they become less useful for two reasons:

– A write in one thread followed by a read in another may retrieve an incomplete value, due to caching of memory updates by the JVM.
– There is no way to atomically modify the value in a reference cell.

Thimble provides *safe* references of type `'a sref` to solve the first problem, and *dynamic* references of type `'a dref` for the second.

1.4 Events

The `get` action on a channel stops a thread as it waits for just one thing to happen. Thimble also offers *events* which allow a thread to specify some more complex set of conditions that it awaits. The `merge` function in Figure 4 uses a choice event to repeatedly gather data from two input channels onto one output channel. One input carries integers, the other floating-point values, and both are transformed into strings before being sent on.

```
(*-----------------------------------------------------------------------------------------
    ini : int chan      out : string chan       read : 'a chan -> ('a->'b) -> 'b event
    inr : real chan                           choose : 'a event list -> 'a
 -----------------------------------------------------------------------------------------*)
open Channel Event

fun merge ini inr out =
    let val s = choose [ read ini (fn i => "Integer: " ^ Int.toString i),
                         read inr (fn r => "Float: " ^ Real.toString r) ]
    in
      write out (s ^ "\n");
      merge ini inr out
    end
```

Fig. 4. Choosing between alternatives.

The structure `Event` introduces the `event` type constructor: if `e` is a value of type `'a event` then `execute e` will cause a thread to block until the chosen event happens, whereupon a value of type `'a` is returned. The nonblocking `attempt e` will return immediately, with a value only if one is available. As with threads, we separate the specification of an event from its enactment.

```
signature Event =
    sig  type event                     val map : ('a -> 'b) -> 'a event -> 'b event
                                        val choice : 'a event * 'a event -> 'a event
         val never : 'a event           val abort : (unit -> unit) -> 'a event -> 'a event
         val immediate : 'a -> 'a event           (* ...plus a few others. *)
   end
```

Fig. 5. Creating and combining events.

Figure 5 lists a selection of operations for constructing and combining events. The `never` event never happens; while `immediate v` is always ready to return value `v`. Function `Event.map` turns an `'a event` into a `'b event` by specifying a postprocessing function `('a->'b)`. The compound event `choice e e'` waits for whichever of `e` and `e'` happens first. Within a compound event, `abort a e` will call `a()` precisely when `e` is not chosen. This is useful, for example, when encoding a complex protocol as an event: an `abort` wrapper can roll back partial commitment.

In the program of Figure 4, basic events are provided by `read`; this pulls a value off a channel and applies a specified function to it. The `choose` function then waits for the first event of a given list; this is a derived operation in the `Event` structure.

```
fun choose es = let val e = foldr choice never es
                in
                    execute e
                end
```

Having postprocessing rolled into the `read` operation helps to match up the result types of different events.

1.5 Pipe of Eratosthenes

The program `epipe` in Figure 6 is a larger example of the Thimble library in use. This calculates prime numbers using a variant on the familiar sieve of Eratosthenes. The numbers from 2 to 200 are passed down a pipeline composed of several filters, each removing multiples of a given prime. Every filter is a separate thread, and the pipe grows at its tail end as new primes are found. At the same time the head of the pipe shortens as filters finish their work. The net effect is that although the primes are always successfully identified, in order, every run of the program creates a different and varying number of interacting threads.

With just one processor, a pipeline like this may be useful because it can offer greater throughput of data; it cannot actually calculate any more quickly. However, the Java virtual machine is designed to take advantage of multiple-processor systems too, and this would allow the same program to run proportionately faster. Linking the threads with channels means that Thimble has already taken care of all the necessary synchronization and shared-memory issues.

2 How it is done

The specification of the Java virtual machine is essentially a portable machine code, with implementations on a very wide range of real machines. As well as MLj there are JVM compilers for other languages, including Java (obviously), Scheme [2] and Ada [15]. It is quite a high-level machine code though: the JVM knows about objects, classes, and a sizeable standard library.

Every JVM can run multiple concurrent threads through the `java.lang.Thread` class. Depending on the implementation these may range from simple coroutines to truly simultaneous threads on multiple processors. This variation is reflected in the specification, which describes a rather loosely-coupled model of threads and shared memory [8, Chap. 8]. For example, updates to memory cells may be seen in a different order from different threads.

To manage synchronization, every JVM object carries both a lock and a semaphore. Any section of code can be marked as a *monitor*, so that a thread must wait to acquire the corresponding lock before entering it. Such a monitor can manage a shared resource through its semaphore: any number of threads can *wait* on this, and when the resource becomes free a *notify* action will wake up one of them.

In the Java language, these features are brought through exactly as they stand. For Thimble, we use them as a foundation and then aim a little higher.

```
open Thread Channel Event Generator

(* Head thread dispatches the numbers 2, 3, ... along a channel. *)
fun raw v limit out = (write out (SOME v); if v>=limit then (write out NONE;
                                                             print "\nAll sent\n")
                                                       else raw (v+1) limit out)

(* Middle threads filter numbers divisible by v. *)
fun mid v inn out = let val x = get inn
                    in
                        case x of SOME w => ( if (w mod v) = 0 then () else (write out x);
                                              mid v inn out )
                                | NONE => write out x
                    end

(* Final thread prints out primes and adds filters to the chain. *)
fun processed inn = let val x = get inn
                        val c = new chan : int option chan
                    in
                        case x of SOME v => ( print ((Int.toString v) ^ " ");
                                              new (thread (fn t=> mid v inn c));
                                              processed c )
                                | NONE => ()
                    end

(* Main program starts up a head thread to feed in the numbers, then becomes
   the tail of the chain itself. *)
fun main args =
    let val c = new chan : int option chan
    in
      new (thread (fn t=> raw 2 200 c));
      processed c
    end
```

Fig. 6. The Pipe of Eratosthenes: calculating prime numbers with a chain of collaborating threads.

```
signature Thread =
    sig  type thread    val thread : (thread -> unit) -> thread Generator.gen

         val getComputation : thread -> (thread -> unit)     val join : thread -> unit
         val currentThread : unit -> thread                  val sleep : Time.time -> unit
         val setPriority : thread -> int -> unit             (* ...and many others. *)
    end

signature Lock = sig  type lock  val lock : lock Generator.gen
                                 val run  : lock -> (unit->unit) -> unit
                 end

signature Flag = sig  type flag                val flag : flag Generator.gen
                      val test : flag -> bool  val wait : flag -> unit
                      val set  : flag -> unit  val test'n'set : flag -> bool
                 end
```

Fig. 7. Threads, locks and flags.

2.1 Threads, locks and flags

Just three structures in the Thimble library define classes directly: one for each of the concurrency features mentioned above. Values from the `Thread`, `Lock`, and `Flag` structures are shown in Figure 7. A Thimble *thread* is a proper subclass of `java.lang.Thread`, with an added field of type `(thread->unit)` that holds the computation a thread will execute. In Java it is rather hard to represent such a computation abstractly, as the necessary mechanism for closures — *inner classes* — is a late addition to the language. In ML, of course, first-class functions are just the thing. We also add one or two other refinements, such as a handler to make sure that exceptions never fall off the end of a thread unnoticed.

Thimble *locks* prevent interference between concurrent threads: any given lock can be held by at most one thread at a time. The `Lock` structure provides a generator for fresh locks, and a `run` operation. This takes a lock and a computation as arguments; it then waits to acquire the lock before carrying out the computation. This corresponds loosely to the `synchronized` method specifier in the Java language; although again ML's first-class functions add flexibility by letting us build the desired computation at run time.

A *flag* in Thimble is a one-shot semaphore. Any number of threads can `wait` on a flag, and they are then blocked until some other thread raises it. The operation `test` checks the status of a flag, `set` raises it, while `test'n'set` atomically does both.

These locks and flags are weaker than traditional monitors and semaphores — there is, for example, no `reset` operation for flags. However, they are sufficient as a basis for the higher-level events and channels to follow.

2.2 Events

The underlying type of a Thimble event is rather complex:

```
datatype 'a event = Event of flag -> unit -> (unit->'a) option .
```

Each of the three function spaces has a role to play. From left to right, they mark the points where an event prepares for something to happen; clears up afterwards; and presents its result. Matching this, the `execute` operation from the `Event` structure proceeds as follows.

– It creates a new flag and passes it to the event. This enables the event, and returns a *completion function* of type `unit -> (unit->'a) option`.
– It waits for someone to raise the flag.
– It calls the completion function. This clears up the aftermath of the event, and returns a possible *result computation*, of type `(unit->'a) option`.
– Finally, it calls the result computation to obtain an actual result.

The non-blocking `attempt` operation raises the flag itself, and then calls the completion function immediately. If the event has not yet happened then the result computation is simply `NONE`.

Each of the event constructors of Figure 5 then fits into this profile. The event `immediate v` raises the flag as soon as it has it; while `never` just throws it away. `Event.map` composes a function onto the result computation of an event, while `abort` adds code into the completion function. The `choice` combinator takes two events and relays their actions appropriately: each gets the flag, and on completion at most one should return a non-empty result computation.

The formulation of events in Thimble has to balance the desire for a powerful mechanism with what is possible on a general Java virtual machine. Without having direct control of the thread scheduler, there is only so much we can do. One aim is to make events low-maintenance: ours do not need separate worker threads, busy waiting, nor any logic for backing off from conflict.

2.3 Channels

The `Channel` structure in Thimble has two distinct faces. It both provides the general interface for using typed channels, and also an implementation of the default channel type. The interface part is fairly straightforward: a channel is a pair of a *source* and a *sink*; in turn an `'a source` is really an `'a event` that can be exercised repeatedly to fetch successive values from the channel; while an `'a sink` is a function of type `('a->unit)` that dispatches them. Conversely, given any values of these types you can build your own variety of `'a chan`.

Implementation of asynchronous channels uses the flags and locks described above. First the enhanced reference types `sref` and `dref` are defined using locks; a channel is then represented by a list stored in a `dref`. If the channel has values to be collected, then this is a list of them. If the channel is empty, then it is a list of *readers* interested in what comes next. An `'a reader` is in turn a `(flag * ('a option) sref)` pair: the flag belongs to the event waiting on the channel, and the `sref` is a place to put a value when it arrives.

The operations of reading and writing to a channel manage this list in appropriate ways. Some care is required: for example, a reader may have lost interest in a value by the time it arrives, because some competing event raised the flag first. Thus the `write` function actually scans the whole list of readers to find which, if any, still have their flags unset. Nevertheless the implementation has reasonably low overheads: all computation is done using the thread of the caller, and there is no busy waiting.

3 Pick your own

With the groundwork in place, this section explains briefly how custom channels can be built to order.

Section 1.3 mentions acknowledged channels, generated by `ackChan : ('a chan) gen`. These are implemented with an asynchronous channel carrying values of type `('a * unit chan)`. To write, the sender creates a fresh `unit chan` and sends it along with the value. The recipient picks off the `'a` value and passes a `()` as acknowledgement back down the `unit chan` to where the sender is waiting.

At first sight this approach seems rather profligate with channels: why not use a single reverse channel for confirmation messages? Unfortunately that would cause complications when there is more than one sender, and each must receive their own acknowledgement. For any protocol, the method of using a new channels to ensure that one message depends on another is a powerful technique for avoiding confusion. The standard FTP scheme for file transfer, for example, creates a separate internet connection for each file sent, even though it could just send it back down the same channel the request arrived. Taking this to an extreme, the πI calculus of Sangiorgi sends only fresh channels, and no data; despite this, it can encode most process calculi operations [13].

The generator `paddedChan` creates a channel that provides values as fast as a reader asks, duplicating them if necessary. To do so it augments a standard channel `c : 'a chan` with two others, `pad : 'a chan` and `empty : unit chan`. These extra channels obey an invariant, in that between them they always carry precisely one value; either the most recent value seen, on `pad`, or a token `()` on `empty` when no values have been sent at all. Writing to the channel `c` is as usual. Reading involves the following more complex event.

```
choice (read c   (fn x => (choose [read pad ignore, read empty ignore];
                           write pad x; x)),
        read pad (fn x => (write pad x; x)))
```

This chooses between two possibilities. First, if there is a value available on the main channel then this is the one we want; the value stored on either the `pad` or `empty` channels is removed, the new value stashed on `pad`, and then returned to the caller. If, on the other hand, there is nothing immediately on `c` then we look to the `pad` for the most recent value seen, and return that instead. Although it is not essential to the semantics, we do in some sense use the least padding required, as the `choice` operator gives priority to its first argument.

The `pacedChan` channel generator uses only two channels, `content : 'a chan` and `empty : unit chan`. Between them they hold at most one value at any time. To read we take an `'a` from `content` and return a `()` to `empty`; to write we take from whichever is full and write to `content`. The net result is that unread values are silently dropped from the channel, at a rate to suit the reader.

All three of these examples are coded in the Thimble library at a high level: no Java extensions, no locks or flags, just channels and events. They are then bundled as sinks and sources to match the standard `'a chan` interface.

Note that we have avoided creating surplus threads, using the resources of calling processes to do all the work. This is not vital, but reflects the fact that for the moment, JVM implementations cannot create fresh threads quite as cheaply as calling a method.

4 Comparison with related systems

There are several other systems that implement a process-algebra style of concurrency over a sequential functional language: Concurrent ML, Facile and Concurrent Haskell are perhaps the best known [12, 6, 11]; [1] give references to some others. Clearly Thimble draws heavily on these for its basic design, and from systems like CCS, the π-calculus and the join calculus for its foundations [9, 10, 5]. To some extent then, Thimble serves simply as a demonstration of how our understanding of this area is maturing: no radically new problems have been uncovered in the development of the library so far. There is more than this though: with maturity comes understanding; so for example it was known from the start what features could and could not be efficiently provided, given the resources of Java's underlying concurrency model.

The most significant feature of Thimble is what it doesn't do: all of the above systems have to do their own scheduling and basic synchronization. We don't, and by using the existing resources of the Java virtual machine we lay down a clear division between high-level language support for concurrent programming, and the raw materials necessary for concurrent execution. We believe that this is a sensible and meaningful separation of concerns, especially in the context of future heterogeneous distributed systems. The development of the UniForM toolkit over Concurrent Haskell has something of this divide, though not so sharply cut [7].

The other language which deserves comparison is Java itself. As mentioned earlier, Java brings through directly the concurrency features of the JVM. While programming with monitors and semaphores alone is quite possible, it does require a certain discipline from the programmer. In the absence of this, a general approach is to mark all methods as `synchronized` until problems go away, then take them off if a program runs too slowly or deadlocks. We hope that by getting these details right within Thimble, user-level programming would be freed from at least some of this.

A closer comparator for Thimble would be with a Java library for channels and events. Unfortunately this kind of general data handling library suffers badly from the constraints of the Java type system: without parametric polymorphism, all channels would have to carry objects, with explicit cast operations and run-time type checks at each end.

5 Future directions

An immediate goal is to integrate Thimble more closely with some of the standard Java libraries; in particular forging a link between channels and Java streams. This would then provide easy access to the file system and the wider internet through the standard `java.net` library.

In the reverse direction, the Thimble combinators for events and channels could be exported back into Java. As noted above, this would involve some loss of value, as Java does not have parameterised types[1]. Its extension GJ does though, and it could make full use of `'a event` and `'a chan`. (GJ would stand for "Generic Java", except that the J-word is trademarked [3].)

More ambitiously, we look to Thimble supporting distributed and even mobile MLj code. The Thimble combinators are all "distribution ready" in that they can safely be implemented for simultaneous threads on remote machines, without changing their semantics: essentially this follows from the existing loose specification of the JVM and the decision not to allow mixed choice. For storage and transmission of objects between machines, the Java libraries support *serialization* of objects into byte strings, and these apply as well to MLj types as Java classes. One problem is to standardise ML data representation: at the moment the MLj optimiser enthusiastically reformats datatypes to fit each program, and this must be reined in a little if one program is to talk to another.

In all of these, the JVM and its standard libraries fix the basic ingredients: what MLj and Thimble can add is a level of abstraction that supports robust and intuitive programming.

Acknowledgements

Although Thimble is in written in pure MLj, it did provide a stress test for some of the more exotic parts of the compiler and its interaction with the JVM. I am most grateful to the MLj team of Nick Benton, Andrew Kennedy and George Russell for their support and forbearance in managing my numerous queries and bug reports.

References

1. D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Conference Record of POPL '92*. ACM Press, 1992.
2. P. Bothner. Kawa - compiling dynamic languages to the Java VM. In *Proceedings of the USENIX 1998 Annual Technical Conference*, 1998. See also `http://sourceware.cygnus.com/kawa/`.
3. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA '98*. ACM Press, 1998. See also `http://www.cs.bell-labs.com/~wadler/pizza/gj/`.
4. *Conference Record of POPL '96*. ACM Press, 1996.
5. C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *POPL '96* [4].
6. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
7. E. W. Karlsen. The UniForM concurrency toolkit and its extensions to Concurrent Haskell. In *Proceedings of the 1997 Glasgow Workshop on Functional Programming*, September 1997.
8. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, April 1999. Online edition at `http://java.sun.com/docs/books/vmspec/`.
9. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
10. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, May 1999.
11. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL '96* [4].
12. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of PLDI '91*. ACM Press, June 1991.
13. D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(1–2):235–274, 1996.
14. I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994.
15. S. T. Taft. Programming the internet in Ada 95. In *Reliable Software Technologies - Ada Europe '96*, Lecture Notes in Computer Science 1088. Springer-Verlag, 1996. See also `http://www.appletmagic.com/`.

[1] Parameterised types are consistently in the top five requested language extensions on Sun's Java bug parade, so there is hope yet.