HAL
open science

Light weight concurrency in OCaml: continuations, monads, events, and friends

Christophe Deleuze

To cite this version:

Christophe Deleuze. Light weight concurrency in OCaml: continuations, monads, events, and friends. 2010. hal-00493213

HAL Id: hal-00493213

https://hal.science/hal-00493213v1

Preprint submitted on 18 Jun 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Light weight concurrency in OCaml: continuations, monads, events, and friends

Christophe Deleuze

April 2010

Abstract

We explore various ways to implement (very) light weight concurrency in OCaml, in both *direct* and *indirect* style, and compare them to system and VM threads approaches. Three simple examples allow us to examine both the coding style and the performances. The cost of context switching, thread creation and the memory footprint of a thread are compared. The trampolined style of programming seems to be the best both at CPU and memory demands.

1 Introduction

Concurrency is a property of systems in which several computations are executing "simultaneously", and potentially interacting with each other. Concurrency doesn't imply that some hardware parallelism be available but just that the computations (that we'll call "threads" in this text) are "in progress" at the same time, and will evolve independently. Of course, threads also need to be able to exchange data.

Besides making potentially easier the exploitation of hardware parallelism,[1] threads allow overlapping I/O and computation (while a thread is blocked on an I/O operation, other threads may proceed) and support a concurrent programming style. Many applications can be expressed more cleanly with concurrency.

Operating systems provide concurrency by time sharing of the CPU between different processes or threads. Scheduling of such threads is preemptive, *i.e.* the system decides when to suspend a thread to allow another to run. This operation, called a context switch, is relatively costly [15]. Since threads can be switched at any time, synchronisation tools such as locks must generally be used to define atomic operations. Locks are low level objects with ugly properties such as breaking compositionnality [20].

Threads can also be implemented in user-space without support from the operating system. Either the language runtime [1] or a library [5, 4] arranges to schedule several "threads" running in a single system thread. In such a setting, scheduling is generally cooperative: threads decide themselves when to suspend to let others execute. Care must be taken in handling I/O since if one such thread blocks waiting for an I/O operation to finish the whole set of threads is blocked. The solution is to use only non blocking I/O operations and switch to another thread if the operation fails. This approach:

- removes the need for most locks (since context switches can only occur at predefined places, race conditions can (more) easily be avoided),
- allows systems with very large numbers of potentially tightly coupled threads (since they can be very light weight both in terms of memory per thread and computation time per context switch),
- can be used to give the application control on the scheduling policy [16].

On the other hand the programmer has to ensure that threads indeed do yield regularly.

In this paper we study several ways to implement very light weight cooperative threads in OCaml without any addition to the language. We first describe thread operations in Section 2. Three

[1] Which OCaml threads currently can't because of the non concurrent garbage collector, by the way.

example applications making heavy use of concurrency are then presented in Section 3. The various implementations are described in Sections 4 and 5. Performance comparisons based on the example applications are given in Section 6. The paper closes with a conclusion and perspectives (Section 7).

2 Principles

2.1 Basic operations

All implementations will provide the following operations :

spawn takes a thunk and creates a new thread for executing it. The thread will actually start running only when the *start* operation is invoked.

yield suspends the calling thread, allowing other threads to execute.

halt terminates the calling thread. If the last thread terminates, *start* returns.

start starts all the threads created by *spawn*, and waits.

stop terminates all the threads. *start* returns, that is, control returns after the call to *start*.

Most systems providing threads do not include something like the *start* operation: threads start running as soon as they are spawned. In our model, the calling ("main") code is not one of the threads but is suspended until all the threads have completed. It is then easy to spawn a set of threads to handle a specific task and resume to sequential operations when they are done. However, this choice has little impact on most of what we say in the following.

2.2 Thread communications

We allow threads to communicate through MVars and synchronous FIFOs. Introduced in Haskell [18], MVars are shared mutable variables that provide synchronization. They can also be thought as one-cell synchronous channels. An MVar can contain a value or be empty. A tentative writer blocks if it already contains a value, otherwise it writes the value and continues. A tentative reader blocks if it is empty, otherwise it takes (removes) the value and continues. Of course blocked readers or writers should be waken up when the MVar is written to or emptied.

We define α *mvar* as the type of MVars storing a value of type α. The following operations are defined:

make_mvar creates a fresh empty MVar.

put_mvar puts a value into the MVar.

take_mvar takes the value out of the MVar.

In the following we assume that only one thread wants to write and only one wants to read in a given MVar. This is not a necessary restriction, though. Roughly, MVars will be defined as a record containing three mutable fields:

- v of type α *option*: contains the stored value if any,
- *read* of some *option* type: contains information on the blocked reader if any,
- *write* of some option type: contains information on the blocked writer and the value it wants to write, if any,

but the details will vary with each implementation.

Contrary to an MVar, a *synchronous FIFO* can store an unlimited amount of values. Adding a value to the FIFO is a non blocking operation while taking one (the one at the head of queue) is blocking if the queue is empty. Operations are similar to MVar's:

make_fifo creates a fresh empty FIFO.

put_fifo adds a value into the FIFO.

take_fifo takes the first value out of the FIFO.

3 Three example applications

We now describe three example applications that will allow us to get a feel of the programming style required by our model, and to collect some performance data on the various implementations. All three examples are process networks.

3.1 Kpn

We consider a problem treated by Dijkstra, and solve it by a *Kahn process network*, as described in [10]. One is requested to generate the first n elements of the sequence of integers of the form $2^a 3^b 5^c$ $(a, b, c \geq 0)$ in increasing order, without omission or repetition. The idea of the solution is to think of that sequence as a single object and to notice that if we multiply it by 2, 3 or 5, we obtain subsequences. The solution sequence is the least sequence containing 1 and satisfying that property and can be computed as illustrated on Figure 1. The thread *merge* assumes two increasing sequences of integers as input and merges them, eliminating duplications on the fly. The thread *times* multiplies all elements of its input FIFO by the scalar a. Finally the thread x prints the flow of numbers and put them in the three FIFOs.[2]

All threads communicate and synchronize through MVars, except that the x thread itself writes its data in three FIFOs for the *times* threads to take it. The computation is initiated by *put*ting the value 1 in the *m235* MVar so that x starts running. Such a computation can be expressed as a *list comprehension* in some languages, such as Haskell[3] [19].

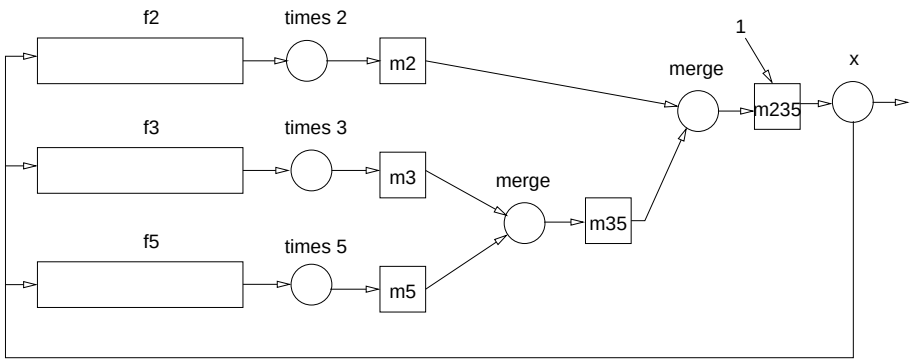


Figure 1: Kpn (threads are circles, MVars squares, FIFOs rectangles)

3.2 Eratosthene sieve

Our second example is Eratosthene sieve. The sieve as a set of concurrent threads is also described in [10] (where it is said to appear the very first time in [17]). A variant can also be found in [9]. The program is structured as a chain of threads exchanging messages.

integers is the *generator*, it sends out all integers starting from 2,

filter n transmits only the numbers it receives if they are not multiple of its n parameter,

sift creates and inserts a new filter in the chain, for each number received,

[2]This description is borrowed from the cited article.

[3]The Haskell code could be `s = 1:merge [ x*2 | x <- s ] (merge [ x*3 | x <- s ] [ x*5 | x <- s ])` with `merge` defined appropriately.

output prints the numbers it receives.

Figure 2: Sieve as a chain of concurrent threads

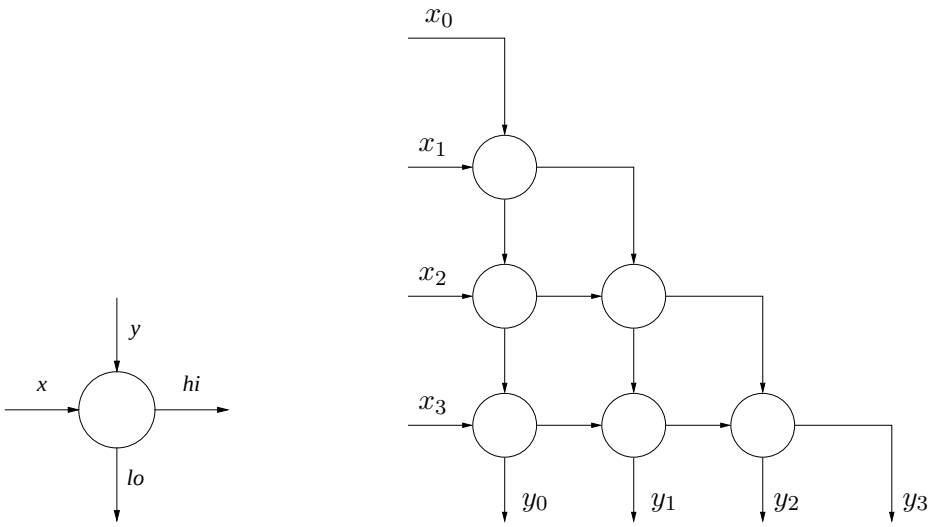
Thus the sieve builds as a chain of *filter* threads with a generator (*integers*) on the left and an expander (*sift*) preceding the consumer (*output*) on the right. Figure 2 shows the threads at startup and after the number 2 has been found to be a prime.

The communication channels between the threads will be implemented by MVars. The code is particularly simple.

3.3 Concurrent sort

Our last example is a concurrent sort described in [9]. As pointed out by the authors, both bubble sort and insertion sort are sequentialized versions of this concurrent algorithm.

This sorting algorithm is made of a network of simple *comparator* threads, each of which is used to sort a pair of values from two input MVars to two output MVars. Such a comparator with inputs x and y and outputs *hi* and *lo* is shown on Figure 3(a). Figure 3(b) shows a network for sorting a group of 4 values.



(a) Comparator (b) Sorting network

Figure 3: Concurrent sort

MVars will be used to store the initial and final values, as well as for the communication between the comparators.

4 Direct style implementations

When programming in *direct style* the primitives will have the signatures shown on Figure 4. Note that some of the operations are potentially blocking. The source code for the examples is given in appendix B. In the following we'll see direct style implementations providing primitives with these exact signatures.

val *yield* : *unit* → *unit*
val *spawn* : (*unit* → *unit*) → *unit*
val *halt* : *unit* → *unit*

val *start* : *unit* → *unit*
val *stop* : *unit* → *unit*

(a) Basic primitives

type α *mvar*
val *make_mvar* : *unit* → α *mvar*
val *take_mvar* : α *mvar* → α
val *put_mvar* : α *mvar* → α → *unit*

type α *fifo*
val *make_fifo* : *unit* → α *fifo*
val *take_fifo* : α *fifo* → α
val *put_fifo* : α *fifo* → α → *unit*

(b) MVars and FIFOs

Figure 4: Direct style signatures for the thread primitives

4.1 preempt

Source code in Section A.1 page 17

We start with a heavyweight version making use of preemptive scheduling as provided by the OCaml `Thread` module. This program can be compiled to use system (*ie* kernel managed) threads or VM threads, this last choice being available only for bytecode executables.

MVars are implemented as a record containing the value (v field) as an option type, a CML-like [21] channel for synchronous events (provided by the `Event` OCaml module) and two booleans *read* and *write* indicating if a thread is blocked on an operation on this MVar. Additionnaly, each MVar has its own lock l (from the *Mutex* module) to ensure MVar manipulations are atomic.

A global lock *gl* is also used to ensure threads do no start doing their job before *start* is called.

The possible number of such threads is quited limited. On a "small" machine (running Linux kernel 2.6.24 i686 with 256 MB of memory) thread creation fails around the 381th one. This limit is much higher on the machines where the measurements presented in Section 6 were conducted. Hopefully the limits are much higher for VM threads. Of course we don't expect this implementation to be very efficient on our examples.

4.2 callcc

Source code in Section A.2 page 18

Since we want to be able to suspend a running thread and activate it again later we need some way to save the current thread state, or rather *continuation*. The continuation of a computation at some point is what remains to be done at this point, in other words the rest of the computation. It is represented by the *context* of the computation [3]. The control flow of a program can be treated in terms of continuations.

The *call − with − current − continuation* primitive (often abreviated as *call/cc*) was first defined in Scheme [11]. It captures (that is, makes a copy of) the current continuation and *reifies*[4] it into a value of type α *cont*. Thus, continuations, which in most languages are implicit, can be explicitly captured and manipulated (passed as parameters, saved in data structures etc) like any other value. These "first class continuations" can also be *throw*n (and given a parameter of type α), meaning that the current continuation is discarded and replaced with the thrown one, so that execution resumes at the point where the continuation was captured[5]. Continuations can be used to implement all sorts of manipulations of the control flow, including multi-threading.

This implementation is designed along the lines described in [7]. There is no scheduler proper, rather each yielding thread enqueues its continuation before dequeueing and throwing the next one. The queue (of type *queue_t*) also stores distinctively the initial continuation e (of the call to *start*) to be thrown when the queue becomes empty so that control returns after the call to *start*.

We could also use a scheduler by letting each yielding thread throw the scheduler continuation, which would consist in dequeuing the next thread and storing back its own continuation before throwing it, but we feel it would be more complex, slow, and offers no advantage.

[4] That is, makes it available in the program.

[5] There are some variations. In Scheme, for example, captured continuations are reified into functions. Thus there is no explicit *throw* operation, the continuation is thrown when the function is applied.

When an operation blocks on an MVar, the continuation of the thread is captured with *callcc* and is stored in the appropriate field.

Callcc for OCaml is provided by a library with a "very naive implementation" whose "performance is terrible" (it copies the whole stack) so that "use in production code is not advised"[14]. Last, it is only available for bytecode. It does work rather well for one of our simple examples, but performs very badly on the others, as we'll discuss later.

Note that we have to fool the typechecker with *Obj.magic* in *take_mvar* and *take_fifo* to ensure these functions are polymorphic. Otherwise, the call to *halt* makes it decide the function must return *unit* and the MVars loose their polymorphism.

4.3 dlcont

Source code in Section A.3 page 19

This version uses delimited continuations provided by the `caml-shift` library [22]. This is only available for bytecode executables (as it manipulates the VM stack). A delimited continuation (also called partial, composable, or sub continuation), is a prefix of the rest of the computation, represented by a delimited part of the context of the computation. Unlike regular continuations, delimited continuations return a value, and thus may be reused and composed. Delimited continuations have been introduced initially as a way to express the semantics of continuations in a denotational setting [6].

Several slightly different operators have been proposed in the litterature but the general idea is that such a continuation is delimited by first pushing a delimiter (often called a *prompt*) on the stack, and later capturing the continuation, up to the first prompt.

In this library, *push_prompt* pushes a prompt on the stack, marking the delimitation, while *take_subcont* turns the part of the stack up to (and not including) the first prompt into a (α, β) *subcont* value and removes it (including the prompt) from the stack.[6] Here α is the type of values that must be given when throwing the continuation, and β is the type of values returned by the continuation. *push_subcont* pushes (*i.e.* throw) a delimited continuation on the stack.

So our plan is:

- the scheduler pushes the prompt and starts a thread (either by calling a function or pushing the corresponding subcont),
- when yielding, the thread uses *take_subcont* to return its current continuation to the scheduler,
- the scheduler queues the subcont, dequeues the next one, pushes the prompt and the new subcont
- and so on...

Alternatively, when yielding, the thread could package the captured subcont into a function pushing it. The scheduler thus does not need to make a difference between "first time" threads (represented as functions) and "already running" threads (represented as subconts). It will just have to push the prompt and call the function. This is alright but it turns out that it's much more efficient to push the prompt and the continuation at the same time (the library provides the *push_prompt_subcont* function for that). So, rather than pushing the prompt inside the scheduler we will use a function that captures the subcontinuation delimited by the passed prompt and builds a function that pushes the prompt and this continuation. This happens to be the behavior of the *shift0* [12] operator (that removes the prompt from the stack and encloses the captured continuation with a prompt).

```
let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt_subcont p sk (fun () → c))))
```

which is an optimized[7] version of

[6] This is the behavior of the operator know as *control0* [12].

[7] Actually, the first version has a subtle memory leak, as explained in Appendix B of [13].

```
let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt p
      (fun () → push_subcont sk (fun () → c)))))
```

Since the prompt is pushed by the thread functions themselves rather than by the scheduler, we also need the "initial" functions to push it. We simply make the *spawn* function (that adds a thread in the queue) insert the *push_prompt* call at the beginning of the function. It also adds a call to *halt* at the end so that the prompt is removed when the thread terminates.

5 Indirect style implementations

The basic idea of *indirect style* is to write the threads so that the continuations are made explicit (as closures) at each potentially blocking point. This way, the continuation can be manipulated without the need for any continuation-capture primitive.

For example, taking a value out of an MVar, which in direct style is written:

```
let v = take_mvar m in ...
```

can instead be written as:

```
take_mvar m >>= fun v → ...
```

where >>= (pronounced *bind*) is the thread sequential composition operator.[8] This operator appears at cooperation points, between a potentially blocking operation and its continuation. The continuation is a closure that will be executed when the blocking operation will have completed. The parameter of the continuation will receive the result of the operation. Imperative loops must be turned into (tail-) recursive functions if they contain a blocking operation. One more operation is useful in indirect style : *skip*, the no-op. It is used in kpn and sieve whose source code (along with the one for sorter) is shown in appendix B.

Trampolined style (derived from *continuation passing style*), monadic style, event-based programming are all variants of the indirect style. We study them in the following.

5.1 tramp

Source code in Section A.4 page 20

Trampolined style [8] is a simple way to provide application level concurrency through cooperative scheduling. The idea is that the code is written so that a function is given explicitly its continuation as a closure. It can then manipulate it just like the direct style continuation-capture based versions. The code must be written in a way similar to continuation passing style [24], but the continuations need to be made explicit only at cooperation points.

Figure 5 shows the signature of the operations. As we can see, each potentially blocking operation is given (as an additional parameter) the (continuation) function to run when the operation has been performed. Note that *put_fifo* does not take such a continuation parameter since this operation never blocks.

For example the *yield* instruction can be used as

```
...
  print_string "hoho";
  yield (fun () →
    print_string "haha";
    yield (fun () →
      print_string "hihi"
        ...
          ))
```

where the argument is the continuation, i.e. the function to be executed when the thread will be resumed. The "bind" infix operator noted >>= can be used as syntactic sugar to obtain a arguably

[8] This is borrowed from monad syntax but does not necessarily represent here "the" bind operator from monads.

```
val yield : (unit → unit) → unit
val spawn : (unit → unit) → unit
val halt : unit → unit

val start : unit → unit
val stop : unit → unit

val skip : (unit → unit) → unit
val ( >>= ) : ((α → unit) → unit) → (α → unit) → unit
```

(a) Basic primitives

```
type α mvar
val make_mvar : unit → α mvar
val take_mvar : α mvar → (α → unit) → unit
val put_mvar : α mvar → α → (unit → unit) → unit

type α fifo
val make_fifo : unit → α fifo
val take_fifo : α fifo → (α → unit) → unit
val put_fifo : α fifo → α → unit
```

(b) MVars and FIFOs

Figure 5: Trampolined style signatures for the thread primitives

more pleasant syntax. >>= takes two arguments and applies its second argument (the continuation) to the first one. Adopting an indentation more fitted to the intended "sequential execution" semantics, the above code is now written:

```
...
  print_string "hoho";
  yield >>= fun () →
  print_string "haha";
  yield >>= fun () →
  print_string "hihi"
  ...
```

Since a potentially blocking function, such as *yield* or *take_mvar*, takes its continuation as an additional parameter, it can execute it immediately or, if it needs to block, store it for later resuming before returning to the scheduler. Also note that this continuation can receive a value if the blocking operation produces a value (as *take_mvar* does).

Writing code so that continuations are explicit is often not as intrusive as one may feel initially. As the code of our example applications show, continuations often do not even appear explicitly. The only case of use in our examples is *print_list* in sorter. Actually, only when using procedural abstractions to build complex blocking operations do we need to manipulate explicitly the extra parameter. Even then it can be easy, as the following two functions show. The first one abstracts the operation of *yield*ing three times and the second one the reading of the value of an MVar by *take*ing it and re*put*ting it immediately (the correctness of this code is not obvious but the implementation of *take_mvar* ensures the continuation of the *take* is executed before any other thread once the value is removed from the MVar, so the operation is indeed atomic):

```
let yield3 k =
  yield >>= fun () →
  yield >>= fun () →
  yield >>=
  k

let read_mvar mv k =
  take_mvar mv >>= fun v →
  put_mvar mv v >>= fun () →
  k v
```

5.2 monad

Source code in Section A.5 page 21

This implementation uses *monads* [18] and is heavily inspired by `Lwt` (light weigth threads) [25], a cooperative thread library for OCaml. Monads are useful in a variety of situations for dealing with

effects in a functional setting. The code is written in *monadic style* which is actually quite close to the *trampolined* style.

Here's a brief description of the implementation. α *thread* is the type of threads returning a value of type α. It is a record with one mutable field denoting the current thread state, the state being a sum type. The three cases correspond respectively to a completed computation, a blocked computation with a list of thunks to execute when it will be completed, and a computation *connect*ed to another one (that means it will behave the same, being just a *Link* to it).

As can be seen on Figure 6, blocking operations return a value of type α *thread* and are thus easily recognized. Also, one more primitive is provided: *return* turns a value of type α in a value of type α *thread*.

```
type α thread
val return : α → α thread
val ( >>= ) : α thread → (α → β thread) →
 β thread

val skip : unit thread
val yield : unit → unit thread
val halt : unit → unit thread
val spawn : (unit → (unit thread)) → unit
val stop : unit → unit thread
val start : unit → unit
```

(a) Basic primitives

```
type α mvar
val make_mvar : unit → α mvar
val put_mvar : α mvar → α → unit thread
val take_mvar : α mvar → α thread

type α fifo
val make_fifo : unit → α fifo
val take_fifo : α fifo → α thread
val put_fifo : α fifo → α → unit
```

(b) MVars and FIFOs

Figure 6: Monadic style signatures for the thread primitives

Consider the evaluation of t >>= f, following the code for >>=. The first argument, not being a function, is evaluated immediately (the operations are executed). If it has completed it is of the form *Return* v, the value v is passed to f. If t has blocked, its value is *Sleep* w. A new sleeping thread *res* is created, a thunk is added to the w list, then *res* is returned. The thunk *connect*s *res* to the value of *bind* t f.

Thus, when t finally completes, the thunk is executed. *bind* t f is evaluated again with t being *Return* v so f is executed with v as argument and returns a value of type β *thread* to which *res* becomes *Link*ed.

As *bind* provides the waking-up of threads, they do not need to be put in *runq*, except those *yield*ing, since they explicitly give back control to the scheduler.

Also, we don't want *spawn*ed threads to start executing before *start* is invoked, so *spawn* makes them wait for completion of a sleeping thread (*start_wait*) that will be woken up by *start*. Threads *spawn*ed later will start running immediately since *start_wait* state will then be *Return* ().

The usage of *bind* also suppresses the need to explicitly manage continuations when composing thread fragments:

```
let yield3 () =
  yield () >>= fun () →
  yield () >>= fun () →
  yield ()

let read_mvar mv =
  take_mvar mv >>= fun v →
  put_mvar mv v
```

For this reason the code for the *print_list* function of sorter differs slightly from the code shown in Section B.6. Here it is:

```
let print_list mvs () =
  let rec loop mvs acc =
    match mvs with
    | [] → return acc
    | h :: t → take_mvar h >>= fun v → loop t (v :: acc)
  in
  loop mvs [] >>= fun l →
    List.iter (fun n → Printf.printf "%i␣" n) (List.rev l); halt ()
```

5.3 lwt

Source code in Section A.6 page 23

The previous implementation is basically a stripped down version of `Lwt`, which is much more elaborate, handling exceptions (mainly by adding a *Fail* state to the sum type), I/O etc. We thus provide an implementation based on `Lwt` where we only need to add the implementation of MVars and FIFOs.

As in `monad`, *spawn* threads wait for the *start_wait* thread waken-up by *start*. *finish* is a thread doing nothing, it just waits to be waken up and then terminates immediately. The `Lwt_unix` module provides a scheduler as the *run* function. It arranges for threads to be scheduled and executed until the passed thread terminates, at which point all the threads are terminated and the function returns.

We cannot stop the scheduler by simply raising an exception since uncaught exceptions in threads are ignored by `Lwt`. Instead, we wakeup the *finish* thread. However, the scheduler has no knowledge of the threads blocked on MVars or FIFOs. So the *stop* operation, in addition to waking up *finish*, sets the *do_stop bool ref* to *true*. The MVar and FIFO blocking operations check its value. If it is set, the thread terminates immediately with a *Fail* state. Yes, this is not very elegant.

There's one pitfall: the system doesn't stop (*start* does not return) after all threads have called *halt*. So, in sorter we make *print_list* call *stop* rather than *halt*. But the point is to compare `Lwt` performance to the other ones.

5.4 equeue

Source code in Section A.7 page 24

A popular paradigm supporting user level concurrency is event-driven programming. The OCaml-Net library [23] provides an `equeue` (for *event queue*) module in which handlers[9] are set up to process events.

We describe it briefly. First an *event system* (called *esys* here) must be created. Events are generated by an event source (here it is the function fun _ → () that generates none) but can also be added by the handlers themselves. Each event in presented to each handler, in turn, until one accepts it (or it is dropped if no handler accepts it). An handler rejects an event by raising the *Reject* exception. Otherwise the event is accepted. In case the handler, having accepted the event, wants to remove itself it must raise the *Terminate* exception.

The event system is activated by the *Equeue.run* function. The function returns when all events have been consumed and the event source does not add any.

In our implementation, handlers will always be "one shot", so they will always raise *Terminate* after having accepted an event. But before to do that, they will have registered a new handler representing the thread continuation.

A thread blocked on a MVar waits for a unique event allowing it to proceed. Blocked writers create a new *eventid* that they register in the control information of the MVar, along with the value they want to write. They then wait for a *Written* event with the correct id. Such an event will be generated when the value will have been actually put in the MVar, operation triggered by the *take*ing of the current MVar value by another thread. Blocked readers create a new *eventid* and wait for the *Read* event that will carry the value taken from the MVar. Again, this event will be generated when some thread puts a value in the MVar.

[9] *Callbacks* is another popular name for these.

The same applies for taking a value out of an empty FIFO. For *yield*ing, a thread creates a new *eventid*, registers its continuation as a handler to the *Go* event with the correct id, and adds this precise event to the system.

Since each blocking operation (in case it actually blocks) registers a new handler and then raises *Terminate*, threads must be running as handlers from the very beginning (for the *Terminate* exception to be catched by the event system). To ensure this, *spawn* registers the thread as a handler for a new *Go* event, then adds the event to the system.

There's one pitfall with this implementation: MVar operations are not polymorphic due to the event system being a monomorphic queue:

val *esys* : *'_a Equeue.t* = *< abstr >*

Thus, all MVars are required to store the same type of value, which is a serious limitation.

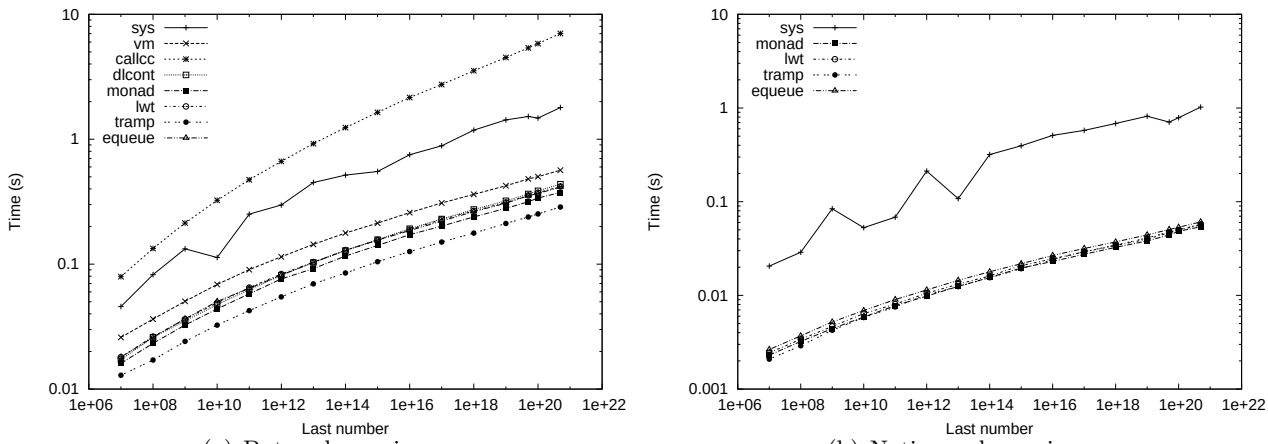
The code for the applications is strictly the same as for the previous implementation. Indeed, the threads are written in trampolined style and the event framework is used to build the scheduler. This implementation can be seen more as a exercise in style[10].

6 Performance

We have measured the time and memory needs for these implementations. The execution times are given by the *Unix.times* function, while the memory usage is measured as the *top_heap_words* given by the *quick_stat* function if the OCaml `Gc` module.

All the programs were run on a PC running the linux kernel version 2.6.24 with x86-64 architecture, powered by an Intel Core 2 Duo CPU clocked at 2.33 GHz with 2 GB of memory. Software versions are OCaml 3.10.2., `Lwt` 1.1.0, `caml-shift` july 2008, `equeue` 2.2.9.

All three examples are made of very simple threads that cooperate heavily. Since there's a fixed number of tightly coupled threads, `kpn` will give us indications on the cost of "context switching" between the threads. `sieve` is interesting because it constantly creates new threads. `sorter` has both a number of threads (created from the start) and a number of operations depending on the problem size. Moreover, its number of threads can easily be made huge (for sorting a list of 3000 numbers, there're about 4.5 million threads). To measure thread creation time alone, we will also run it with a parameter that terminates the program as soon as the sorting network has been set up.

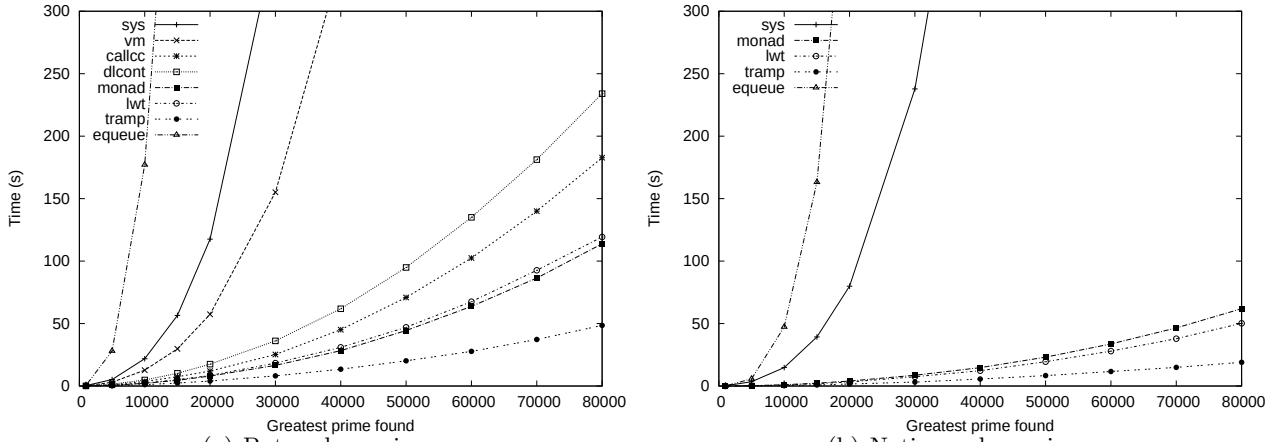


(a) Bytecode version (b) Native-code version

Figure 7: `kpn`, execution time

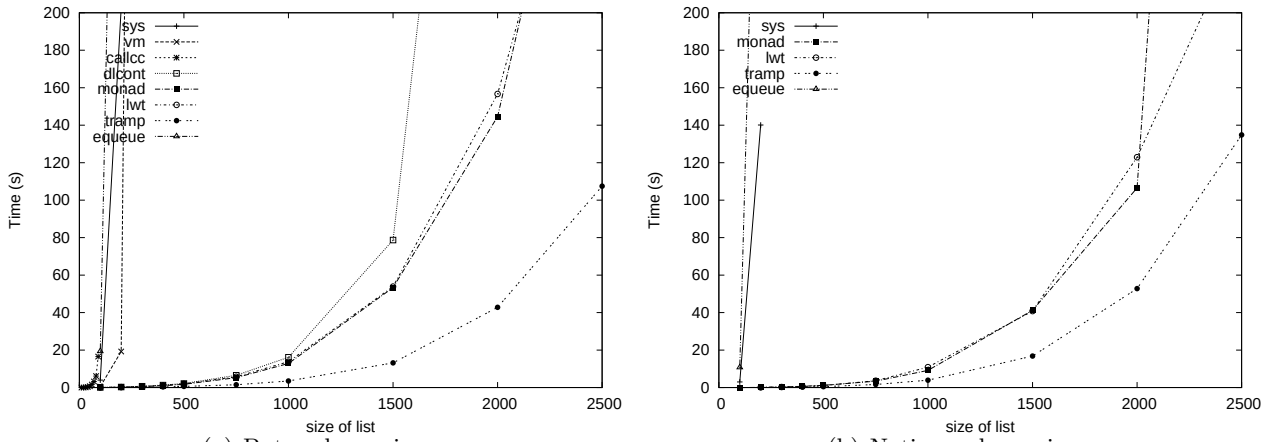
Execution time Figure 7 shows the execution time for `kpn`, on a log-log graph. `Callcc` is notably slow. Even heavy-weight `sys` is much faster, `vm` being ten times faster. The other implementations are rather similar, `tramp` being slightly better in the bytecode version.

[10]But one could argue that **all** our implementations are!



(a) Bytecode version (b) Native-code version

Figure 8: sieve, execution time



(a) Bytecode version (b) Native-code version

Figure 9: sorter, execution time

It has been noted [2] that continuations used for implementing threads are *one shot continuations* and are thus amenable to particular optimized implementations. No such implementation currently exists for OCaml. We note that dlcont performance is on par with `lwt` and `monad`. VM threads are much better than system threads and are only slightly slower than the light weigth implementations.

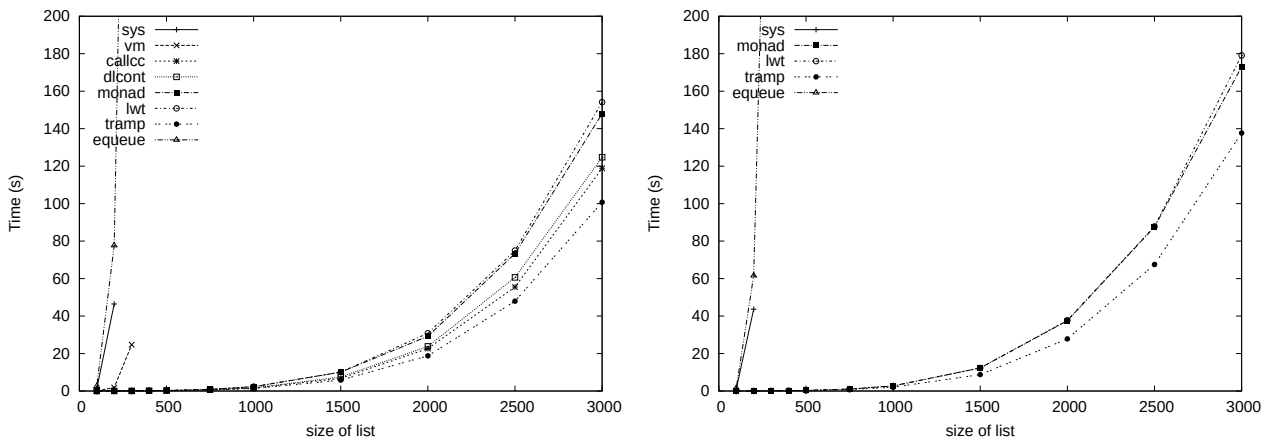
Figure 8 shows the execution times for the sieve. `Equeue` performance is terrible (much worse than `sys`) both for sieve and sorter. The problem is in the implementation of the `Equeue` module. As we said, events are presented to each handler in turn until one accepts it. In effect the threads are performing active wait on the MVars. Thus, `equeue` does not scale with the number of threads. Clearly, this module has not been designed with massive concurrency in mind.

As a side note, it seems that a simple change in `equeue` implementation would dramatically improve performance for the sieve: events should be presented only to (or starting with) handlers *set up after the handler that generated them*, in the order they were set up. This way each event would be presented immediately to the handler that is waiting for it. This would take advantage of the very particular communication pattern of the concurrent sieve and is not generally applicable, of course.

`vm` and `sys` are both much slower than the light weight implementations, among which `tramp` is the faster, `callcc` being not bad at all.

For sorter (Figure 9), performance for `sys` is shown only for lists of size 100 and 200. The number of threads used by sorter is about $n \times (n-1)/2$ with n the list size, which means 19900 threads for n=200 and 44850 for 300. We have found experimentally that only about 32000 system threads can be created on the system used.

`callcc` performs extremely badly, as does `vm`. Here again `tramp` is notably better, while `monad`



(a) Bytecode version (b) Native-code version

Figure 10: sorter -d, execution time

and `lwt` are very similar.

Figure 10 shows the time to set up the sorter network but not running it. The main difference is `callcc` being rather good this time (`dlcont` is better here too). Indeed, since the threads are not running, no continuation captures are performed...

This is the only figure where the performances for bytecode are (slighlty) better than those for native code. According to OCaml's documentation native code executables are faster but also bigger which can translate into larger startup time but this wouldn't alone explain what we see here. Memory allocation may be slower since sorter -d essentially allocates threads and MVars.

Memory usage Figure 11 shows on a log-log graph the memory requirements for the kpn example. `vm`, `sys`,[11] `lwt` and `tramp` are all identical both in bytecode and native code. `Dlcont` is a bit (well, twice) above. The problem with `callcc` becomes clear: its memory usage grows linearly while no new threads (or whatever data) are created. There's a memory leak, but the authors had warned us of the experimental status of the `callcc` library. In native code, all the implementations have the same memory requirements.

The graphs for sieve are shown in Figure 12. `tramp` is clearly the best. `equeue` is good too but values are shown only for the first few points since the program is so slow... `sys` is better than `vm` but again we don't measure the memory used by the operating system itself.

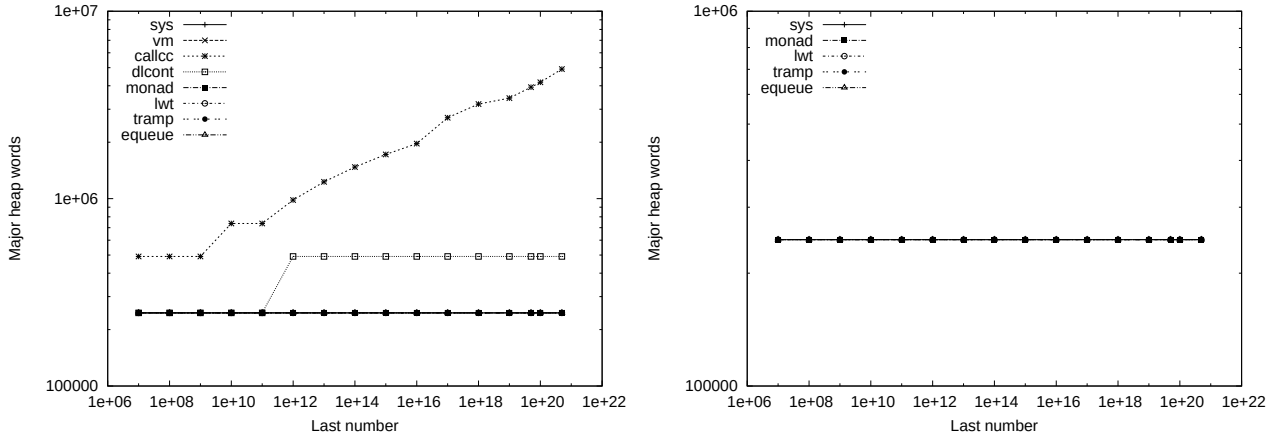
The problem with `callcc` is again obvious with sorter, on Figure 13: it uses huge amounts of memory, making the system trash. `dlcont` is much above the other implementations, and is quite good with `sorter -d` since no continuation capture occurs. `monad` and `lwt` are very close. Actually the graph for `lwt` is hidden under the one for `monad` on Figure 13(b). We don't include the graphs for native code since they don't show anything particularly interesting.

Finally Figures 14, 15, and 16 present the same data with a different view: they show the memory requirements per thread. `tramp` is always under the other implementations. It's interesting to see on Figures 15 and 16 that its advantage is much larger when the threads are running. The advantage over `monad` and `lwt` is probably caused by the relative complexity of the *thread* type (and the associated *bind* operator) they are based on.

7 Conclusion and perspectives

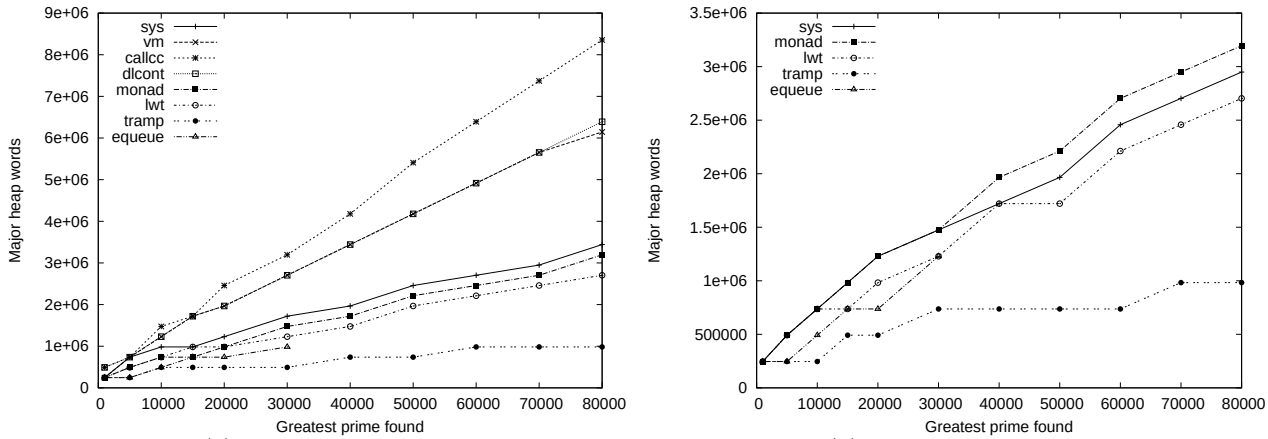
We have described, implemented, and tested several ways to implement light weight concurrency in OCaml. Direct style implementations involve capturing continuations, which is relatively costly (although much less than what is incurred by VM or system threads). Indirect style implementations perform better but force the programmer to write in a specific style.

[11] Of course, one should also consider the memory used by the system to manage threads, but we euh... haven't.



(a) Bytecode version (b) Native-code version

Figure 11: kpn, memory usage



(a) Bytecode version (b) Native-code version

Figure 12: sieve, memory usage

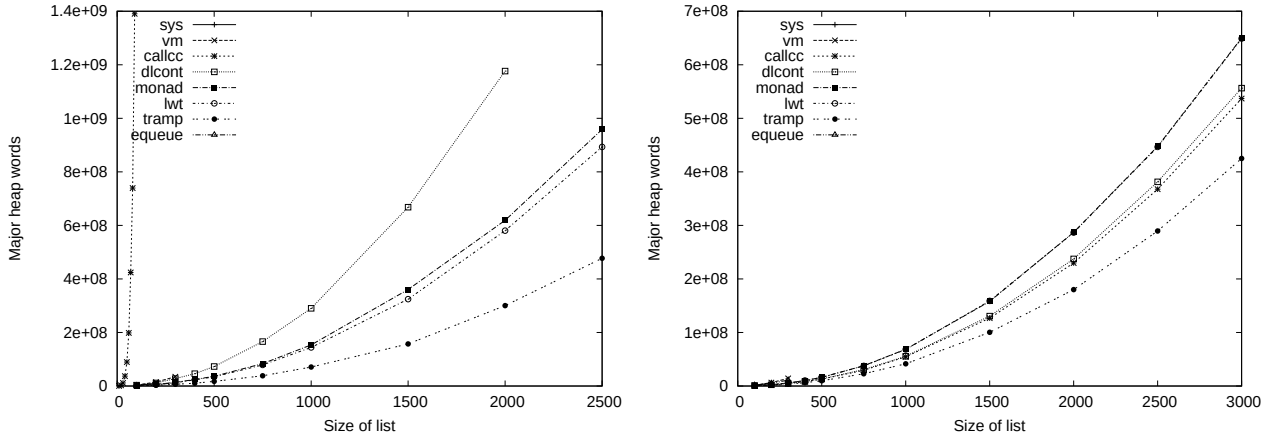
As we saw, event-based programming can be seen as a form of trampolined style programming with an event-based scheduling strategy. We didn't realize this immediately since event-based programming is mostly associated with imperative languages while trampolined style is with functional ones.

Apart from `callcc` that relies on a toy implementation of continuation capture and `equeue` that is not designed for massive concurrency, the light weight implementations can easily handle millions of threads.

The trampolined implementation is the lighter. Monad based ones (`monad` and `lwt`) are more costly due to the more complex implementation. Of course our examples are minimal, and all our implementations are obviously only skeletons (except `Lwt` of course), this should be kept in mind when looking at the performance results.

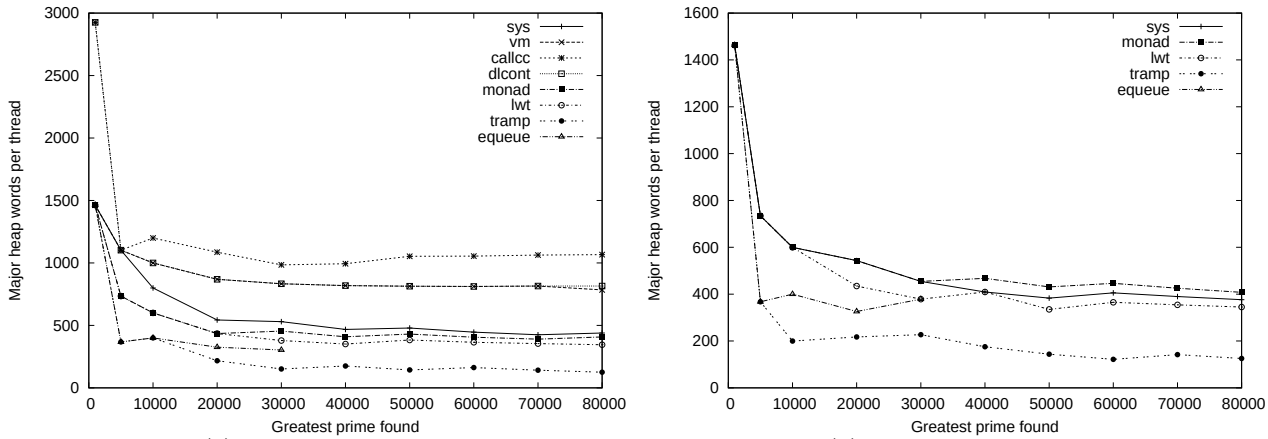
Realistic libraries should at least deal properly with I/O and exceptions. Concerning exceptions, OCaml's callcc is known not to, while dlcont is reported to handle them completely [13]. As we said, lwt deals with them (altough not with the standard syntax) so it could easily be added to monad.

We are currently developping a library (called μthreads) for light weight concurrency in OCaml. It is based on the trampolined style. Apart from exceptions and I/O it also implements delays, timed operations and synchronous events ala CML. More realistic applications, such as an FTP server are also being developed.



(a) sorter (b) sorter -d

Figure 13: sorter, memory usage for bytecode



(a) Bytecode version (b) Native-code version

Figure 14: sieve, memory usage per thread

References

[1] Joe Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6.1–6.26, New York, NY, USA, 2007. ACM.

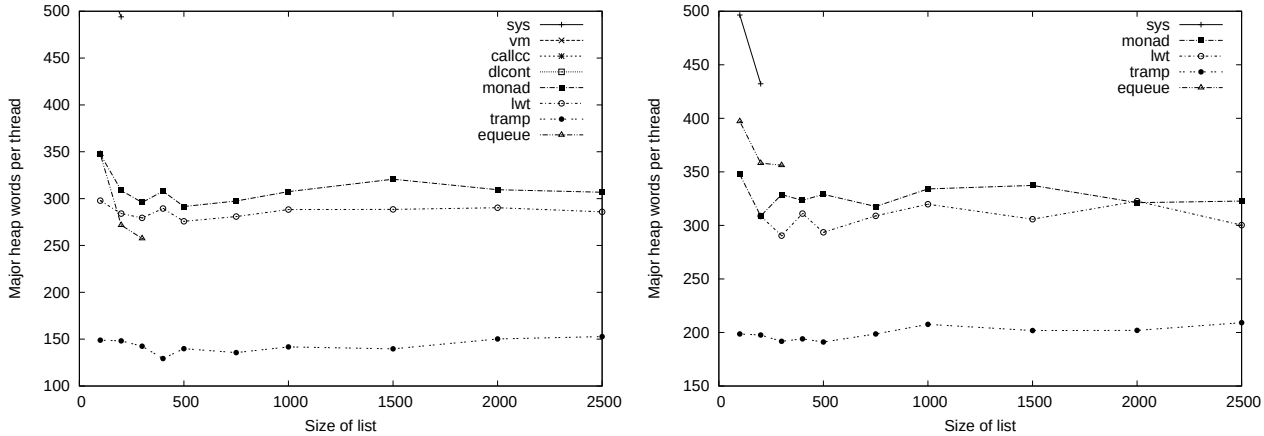
[2] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *In Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[3] Chung chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 5th workshop on Scheme and functional programming*, pages 99–107. Indiana University, 2004.

[4] Russ Cox. Libtask: a coroutine library for C and Unix. Software library. `http://swtch.com/libtask/`

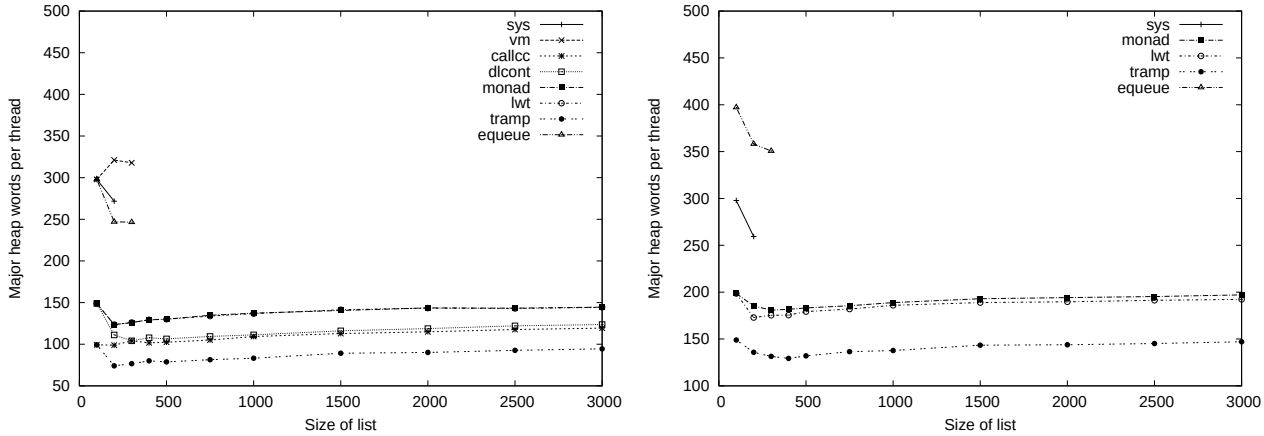
[5] Ralf S. Engelschall. GNU Pth - the GNU portable threads. Software library. `http://www.gnu.org/software/pth/`

[6] Matthias Felleisen, Daniel P. Friedman, Bruce F. Duba, and John Merrill. Beyond continuations. Technical Report Computer Science Dept. Technical Report 216, Indiana University, February 1987. `http://www.ccs.neu.edu/scheme/pubs/felleisen-beyond.pdf`



(a) Bytecode version (b) Native-code version

Figure 15: sorter, memory usage per thread



(a) Bytecode version (b) Native-code version

Figure 16: sorter -d, memory usage per thread

[7] D. P. Friedman. Applications of continuations. Invited Tutorial, Fifteenth Annual ACM Symposium on Principles of Programming Languages, January 1988. `http://www.cs.indiana.edu/~dfried/appcont.pdf`

[8] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999. `http://citeseer.ist.psu.edu/ganz99trampolined.html`

[9] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, New Haven, CT 06520-2158, 1993.

[10] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information processing*, pages 993–998, Toronto, August 1977.

[11] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.

[12] Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Indiana University, March 2005. `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR611`

[13] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely system description. Technical report, March 2010. `http://okmij.org/ftp/Computation/caml-shift.pdf`

[14] Xavier Leroy. OCaml-callcc: call/cc for OCaml. OCaml library. http://pauillac.inria.fr/~xleroy/software.html

[15] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[16] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services. In *Proceedings of 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 189–199, 2007. http://www.seas.upenn.edu/~lipeng/homepage/unify.html

[17] M. Douglas McIlroy. Coroutines. Internal report, Bell telephone laboratories, Murray Hill, New Jersey, May 1968.

[18] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdof Summer School 2000*, pages 47–96. IOS Press, 2001. http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/

[19] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[20] Simon Peyton Jones. *Beautiful code*, chapter Beautiful concurrency. O'Reilly, 2007.

[21] John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999.

[22] Amr Sabry, Chung chieh Shan, and Oleg Kiselyov. Native delimited continuations in (bytecode) OCaml. OCaml library, 2008. http://okmij.org/ftp/Computation/Continuations.html#caml-shift

[23] Gerd Stolpmann. Ocamlnet. http://projects.camlcity.org/projects/ocamlnet.html

[24] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, December 1998. Reprint from AI memo 349, December 1975.

[25] Jérôme Vouillon. Lwt: a cooperative thread library. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12, New York, NY, USA, 2008. ACM.

A Source code of the implementations

A.1 preempt

```
type α thread = α → unit

let gl = Mutex.create ()

let spawn t = ignore (Thread.create (fun () → Mutex.lock gl; Mutex.unlock gl; t ()) ())

let stop_event = Event.new_channel ()
let start () = Mutex.unlock gl; Event.sync (Event.receive stop_event)
let stop () = Event.sync (Event.send stop_event ())

let halt = Thread.exit
let yield = Thread.yield

type α mvar =
    { mutable v : α option; ch : α Event.channel;
      mutable read : bool; mutable write : bool; l : Mutex.t }
```

```
let make_mvar () =
  { v = None; ch = Event.new_channel (); read = false; write = false; l = Mutex.create () }

let put_mvar out v =
  let ul () = Mutex.unlock out.l in
  Mutex.lock out.l;
  match out with
  | { v = Some v'; ch = c; read = _; write =false } → out.write ← true; ul ();
      Event.sync (Event.send c v)

  | { v = None; ch = c; read =true; write =false } → ul (); out.read ← false;
      Event.sync (Event.send c v)

  | { v = None; ch = c; read =false; write =false } → out.v ← Some v; ul ()

let take_mvar inp =
  let ul () = Mutex.unlock inp.l in
  Mutex.lock inp.l;
  match inp with
  | { v = Some v; ch = c; read =false; write =false } → inp.v ← None; ul (); v

  | { v = Some v; ch = c; read =false; write =true } →
      inp.write ← false; ul (); let v' = Event.sync (Event.receive c) in
      Mutex.lock inp.l; inp.v ← Some v'; ul (); v

  | { v = None; ch = c; read =false; write = _ } →
      inp.read ← true; ul (); Event.sync (Event.receive c)

type α fifo = { q : α Queue.t; mutable w : α Event.channel option }
let make_fifo () = { q = Queue.create (); w = None }

let take_fifo f =
    if Queue.length f.q = 0 then
      let e = Event.new_channel () in
      f.w ← Some e;
      Event.sync (Event.receive e)
    else
      Queue.take f.q

let put_fifo f v =
    match f.w with
    | None → Queue.add v f.q
    | Some e → f.w ← None; Event.sync (Event.send e v)

Mutex.lock gl
```

A.2 callcc

```
open Callcc

type α thread = α → unit

type queue_t = { mutable e :unit thread; q :unit thread Queue.t }

let q = { e = (fun() → ()); q = Queue.create () }

let enqueue t = Queue.push t q.q
let dequeue () = try Queue.take q.q with Queue.Empty → q.e

let halt () = dequeue () ()

let yield () =
  callcc (fun k → enqueue (fun () → throw k ()); dequeue () ())

let spawn p = enqueue (fun () → p (); halt ())

exception Stop
let stop () = raise Stop
```

```
let start () =
  try
    callcc (fun exitk →
      q.e ← (fun () → throw exitk ());
      dequeue () ())
  with Stop → ()

type α mvar = { mutable v : α option;
                mutable read : α thread option;
                mutable write : (unit thread × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v =
    match out with
    | { v = Some v; read = _; write = None } →
        callcc (fun k →
          out.write ← Some ((fun () → throw k ()), v); halt ())

    | { v = None; read = Some r; write = None } →
        out.read ← None; enqueue (fun () → r v)

    | { v = None; read = None; write = None } → out.v ← Some v; ()

let take_mvar inp =
    match inp with
    | { v = Some v; read = None; write = None } → inp.v ← None; v

    | { v = Some v; read = None; write = Some(c, v') } →
        inp.v ← Some v'; inp.write ← None; enqueue c; v

    | { v = None; read = None; write = _ } →
        callcc (fun k →
          inp.read ← Some (fun v → throw k v);
          Obj.magic halt ())

type α fifo = { q : α Queue.t; mutable w : α thread option }
let make_fifo () = { q = Queue.create (); w = None }

let take_fifo f =
  if Queue.length f.q = 0 then
    Callcc.callcc (fun k → f.w ← Some (fun v → Callcc.throw k v);
      Obj.magic halt ())
  else
    Queue.take f.q

let put_fifo f v =
  Queue.add v f.q;
  match f.w with
  | Some k → enqueue (fun () → k (Queue.take f.q)); f.w ← None
  | None → ()
```

A.3 dlcont

```
open Delimcc

type α thread = α → unit

let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq

let prompt = new_prompt ()

let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt_subcont p sk (fun () → c))))

let yield () = shift0 prompt (fun f → enqueue f)
```

could use abort in halt, we just want to remove the prompt

```
let halt () = shift0 prompt (fun f → ())
```

enqueue a new thread

```
let spawn p = enqueue (fun () → push_prompt prompt (fun () → p (); halt ()))
```

```
exception Stop
let stop () = raise Stop
```

```
let start () =
  try
    while true do
      dequeue () ()
    done
  with Queue.Empty | Stop → ()
```

```
type α mvar = { mutable v : α option;
                mutable read : α thread option; (* thread blocked on take *)
                mutable write : (unit thread × α) option } (* ... on put *)
```

```
let make_mvar () = { v = None; read = None; write = None }
```

```
let put_mvar out v =
  match out with
  | { v = Some v'; read = _; write = None } →
      shift0 prompt (fun f → out.write ← Some (f, v))
  | { v = None; read = Some r; write = None } →
      out.read ← None; enqueue (fun () → r v)
  | { v = None; read = None; write = None } → out.v ← Some v
```

```
let take_mvar inp =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; v
  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; enqueue c; v
  | { v = None; read = None; write = _ } →
      shift0 prompt (fun f → inp.read ← Some((fun i → f i)))
```

```
type α fifo = { q : α Queue.t; mutable w : α thread option }
let make_fifo () = { q = Queue.create (); w = None }
```

```
let take_fifo f =
  if Queue.length f.q = 0 then
    shift0 prompt (fun k → f.w ← Some k)
  else
    Queue.take f.q
```

```
let put_fifo f v =
  Queue.add v f.q;
  match f.w with
  | Some k → enqueue (fun () → k (Queue.take f.q)); f.w ← None
  | None → ()
```

A.4 tramp

```
type α thread = α → unit
```

```
let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq
```

```
let skip k = k ()
let yield k = enqueue k
let halt () = ()
let spawn t = enqueue t
```

```
exception Stop
let stop () = raise Stop

let start () =
  try
    while true do
      dequeue () ()
    done
  with Queue.Empty | Stop → ()

let (>>=) inst (k : α thread) : unit = inst k

type α mvar = { mutable v : α option;
                mutable read : α thread option;
                mutable write : (unit thread × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v k =
  match out with
  | { v = Some v'; read = _; write = None } → out.write ← Some (k, v)

  | { v = None; read = Some r; write = None } →
        out.read ← None; spawn (fun () → r v); k ()

  | { v = None; read = None; write = None } → out.v ← Some v; k ()

let take_mvar inp k =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; k v

  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; spawn c; k v

  | { v = None; read = None; write = _ } → inp.read ← Some(k)

type α fifo = { q : α Queue.t; mutable w : α thread option }
let make_fifo () = { q = Queue.create (); w = None }

let take_fifo f k =
  if Queue.length f.q = 0 then
    f.w ← Some k
  else
    k (Queue.take f.q)

let put_fifo f v =
  Queue.add v f.q;
  match f.w with
  | Some k → enqueue (fun () → k (Queue.take f.q)); f.w ← None
  | None → ()
```

A.5 monad

```
type α state =
  | Return of α
  | Sleep of (α thread → unit) list ref
  | Link of α thread

and α thread = { mutable st : α state }

let rec repr t =
  match t.st with
  | Link t' → repr t'
  | _ → t

let wait () = { st = Sleep (ref []) }
let return v = { st = Return v }
```

```
let wakeup t v =
  let t = repr t in
  match t.st with
  | Sleep w →
      t.st ← Return v;
      List.iter (fun f → f t) !w
  | _ → failwith "wakeup"

let connect t t' =
  let t' = repr t' in
  match t'.st with
  | Return v → wakeup t v
  | Sleep w' →
      let t = repr t in
      match t.st with
      | Sleep w → w := !w @ !w'; t'.st ← Link t
      | _ → failwith "connect"

let rec (>>=) t f =
  match (repr t).st with
  | Return v → f v
  | Sleep w → let res = wait () in
    w := (fun t → connect res (t >>= f)) :: !w;
    res

let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq

let skip = return ()
let yield () = let res = wait () in enqueue res; res
let halt () = return ()

let wait_start = wait ()

let spawn t = wait_start >>= t; ()

exception Stop
let stop () = raise Stop

let start () =
  try
    wakeup wait_start ();
    while true do
      wakeup (dequeue ()) ()
    done
  with Queue.Empty | Stop → ()

type α mvar = { mutable v : α option;
                mutable read : α thread option;
                mutable write : (unit thread × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v =
  match out with
  | { v = Some v'; read = _; write = None } →
      let w = wait () in out.write ← Some (w, v); w

  | { v = None; read = Some r; write = None } →
      out.read ← None; wakeup r v; return ()

  | { v = None; read = None; write = None } → out.v ← Some v; return ()

let take_mvar inp =
  match inp with
  | { v = Some v; read = None; write = None } →
      inp.v ← None; return v
```

```
  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; wakeup c ();
      return v

  | { v = None; read = None; write = _ } →
      let w = wait () in inp.read ← Some(w); w

type α fifo = { q : α Queue.t; mutable w : α thread option }
let make_fifo () = { q = Queue.create (); w = None }

let take_fifo f =
  if Queue.length f.q = 0 then
    let k = wait () in (f.w ← Some k; k)
  else
    return (Queue.take f.q)

let put_fifo f v =
  Queue.add v f.q;
  match f.w with
  | Some k → f.w ← None; wakeup k (Queue.take f.q)
  | None → ()
```

A.6 lwt

```
type α thread = α Lwt.t

let (>>=) = Lwt.bind

let wait_start = Lwt.wait ()

exception Stop
let do_stop = ref false

let spawn t = (wait_start >>= t); ()
let finish = Lwt.wait () >>= fun () → Lwt.fail Stop
let start () = Lwt.wakeup wait_start (); Lwt_unix.run finish
let stop () = do_stop := true; Lwt.wakeup finish (); Lwt.return ()
let halt () = Lwt.return ()
let return a = Lwt.return a
let skip = Lwt.return ()
let yield = Lwt_unix.yield

type α mvar = { mutable v : α option;
                mutable read : α Lwt.t option;
                mutable write : (unit Lwt.t × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v =
  if !do_stop then Lwt.fail Stop else
  match out with
  | { v = Some v'; read = _; write = None } →
      let w = Lwt.wait () in out.write ← Some (w, v); w

  | { v = None; read = Some r; write = None } →
      out.read ← None; Lwt.wakeup r v; Lwt.return ()

  | { v = None; read = None; write = None } → out.v ← Some v; Lwt.return ()

let take_mvar inp =
  if !do_stop then Lwt.fail Stop else
  match inp with
  | { v = Some v; read = None; write = None } →
      inp.v ← None; Lwt.return v

  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; Lwt.wakeup c ();
      Lwt.return v
```

```
  | { v = None; read = None; write = _ } →
      let w = Lwt.wait () in inp.read ← Some(w); w

type α fifo = { q : α Queue.t; mutable w : α thread option }
let make_fifo () = { q = Queue.create (); w = None }

let take_fifo f =
  if !do_stop then Lwt.fail Stop else
  if Queue.length f.q = 0 then
    let k = Lwt.wait () in (f.w ← Some k; k)
  else
    Lwt.return (Queue.take f.q)

let put_fifo f v =
  Queue.add v f.q;
  match f.w with
  | Some k → f.w ← None; Lwt.wakeup k (Queue.take f.q)
  | None → ()
```

A.7 equeue

```
let skip k = k ()

let (>>=) inst k = inst k

type eventid = unit ref
type α event = Written of eventid | Read of eventid × α | Go of eventid

let make_eventid () = ref ()

let esys : int event Equeue.t = Equeue.create (fun _ → ())

let yield k =
  let id = make_eventid () in
  Equeue.add_handler esys (fun esys e →
    match e with
    | Go id′ when id′ ≡ id → k ()
    | _ → raise Equeue.Reject);
  Equeue.add_event esys (Go id);
  raise Equeue.Terminate

let spawn t =
  let id = make_eventid () in
  Equeue.add_handler esys (fun esys e →
    match e with
    | Go id′ when id′ ≡ id → t ()
    | _ → raise Equeue.Reject);
  Equeue.add_event esys (Go id)

let halt () = raise Equeue.Terminate

exception Stop
let stop () = raise Stop

let start () =
  try
    Equeue.run esys
  with Stop → ()

type α mvar = { mutable v : α option;
                mutable read : eventid option;
                mutable write : (eventid × α) option }

let make_mvar () = { v = None; read = None; write = None }
```

```
let put_mvar out v k =
  match out with
  | { v = Some v'; read = _; write = None } →
      let id = make_eventid () in out.write ← Some (id, v);
      Equeue.add_handler esys (fun esys e →
        match e with
        | Written id' when id' ≡ id → k ()
        | _ → raise Equeue.Reject);
      raise Equeue.Terminate

  | { v = None; read = Some id; write = None } →
      out.read ← None;
      Equeue.add_event esys (Read(id, v));
      k ()

  | { v = None; read = None; write = None } → out.v ← Some v; k ()

let take_mvar inp k =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; k v

  | { v = Some v; read = None; write = Some(id, v') } →
      inp.v ← Some v'; inp.write ← None;
      Equeue.add_event esys (Written id); k v

  | { v = None; read = None; write = _ } →
      let id = make_eventid () in
      inp.read ← Some id;
      Equeue.add_handler esys (fun esys e →
        match e with
        | Read(id', arg) when id' ≡ id → k arg
        | _ → raise Equeue.Reject);
      raise Equeue.Terminate

type α fifo = { q : α Queue.t; mutable w : eventid option }
let make_fifo () = { q = Queue.create (); w = None }

let take_fifo f k =
  if Queue.length f.q = 0 then
    let id = make_eventid () in
    f.w ← Some id;
    Equeue.add_handler esys (fun esys e →
      match e with
      | Read(id', arg) when id' ≡ id → k arg
      | _ → raise Equeue.Reject);
    raise Equeue.Terminate
  else
    k (Queue.take f.q)

let put_fifo f v =
  Queue.add v f.q;
  match f.w with
  | Some id → Equeue.add_event esys (Read(id, Queue.take f.q)); f.w ← None
  | None → ()
```

B Source code of the examples

B.1 KPN, direct style

```
open Lwc
open Big_int

let (<) = lt_big_int
let (>) = gt_big_int

let ( × ) = mult_big_int
```

Merge thread

```
let rec mergeb q1 q2 qo v1 v2 =
  let v1, v2 =
    if v1 < v2 then begin
```

```
      put_mvar qo v1;
      (take_mvar q1, v2)
    end
    else if v1 > v2 then begin
      put_mvar qo v2;
      (v1, take_mvar q2)
    end
    else begin
      put_mvar qo v1;
      (take_mvar q1, take_mvar q2)
    end
  in
  mergeb q1 q2 qo v1 v2
```

Initializer for merge thread

```
let merge q1 q2 qo () =
  let v1 = take_mvar q1
  and v2 = take_mvar q2 in
  mergeb q1 q2 qo v1 v2
```

Multiplier thread

```
let rec times a f qo () =
  let v = take_fifo f in
  put_mvar qo (a × v);
  times a f qo ()
```

The x thread itself

```
let rec x mv f2 f3 f5 () =
  let v = take_mvar mv in
  if v > !last then stop ();
  if !print then
    Printf.printf "%s␣" (string_of_big_int v);
  put_fifo f2 v;
  put_fifo f3 v;
  put_fifo f5 v;
  x mv f2 f3 f5 ()
```

Set up and start

```
let main () =
  (* fifo + times = mult *)
  let make_mult a =
    let f = make_fifo ()
    and mv = make_mvar () in
    let t = times a f mv
    in
    spawn t; (f, mv)
  in
  let make_merge q1 q2 =
    let qo = make_mvar () in
    let m = merge q1 q2 qo
    in
    spawn m; qo
  in
  let f2, m2 = make_mult (big_int_of_int 2)
  and f3, m3 = make_mult (big_int_of_int 3)
  and f5, m5 = make_mult (big_int_of_int 5) in
  let m35 = make_merge m3 m5 in
  let m235 = make_merge m2 m35
  in
  spawn (x m235 f2 f3 f5);
  put_mvar m235 unit_big_int; start ()
```

B.2 KPN, indirect style

```
open Lwc
open Big_int

let (<) = lt_big_int
let (>) = gt_big_int
let ( × ) = mult_big_int
```

Merge thread

```
let rec mergeb q1 q2 qo v1 v2 =
  if v1 < v2 then begin
    put_mvar qo v1 >>= fun () →
    take_mvar q1 >>= fun v1 →
    mergeb q1 q2 qo v1 v2
  end
  else if v1 > v2 then begin
    put_mvar qo v2 >>= fun () →
    take_mvar q2 >>= fun v2 →
    mergeb q1 q2 qo v1 v2
  end
  else begin
    put_mvar qo v1 >>= fun () →
    take_mvar q1 >>= fun v1 →
    take_mvar q2 >>= fun v2 →
    mergeb q1 q2 qo v1 v2
  end
```

Initializer for merge thread

```
let merge q1 q2 qo () =
  take_mvar q1 >>= fun v1 →
  take_mvar q2 >>= fun v2 →
  mergeb q1 q2 qo v1 v2
```

Multiplier thread

```
let rec times a f qo () =
  take_fifo f >>= fun v →
  put_mvar qo (a × v) >>=
  times a f qo
```

The x thread itself

```
let rec x mv f2 f3 f5 () =
  take_mvar mv >>= fun v →
  if v > !last then stop ()
  else skip >>= fun () →
    if !print then
      Printf.printf "%s␣" (string_of_big_int v);
    put_fifo f2 v;
    put_fifo f3 v;
    put_fifo f5 v;
    x mv f2 f3 f5 ()
```

Set up and start

```
let main () =
  (* fifo + times = mult *)
  let make_mult a =
    let f = make_fifo ()
    and mv = make_mvar () in
    let t = times a f mv
    in
    spawn t; (f, mv)
  in
  let make_merge q1 q2 =
    let qo = make_mvar () in
    let m = merge q1 q2 qo
    in
    spawn m; qo
  in
  let f2, m2 = make_mult (big_int_of_int 2)
  and f3, m3 = make_mult (big_int_of_int 3)
  and f5, m5 = make_mult (big_int_of_int 5) in
  let m35 = make_merge m3 m5 in
  let m235 = make_merge m2 m35
  in
  spawn (x m235 f2 f3 f5);
  spawn (fun () → put_mvar m235 unit_big_int >>=
                  halt);
  start ()
```

B.3 Sieve, direct style

```
open Lwc

let rec integers out i () =
  put_mvar out i;
  integers out (i + 1) ()

let rec output inp () =
  let v = take_mvar inp in
  if !print then (Printf.printf "%i␣" v; flush stdout);
  if v < !last then output inp () else stop ()

let rec filter n inp out () =
  let v = take_mvar inp in
  if v mod n ≠ 0 then put_mvar out v;
  filter n inp out ()

let rec sift inp out () =
  let v = take_mvar inp in
  put_mvar out v;
  let mid = make_mvar () in
  spawn (filter v inp mid);
  sift mid out ()

let sieve () =
  let s1 = make_mvar () in
  let s2 = make_mvar () in
  spawn (integers s1 2);
  spawn (sift s1 s2);
  spawn (output s2);
  start ()
```

B.4 Sieve, indirect style

```
open Lwc

let rec integers out i () =
  put_mvar out i >>= integers out (i + 1)

let rec output inp () =
  take_mvar inp >>= fun v →
  if !print then (Printf.printf "%i␣" v; flush stdout);
  if v < !last then output inp () else (stop (); halt())

let rec filter n inp out () =
  take_mvar inp >>= fun v →
  (if v mod n ≠ 0 then put_mvar out v else skip) >>=
  filter n inp out

let rec sift inp out () =
  take_mvar inp >>= fun v →
  put_mvar out v >>= fun () →
  let mid = make_mvar () in
  spawn (sift mid out);
  filter v inp mid ()

let sieve () =
  let s1 = make_mvar () in
  let s2 = make_mvar () in
  spawn (integers s1 2);
  spawn (sift s1 s2);
  spawn (output s2);
  start ()
```

B.5 Sorter, direct style

```
open Lwc

let minmax a b =
  if a < b then (a, b) else (b, a)

let rec comparator x y hi lo =
  let a = take_mvar x
  and b = take_mvar y in
  let (l, h) = minmax a b
  in
  put_mvar lo l;
  put_mvar hi h;
  comparator x y hi lo

let make_list n fct =
  let rec loop n acc =
    if n = 0 then acc
    else
      loop (n − 1) (fct n :: acc)
  in
  loop n []

let make_n_mvars n =
  make_list n (fun _ → make_mvar ())

let rec iter4 fct l1 l2 l3 l4 =
  match (l1, l2, l3, l4) with
  | [], [], [], [] → []
  | l1 :: l1s, l2 :: l2s, l3 :: l3s, l4 :: l4s →
      fct (l1, l2, l3, l4);
      iter4 fct l1s l2s l3s l4s
  | _ → failwith "iter4"
```

```
let column (i :: is) y =
  let n = List.length is in
  let ds = make_n_mvars (n − 1) in
  let os = make_n_mvars n
  in
  iter4
    (fun (i, di, o, od) →
      spawn (fun () → comparator i di o od))
    is (i :: ds) os (ds @ [y]);
  os

let sorter xs ys =
  let rec help is ys n =
    if n > 2 then
      let os = column is (List.hd ys) in
      help os (List.tl ys) (n − 1)
    else
      spawn (fun () → comparator
            (List.hd (List.tl is)) (List.hd is)
            (List.hd (List.tl ys)) (List.hd ys))
  in
  help xs ys (List.length xs)

let set_list ms l () =
  List.iter (fun (mv, v) → put_mvar mv v)
    (List.map2 (fun a b → (a, b)) ms l);
  halt ()

let print_list ms () =
  List.iter (fun n → Printf.printf "%i␣" n)
    (List.map take_mvar ms);
  flush stdout; stop ()

let sort l =
  let n = List.length l in
  let xs = make_n_mvars n
  and ys = make_n_mvars n
  in
  sorter xs ys;
  spawn (set_list xs l);
  spawn (print_list ys);
  if ¬ !dont then start ()

let doit () =
  let l = make_list !last (fun _ → Random.int 999) in
  sort l
```

B.6 Sorter, indirect style

```
open Lwc

let minmax a b =
  if a < b then (a, b) else (b, a)

let rec comparator x y hi lo () =
  take_mvar x >>= fun a →
  take_mvar y >>= fun b →
  let (l, h) = minmax a b
  in
  put_mvar lo l >>= fun () →
  put_mvar hi h >>=
  comparator x y hi lo

let make_list n fct =
  let rec loop n acc =
    if n = 0 then acc
    else
      loop (n − 1) (fct n :: acc)
  in
  loop n []

let make_n_mvars n =
  make_list n (fun _ → make_mvar ())

let rec iter4 fct l1 l2 l3 l4 =
  match (l1, l2, l3, l4) with
  | [], [], [], [] → []
  | l1 :: l1s, l2 :: l2s, l3 :: l3s, l4 :: l4s →
      fct (l1, l2, l3, l4);
      iter4 fct l1s l2s l3s l4s
  | _ → failwith "iter4"

let column (i :: is) y =
  let n = List.length is in
  let ds = make_n_mvars (n − 1) in
  let os = make_n_mvars n
  in
  iter4
    (fun (i, di, o, od) →
      spawn (comparator i di o od))
    is (i :: ds) os (ds @ [y]);
  os

let sorter xs ys () =
  let rec help is ys n =
    if n > 2 then
      let os = column is (List.hd ys) in
      help os (List.tl ys) (n − 1)
    else
      spawn (comparator
            (List.hd (List.tl is)) (List.hd is)
            (List.hd (List.tl ys)) (List.hd ys))
  in
  help xs ys (List.length xs)

let rec set_list mvs l () =
  match mvs, l with
  | [], [] → halt ()
  | m :: r, h :: t → put_mvar m h >>= set_list r t

let print_list mvs () =
  let rec loop mvs acc k =
    match mvs with
    | [] → k acc
    | h :: t → take_mvar h >>= fun v →
                loop t (v :: acc) k
  in
  loop mvs [] >>= fun l →
    List.iter (fun n → Printf.printf "%i␣" n) (List.rev l);
    halt ()

let sort l =
  let n = List.length l in
  let xs = make_n_mvars n
  and ys = make_n_mvars n
  in
  sorter xs ys ();
  spawn (set_list xs l);
```

```
  spawn (print_list ys);
  if ¬ !dont then start ()

let doit () =
  let l = make_list !last (fun _ → Random.int 999) in
  sort l
```