Compiler support for lightweight concurrency

Kathleen Fisher
AT&T Labs, Research
`kfisher@research.att.com`

John Reppy
Bell Labs, Lucent Technologies
`jhr@research.bell-labs.com`

April 3, 2020

Abstract

This paper describes the design of a *direct-style*, λ-calculus-based compiler intermediate representation (IR) suitable for implementing a wide range of surface-language concurrency features while allowing flexibility in the back-end implementation. The features that this IR includes to support concurrency include a weak but inexpensive form of continuations and primitives for thread creation, scheduling, and termination. Although this work has been done in the context of implementing concurrency for the MOBY programming language, the model is general and could be used in the context of other languages. In this paper, we describe the IR's continuations and thread manipulation primitives and illustrate their expressiveness with examples of various concurrency operations. We define an operational semantics to specify the operations precisely and to provide a tool for reasoning about the correctness of implementations. To illustrate the flexibility of our the approach, we describe two existing implementations of the threading model — one based on a one-to-one mapping of language threads to POSIX threads and one based on a many-to-many mapping.

1 Introduction

Concurrency provides an important abstraction mechanism for programming various kinds of reactive systems, such as user interfaces and distributed services [Pik89, GR93, HJT$^+$93, Rep99]. Thread libraries for sequential languages like C or C++ are currently the most common form of concurrent programming, but languages that support concurrency directly are becoming more common. As with any abstraction, it is important that concurrency be implemented efficiently, otherwise programmers will not take advantage of it.

The important performance issues for concurrency are the time it takes to create threads, the speed of synchronization and communication, and the space required to represent threads. Experience with Concurrent ML (CML) has shown that a very lightweight implementation of concurrency primitives encourages a programming model where thread abstraction can be used liberally [Rep99].[1] We are currently working on the implementation of concurrency for the MOBY programming language, which is an ML-like language with support for both object-oriented [FR99] and CML-style concurrent programming. Our goal is to provide an implementation of MOBY's concurrency features that is as close in efficiency to CML's implementation as possible.[2] More specifically, our goals are to support low-overhead synchronization and communication, fast thread creation, space efficient thread data structures, and easy access to the multiprocessing and concurrency features of the underlying operating system.

[1]Many CML applications use hundreds, or even thousands, of threads.

[2]We do not expect to achieve parity with the CML implementation, because our implementation supports multiprocessing and multiple system threads, whereas CML's implementation uses a single system process.

One common approach to implementing threading libraries for sequential languages has been to use *continuations* as a representation of threads [Wan80, Shi97, Rep99]. This approach includes many examples using the first-class continuations of SCHEME [ADH^{+}98] and SML/NJ [AM91], as well the use of `setjmp`/`longjmp` in C. Depending on the efficiency of the underlying continuation mechanism, this approach can result in extremely efficient threading libraries. Our approach builds on this idea that continuations are an effective tool for implementing concurrency mechanisms, but instead of adding continuations to MOBY, we added them to our compiler's intermediate representation (IR). Putting continuations into the IR has several advantages: we avoid the complexity of continuations as a surface-language feature, we can use a more restricted form of continuations that admits efficient implementation, and we can take advantage of existing compiler optimizations to improve the performance of the concurrency mechanisms.

This paper describes our approach to implementing concurrency in the MOBY compiler. The main contribution of the paper is the design of a compiler IR, called BOL, that effectively supports the efficient implementation of concurrent language mechanisms. This design has two major components: a restricted form of continuations and a collection of operations that support thread creation and scheduling. Our continuations are one-shot and have a lifetime delimited by their scope. Our scheduling model is abstract, which allows multiple implementations with different performance characteristics. We describe two implementations that illustrate the flexibility of our design. While this work has been done in the context of compiling MOBY, the approach is general and could be used to implement other concurrent languages.

We organize the remainder of this paper as follows. In the next section, we introduce BOL, describe BOL continuations, and illustrate their use by showing how to use such continuations to implement non-local control flow. In Section 3, we describe the BOL primitives for thread management and scheduling. To illustrate the expressiveness of BOL's thread model, we show in Section 4 how to use BOL operations to implement a variety of surface-language concurrency features. Section 5 presents an operational semantics for BOL, providing a tool for reasoning about the limited lifetimes of BOL continuations and documentation for the behavior of the various scheduling primitives. We have two implementations of our design, which we describe in Section 6. The first implementation maps each MOBY thread onto its own system thread, while the second maps multiple MOBY threads onto each system thread. Section 8 describes some areas for improvement in these implementations and Section 9 describes related work. We conclude with a summary in Section 10.

2 BOL

The MOBY compiler translates the typed abstract syntax tree produced by typechecking to a weakly typed, direct-style intermediate representation called BOL.[3] This IR is used both for optimizing the code from a single module and also for cross-module inlining [FPR01]. Because BOL is a compiler IR, it does not perform runtime checking to avoid executing buggy code: it assumes that the language front-end has already done such checking on user code and has generated correct BOL code. This assumption allows BOL to avoid expensive runtime checks.

BOL is a *normalized* representation that makes dataflow explicit by binding each intermediate result to a variable [FSDF93], but in this paper we use the denormalized syntax of Figure 1 to make the examples more readable.

BOL includes four syntactic classes: *blocks*, *statements*, *expressions*, and *primops*. Blocks group statements and serve to delineate the scope of bindings. A statement is either an expression or a binding followed by a statement. Bindings include function bindings, **`let`** bindings, and a "**`do`**" binding, which evaluates

[3]BOL's type system is roughly equivalent to C's without recursive types. It is used to guide the mapping of BOL variables to machine registers and to provide representation information for the garbage collector. Since the type system is largely orthogonal to concurrency issues, we have omitted the BOL types from the examples in this paper.

$$
\begin{array}{rcl}
\textit{blk} & ::= & \{\ \textit{stmt}\ \} \\
\\
\textit{stmt} & ::= & \textit{exp} \\
& \mid & \mathbf{fun}\ f\ (x_1,\ \ldots,\ x_n)\ \textit{blk}\ \textit{stmt} \\
& \mid & \mathbf{let}\ (x_1,\ \ldots,\ x_n)\ =\ \textit{exp}\ \textit{stmt} \\
& \mid & \mathbf{do}\ \textit{exp}\ \textit{stmt} \\
& \mid & \mathbf{letcont}\ k\ (x_1,\ \ldots,\ x_n)\ =\ \textit{exp}\ \mathbf{in}\ \textit{blk} \\
& \mid & \mathbf{let}\ k'\ =\ \mathbf{apply\_cont}\ k\ (\textit{exp}_1,\ \ldots,\ \textit{exp}_n)\ \textit{stmt} \\
& \mid & \ldots \\
\\
\textit{exp} & ::= & \textit{blk} \\
& \mid & \mathbf{if}\ \textit{exp}\ \mathbf{then}\ \textit{exp}\ \mathbf{else}\ \textit{exp} \\
& \mid & f\ (\textit{exp}_1,\ \ldots,\ \textit{exp}_n) \\
& \mid & \mathbf{alloc}\ (\textit{exp}_1,\ \ldots,\ \textit{exp}_n) \\
& \mid & \textit{exp}\,\mathbf{\#}i \\
& \mid & \textit{exp}\,\mathbf{\#}i\ \mathbf{:=}\ \textit{exp} \\
& \mid & \mathbf{throw}\ k\ (\textit{exp}_1,\ \ldots,\ \textit{exp}_n) \\
& \mid & \textit{primop}(\textit{exp}_1,\ \ldots,\ \textit{exp}_n) \\
& \mid & \ldots \\
\\
\textit{primop} & ::= & \texttt{AcquireLock}\ \mid\ \texttt{ReleaseLock} \\
& \mid & \texttt{I32Add}\ \mid\ \ldots
\end{array}
$$

Figure 1: BOL syntax

its right-hand-side expression for side effects. Expressions include blocks, conditionals, function calls, heap-object allocation, heap-object field selection (zero-based indexing), imperative update of heap-object fields, the application of primitive operations to arguments, variables, and literals. Statements and expressions also include the continuation operations that we describe in the next section. Primitive operations include `AcquireLock` and `ReleaseLock` for manipulating spinlocks and arithmetical operations such as `I32Add` for adding 32-bit integers and `I32Eq` for testing two 32-bit integers for equality.

2.1 BOL continuations

To support non-local control flow, BOL has a weak form of explicit continuations. The binding

$$\mathbf{letcont}\ k\ (x_1,\ \ldots,\ x_n)\ =\ \textit{exp}\ \mathbf{in}\ \textit{blk}$$

reifies the current continuation by binding k to the current continuation prefixed by $\lambda\,(x_1,\ \ldots,\ x_n)\,.\,\textit{exp}$. The scope of k is *blk*. We use a different syntax from the other binding forms to make the scope of a **letcont** clear. The expression

$$\mathbf{throw}\ k\ (\textit{exp}_1,\ \ldots,\ \textit{exp}_n)$$

transfers control to the continuation bound to k, with the values of the $\textit{exp}_i$ bound to the parameters of k. Finally, the binding

$$\mathbf{let}\ k'\ =\ \mathbf{apply\_cont}\ k\ (\textit{exp}_1,\ \ldots,\ \textit{exp}_n)$$

partially applies the continuation k to its arguments and binds k' to a parameterless continuation that represents the control part of k. We illustrate the use of **apply_cont** to pass values from one thread to another in Section 4.3.

BOL continuations are not first-class, but instead have a lifetime that is restricted to the lifetime of their scope. This restriction makes it possible to stack allocate continuations and is the reason that the **apply_cont** operation is primitive. In terms of expressiveness, BOL continuations are weaker than Bruggeman *et al.*'s one-shot continuations [BWD96] because of their limited lifetimes. They are similar to the continuations of C-- [RP00b] except that C-- continuations are not one-shot. BOL continuations are perhaps closest to C's setjmp/longjmp mechanism [ANS90] in terms of expressiveness. The formal semantics for the fragments of BOL described in this section appear in Section 5.1.

2.2 Example — exception handlers

To illustrate how **letcont** and **throw** may be used to implement non-local control flow features, we give an example illustrating how they are used in the MOBY compiler to implement exception handling. Figure 2(a) gives a simple MOBY function that has an exception handler. In this example, the function f applies g to x divided by y. The application is wrapped by an exception handler that catches all exceptions and returns 17. The result of the call to g or the handler is incremented by 1 and returned as the result of f. The handler in this example will handle uncaught exceptions raised in g as well as the DivByZero

```
fun f (x : Int, y : Int) -> Int {
  (try g(x/y) except { ex => 17 }) + 1
}
```

(a) A MOBY exception handler

```
fun f (x, y, exh) = (
  let t1 =
    letcont exhK (ex) = 17
    in (
      let t2 = if I32Eq (y, 0)
           then throw exhK (DivByZero)
           else I32Div (x, y)
      in g (t2, exhK)
    )
  in I32Add (t1, 1)
)
```

(b) The BOL representation

Figure 2: A MOBY exception handler and its BOL representation

exception raised when y is zero.

Figure 2(b) shows the translation of f into BOL. When translating to BOL, we add an extra parameter to each function, which is its exception handler continuation (*e.g.*, the parameter exh in f). The exception handler in Figure 2(a) is translated to a **letcont** binding of the continuation exhK, which takes the exception packet as an argument (unused in this case). In this example, the continuation bound to exhK is used both in a local **`throw`** and is passed as the exception handler to `g`.

3 BOL thread operations

Although the continuations we have presented so far are an important part of our concurrency implementation, they are not sufficient by themselves to support concurrency and multiprocessing. The restricted lifetime of BOL continuations means that we cannot use them to implement thread creation, so we extend BOL with operations for thread creation and termination. We also extend BOL with operations for manipulating an abstract scheduling queue. By using abstract scheduling operations, we can support different runtime threading models without changing the front-end or libraries. The only changes are in the code generator, which maps BOL scheduling operations to machine code, and in the runtime system code. In this section, we describe our extensions to BOL, which collectively define an abstract interface to the thread support provided by the code generator and runtime system. A formal semantics for these operations is given in Section 5.2, and we describe two different implementations of them in Section 6.

3.1 Tasks and threads

The BOL thread model distinguishes between *tasks* and *threads*. Tasks correspond to operating-system threads, such as POSIX threads [But97], or processes, whereas threads correspond to language-level threads (*i.e.*, MOBY threads). Tasks provide the computation resources for running (or hosting) threads. We assume that tasks may run in parallel and that task creation, termination, and scheduling are handled by the underlying operating system.

There are several different ways to map threads to tasks. In the *one-to-one* mapping, there is a one-to-one correspondence between threads and tasks; in the *many-to-one* mapping, multiple threads are multiplexed on a single task, while in the *many-to-many* mapping, multiple threads are multiplexed over multiple tasks. When there are multiple threads per task, the implementation of the threading model may support preemptive scheduling between threads on a given task. The BOL thread model hides the choice of mapping from threads to tasks, along with other implementation details, which allows us to support different implementations of the threading model. This flexibility has two advantages: it supports improved application performance by allowing one to choose a threading model that best matches the needs of the application and it provides a testbed for experimenting with different implementation techniques.

BOL does not have an explicit representation of tasks; instead there is an implicit notion of the *host*, which is the task running the code. BOL does have an abstract *thread ID* type, which is an abstraction of thread-specific data (*e.g.*, its stack, locks, *etc.*). Each task has a *current* thread; the BOL operation **`get_thread_id`** is used to get the current thread ID. Threads that are not running (called *suspended*) are represented by a pair of a thread ID and a *resume* continuation.

3.2 Thread creation

Because BOL continuations have limited lifetimes, it is not possible to bootstrap new execution contexts using **`letcont`**. Instead, BOL provides a thread creation statement

$$\texttt{let}\ (\mathit{tid}, k)\ =\ \texttt{create}\,(f)$$

This statement creates and returns a new (suspended) thread for evaluating the function *f*. The new thread is represented by its ID (*tid*) and an initial resume continuation (k) that will execute the application $f()$. Unlike continuations created by **`letcont`**, k has unlimited lifetime, although it is still one-shot.

3.3 Scheduling support

Implementing synchronization and communication primitives requires operations for scheduling threads. To provide explicit control over such scheduling in an implementation-independent fashion, we augment BOL with a scheduling queue abstraction. The basic idea is that there is a collection, called the *ready queue*, of suspended threads that are *ready* to run and operations for adding and removing threads from this collection. The following BOL forms define the interface to the scheduling queue abstraction:

do lock_self ()

This operation locks the state of the current thread; the state is unlocked when the host task starts executing some other thread (see **dispatch** below). Threads must be locked prior to being added to the ready queue or stored in some communication data structure (*e.g.*, channel waiting queue).

do enqueue (*tid*, *k*)

This operation marks the argument thread as ready to run. It is an *unchecked* run-time error for this operation to be called on a thread that is either running or ready.

do enqueue_self (*k*)

This operation adds the current thread to the ready queue with resume continuation *k*. It is an *unchecked* runtime error to call this operation when the current thread is unlocked.

let (*tid*, *k*) = **dequeue**()

This operation removes a thread from the ready queue. This operation should succeed even when there are no ready threads; in that case either an idle thread or special scheduler thread should be returned by the implementation.

dispatch(*tid*, *k*)

This expression form is used to switch control to the thread *tid* at the continuation *k*. It is only enabled when the thread *tid* is unlocked (except in the special case where *tid* is the host's current thread). In addition to throwing to *k*, this expression unlocks the host task's current thread and makes *tid* be the host's new current thread.

These operations manage any concurrency control required to protect the scheduler data structures. Concurrency control on the thread-specific data (*e.g.*, stack) is provided by the combination of **lock_self** and **dispatch**. The typical usage pattern of these primitives has the form

```
letcont k () = ()
  in {
    do lock_self()
    scheduling code
    dispatch (nextTid, nextK)
  }
```

where the current thread's resume continuation is bound to `k` and then "*scheduling code*" enqueues `k` in either the ready queue or some communication primitive's waiting queue. The "*scheduling code*" is executed with the current thread's state being locked. Conceptually, the **lock_self** primitive can be thought of as marking the transition of the current thread from running to suspended. We need the locking because the host task is using the current thread's stack to execute the "*scheduling code*."

3.4 Thread termination

To release the resources associated with a given thread, we provide expression form

terminate()

which terminates the host task's current thread. Once a thread has been terminated, its host task is free to execute some other ready thread (*i.e.* via **dequeue** and **dispatch**).

4 Implementing concurrency in BOL

In this section, we give a few examples of how the combination of BOL continuations and thread operations are used to implement various concurrency mechanisms. These examples are implemented in BOL; in some cases this code is generated by the compiler as it translates MOBY concurrency features to BOL, whereas in other cases, the code is defined in library modules. In either case, the compiler is able to inline these definitions at their points of use. An extensive set of examples of implementing concurrency features using continuations can be found in Chapter 10 of [Rep99].

4.1 Context switching

Our first example is the implementation of `yield`, which performs explicit context switching. While `yield` is not necessary in a preemptively scheduled system, its implementation illustrates the dispatch pattern found in many other operations. It is implemented in BOL as follows:

```
fun yield (exh) {
  letcont myK () = ()
  in {
    do lock_self ()
    do enqueue_self (myK)
    let (tid, nextK) = dequeue()
    dispatch (tid, nextK)
  }
}
```

This code first reifies the return continuation of the `yield` function as the variable `myK`. It then enqueues this continuation and the current thread's ID in the scheduling queue. Finally, it dequeues another thread from the scheduling queue and dispatches to it.

4.2 Thread spawning

Another standard operation is spawning a new thread to execute some expression. This operation is more complicated than just using the **create** operation, since we need to protect against uncaught exceptions in the new thread. Figure 3 has the BOL implementation of a `spawn` function that creates a new thread to evaluate the function `f` that is passed to `spawn` as an argument. To handle uncaught exceptions in the evaluation of `f`, we define a new function (`f'`) that *wraps* the application of `f` with an exception handler of last resort and termination code. The implementation of `spawn` creates a new thread to evaluate `f'` and enqueues it in the scheduling queue. Then it yields control to the next ready thread.[4]

Because we are defining these higher-level concurrency operations in BOL, the compiler may be able to apply optimizations. For example, if the compiler inlines `spawn` at the site of its application to a known function, then it can inline the body of `f` in `f'`.[5] In a program that uses many small, short-lived threads,[6] this inlining can have a significant impact by reducing the number of function closures created and enabling other optimizations. For example, the compiler can sometimes detect the case where the thread does not raise an exception, which allows the "last resort" exception handler to be eliminated.

[4]Yielding is not necessary, but it lets the child thread run before the parent runs again.

[5]In the MOBY compiler, `spawn` is always applied to a known function because of the way translation from the surface language works.

[6]We have found such usage patterns common in CML programs.

```
fun spawn (f, exh) {
  fun f' () {
    do { letcont handler (exn) = ()
         in { f(handler) }
       }
    terminate()
  }
  let (tid, childK) = create (f')
  do enqueue (tid, childK)
  yield (exh)
}
```

Figure 3: BOL code for spawning a thread

4.3 MVars

Our third example is the implementation of *MVars*, which are a form of synchronous memory [Mil90, BNA91, PGF96, Rep99]. An MVar is a memory cell that is either *empty* or *full*. There are two operations on MVars:

`take` (*mv*)
: This operation takes a value out of the MVar *mv* and makes it empty. If no value is available (*i.e.*, *mv* is empty), then the caller is blocked until *mv* becomes full.

`put` (*mv*, *v*)
: This operation puts the value *v* into the MVar *mv*, changing *mv*'s state from empty to full. If *mv* is already full, then an exception is raised.[7]

In our implementation, we represent an MVar as a triple of a spinlock, the MVar state, and the MVar cell. This example uses the spinlock operations `AcquireLock` and `ReleaseLock` to provide lightweight concurrency control for the MVar's state. The state is an integer count of the number of threads waiting on the MVar, with `-1` signifying the full state. The cell stores the value in a full MVar and the list of waiting threads in an empty MVar (to simplify this example, we use a LIFO scheduling policy).

The BOL code that implements `take` and `put` appears in Figures 4 and 5. The `take` operation acquires the MVar's spin lock and then checks to see if the MVar is full. If it is full, then `take` gets the value, sets the state to empty, clears the cell, and releases the lock before returning the value. When the MVar is empty, the `take` operation must add the calling thread to the waiting list. Notice that the continuations in the waiting queue take the result as an argument; when `put` is called on an MVar that has waiting threads, it passes the value directly to the waiting taker as part of its wakeup sequence.

The `put` operation first checks if there are any waiting threads. If there are none (*i.e.*, the state field `mv#1` is `0`), it installs its value into the MVar. If there are no waiting threads, but the MVar is full (*i.e.*, the state field `mv#1` is `-1`), it raises the `FullMVar` exception. Otherwise, there are threads waiting on the empty MVar and the state field `mv#1` holds the number of waiting threads. In this case, the `put` operation decrements the number of waiting threads and removes the first waiting taker thread in the list. Since the waiting taker's continuation expects the MVar's value, the `put` operation uses **`apply_cont`** to bind the

[7]Some forms of MVars block the caller until the MVar is empty [Mil90].

```
fun take (mv, exh) {
  do AcquireLock(mv#0)
  if I32Eq(mv#1, -1)
    then {
      let res = mv#2
      do mv#1 := 0
      do mv#2 := 0
      do ReleaseLock(mv#0)
      res
    }
    else {
      letcont k(res) = res
      in {
        do lock_self ()
        do mv#1 := I32Add(mv#1, 1)
        do mv#2 := alloc(get_thread_id(), k, mv#2)
        do ReleaseLock(mv#0)
        let (tid, nextK) = dequeue()
        dispatch (tid, nextK)
      }
    }
}
```

Figure 4: BOL implementation of MVar take operation

value in a resume continuation that can be added to the scheduling queue. This approach to passing the value directly to the waiting taker contrasts with the typical imperative approach, in which the value is stored in shared memory that is protected by a mutex and a condition variable is used to signal the waiting taker. Note that we do not force a context switch when there is a waiting thread, although we might want to do so.

While these functions are too large to be good candidates for inlining, we can restructure them into a wrapper that handles the *fastpath* (*i.e.*, taking from a full MVar and putting into an empty one) and which calls a separate BOL function for the the more complicated case that involves synchronization. Then the wrapper function is a good candidate for inlining.

5 The semantics of BOL

In this section, we present a two-tiered formal system that defines the semantics of core elements of BOL. In Section 5.1 we present a model of single-threaded BOL programs; in Section 5.2, we lift this model to multi-threaded programs.[8]

5.1 Core BOL

The syntax of the single-threaded fragment of core BOL appears in Figure 6. Our semantics for this fragment, which appears in full in Appendix A, is patterned after the CEK machine [FSDF93]. It is intended

[8]For notational convenience, we do not require core BOL terms to be written in a direct style.

```
fun put (mv, v, exh) {
  do AcquireLock(mv#0)
  if I32Eq(mv#1, 0)
    then {
      do mv#1 := -1
      do mv#2 := v
      ReleaseLock(mv#0)
    }
  else if I32Lt(mv#1, 0)
    then {
      do ReleaseLock(mv#0)
      throw exh(FullMVar)
    }
    else {
      do mv#1 := I32Sub(mv#1, 1)
      let tid = mv#2#0
      let rk = mv#2#1
      do mv#2 := mv#2#2
      do ReleaseLock(mv#0)
      let rk' = apply_cont rk(v)
      enqueue (tid, rk')
    }
}
```

Figure 5: BOL implementation of MVar put operation

to serve as a tool for reasoning about the limited lifetimes of BOL continuations. To that end, we have replaced the continuation portion of the CEK machine with a pair consisting of a stack of continuations S and a label l indicating which location in the stack contains the next continuation. Whenever the evaluation of an expression reaches a value, the label is used to look up the next continuation k and the continuation stack is cut to one below l using the operation $S \downarrow l$. This cutting ensures that any continuations that have gone out of scope are no longer available and that the continuation k can be used only once. Similarly, when a continuation is thrown to, the continuation stack is cut to one below the label of the thrown-to continuation.

The evaluation of **let** k = **apply_cont** $e_1(e_2)$ **in** e_3 evaluates e_1 to a label l denoting a continuation and e_2 to a value V_d. It then modifies in place the continuation at label l to store the value V_d in the continuation. Evaluation then proceeds with expression e_3. If later code throws to k with a unit argument, we use the stored value as the argument for the continuation.

5.2 Core BOL with scheduling primitives

To model BOL's scheduling primitives, we extend the grammar for Core BOL with the operations shown in Figure 7. We then layer a machine abstraction on top of the thread states from the single-threaded semantics. Appendix B presents the full system in detail.

In this model, a machine state M is a triple of the form:

$$\{\!: P \quad ; \quad Q \quad ; \quad T \quad :\!\}$$

where intuitively P is a pool of threads, Q records the threads waiting to run, and T gives the state of each

$$
\begin{array}{rcl}
e & ::= & V \\
 & | & \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2 \\
 & | & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
 & | & e_1 e_2 \\
 & | & \mathbf{letcont}\ k(x)\ = e_1\ \mathbf{in}\ e_2 \\
 & | & \mathbf{let}\ k = \mathbf{apply\_cont}\ e_1(e_2)\ \mathbf{in}\ e_3 \\
 & | & \mathbf{throw}\ e_1(e_2) \\
 & & \\
V & ::= & c \mid x \mid \lambda x.e
\end{array}
$$

$$
\begin{array}{rcl}
e & \in & \textit{CBOL} \\
V & \in & \textit{Values} \\
x,\ k,\ tid & \in & \textit{Variables} \\
c, \bullet & \in & \textit{Constants} \\
 & & \text{We use } \bullet \text{ to denote the unit constant.}
\end{array}
$$

Figure 6: Abstract Syntax of Core BOL (*CBOL*).

underlying task. Technically, P is a finite map from thread IDs to thread states; Q is a set of (thread ID, continuation label) pairs; and T is a finite map from task IDs to the state of the associated task: `Idle` for an idle task and (i, Md) for a task executing thread i in mode Md (Md is either U for normal execution or L for privileged).

Active threads are those which have a task executing them. *Suspended* threads are represented as a pair of a thread ID and a continuation label, whether this pair is stored in the ready queue Q or in a data structure. Intuitively, the thread ID identifies the suspended thread in the pool P, while the continuation label indicates where in the suspended thread's continuation stack computation should resume.

The semantics includes two kinds of transitions: thread-local and scheduling-related. A thread-local transition occurs in machine M whenever there is an *active* thread i whose state $P(i)$ matches the left-hand side of one of the transition rules of the single-threaded semantics. In contrast, a scheduling-related transition manipulates the thread pool, the ready "queue," and/or the tasks. The semantics of each of the BOL scheduling primitives may involve some initial thread-local transitions to evaluate arguments, after which there is a scheduling-related transition to describe the desired behavior. We give an intuitive description of the key rules here.

The semantics of the **`create`** construct adds a new thread to the thread pool for evaluating the argument function. The thread ID of the new thread and the continuation label for applying its function to the unit value are bound in the thread invoking **`create`**. Note that this pair constitutes a suspended thread for evaluating the argument function.

When run in a thread with ID i, the **`lock_self`** primitive updates the state of i to be privileged, L, ensuring that the thread will not be interrupted until it executes a **`dispatch`**, which (among other things) unlocks the state of the dispatching thread. The **`lock_self`** primitive allows a thread to execute "scheduling" code after it has finished its main execution.

The two enqueue operations add a (thread ID, continuation label) pair to the collection of pending threads Q. The first, **`enqueue_self`**, adds the ID of the executing thread and the argument continuation label to Q. In this case, the executing thread should be in the L mode. The second, **`enqueue`**, adds the argument thread ID and continuation label to Q. In this case, the argument thread should not be in the ready queue already nor should it be running.

The **dequeue** operation uses the (unspecified) $\mathrm{next}(Q)$ function to choose some ready thread to pass to the executing thread, presumably to schedule for execution using the **dispatch** operation. Different implementations of the next function correspond to different scheduling policies.

The **dispatch** primitive causes control of the underlying task to transfer from the currently executing thread i to the suspended thread denoted by the argument thread ID and continuation label l. The dispatching thread should be executing in the locked state. If the dispatch were self-inflicted, this operation essentially throws to l and unlocks the state of the thread. In the non-self-inflicted case, thread i becomes suspended (and consequently unlocked) after executing **dispatch**.

Executing the **get_thread_id** primitive in thread i returns i, while executing the **terminate** primitive causes the host thread to be removed from the thread pool P.

Finally, the semantics includes non-deterministic rules for managing task resources. Two rules permit idle tasks to be added or removed from the task collection T at any time, allowing an implementation freedom in how it manages underlying system threads. A third non-deterministic rule allows a busy task to swap out its computation to the ready queue, while a fourth such rule allows an idle task to swap in a thread from the ready queue. Different choices about when to apply such rules, paired with different implementations of the next function, correspond to different scheduling and preemption policies.

$$
\begin{array}{rcl}
e & ::= & \ldots \\
 & | & \textbf{let}\ (tid, k) = \textbf{create}\ e_1\ \textbf{in}\ e_2 \\
 & | & \textbf{let}\ tid = \textbf{get\_thread\_id}()\ \textbf{in}\ e \\
 & | & \textbf{do}\ \textbf{lock\_self}()\ \textbf{in}\ e \\
 & | & \textbf{do}\ \textbf{enqueue\_self}\ (e_1)\ \textbf{in}\ e_2 \\
 & | & \textbf{do}\ \textbf{enqueue}\ (e_1, e_2)\ \textbf{in}\ e_3 \\
 & | & \textbf{let}\ (tid, k) = \textbf{dequeue}()\ \textbf{in}\ e \\
 & | & \textbf{dispatch}\ (e_1, e_2) \\
 & | & \textbf{terminate}()
\end{array}
$$

Figure 7: Abstract Syntax of Core BOL refined to include scheduling primitives.

---

6 Implementing BOL threads

The BOL threading model, described in Section 3, is implemented as a collaboration between the MOBY compiler's code generator and the runtime system. In this section, we describe two different implementations of this model. The first is a one-to-one implementation that maps each language-level thread to its own host task. The second is a simple many-to-many implementation that multiplexes language-level threads over a collection of tasks. Each of these implementations does most of its work in its runtime system, which is written in C (with some **asm** directives). We expect to migrate some of the work to the code generator in the future. We describe these implementations below.

6.1 Representation of continuations

In both implementations, we represent a BOL continuation as a pair of the continuation's code address and the stack pointer to the continuation's stack frame. We allocate space for a continuation in the stack frame of its procedure. Additional space is allocated for the continuation's parameters. With this representation, constructing a continuation takes only a couple machine instructions. Throwing to a continuation requires storing its arguments, loading its stack pointer, and jumping to its address. Because BOL continuations do not require stack segment manipulations, they can be implemented more efficiently than the one-shot continuations of Bruggeman *et al.* [BWD96]. Furthermore, BOL continuations do not force a particular stack implementation; they can be implemented on a contiguous stack or a segmented stack.

6.2 Tasks

Tasks are the computation agents that host the execution of threads; they typically correspond to operating-system threads or processes. On multiprocessor systems tasks can run in parallel. We assume the underlying system provides mutex locks, condition variables, and semaphores as found in the POSIX thread API [But97]. Both of our current implementations map each runtime system task onto a POSIX thread.

6.3 The scheduler interface

Both of our current implementations map the BOL threading model directly onto a collection of C functions, whose prototypes appear in Figure 8. These functions manipulate four abstract C types: we use `Closure`

```
void Create (Closure *clos, ThreadID *tidOut, Cont *kOut);

void LockSelf (Task *host);
void Enqueue (Task *host, ThreadID tid, Cont k);
void EnqueueSelf (Task *host, Cont k);
void Dequeue (Task *host, ThreadID *tidOut, Cont *kOut);
void Dispatch (Task *host, ThreadID tid, Cont k);

void Terminate (Task *host);
```

Figure 8: Runtime interface of BOL thread operations

to represent closure values, `Cont` to represent continuations, `ThreadID` to represent thread-local data, and `Task` to represent task-specific information.

6.4 A one-to-one scheduler

Our one-to-one implementation of the BOL threading model uses the underlying operating system's scheduling mechanisms to provide thread-level concurrency. This implementation maintains a one-to-one correspondence between threads and tasks. Each time a new thread is created, a new task is created as well (or recycled from a cache of idle tasks). Each thread always runs on its associated task, and a given task runs only the code for its thread or special scheduling code (which we represent as a BOL continuation). The dequeue operation returns either the running thread's resume continuation (if it has been previously enqueued) or the associated task's scheduling continuation.

In this implementation, the `Task` type is a C data structure with following fields:

`Mutex lock;`
: This field holds a lock that controls access to the task's state.

`Cond wait;`
: We use this condition variable to suspend the task when the associated thread is idle.

`Cont resumeCont;`
: If non-`NULL`, this field holds the continuation at which to resume executing the task's thread. A `NULL` value indicates no resume continuation has been scheduled via an enqueue operation.

`Cont schedCont;`
: This field holds the scheduling continuation for the task.

`ThreadID myThread;`
: This field holds the thread ID of the task's thread.

The `ThreadID` type is a pointer to a data structure that has the following field:

`Task myTask;`
: This field points to the corresponding task data structure.

6.4.1 Thread creation

Thread creation involves creating a new task, initializing the task's scheduling continuation, and creating a new thread ID. This process requires coordination between the routine that starts the new task (`Create`) and the task startup code. Space limitations prevent us from presenting the code, but the basic idea is that `Create` passes the address of a stack-allocated data structure to the new task startup code. On entry, this structure contains the task data structure and the closure containing the code for the new thread. On return, the structure contains the initial continuation for the new thread. The data structure also contains a semaphore that the new task uses to signal when it is ready to begin execution.

6.4.2 Scheduling operations

The most interesting aspect of our one-to-one implementation is how the BOL scheduling abstraction is mapped onto a one-thread-per-task implementation model. In this model, threads are not multiplexed across tasks, and so the runtime system does not need a scheduling queue. Instead, it uses the thread scheduling operations to suspend and resume the associated task.

The `LockSelf` operation is a noop in this implementation, since a thread is always hosted by the same task.

The `Enqueue` and `EnqueueSelf` operations record the given continuation in the resume continuation field of the task associated with the enqueued thread. In the case of the `Enqueue` operation, we must first acquire the associated lock to guard against a potential race condition with the `Dequeue` operation and/or task scheduling code. We must also wakeup the underlying task of the thread being enqueued.

```
void Enqueue (Task *host, ThreadID tid, Cont k)
{
    Task *task = tid->myTask;
    MutexLock (task->lock);
    task->resumeCont = k;
    CondSignal (task->wait);
    MutexUnlock (task->lock);
}
void EnqueueSelf (Task *host, Cont k)
{
  host->resumeCont = k;
}
```

The `Dequeue` operation returns the current thread ID and a continuation that depends on the thread's state. If the thread is suspended (*i.e.*, its host's resume continuation is `NULL`), then the returned continuation is the task's scheduler continuation. If the thread is enabled, the stored resume continuation is returned and the resume field is nullified.

```
void Dequeue (Task *host, ThreadID *tidOut, Cont *kOut)
{
    MutexLock (host->lock);
    if (host->resumeCont == NULL)
        *kOut = host->schedCont;
    else {
        *kOut = host->resumeCont;
        host->resumeCont = NULL;
    }
    MutexUnlock (host->lock);
    *tidOut = host->myThread;
}
```

The scheduler continuation (stored in the `schedCont` field) is implemented as an assembly-code wrapper around the following function:

```
void Scheduler (Task *host)
{
    Cont k;
    MutexLock (host->lock);
    if (host->resumeCont == NULL)
        CondWait (host->lock, host->wait);
    k = host->resumeCont;
    host->resumeCont = NULL;
    MutexUnlock (host->lock);
    THROW(k);
}
```

Once the scheduler has the task's lock, it checks the task's resume continuation. If the continuation is `NULL`, the task blocks on its condition variable to wait for a resume continuation. Although this check might seem redundant, as the `Dequeue` code only returns the scheduler continuation when the resume continuation is `NULL`, the check is necessary because some other thread might have `Enqueue`d this thread between the time that the `Dequeue` code released the task lock and the `Scheduler` code acquired it.[9] Once the thread is signaled by `Enqueue`, control is transfered to the resume continuation using the `THROW` macro, which expands to a sequence of **`asm`** directives that throw to the continuation `k`.

The dispatch operation in this implementation is equivalent to a BOL **`throw`** operation.

[9]An alternative would be to hold the lock between the `Dequeue` and `Scheduler` code, but it is simpler and more robust to have each operation manage the lock independently.

```
void Dispatch (Task *host, ThreadID tid, Cont k)
{
    THROW(k);
}
```

6.4.3 Thread termination

Thread termination involves freeing the task data structure and terminating the underlying system thread. As an optimization, we can keep a pool of idle system threads around to reduce the cost of thread creation.

6.5 A many-to-many scheduler

We have also implemented the BOL thread abstraction using a simple many-to-many scheme in which threads are multiplexed over multiple tasks. In this implementation, threads execute in a conventional stack-based environment, but the stacks are smaller than the default OS-thread stack [10] and are managed by the runtime system. To support multiplexing, the runtime system maintains a single global queue of ready threads.

The `Task` type is much simpler in the many-to-many implementation. For scheduling purposes, it has the following fields:

`Cont schedCont;`
: This field holds a continuation that is used when there are no ready threads.

`ThreadID myThread;`
: If the task is executing a thread, this field holds the associated thread ID. Otherwise, it holds the special value `idleTid`.

The `ThreadID` type is a pointer to a *thread descriptor*, which is a data structure that has the following fields:

`Task myTask;`
: If the thread is executing, this field points to the underlying task data structure. Otherwise, it contains the value `0`.

`int lock;`
: This field is used to lock the thread's state during context switches.

6.5.1 Thread creation

Thread creation is simpler in the many-to-many case, because it is independent of task creation. The `Create` function allocates a new stack and thread descriptor and initializes the stack with a continuation that will launch the thread. The launching code is a small piece of assembly code that calls the thread's initial function. The `Create` function returns a pointer to the new thread descriptor (the thread ID) and the initial continuation.

6.5.2 Scheduling operations

The many-to-many scheduler uses a single global queue of ready threads with the following interface:[11]

[10] Our stacks are 128K bytes vs. the 2Mb default stack used by the Linux implementation of POSIX threads.

[11] We expect to move to a per-task scheduling queue at some point, which is one reason for the `host` argument to the scheduling operations.

```
void RdyEnq (ThreadID tid, Cont k, Bool wake)
void RdyDeq (ThreadID *tidOut, Cont *kOut)
```

The ready queue is protected by a mutex lock and has an associated condition variable that is used to block tasks when the queue is empty (see the description of `Dequeue` below). Tasks blocked on the ready queue's condition variable are signaled by `RdyEnq` when its `wake` parameter is true.

The `LockSelf` operation sets the current thread's lock, which protects against the thread being rescheduled by some other task before it dispatches a new thread.

```
void LockSelf (Task *host)
{
  host->myThread->lock = LOCKED;
}
```

Depending on the underlying hardware, it may be necessary to use a write barrier to guarantee that all processors see the effect of `LockSelf` before subsequent scheduling operations. The thread's lock is released when the next thread is dispatched.

The `Enqueue` and `EnqueueSelf` operations simply add a (thread ID, continuation) pair to the global ready queue.

```
void Enqueue (Task *host, ThreadID tid, Cont k)
{
    RdyEnq (tid, k, TRUE);
}
void EnqueueSelf (Task *host, Cont k)
{
    RdyEnq (host->myThread, k, FALSE);
}
```

Because `EnqueueSelf` does not increase the net number of ready threads, there is no advantage to waking a blocked task.

Dequeueing a thread from the global queue requires checking to see if any ready threads are available. If not, we schedule a special continuation that will block this task on the condition variable for the ready queue.

```
void Dequeue (Task *host, ThreadID *tidOut, Cont *kOut)
{
    if (Empty(ReadyQ)) {
        tidOut = idleTid;
        kOut = host->schedCont;
    }
    else
        Dequeue (ReadyQ, tidOut, kOut);
}
```

Dispatching a thread in the many-to-many implementation requires some bookkeeping in the task and thread structures and also clearing the current thread's lock. We also need to verify that the thread we are about to start executing is not locked.

```
void Dispatch (Task *host, ThreadID tid, Cont k)
{
    int *lock = &(host->myThread->lock);
    host->myThread->myTask = 0;
    host->myThread = tid;
    tid->myTask = host;
    *lock = UNLOCKED;
    while (tid->lock == LOCKED)
        continue;
    THROW(k);
}
```

Note that once we have unlocked the current thread we cannot use its stack; we use **asm** directives to ensure this property.

An important property of our thread locking scheme is that it is deadlock free. Assuming that the BOL code follows the guidelines of Section 3, any task that locks its current thread in `LockSelf` cannot be blocked before unlocking the thread in `Dispatch` (this property is why `Dequeue` returns a scheduler continuation when the ready queue is empty instead of blocking immediately). The thread locking scheme also ensures that two tasks will not try to execute using the same thread's stack at the same time.

6.5.3 Thread termination

Thread termination in the many-to-many case requires some care to avoid reusing the thread's stack before it is fully terminated. Fortunately, we can use the existing locking mechanism to protect against this race condition. The `Terminate` function first locks the current thread's stack and then puts it into the list of free stacks. It then dequeues and dispatches the next thread, which has the side-effect of unlocking the stack. The stack allocation code use by `Create` will not allocate a locked stack.

6.6 Preemption

Preemptive scheduling is an important feature for concurrent programming. Like garbage collection, it supports modularity by freeing the programmer from having to manage resources explicitly. To this end, tasks in our many-to-many implementation periodically preempt the current thread, causing another ready thread to be run. We also use preemption in both implementations to synchronize tasks for garbage collection (see Section 6.7). Our runtime system uses UNIX signals to preempt tasks. To avoid problems in the case where a signal arrives in the midst of a context switch, we use the UNIX `sigaltstack` mechanism to set a different stack for handling signals.

A common way to implement preemption is by defining *safe points* where the state of the thread can be captured in a predictable way [LCJS87, Rep90]. Safe points have the disadvantage that they add execution overhead. For example, in CML GC tests are used as safe points, which means that even non-allocating loops must have GC tests.

In MOBY, we don't use safe points, so we can avoid their execution overhead. Instead, we support preemption at any program point using *PC maps*. The compiler generates such maps, which map program counter (PC) values to information about the generated code. This information includes liveness and type information used by the garbage collector. The PC map is also used to mark heap allocations that must be *atomic* with respect to preemption. If the runtime attempts to preempt a thread inside an atomic allocation, the PC-map entry provides enough information to *roll back* execution to the start of the allocation [SCM99]; the allocation is then attempted from the beginning when the preempted thread is resumed. The PC-map entry for an allocation also contains a bound on the amount of memory required before the next heap limit check; this information is used to perform a heap limit check prior to resuming the thread. One other detail that the compiler must handle is ensuring that the values used to initialize the allocated object are kept live until the end of the atomic region. We ensure this property by generating a special pseudo operation at the very end of the atomic allocation sequence that *uses* all of the initial values, which forces the register allocator to preserve the initialization values until the end of the allocation. An alternative proposed by Shivers *et al.* is to preallocate the object atomically and then incrementally initialize it [SCM99]. Shivers' technique requires additional information in the PC maps to help the GC manage partially initialized objects.

6.7 Allocation and garbage collection

Our current implementation uses a "stop-the-world" garbage collector based on Smith and Morrisett's mostly-copying collector [SM97]. The collector organizes the heap into small (8Kb) pages and each task has its own list of free pages from which to allocate. We dedicate a register, called the *allocation pointer*, to point to the next word to allocate in the current page. The compiler generates inline allocation code and heap limit checks. As is typical in ML-style languages, objects are initialized when they are allocated. Since allocation pages are a power of two in size and alignment, we can compute the start and end of the current allocation page easily from the allocation pointer. We use the first few words in the allocation page to hold task-specific data, including the current task descriptor and the task's list of free pages. When a heap limit check hits the end of the current allocation page for a given task, a small assembly stub is called that gets another page from the task's free list. Since each task has its own list of free allocation pages, synchronization is only required when the task's free list is empty. In this case, control passes to the run-time system, where a small number of pages are grabbed from the global free list. Access to the global free list is protected by a global lock.

If the global free list is empty when a task T requires more memory, then a garbage collection is required. To perform a garbage collection, we must first suspend all other tasks and record their registers so that the collector can access the root set. To initiate this process, T sends a POSIX signal to each of the other tasks. The signal handler for each task saves its register state and transfers control back to a special runtime system routine that suspends T. This runtime routine uses the PC maps described in Section 6.6 to rollback preempted allocations. T then waits for the other tasks to record their states and suspend themselves before proceeding with the collection. Once the collection is finished, the other tasks are resumed.

7 Compiler issues

In this section, we describe a number of issues in the implementation of BOL continuations and threads. These issues include continuation-specific optimizations, continuation representations, register allocation, and supporting thread preemption.

7.1 Cluster conversion

A cluster is a collection of *fragments*, which are BOL functions that contain no nested function bindings. At runtime, these fragments share the same stack frame; intuitively, a cluster is a representation of a procedure's control-flow graph. One fragment in a cluster is the *entry fragment* and all calls to the cluster target it; the other fragments in the cluster are only called by internal tail calls, which we write as **`goto`**s.

An important part of the cluster conversion phase is *Local CPS* (LCPS) conversion, which is used to collapse nested recursions into a single cluster [Rep01]. We also use LCPS conversion to introduce join-point functions when we restructure **`letcont`** bindings by converting their right-hand-side into a **`goto`** to a separate fragment. This transformation serves two purposes: it makes management of free variables easier and it allows a later optimization phase to convert local **`throw`**s to **`goto`**s. For example, the BOL code from Figure 2(b) is translated into the representation in Figure 9. In this figure, we have written the

```
fun f (x, y, exh) =
{
  letcont exhK (exn) = goto exhFrag ()
    if I32Eq (y, 0)
      then goto exhFrag ()
      else {
        let t2 = I32Div(x, y)
        let t3 = g (t2, exhK)
          goto rcont(t3)
      }
}
and label exhFrag () = goto rcont (17)
and label rcont (t1) = return I32Add(t1, 1)
```

Figure 9: The cluster representation of Figure 2(b)

cluster as a collection of mutually recursive fragment definitions, where `f` is the cluster entry point. Control transfers from one fragment to another are written as **`goto`**s with arguments.[12] The fragment `exhFrag` is the right-hand-side of the original **`letcont`** binding; notice that the local **`throw`** to **`exhK`** when `y` is zero was converted to a **`goto`** to `exhFrag`. The fragment `rcont` was introduced by LCPS conversion to be the join point for the right-hand-side and body of the original **`letcont`** binding.

7.2 Register allocation issues

The most complicated aspect of compiling the **`letcont`** binding is ensuring that the register allocator will manage callee-save registers correctly. The recommended way to manage callee-save registers in MLRISC is to copy them to fresh pseudo-registers (we call these *shadow registers*) on function entry and restore the callee save registers from the shadow registers on function exit. If the register allocator needs to use a callee-save register, the incoming value will be spilled at the function's entry and restored at exit, otherwise the moves are coalesced [GA96].

With the introduction of continuations, we need to provide the MLRISC framework with information about possible non-local control transfers [RP00b, Rei01]. Consider the **`letcont`** binding of a continuation `k`. For any call site in `k`'s cluster that has `k` as an argument, we add an implicit control-flow edge from the call to the entry label associated with `k`. Since the callee-save registers are in an unknown state upon entry to the continuation fragment, we place a special pseudo instruction at the fragment's entry that *defines* the callee-save registers. This pseudo instruction is followed by a parallel copy that restores the callee-save registers from the shadow registers. Figure 10 illustrates this technique for the cluster in Figure 9. In this figure, each fragment is in its own box; normal controlflow edges are solid arrows and the dotted arrow represents the implicit controlflow induced by the passing of `exhK` as an argument to `g`. The shaded italic code is MLRISC instructions introduced to manage the callee-save registers, where *cs* are the callee-save registers and *srs* are the shadow registers. In practice, we introduce additional pseudo-registers to split the live range of the shadow registers and we use shrink-wrapping to localize register saving and restoring.

[12]Note that if the cluster were a recursive function, then there would be a call to its entry fragment.

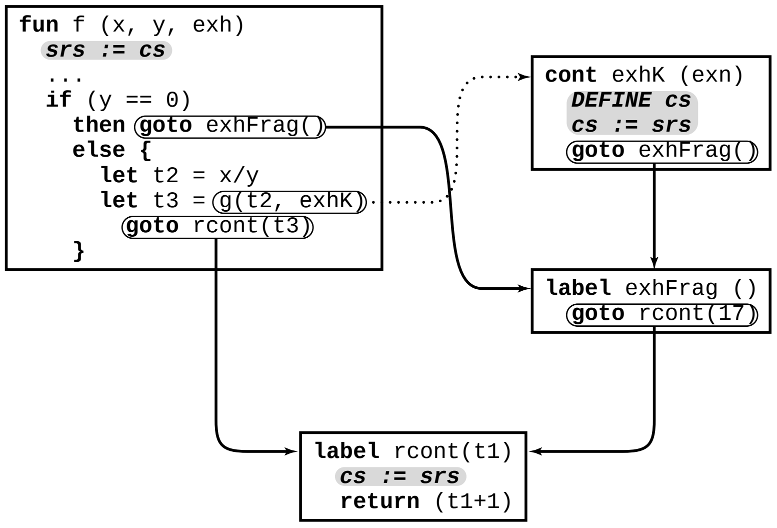

Figure 10: CFG representation of Figure 9

8 Future work

While our current implementations demonstrate the flexibility and expressiveness of our design and while they are well suited for some workloads, there is much room for improvement. We are especially interested in improving the many-to-many implementation, since most system thread implementations are too expensive to support very large numbers of threads. In this section, we briefly discuss some possible improvements.

8.1 Scheduling

Our many-to-many implementation uses a single global scheduling queue for ready threads. This choice is problematic both because the lock on the queue is a point of contention and because threads bounce from task to task, which does not preserve cache locality on multiprocessors. Our design can support per-task scheduling queues easily, since the scheduling operations already take the host task as an argument. This scheme will require a mechanism for load balancing across threads.

8.2 Stack management

Our many-to-many implementation uses fixed-size contiguous stacks, which is both wasteful for threads that require only a few stack frames to execute and inadequate for threads that have deep recursion. There are several alternative stack management schemes that can provide more flexibility:

- *Stack copying* — threads run on a contiguous stack; if a thread overflows its stack, then the runtime grows the stack, which may require copying it.
- *Segmented stacks* — threads run on a segmented stack; when a thread overflows one segment, another segment is added [HDB90, BWD96].

- *Linked stack frames* — threads heap allocate their stack frames and the stack is represented as a linked list. This scheme provides the lowest per-thread space overhead, but increases the cost of function applications and complicates interoperability.

Our current architecture is compatible with any of these schemes, although implementing them will require changes in the calling conventions.

8.3 Compiler support

Over time, we plan to move parts of the implementation from the runtime system into the compiler. In particular, we would like to generate inline code for the fast paths through the scheduling operations.

9 Related work

The correspondence between continuations and multiple threads of control dates back to Wand's seminal work [Wan80] and has been exploited in many concurrency libraries [HFW84, CM90, Ram90, Rep91]. Various researchers have observed that the full power of first-class continuations is not necessary for implementing either exceptions or concurrency. Bruggeman *et al.* propose a form of one-shot continuations that are sufficient to support concurrency [BWD96]. These one-shot continuations are more expensive to create than BOL continuations because they require beginning a new stack segment; the lifetime restrictions that we place on BOL continuations make them cheap to create in contrast. This cheapness comes at a cost, however, as it requires an additional thread creation operation to support concurrency. Essentially, we have isolated the expensive case in the **`create`** operator.

C-- is a "portable assembly language" with C-like syntax [PNO97]. It has a restricted form of continuation that is similar in power to ours and can be used to implement non-local controlflow [RP00b]. Ramsey and Peyton Jones have proposed using C-- continuations to implement concurrency [RP00a], although to our knowledge there is currently no implementation of the C-- concurrency mechanisms. Like our approach, their goal is to support a range of different implementation techniques in a common framework. In contrast to BOL, C-- is a compiler *target*, not an IR. In our approach, the source language compiler can optimize the BOL concurrency primitives using the same optimizations it uses on the rest of the IR; the C-- compiler converts C-- code output from the source language compiler into machine code and performs its own optimizations in the process. In many ways, the C-- framework plays a rôle similar to the MLRISC [GGR94] framework that we use for code generation. Indeed, one could imagine retargeting the MOBY backend to generate C-- code from our BOL representation.

Dealing with the combination of preemption and atomic code sequences, such as allocation, has been addressed in a number of systems. The Trellis/Owl system used a VAX instruction interpreter in its runtime to execute the preempted thread's code to the end of the critical region [MK87]. This approach is not very portable and introduces significant interpretive overhead to preemption. Bershad *et al.* proposed using roll backs as a way to implement efficient mutual exclusion on uniprocessors [BRE92]. Shivers *et al.* applied this idea to heap allocation [SCM99]. In their implementation, they use the low bit of the allocation register as a flag to signal when execution is inside an atomic allocation and their runtime system uses PC maps to determine how to roll back the program counter. Our approach dispenses with using a bit of the allocation pointer as a flag, since the PC map itself can be used to determine when execution is inside an atomic allocation. In a many-to-many implementation of the BOL model, we may also use PC maps to delay preemption of threads in critical regions [ABLL92]. One disadvantage of our approach is that PC maps can consume significant storage. We currently see a doubling of the code size on the IA32, but other researchers have reported success with more compact representations of this sort of information [DMH92, SLC99, Tar00], so we expect that the space overhead of PC maps can be reduced significantly.

JAVA is probably the most widely used concurrent language. It supports a shared-memory programming model with objects playing the rôle of monitors. Because any JAVA object can be a monitor, a naïve implementation of JAVA will incur significant space and time overhead. The main focus on implementing concurrency in JAVA has been on reducing the space overhead of per-object locks [BKMS98] and statically eliminating unnecessary synchronizations [OOP99]. Our work is complementary to these efforts.

10 Conclusion

This paper describes the design of a direct-style compiler IR suitable for implementing a wide range of surface-language concurrency features while allowing flexibility in the back-end implementation. The features that this IR includes to support concurrency include weak continuations and primitives for thread creation, scheduling, and termination. Although this work has been done in the context of implementing concurrency for the MOBY programming language, the model is general and could be used in the implementation of other concurrent languages.

The examples in Section 4 illustrate that the IR is expressive enough to support a variety of surface language concurrency mechanisms. The back-end flexibility allows us to experiment with a range of implementation techniques and to provide implementations with different performance characteristics, each suitable for a different class of application. For example, a computationally bound application with few threads would perform best on a one-to-one implementation, whereas a highly threaded application would benefit more from the low-overhead threads of a many-to-many implementation. The implementations described in Sections 6.4 and 6.5 illustrate that the model can accommodate a wide range of implementations.

The operational semantics, presented informally in Section 5 and described in detail in the appendices, precisely specify the behavior of the IR and provide a tool for reasoning about the correctness of a given implementation.

The performance goals for our implementations are to provide low-overhead synchronization and communication, fast thread creation, space-efficient thread data structures, and access to the multiprocessing and concurrency mechanisms of the underlying operating system. We have implemented both the one-to-one mapping of Section 6.4 and the many-to-many mapping of Section 6.5 on top of the Linux implementation of POSIX threads. The one-to-one implementation satisfies the first and fourth of our performance goals, but because it maps each MOBY thread onto its own POSIX thread, this implementation cannot provide fast thread creation or space efficient threads. In contrast, the many-to-many implementation of Section 6.5 uses less space per thread and supports efficient thread creation. We expect that the refinements described in Section 8 will let us achieve our performance goals.

Acknowledgments

Daniel Grossman implemented the first version of PC maps and multiprocessing runtime support during the summer of 2000. We had significant help and support with the MLRISC framework from Lal George and Allen Leung. Discussions with Matthias Blume and Lal George helped us work out the implementation details.

References

[ABLL92] Anderson, T. E., B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, **10**(1), February 1992, pp. 53–79.

[ADH$^+$98] Abelson, H., R. Dybvig, C. Haynes, G. Rozas, N. Adams IV, D. Friedman, E. Kohlbecker, G. Steele Jr., D. Bartley, R. Halstead, D. Oxley, G. Sussman, G. Brooks, C. Hanson, K. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, **11**(1), August 1998, pp. 7–105.

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, August 1991, pp. 1–26.

[ANS90] American National Standards Institute, Inc., New York, NY. *American National Standard for Information Systems — Programming Language — C*, 1990.

[BKMS98] Bacon, D. F., R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998, pp. 258–268.

[BNA91] Barth, P., R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, vol. 523 of *Lecture Notes in Computer Science*, New York, N.Y., August 1991. Springer-Verlag, pp. 538–568.

[BRE92] Bershad, B. N., D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, September 1992, pp. 223–233.

[But97] Butenhof, D. R. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA, 1997.

[BWD96] Bruggeman, C., O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*, May 1996, pp. 99–107.

[CM90] Cooper, E. C. and J. G. Morrisett. Adding threads to Standard ML. *Technical Report CMU-CS-90-186*, School of Computer Science, Carnegie Mellon University, December 1990.

[DMH92] Diwan, A., J. E. B. Moss, and R. L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, June 1992, pp. 273–282.

[FPR01] Fisher, K., R. Pucella, and J. Reppy. A framework for interoperability. In N. Benton and A. Kennedy (eds.), *Electronic Notes in Theoretical Computer Science*, vol. 59, New York, NY, 2001. Elsevier Science Publishers.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999, pp. 37–49.

[FSDF93] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 237–247.

[GA96] George, L. and A. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, **18**(3), May 1996, pp. 300–324.

[GGR94] George, L., F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, April 1994, pp. 83–97.

[GR93] Gansner, E. R. and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, vol. 1 of *Software Trends*, pp. 61–80. John Wiley & Sons, 1993.

[HDB90] Hieb, R., R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 66–77.

[HFW84] Haynes, C. T., D. P. Friedman, and M. Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, July 1984, pp. 293–298.

[HJT$^+$93] Hauser, C., C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993, pp. 94–105.

[LCJS87] Liskov, B., D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, November 1987, pp. 111–122.

[Mil90] Milewski, J. Functional data structures as updatable objects. *IEEE Transactions on Software Engineering*, **16**(12), December 1990, pp. 1427–1432.

[MK87] Moss, J. and W. Kohler. Concurrency features for the trellis/owl language. In *ECOOP'87*, vol. 276 of *Lecture Notes in Computer Science*, 1987, pp. 171–180.

[OOP99] *OOPSLA'99 Conference Proceedings*, November 1999. This proceedings contains several papers on eliminating synchronizations in JAVA using escape analysis.

[PGF96] Peyton Jones, S., A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, January 1996, pp. 295–308.

[Pik89] Pike, R. A concurrent window system. *Computing Systems*, **2**(2), 1989, pp. 133–153.

[PNO97] Peyton Jones, S., T. Nordin, and D. Oliva. C–: A portable assembly language. In *9th International Workshop on the Implementation of Functional Languages*, vol. 1467 of *Lecture Notes in Computer Science*, New York, N.Y., September 1997. Springer-Verlag, pp. 1–19.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.

[Rei01] Reig, F. Annotations for portable intermediate languages. In N. Benton and A. Kennedy (eds.), *Electronic Notes in Theoretical Computer Science*, vol. 59, New York, NY, 2001. Elsevier Science Publishers.

[Rep90] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Department of Computer Science, Cornell University, Ithaca, NY, August 1990.

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep99] Reppy, J. H. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, England, 1999.

[Rep01] Reppy, J. Local CPS conversion in a direct-style compiler. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, January 2001, pp. 13–22.

[RP00a] Ramsey, N. and S. Peyton Jones. Featherweight concurrency in a portable assembly language. Unpublished paper available at `http://www.cminusminus.org/abstracts/c--con.html`, November 2000.

[RP00b] Ramsey, N. and S. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the SIGPLAN'00 Conference on Programming Language Design and Implementation*, June 2000, pp. 285–298.

[SCM99] Shivers, O., J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, September 1999, pp. 48–59.

[Shi97] Shivers, O. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, January 1997.

[SLC99] Stichnoth, J. M., G.-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999, pp. 118–127.

[SM97] Smith, F. and G. Morrisett. Mostly-copying collection: A viable alternative to conservative mark-sweep. *Technical Report 97-1644*, Department of Computer Science, Cornell University, August 1997.

[Tar00] Tarditi, D. Compact garbage collection tables. In *ACM SIGPLAN International Symposium on Memory Management*, October 2000, pp. 50–58.

[Wan80] Wand, M. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, August 1980, pp. 19–28.

A The semantics of single-threaded Core BOL

Semantics of single-threaded program e:

$$eval(e) = c \quad \text{if} \quad \langle e,\ \emptyset,\ 0,\ [0 \mapsto \texttt{stop}]\rangle \longmapsto^{*} \langle \texttt{stop},\ c,\ []\rangle.$$

Data specifications:

$$
\begin{array}{rcll}
St & \in & State_d & = \mathit{CBOL} \times \mathit{Env}_d \times \mathit{Lab}_d \times \mathit{Stack}_d \;\mid\; \mathit{Cont}_d \times Value_d \times \mathit{Stack}_d \quad \text{(thread states)} \\
E & \in & \mathit{Env}_d & = \mathit{Variables} \overset{\text{fin}}{\to} \mathit{Value}_d \quad \text{(environments)} \\
l & \in & \mathit{Lab}_d & = Nat \quad \text{(continuation labels)} \\
S & \in & \mathit{Stack}_d & = \mathit{Lab}_d \overset{\text{fin}}{\to} \mathit{Cont}_d \quad \text{(continuation stack)} \\
V_d & \in & Value_d & = c \mid \{\!|\mathbf{cl}\, x.e,\ E|\!\} \mid \{\!|\mathbf{cb}\, x,\ V_d,\ e,\ E,\ l|\!\} \mid l \quad \text{(machine values)} \\
K & \in & \mathit{Cont}_d & = \mathbf{stop} \mid \mathbf{error} \quad \text{(continuations)} \\
 & & & \mid \{\!|\mathbf{lt}\, x,\ e_{\mathit{body}},\ E,\ l|\!\} \quad \text{– evaluating let binding} \\
 & & & \mid \{\!|\mathbf{if},\ e_{\mathit{true}},\ e_{\mathit{false}},\ E,\ l|\!\} \quad \text{– evaluating conditional} \\
 & & & \mid \{\!|\mathbf{ap1},\ e_{\mathit{arg}},\ E,\ l|\!\} \quad \text{– evaluating function} \\
 & & & \mid \{\!|\mathbf{ap2},\ V_d,\ E,\ l|\!\} \quad \text{– evaluating function argument} \\
 & & & \mid \{\!|\mathbf{thr1},\ e_{\mathit{arg}},\ E,\ l|\!\} \quad \text{– evaluating continuation} \\
 & & & \mid \{\!|\mathbf{thr2},\ V_d,\ E,\ l|\!\} \quad \text{– evaluating continuation argument} \\
 & & & \mid \{\!|\mathbf{cb}\, x,\ V_d,\ e_{\mathit{wrap}},\ E,\ l|\!\} \quad \text{– evaluating letcont binding, with space for arg} \\
 & & & \mid \{\!|\mathbf{apk1}\, k,\ e_{\mathit{arg}},\ e_{\mathit{body}},\ E,\ l|\!\} \quad \text{– evaluating apply\_cont binding} \\
 & & & \mid \{\!|\mathbf{apk2}\, k,\ V_d,\ e_{\mathit{body}},\ E,\ l|\!\} \quad \text{– evaluating apply\_cont argument}
\end{array}
$$

Operations on continuation stacks:

$$
\begin{array}{rcl}
\mathbf{top}(S) & = & \text{least } n \text{ such that } n \notin dom(S) \\
S \downarrow l & = & \begin{cases} S(l') & \text{if } l' < l \\ \mathbf{error} & \text{otherwise} \end{cases}
\end{array}
$$

Converting syntactic values to machine values:

$$
\begin{array}{rcl}
\gamma(c,\ E) & = & c \\
\gamma(x,\ E) & = & E(x) \\
\gamma(\lambda x.e,\ E) & = & \{\!|\mathbf{cl}\, x.e,\ E|\!\}
\end{array}
$$

Thread transition rules:

$$\begin{array}{rcl}
\langle V,\ E,\ l,\ S\rangle & \longmapsto & \langle S(l),\ \gamma(V,\ E),\ S\downarrow l\rangle \\
\langle \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{lt}\ x,\ e_2,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{if},\ e_2,\ e_3,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle e_1 e_2,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{ap1},\ e_2,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \mathbf{letcont}\ k(x)\ = e_1\ \mathbf{in}\ e_2,\ E,\ l,\ S\rangle & \longmapsto & \langle e_2,\ E[k \mapsto l'],\ l,\ S[l' \mapsto \{\!|\mathbf{cb}\ x,\ \bullet,\ e_1,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \mathbf{let}\ k = \mathbf{apply\_cont}\ e_1(e_2)\ \mathbf{in}\ e_3,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{apk1}\ k,\ e_2,\ e_3,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \mathbf{throw}\ e_1(e_2),\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{thr1},\ e_2,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S)
\end{array}$$

$$\begin{array}{rcl}
\langle \{\!|\mathbf{lt}\ x,\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E[x \mapsto V_d],\ l,\ S\rangle \\
\langle \{\!|\mathbf{if},\ e_2,\ e_3,\ E,\ l|\!\},\ \mathbf{true},\ S\rangle & \longmapsto & \langle e_2,\ E,\ l,\ S\rangle \\
\langle \{\!|\mathbf{if},\ e_2,\ e_3,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_3,\ E,\ l,\ S\rangle \\
 & & \quad \text{if } V_d \neq \mathbf{true} \\
\langle \{\!|\mathbf{ap1},\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{ap2},\ V_d,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \{\!|\mathbf{ap2},\ V_d',\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e',\ E'[x \mapsto V_d],\ l,\ S\rangle \\
 & & \quad \text{if } V_d' = \{\!|\mathbf{cl}\ x.e',\ E'|\!\} \\
\langle \{\!|\mathbf{apk1}\ k,\ e_2,\ e_3,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{apk2}\ k,\ V_d,\ e_3,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \{\!|\mathbf{apk2}\ k,\ l'',\ e_3,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_3,\ E[k \mapsto l''],\ l,\ S[l'' \mapsto \{\!|\mathbf{cb}\ x,\ V_d,\ e',\ E',\ l'|\!\}]\rangle \\
 & & \quad \text{if } S(l'') = \{\!|\mathbf{cb}\ x,\ \_,\ e',\ E',\ l'|\!\} \\
\langle \{\!|\mathbf{thr1},\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E,\ l',\ S[l' \mapsto \{\!|\mathbf{thr2},\ V_d,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \mathbf{top}(S) \\
\langle \{\!|\mathbf{thr2},\ l',\ E,\ l''|\!\},\ \bullet,\ S\rangle & \longmapsto & \langle e',\ E'[x \mapsto V_d],\ l,\ S\downarrow l'\rangle \\
 & & \quad \text{if } S(l') = \{\!|\mathbf{cb}\ x,\ V_d,\ e',\ E',\ l|\!\} \\
\langle \{\!|\mathbf{thr2},\ l',\ E,\ l''|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e',\ E'[x \mapsto V_d],\ l,\ S\downarrow l'\rangle \\
 & & \quad \text{if } S(l') = \{\!|\mathbf{cb}\ x,\ \_,\ e',\ E',\ l|\!\} \text{ and } V_d \neq \bullet.
\end{array}$$

B The semantics of multi-threaded Core BOL

Initial configuration for multi-threaded program e:

$$\{\!:\ \langle e,\ \emptyset,\ 0,\ [0 \mapsto \mathbf{stop}]\rangle_0 \quad ; \quad \emptyset \quad ; \quad [0 \mapsto (0, U)] \quad :\!\}$$

Data specifications:

$$
\begin{array}{rcllr}
M & \in & Machine & = \{\!:\ P\ ;\ Q\ ;\ T\ :\!\} & \text{(machine states)}\\
P & \in & \mathit{ThreadPool} & = \mathit{ThreadId} \overset{\text{fin}}{\to} \mathit{State}_d & \text{(all threads)}\\
 & & & \quad \text{Note: } P \text{ written as list of elements of the form } St_i & \\
i,\ j & \in & \mathit{ThreadId} & = Nat & \text{(thread identification number)}\\
Q & \in & \mathit{ReadyThreads} & = \wp(\mathit{ThreadID} \times \mathit{Lab}_d) & \text{(set of ready threads)}\\
 & & & \quad \text{Note: We will sometimes treat } Q \text{ as a finite map} & \\
 & & & \quad \text{from } \mathit{ThreadID}\text{'s to } \mathit{Lab}'_d s. & \\
T & \in & \mathit{Tasks} & = \mathit{TaskId} \overset{\text{fin}}{\to} \mathit{TaskState} & \text{(tasks for executing threads)}\\
m & \in & \mathit{TaskId} & = Nat & \text{(task identification number)}\\
ts & \in & \mathit{TaskState} & = \texttt{Idle} & \text{(task states)}\\
 & & & \mid\ (i, U) \quad \text{– thread } i \text{ running in normal mode.} & \\
 & & & \mid\ (i, L) \quad \text{– thread } i \text{ running in privileged mode.} & \\
K & \in & \mathit{Cont}_d & = \ldots & \text{(continuations)}\\
 & & & \mid\ \textbf{wait} \quad \text{– thread suspended} & \\
 & & & \mid\ \{\!|\textbf{crt}\ (tid,\ k),\ e_{body},\ E,\ l|\!\} \quad \text{– evaluating function to create} & \\
 & & & \mid\ \{\!|\textbf{enqs},\ e_{body},\ E,\ l|\!\} \quad \text{– evaluating enqueue self} & \\
 & & & \mid\ \{\!|\textbf{enq1},\ e_{cont},\ e_{body},\ E,\ l|\!\} \quad \text{– evaluating enqueue thread id} & \\
 & & & \mid\ \{\!|\textbf{enq2},\ V_{thd},\ e_{body},\ E,\ l|\!\} \quad \text{– evaluating enqueue continuation} & \\
 & & & \mid\ \{\!|\textbf{dsp1},\ e_{cont},\ E,\ l|\!\} \quad \text{– evaluating dispatch thread id} & \\
 & & & \mid\ \{\!|\textbf{dsp2},\ V_{thd},\ E,\ l|\!\} \quad \text{– evaluating dispatch continuation} &
\end{array}
$$

Auxiliary functions:

$$
\begin{array}{lcl@{\qquad}lcl}
\text{fresh}(N) & = & \text{least } n \text{ such that } n \notin N, N \subset Nat & \text{locked}(T) & = & \{i \mid \exists m \in dom(T).T(m) = (i, L)\}\\
\text{next}(Q) & = & \text{some } (i, l) \in Q & \text{active}(T) & = & \{i \mid \exists m \in dom(T).T(m) = (i, \_)\}\\
\text{stack}(St) & = & \begin{cases} S & \text{if } St = \langle e,\ E,\ l,\ S\rangle \\ S & \text{if } St = \langle K,\ V_d,\ S\rangle \end{cases} & \text{taskId}(T, i) & = & \begin{cases} m & \text{if } T(m) = (i, \_) \\ \mathit{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

Additional thread transition rules:

$$
\begin{array}{rcl}
\langle \textbf{let}\ (tid, k)\ \textbf{= create}\ e_1\ \textbf{in}\ e_2,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\textbf{crt}\ (tid,\ k),\ e_2,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \textbf{top}(S) \\
\langle \textbf{do enqueue\_self}\ (e_1)\ \textbf{in}\ e_2,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\textbf{enqs},\ e_2,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \textbf{top}(S) \\
\langle \textbf{do enqueue}\ (e_1, e_2)\ \textbf{in}\ e_3,\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\textbf{enq1},\ e_2,\ e_3,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \textbf{top}(S) \\
\langle \textbf{dispatch}\ (e_1, e_2),\ E,\ l,\ S\rangle & \longmapsto & \langle e_1,\ E,\ l',\ S[l' \mapsto \{\!|\textbf{dsp1},\ e_2,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \textbf{top}(S) \\
\langle \textbf{terminate}(),\ E,\ l,\ S\rangle & \longmapsto & \langle \textbf{stop},\ \bullet,\ S\rangle \\
 & & \\
\langle \{\!|\textbf{enq1},\ e_2,\ e_3,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E,\ l',\ S[l' \mapsto \{\!|\textbf{enq2},\ V_d,\ e_3,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \textbf{top}(S) \\
\langle \{\!|\textbf{dsp1},\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E,\ l',\ S[l' \mapsto \{\!|\textbf{dsp2},\ V_d,\ E,\ l|\!\}]\rangle \\
 & & \quad \text{where } l' = \textbf{top}(S) \\
\langle \{\!|\textbf{cb}\, x,\ V_{arg},\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle & \longmapsto & \langle e_2,\ E[x \mapsto V'_d],\ l,\ S\rangle \\
 & & \quad \text{where} V'_d = \begin{cases} V_{arg} & \text{if } V_d = \bullet \\ V_d & \text{otherwise} \end{cases}
\end{array}
$$

Machine transition rules:

$$\{\!:\ P, St_i \ ;\ Q \ ;\ T \ :\!\} \implies \{\!:\ P, St'_i \ ;\ Q \ ;\ T \ :\!\}$$ – thread execution

where $i \in \text{active}(T)$ and $St \longmapsto St'$

$$\{\!:\ P \ ;\ Q \ ;\ T \ :\!\} \implies \{\!:\ P \ ;\ Q \ ;\ T[m \mapsto \texttt{Idle}] \ :\!\}$$ – add task

where $m \notin dom(T)$

$$\{\!:\ P \ ;\ Q \ ;\ T \ :\!\} \implies \{\!:\ P \ ;\ Q \ ;\ T \backslash m \ :\!\}$$ – remove task

where $T(m) = \texttt{Idle}$

$$\{\!:\ P, \langle \{\!|\mathbf{crt}\,(tid,\ k),\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$ – create new thread

$$\implies \{\!:\ P, \langle e_2,\ E[tid \mapsto j,\ k \mapsto 1],\ l,\ S\rangle_i, \langle \mathbf{wait},\ \bullet,\ S_{new}\rangle_j \ ;\ Q \ ;\ T \ :\!\}$$

where $i \in \text{active}(T)$ and $j = \text{fresh}(dom(P) \cup \{i\})$
$S_{new} = [0 \mapsto \mathbf{stop},\ 1 \mapsto \{\!|\mathbf{ap2},\ V_d,\ \emptyset,\ 0|\!\}]$

$$\{\!:\ P, \langle \mathbf{let}\ tid = \mathbf{get\_thread\_id}()\ \mathbf{in}\ e,\ E,\ l,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$ – get current tid

$$\implies \{\!:\ P, \langle e,\ E[tid \mapsto i],\ l,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$

where $i = \text{active}(T)$

$$\{\!:\ P, \langle \mathbf{do}\ \mathbf{lock\_self}()\ \mathbf{in}\ e,\ E,\ l,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$ – lock self

$$\implies \{\!:\ P, \langle e,\ E,\ l,\ S\rangle_i \ ;\ Q \ ;\ T(m)[i \mapsto L] \ :\!\}$$

where $m = \text{taskId}(T, i)$

$$\{\!:\ P, \langle \{\!|\mathbf{enqs},\ e_2,\ E,\ l|\!\},\ V_d,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$ – enqueue self

$$\implies \{\!:\ P, \langle e_2,\ E,\ l,\ S\rangle_i \ ;\ Q, (i, V_d) \ ;\ T \ :\!\}$$

where $i \in \text{locked}(T)$

$$\{\!:\ P, \langle \{\!|\mathbf{enq2},\ j,\ e,\ E,\ l|\!\},\ l,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$ – enqueue

$$\implies \{\!:\ P, \langle e,\ E,\ l,\ S\rangle_i \ ;\ Q, (j, l) \ ;\ T \ :\!\}$$

where $i \in \text{active}(T),\ j \notin \text{active}(T),$ and $j \notin dom(Q)$

$$\{\!:\ P, \langle \mathbf{let}\ (tid, k) = \mathbf{dequeue}()\ \mathbf{in}\ e,\ E,\ l,\ S\rangle_i \ ;\ Q \ ;\ T \ :\!\}$$ – dequeue

$$\implies \{\!:\ P, \langle e,\ E[tid \mapsto j,\ k \mapsto l_j],\ l,\ S\rangle_i \ ;\ Q \backslash j \ ;\ T \ :\!\}$$

where $i \in \text{active}(T)$ and $(j, l_j) \in \text{next}(Q)$

$\{\!:\ P, \langle \{\!|\mathbf{dsp2},\ i,\ E,\ l|\!\},\ l_i,\ S\rangle_i \quad ; \quad Q \quad ; \quad T \quad :\!\}$ – dispatch (self)

$\quad \Longrightarrow \quad \{\!:\ P, \langle S(l_i),\ \bullet,\ S \downarrow l_i\rangle_i \quad ; \quad Q \quad ; \quad T[m \mapsto (i, U)] \quad :\!\}$

where $m = \text{taskId}(T, i)$

$\{\!:\ P, \langle \{\!|\mathbf{dsp2},\ j,\ E,\ l|\!\},\ l_j,\ S\rangle_i \quad ; \quad Q \quad ; \quad T \quad :\!\}$ – dispatch (other)

$\quad \Longrightarrow \quad \{\!:\ P', \langle S_j(l_j),\ \bullet,\ S_j \downarrow l_j\rangle_j, \langle \mathbf{wait},\ \bullet,\ S\rangle_i \quad ; \quad Q \quad ; \quad T[m \mapsto (j, U)] \quad :\!\}$

where $m = \text{taskId}(T, i)$, $j \notin \text{active}(T)$, $j \notin dom(Q)$,
$P = P', St_j$ and $S_j = \text{stack}(St_j)$

$\{\!:\ P, \langle \mathbf{stop},\ V_d,\ S\rangle_i \quad ; \quad Q \quad ; \quad T \quad :\!\} \quad \Longrightarrow \quad \{\!:\ P \quad ; \quad Q \quad ; \quad T[m \mapsto \texttt{Idle}] \quad :\!\}$ – terminate

where $m = \text{taskId}(T, i)$

$\{\!:\ P \quad ; \quad Q \quad ; \quad T \quad :\!\} \quad \Longrightarrow \quad \{\!:\ P', \langle S_i(l_i),\ \bullet,\ S_i \downarrow l_i\rangle_i \quad ; \quad Q \backslash i \quad ; \quad T[m \mapsto (i, U)] \quad :\!\}$ – swap in

where $T(m) = \texttt{Idle}$ and $(i, l_i) = \text{next}(Q)$ and $P = P', St_i$ and $S_i = \text{stack}(St_i)$

$\{\!:\ P \quad ; \quad Q \quad ; \quad T \quad :\!\} \quad \Longrightarrow \quad \{\!:\ P', \langle \mathbf{wait},\ \bullet,\ S_{new}\rangle_i \quad ; \quad Q, (i, l_i) \quad ; \quad T[m \mapsto \texttt{Idle}] \quad :\!\}$ – swap out

where $T(m) = (i, U)$ and $P = P', \langle e,\ E,\ l,\ S\rangle_i$ and $y \notin FreeVars(e)$
$S_{new} = S[l_i \mapsto \{\!|\mathbf{ap2},\ \{\!|\mathbf{cl}\, y.e,\ E|\!\},\ E,\ l|\!\}]$