
HIGHER-ORDER AND SYMBOLIC COMPUTATION 11, 209–225 (1998)

Safe-for-Space Threads in Standard ML*

EDOARDO BIAGIONI, KEN CLINE, PETER LEE, CHRIS OKASAKI AND CHRIS STONE
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. Threads can easily be implemented using first-class continuations, but the straightforward approaches for doing so lead to space leaks, especially in a language with exceptions like Standard ML. We show how these space leaks arise and give a new implementation for threads that is safe for space.

Keywords: continuations, coroutines, Standard ML, space safety, threads

1. Introduction

The ability to provide simple implementations of lightweight, concurrent threads is often cited as one of the great attractions of first-class continuations. We show that this task is not nearly as simple as previously thought, at least if one is concerned about space safety.

The term "space-safety" refers informally to the notion that the implementation of some feature or mechanism will not, through normal use, leak heap or stack storage. This notion is almost always informal (although Clinger [3] has attempted a formal characterization) because it often depends on the intricate details of a particular implementation and what constitutes "normal use." Still, there are common practices in areas such as automatic garbage collection that allow one to make useful conclusions about the space-safety of mechanisms such as threads.

Programming with threads is common in domains such as networking, operating systems, and user interfaces. Threads are not strictly necessary for such applications, but designing these systems with threads leads to an overall system structure that is much easier to understand and modify. Principles for programming with threads can be found in any undergraduate textbook on operating-system design. An excellent source for advice is Nelson's book on Modula-3 [11].

As is well known, threads can be implemented elegantly in a language with first-class continuations, such as Scheme [8, 14] or Standard ML extended with `callcc` [4, 12]. However, naive implementations are likely to suffer from two potential space leaks, one involving continuations and one involving exceptions. The space leak involving continuations

*This is a revised version of a paper presented at the 1997 ACM SIGPLAN Workshop on Continuations. The research was sponsored in part by the Advanced Research Projects Agency ITO under the title "The Fox Project: Advanced Languages for Systems Software," DARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

can easily be optimized away by a good compiler, but the space leak involving exceptions cannot. In this paper, we describe the contortions necessary to implement safe-for-space threads in Standard ML [10], using first-class continuations as provided by the Standard ML of New Jersey system (SML/NJ) [1].

We begin by developing a simple threads package, and then point out and fix some bizarre behavior that can be caused by exceptions. Next, we describe the two potential space leaks, and show how they can be avoided with a clever use of `callcc`. We then sketch some of the difficulties in achieving a comparable implementation using other control operators, such as Felleisen's `control/prompt` [6], Danvy and Filinski's `shift/reset` [5], or Gunter, Rémy, and Riecke's `set/cupto` [7]. Finally, we draw some conclusions.

2. A simple threads package

We begin by considering the simple threads interface shown in figure 1. Three operations are specified in the signature `COROUTINE`: `fork`, `yield`, and `exit`. The `fork` procedure takes a function `f` as an argument and then evaluates the expression `f ()` in a newly created thread. The new thread is called the *child* whereas the thread that called `fork` is the *parent*. The computation of the two threads is expected to occur "concurrently". There is also a notion of a "main" thread, which is the one thread that was not created by a call to `fork`. The main thread is the only thread whose return value is significant; its result is the result of the entire program.

Concurrency amongst threads is obtained by having individual threads voluntarily suspend themselves, thereby giving other threads a chance to execute. In this sense, our threads are cooperative coroutines rather than parallel or pre-emptable (time-sliced) processes.[1] A thread calls `yield` to place itself on a queue of "ready" threads and activate the next thread. The ready threads are typically executed in first-in-first-out order, although it is considered bad programming style to depend on this ordering.

The `exit` procedure terminates the current thread and activates the next ready thread. Unlike `yield`, a call to `exit` never returns. Instead, it either transfers control directly to a waiting thread or raises the `NoReadyThread` exception if there are no other threads remaining in the queue. A child thread implicitly exits if the expression it is evaluating complete (returns the unit value).

Though we do not formally specify these operations, there are certain properties we would like to hold. Calls to `exit` should never return[2] and if a thread calls `exit`, any resources belonging only to that thread should be reclaimable. A call to `fork` and `yield`

```
signature COROUTINE = sig
    exception NoReadyThread

    val fork :  (unit -> unit) -> unit
    val yield:  unit -> unit
    val exit :  unit -> 'a
end
```

Figure 1. An interface for a simple threads package.

should return exactly once (strictly speaking, should return at most once and returns exactly once if all other threads `yield`) and should never raise an exception.

There are some subtleties involving what should happen when the main thread returns. Should it implicitly wait for all the other threads to also finish? From the point of view of the implementor, the simplest approach is to make all the other threads silently disappear when the main thread returns. This is the approach we will describe. There are also questions about what should happen if the main thread calls `exit` rather than returning, or even if it should be allowed to do so. We shall return to this matter in Section 4.

3. A first implementation

We use first-class continuations as provided by SML/NJ [1], with the built-in type `'a cont` and primitive operators `callcc` and `throw`.

Following the by-now standard approach, first advocated by Wand [14], we represent the state of a thread as a continuation.

```
type thread = unit cont
```

A sleeping thread is activated by throwing to its continuation.

The queue of ready threads is then easily implemented as a queue of continuations. Using the standard structure `Queue` (an implementation of imperative queues provided by the SML/NJ library), we have the following definition of the ready queue:

```
val readyQueue : thread Queue.queue = Queue.mkQueue ()
```

A couple of auxiliary functions turn out to be useful. The first one, called `dispatch`, activates the next thread on the ready queue. If no threads are waiting on the ready queue, the `NoReadyThread` exception is raised.

```
exception NoReadyThread

fun dispatch () =
    let val t = Queue.dequeue readyQueue
                handle Queue.Dequeue => raise NoReadyThread
    in throw t () end
```

The second auxiliary function is simply a shorthand for enqueuing a continuation on the ready queue:

```
fun enqueue t = Queue.enqueue (readyQueue, t)
```

With these helper functions in hand, we can now make simple definitions of the main thread routines. The first is `fork`:

```
fun fork f =
    callcc (fn parent =>
        (enqueue parent;
         f ();
         exit ()))
```

To start a new thread, we first capture the continuation of the parent thread and enqueue it. We then activate the child function `f`. The return value of `f` is irrelevant, so when and if `f` returns, we end this thread by calling `exit`. Thus the code sequence `f (); exit ()` comprises the child thread. There are several other ways to schedule the parent and child threads during a `fork`. For instance, the following code enqueues the state of the child and continues with the parent.

```
fun fork' f =
    let val child =
           callcc (fn return =>
              (callcc (fn child => throw return child);
               f ();
               exit ()))
    in enqueue child end
```

Alternatively, we could enqueue both threads and then call `dispatch` to run the next thread on the ready queue.

For `yield`, we have the following definition:

```
fun yield () =
    callcc (fn thread =>
       (enqueue thread;
        dispatch ()))
```

We first capture and enqueue the current continuation, and then call `dispatch` to run the next thread.

Finally, we have `exit`, which immediately starts the next thread.

```
fun exit () = dispatch ()
```

Note that `exit` does not capture the current continuation before calling `dispatch`. Therefore, the current thread is lost.

This implementation is essentially similar to Wand's implementation of threads in Scheme [14]. Existing implementations of threads in Standard ML, such as ML-Threads [4] and CML [12, 13], differ mainly in that they must handle exceptions specially, as discussed in the following section.

4. The problem with exceptions

Although appealingly simple, the above implementation exhibits bizarre behavior in the presence of exceptions. The key question is: when a thread raises an exception that is not caught within that thread, where is the exception handled? This is not always obvious when `callcc` is involved. We choose the interpration in which handler stack is part of the context that is captured by `callcc` and restored by `throw`. (We present an informal semantics for this combination of exceptions and first-class continuations in Section 5.) This is the behavior of `callcc` in the SML/NJ compiler, and is the most useful behavior for implementing a threads package. We briefly discuss the alternative (where `callcc` and `throw` do not affect exception handlers in Section 8.

Armed with this knowledge, we can easily see that `yield` has no effect on the exception handlers of a particular thread—the same handlers that were active before the `yield` are again active when the yielding thread resumes.

```
fun yield () =
    callcc (fn thread =>
       (enqueue thread;
        dispatch ()))
```

The current handlers are captured by the `callcc`. Later, when this thread reaches the head of the ready queue, the handlers are restored by the `throw` in `dispatch`.

By the same reasoning, `fork` has no effect on the exception handlers of the parent thread. However, note that although `callcc` saves the current handlers, it does not change them. Thus, inspecting the code for `fork`

```
fun fork f =
    callcc (fn parent =>
        (enqueue parent;
         f ();
         exit ()))
```

we see that `f` is executed in an exception context inherited from its parent. So, an exception that escapes a child thread may be caught within its parent thread. For example, the following program will print `Surprise!`[3]

```
let fun child () = (1 div 0; ())
    fun parent () = fork child handle Div => print "Surprise!"
in
    fork parent;
    ...
end
```

This shows that the above definition of `fork` fails to obey the "`fork` does not raise an exception" property mentioned in Section 2. This is not an absolutely vital property; unfortunately, the other more important properties fail as well:

```
let fun child () = (1 div 0; ())
    fun parent () = ((fork child) handle Div => ();
                     x := !x + 1)
in
    fork parent;
    ...
end
```

Now, when `child` raises the `Div` exception, `parent` catches the exception and increments `x`. But note that the parent thread is still in the ready queue. When it eventually reaches the head of the ready queue and is dispatched, it will increment `x` a second time!

In the above example, the exception prematurely woke an inactive thread. With another slight change, the exception can actually resurrect a thread that has already exited.

```
let fun child () = (yield (); 1 div 0; ())
    fun parent () = ((fork child) handle Div => ();
                     x := !x + 1)
in
    fork parent;
    ...
end
```

In this example, the child thread voluntarily yields control and the parent thread executes to completion. Later, when the child thread resumes, it raises an exception that resurrects and re-executes the parent.

The situation becomes even more unpredictable given a slightly different implementation strategy. For example, threads are sometimes represented as functions of type `unit -> unit` rather than continuations. The function passed to `fork` can be transferred directly to the ready queue. (Later, when a thread is suspended, its continuation is coerced into a function by partially applying `throw`.) In this setup `dispatch` simply removes and calls the first function in the queue. However, calling the function passed to `fork` does not affect the exception handlers; this function is executed not in the exception context of its parent, but rather in the exception context of whatever thread first yielded control to the child. Any exceptions escaping the child thread are thus caught by this unrelated thread rather than the parent.

Returning to our implementation of thread using continuations, we take a first step towards solving these kinds of problems by guaranteeing that no exception escapes its thread. We accomplish this by installing around each new thread a universal handler that will catch and discard any errant exceptions.

```
fun fork f =
    callcc (fn parent =>
        (enqueue parent;
         f () handle _ => ();
         exit ()))
```

This wrapper appears in both CML and ML-Threads.

However, this is not quite enough. What if `exit` itself was the culprit? In other words, what if `f` raised the `NoReadyThread` exception by calling `exit` when the ready queue was empty (i.e., when all other threads had already exited)? The handler installed by `fork` will catch the exception, but the exception will be immediately re-raised by the subsequent `exit`. This exception will be caught either by an internal handler of `f`'s parent, resurrecting `f`'s parent from the dead, or by the handler wrapped around `f`'s parent by `fork`. In the latter case, the exception will again be immediately re-raised by the subsequent `exit`. In this fashion, the `NoReadyThread` exception can propagate through each of `f`'s ancestors all the way out to the main thread.

This problem can arise only if a child thread calls `exit` after the main thread has already exited. However, since the result of the main thread is the result of the entire program, it is reasonable to forbid the main thread from calling `exit`. This is easily accomplished by keeping track of whether the current thread is the main thread or a child thread and raising a new exception `MainThreadCantExit` if the main thread attempts to call `exit`. With this approach, there will always be at least one thread in the ready queue whenever we are executing a child thread—namely, the main thread. Therefore, `dispatch` can never fail and there is no longer any need for the `NoReadyThread` exception.

Recall that when the main thread returns (as opposed to exiting), any sleeping threads silently disappear. Therefore, if we forbid the main thread from exiting, then we should also provide a way for the main thread to find out when it is safe for it to return (i.e., when all the other threads have exited). It is not difficult to provide a primitive to allow the main thread to sleep until the other threads have all exited. For the details of this primitive, as well as the changes necessary to prevent the main thread from calling `exit`, see the full implementation in Appendix.

5. Space-safety of the threads package

Neither of the threads implementations presented so far is safe for space. Both suffer from two kinds of potential space leaks. In the first kind of leak, a child thread unnecessarily retains its parent's continuation. This potential leak is not too worrisome because the control part of the continuation can be optimized away by a good compiler. The second kind of leak, however, is more serious. In this leak, a child thread unnecessarily retains its parent's exception handlers. Unfortunately, this second leak is unlikely to be optimized away without enormous advances in compiler technology.

The fact that these implementations leak storage is surprising, because implementations like these (e.g., ML-Threads and CML [12, 13]) have been in use for many years. We can only speculate that the kinds of idioms for which these space leaks prove problematic have

$$
\begin{array}{ll}
[x] & = \lambda h.\lambda k.k\,x \\
[\lambda x.M] & = \lambda h.\lambda k.k(\lambda x.[M]) \\
[M\ N] & = \lambda h.\lambda k.[M]\,h\,(\lambda m.[N]\,h\,(\lambda n.m\,n\,h\,k)) \\
[\texttt{callcc}\,M] & = \lambda h.\lambda k.[M]\,h\,(\lambda m.m\,k\,h\,k) \\
[\texttt{throw}\,M\,N] & = \lambda h.\lambda k.[M]\,h\,(\lambda m.[N]h\,m) \\
[\texttt{raise}\,M] & = \lambda h.\lambda k.[M]h\,h \\
[\texttt{handle}\,M\,\texttt{with}\,N] & = \lambda h.\lambda k.[M](\lambda e.[N]\,h\,(\lambda n.n\,e\,h\,k))k
\end{array}
$$

Figure 2. CPS translation with exceptions.

not arisen until now, or that these leaks have been hidden by other, more obvious leaks. In our experience, these leaks have shown to be fatal for long-running applications such as the FoxNet web server [2].

For explanatory purposes, we assume that the compiler uses a CPS intermediate representation. This yields the most direct description of the implementation of exception-handling and continuation primitives, as well as allowing a succinct explanation of the space leaks. However, the leaks are not specific to CPS-based compilers, as any other compilation strategy will have the same problems.

Figure 2 shows a translation of exception and continuation primitives into CPS; we write $[M]$ to denote the translation of a term M. Translated expressions are parameterized by the standard continuation k and an exception-handling continuation h to be invoked by `raise`. The translation is relatively straightforward, and in most cases is simply the standard call-by-value CPS translation augmented to pass along the exception continuation. The unusual cases are for `raise`, which discards the standard continuation and invokes the exception continuation, and `handle`, which extends the exception continuation. (For simplicity, we assume that `handle` takes two expressions, the latter evaluating to a handler function that accepts an exception as its argument. That is, the Standard ML code e_1 `handle` x `=>` e_2 is represented as `handle` e_1 `with` $\lambda x.e_2$. In a fuller presentation there would be further operations so that e_2 could do case analysis on the particular exception caught.)

In this setting, the two space leaks arise when a child thread unnecessarily retains its parent's standard continuation, k, and exception-handling continuation, h. To see why this occurs, consider the `fork` function.

```
fun fork f =
    callcc (fn parent =>
       (enqueue parent;
        f () handle _ => ();
        exit ()))
```

Applying our CPS translation to a call `fork f`, we get

$$
\begin{array}{l}
[\texttt{fork f}] \equiv \lambda h.\lambda k.[\texttt{enqueue}]\,k\,h\,(\lambda().[\texttt{f}]\,()\,(\lambda e.k_{exit}\,())\,k_{exit}) \\
\qquad\qquad \text{where } k_{exit} = \lambda().[\texttt{exit}]\,()\,h\,k
\end{array}
$$

Because the continuation k_{exit} mentions both the standard continuation k and the exception continuation h of the parent, a simple tracing garbage collector will hang on to these

continuations in case `exit` ever returns (invokes k) or raises an exception (invokes h). Even if the parent thread exits, k and h cannot be garbage collected until the child thread releases them. In fact, these continuations will also be part of the context of any descendants of the child thread, so the parent thread's memory may be retained until not only the child thread, but all of the child thread's descendants, have exited.

Fortunately, it is reasonable to hope that a good compiler would recognize that `exit` ends in a call to `throw`, and therefore never invokes its standard continuation. If we rewrite `exit` slightly as

```
fun dequeue () = Queue.dequeue readyQueue
fun exit () = throw (dequeue ()) ()
```

then the CPS translation of `exit` is

$$[\texttt{exit}\,()] \equiv \lambda h.\lambda k.[\texttt{dequeue}]\,()\,h\,(\lambda k'.k'\,())\,.$$

Since it is syntactically obvious that `exit` does not use its standard continuation, a smart compiler could optimize the translation of `fork`, either by inlining `exit` so that the reference to k disappears

$$[\texttt{fork f}] \equiv \lambda h.\lambda k.[\texttt{enqueue}]\,k\,h\,(\lambda().[\texttt{f}]\,()\,(\lambda e.k_{exit}\,())\,k_{exit}) \quad \text{where} \quad k_{exit} = \lambda().[\texttt{dequeue}]\,()\,h\,(\lambda k'.k'\,())$$

or by simply replacing k_{exit} with $k'_{exit} = \lambda().[\texttt{exit}]\,()\,h\,k_{dummy}$, where k_{dummy} is an arbitrary (small) continuation. In that case, the system can release the extraneous pointer to the parent continuation and avoid this space leak. Surprisingly, SML/NJ[4] does not appear to optimize throws in this fashion.

The picture is not so rosy when we turn to the second space leak, in which a child thread retains its parent's exception-handling continuation. Even if we eliminate the reference to k in k_{exit}, there is still a reference to h. Consider again the version of k_{exit} in which `exit` is inlined.

$$\lambda().[\texttt{dequeue}]\,()\,h\,(\lambda k'.k'\,())$$

We happen to know that, since we have forbidden the main thread from calling `exit`, the queue of ready threads is never empty while a child thread is executing and hence `dequeue` will never raise an exception. Therefore, it would be safe to replace h in k_{exit} with a dummy continuation h_{dummy}. However, *it is unreasonable to expect the compiler or run-time system to be able to prove this fact.* Without extremely sophisticated analysis tools, or at least better tools for communicating these kinds of system invariants to the compiler, we have no realistic hope that the compiler will eliminate this reference to h.

This leak is a problem even for non-CPS based compilers. If `exit` raises an exception when the queue is empty, as discussed above there will be a chain of exceptions raised by `exit` in the thread's parent, the parent's parent, and so on up to the main thread. Because

we do not expect any compiler to prove that the ready queue never empties, the compiler will keep this chain of handlers at run-time, which takes space proportional to the `fork` depth.

Because a thread may retain context from each of its ancestors, both of these space leaks are most noticeable when there are deeply nested threads. For example, imagine a server architecture in which each request is processed in its own thread. When a thread receives a request, it immediately forks off a new thread to wait for the next request. In this architecture, the dth request is processed in a thread of depth d. Obviously, such a server cannot afford a space leak that grows with each new thread. Exactly this kind of architecture appears in the FoxNet system [2].

6. Safe-for-space threads

In both implementations so far, problems arise because a child thread hangs on to some of its parent's context. We can avoid these problems by arranging for every thread to be executed in a top-level context rather than in its parent's context. We do this by using first-class continuations to capture a "thread-activating" context at the top level.

```
val threadActivator : (unit -> unit) cont =
    callcc (fn return =>
        let val f = callcc (fn fc => throw return fc)
        in
            f () handle _ => ();
            exit ()
        end)
```

Then, `fork` can start each new thread in this context.

```
fun fork f =
    callcc (fn parent =>
       (enqueue parent;
        throw threadActivator f))
```

Looking at the CPS translation

$$[\texttt{fork}\,\texttt{f}] \equiv \lambda h.\lambda k.[\texttt{enqueue}]\,k\,h\,(\lambda().[\texttt{threadActivator}][\texttt{f}])$$

we see that the child now discards the parent's context (k and h).

This is enough to avoid the space leak, but in fact we can do slightly better with only a small change. Each child thread now creates its own exception handler. Since these exception handlers are all the same, we would prefer for all children to share the same handler. This can be accomplished by capturing the thread-activating context *inside* the exception handler that checks for uncaught exceptions.

```
val threadActivator : (unit -> unit) cont =
    callcc (fn return =>
       (let val f = callcc (fn fc => throw return fc)
        in f () end
        handle _ => ();
        exit ()))
```

Now each child retains no more extra context than the context of the continuation `threadActivator`. This includes the handler inside `threadActivator` and any other handlers that were active when `threadActivator` was created. If we are careful, this should only be the default handler that catches uncaught exceptions at the top level. (If the compiler is not smart enough to realize that `exit` can never return, `threadActivator` may accidently retain `exit`'s continuation, `return`, which could potentially be very large.)

7. Measurements

To determine the severity of the space leaks in the naive implementation of threads, we conducted several experiments. The results are shown in Table 1.

We compared five implementations of `fork` on four versions of a test loop. The first four implementations of `fork` are variations of the naive implementation given in Section 3.

```
fun fork f =
    callcc (fn parent =>
       (enqueue parent;
        f ();
        exit ()))
```

Table 1. Bytes Leaked Per Fork. All tests were run on a DEC AlphaStation 250 4/266 with 96 MB of memory, using SML/NJ Version 109.30. We measured the amount of live data (using `exportML`) every 50000 iterations. The reported sizes are the average increase per iteration over 300000 iterations. There were unexplained variations of up to about 30 KB (<0.7 bytes per iteration) between runs of 50000 iterations, which we assume are due to vagaries in garbage collection.

		Version of test loop			
		Without handler		With handler	
Version of fork		No cont	Cont	No cont	Cont
Without handler	No raise	24	40	52	56
	Raise	0	0	28	32
With handler	No raise	44	60	72	76
	Raise	24	24	60	64
With top-level activator		0	0	0	0

The versions of `fork` labeled "With Handler" add an exception handler around each child thread as discussed in Section 4.

```
...
f () handle _ => ();
...
```

The versions labeled "Raise" add a spurious `raise` after the `exit` to trick the compiler into releasing the continuation. (Note that SML/NJ optimizes `raise` to release its continuation but does not optimize `throw`.)

```
...
exit ();
raise TrickTheCompilerIntoReleasingCont
```

The final version of `fork` uses a top-level thread activator as in Section 6.

The test loop recursively forks new threads up to a given depth:

```
fun loop i =
       (yield (); (* let your parent finish *)
        if i < alldone then
           fork (fn () => loop (i+1))
        else
           ())
```

Because each iteration of the loop begins by yielding back to its parent, which immediately exits, no more than two threads are active at any given time. The versions of the test loop labeled "With Handler" add a new exception handler around each `fork`.

```
...
fork (fn () => loop (i+1)) handle _ => ()
...
```

The versions of the test loop labeled "Cont" add an extra action after the `fork`, so that `fork`'s continuation must contain at least an extra integer.

```
...
i+i; (* continuation must save i *)
()
```

Inspecting Table 1, we see that the space leaks range from 24 bytes per `fork` to 76 bytes per `fork`. These numbers can be made arbitrarily large if the user code adds more exception handlers around each `fork` and performs more non-trivial actions after each `fork`. The bottom row confirms that our final implementation of threads is in fact safe for space.

Interestingly, the version of `fork` without an exception handler but with a spurious `raise` is also safe for space, *provided the user never installs his own exception handlers*. However, this version suffers from both a space leak and the sorts of bizarre behavior described in Section 4 when the user does use exceptions.

8. Other control operators

Researchers have studied many control operators besides `callcc` and `throw`. In this section, we briefly summarize some of the difficulties one encounters when trying to implement safe-for-space threads using a few of these alternative control operators.

First, we consider Felleisen's `control/prompt` [6] and Danvy and Filinski's `shift/reset` [5]. These seem like natural choices for implementing safe-for-space threads since they essentially allow one to run a thread in a top-level context, much like the `threadActivator` continuation does. If the user also has access to these operators, then it becomes difficult to give *any* reasonable implementation of threads, much less one that is safe-for-space—even if we assume that the scheduler can set the outermost prompt. The problem is that a prompt set by the user might mask the prompt set by the threads package, or vice versa.

Gunter, Rémy, and Riecke's `set/cupto` [7] allow named prompts, and present as an example an implementation of threads similar in spirit to those described in this paper. Unfortunately, their implementation of threads suffers from several space leaks; one leak stems from the use of exception handlers. Fixing this leak is complicated by the fact that their implementation of `set/cupto` is buggy with respect to exceptions. Still, assuming a correct implementation of `set/cupto`, it should be possible to avoid this leak with careful programming. A second, more serious leak involves the stack of control points maintained by the control operators. Every time a thread yields and resumes, it pushes an extra control point on the stack, so that a thread that has yielded and resumed n times has a control stack of at least depth n. It is unclear whether this leak is inherent in any implementation of these control operators, or an artifact of the particular implementation of `set/cupto` presented in [7].

Finally, Reppy has proposed variants of `callcc` and `throw`, called `capture` and `escape`, that do not save and restore the exception handler stack [13, p. 136]. On the surface, these operators sound like they might help prevent the retention of unnecessary exception handlers. In fact, however, just the opposite is true. Suppose that we replace `callcc` and `throw` in our threads package with `capture` and `escape`, and consider the following program fragment:

```
(fork (fn () => exit ()); raise E) handle E => ...
```

First, we install the `E` handler, and then fork the child thread, which installs a new universal handler. The child thread then exits, and the parent thread resumes, with both handlers still active. Thus, when we raise `E`, it will be caught by the universal handler of the now defunct thread. Not only can this lead to faster space leaks by retaining too many exception handlers, it also causes the kinds of bizarre behavior about which we complained

in Section 4, only worse. Now, a thread's exceptions may be caught by another thread even when we specifically attempt to handle the exception in the current thread.

9. Conclusions

This work was motivated by the discovery of a space leak in the FoxNet HTTP server [2]. We have experimentally observed such leaks for many different implementations of threads. Although the leaks can be as small as 24 bytes per `fork`, the example of the FoxNet HTTP server—expected to run for months at a time—shows that even slow leaks can be intolerable. We have also modeled these space leaks in a semantics combining both exceptions and first-class continuations.

Threads are an important structuring tool for real-world systems, and the ability to implement light-weight threads is often cited as one of the major benefits of first-class continuations. However, as we have shown, implementations of threads that are both safe-for-space and predictable in the presence of exceptions can be quite subtle and somewhat complicated. This suggests, possibly, that threads ought to be a primitive notion in the language, instead of constructed out of continuations. But until functional languages such as Standard ML incorporate threads as primitive features, we can build on the techniques shown in this paper to build threads that are fast and behave well even in long-running programs.

Acknowledgments

This paper grew from a tutorial presented at the Second International Summer School on Advanced Functional Programming Techniques in Olympia, Washington [9]. We have benefitted greatly from discussions with, and comments from, Olin Shivers, Olivier Danvy, Bob Harper, the participants of the summer school, and the participants and referees of the 1997 Workshop on Continuations.

Appendix: Final implementation

```
signature COROUTINE =
sig
    exception MainThreadCantExit
    exception ChildThreadCantSync

    val fork :  (unit -> unit) -> unit
    val yield:  unit -> unit
    val exit :  unit -> 'a

    val sync :  unit -> unit
       (* sync () yields until all other threads have completed *)
end
```

```
structure Coroutine : COROUTINE =
struct
    exception MainThreadCantExit
    exception ChildThreadCantSync

    val callcc = SMLofNJ.callcc
    val throw  = SMLofNJ.throw

    datatype threadType = Main | Child
    type thread = unit cont * threadType

    val readyQueue : thread Queue.queue = Queue.mkQueue ()
    val syncCont : thread option ref = ref NONE
    val currentThreadType = ref Main

    fun enqueue thread = Queue.enqueue (readyQueue, thread)

    fun dispatch () =
        let val (t, typ) =
                Queue.dequeue readyQueue
                handle Queue.Dequeue =>
                    (* syncCont cannot be NONE *)
                    case !syncCont of SOME main => main
        in
            currentThreadType := typ;
            throw t ()
        end

    fun exit () =
        case !currentThreadType of
            Main  => raise MainThreadCantExit
          | Child => dispatch ()

    fun sync () =
        case !currentThreadType of
            Main => callcc (fn t =>
                        (syncCont := SOME (t, Main);
                         dispatch ()))
        | Child => raise ChildThreadCantSync

    fun yield () =
        callcc (fn thread =>
           (enqueue (thread, !currentThreadType);
            dispatch ()))
```

```
    val threadActivator : (unit -> unit) cont =
        callcc (fn return =>
          (let val f = callcc (fn fc => throw return fc)
           in f () end
           handle _ => ();
           exit ();
           (* raise dummy exception as hint to compiler *)
           raise MainThreadCantExit))

    fun fork f =
        callcc (fn parent =>
          (enqueue (parent, !currentThreadType);
           currentThreadType := Child;
           throw threadActivator f))
end
```

Notes

1. Extending the system for pre-emption is a straightforward exercise, given suitable primitives for interrupting programs at regular intervals. To see how this is done in SML/NJ, the interested reader can consult the source code for ML-Threads or CML, which are both part of the SML/NJ standard distribution, available from `ftp.research.bell-labs.com`. The space-safety considerations discussed in this paper are applicable to both coroutining and pre-emptable threads.
2. We make one exception to this; see Section 3.
3. In each of the following examples, we rely on the fact that `fork` suspends the parent thread and immediately executes the child thread. Similar examples can be devised for different scheduling policies.
4. Version 109.30.

References

1. Appel, A.W. and MacQueen, D.B. A Standard ML compiler. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science. Springer-Verlag, 1987, vol. 274, pp. 301–324.
2. Biagioni, E., Harper, R., Lee, P., and Milnes, B.G. Signatures for a protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, pp. 55–64, June 1994.
3. Clinger, W.D. Proper tail recursion and space efficiency. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 174–185, June 1998.
4. Cooper, E.C. and Morrisett, J.G. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, Dec. 1990.
5. Danvy, O. and Filinski, A. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160, June 1990.
6. Felleisen, M. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pp. 180–190, Jan. 1988.
7. Gunter, C.A., Rémy, D., and Riecke, J.G. A generalization of exceptions and control in ML-like languages. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture*, pp. 12–23, June 1995.

8. Haynes, C.T., Friedman, D.P., and Wand, M. Obtaining coroutines with continuations. *Computer Languages*, **11**(3–4):143–153, 1986.
9. Lee, P. Implementing threads in Standard ML. In *Advanced Functional Programming*, Lecture Notes in Computer Science, vol. 1129, pp. 115–130, Aug. 1996.
10. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.
11. Nelson, G. (Ed.) *Systems Programming with MODULA-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.
12. Reppy, J.H. CML: A higher-order concurrent language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 293–305, June 1991.
13. Reppy, J.H. Higher-order concurrency. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, New York, Jan. 1992.
14. Wand, M. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pp. 19–28, Aug. 1980.