Putting the Spine back in the Spineless Tagless G-Machine: An Implementation of Resumable Black-Holes

Alastair Reid

Department of Computer Science, Yale University
reid-alastair@cs.yale.edu

Abstract. Interrupt handling is a tricky business in lazy functional languages: we have to make sure that thunks that are being evaluated can be halted and later restarted if and when they are required. This is a particular problem for implementations which use black-holing. Black-Holing deliberately makes it impossible to revert such thunks to their original state to avoid a serious space leak. Interactive Haskell implementations such as Hugs and hbi catch interrupts and avoid the problem by omitting or disabling black-holing. Batch mode Haskell implementations such as HBC and the Glasgow Haskell Compiler (GHC) avoid this problem by disabling black-holing or by providing no way to catch interrupts. This paper describes a modification to GHC's abstract machine (the Spineless Tagless G-Machine) which simultaneously supports both interrupts and black-holing.

1 Introduction

Black-Holing [6] is an important technique for avoiding space leaks in lazy functional languages. When a program starts to evaluate an unevaluated thunk, it copies the contents of the thunk onto the stack (or into registers) and overwrites the thunk with an object known as a "black-hole." When the program completes evaluation of the unevaluated thunk, the thunk is overwritten a second time with the value of the thunk. If the program tries to evaluate a thunk which is already being evaluated, it reports an error. This is the correct behaviour in a sequential evaluator: it can only happen if the value of the original thunk depends on itself and would have caused an infinite loop in a system which did not support black-holes. (Concurrent evaluation requires different behaviour and is discussed in section 5.4.) Black-Holing a thunk is important because it removes references to the free variables of the thunk; if one of these references is the last reference to the variable, the variable can be garbage collected immediately — reducing the heap usage of the program. Jones [6] shows that simple tail-recursive functions such as `last` can run in constant space with black-holing but require linear space without black-holing.

The problem with black-holing is that it assumes that evaluation of a thunk will not stop until the value of the thunk has been found. This is a problem if

we wish to pause evaluation of a thunk to handle an interrupt or if we wish to speculatively evaluate a thunk while waiting for user input and pause evaluation when user input arrives. In both circumstances, black-holed thunks are left in the heap and incorrectly report errors if they are subsequently evaluated.

An obvious fix is to revert black-holes to their original form when an interrupt occurs. There are two problems with this:

1. To revert a black-hole to its original form, we have to preserve the contents of the original thunk until evaluation of that thunk completes (i.e., until we're certain it will not need to be reverted). Doing so retains references to the thunk's free variables restoring the space leak that black-holing is designed to fix.
2. Reverting the black-hole to its original form causes us to discard a lot of the work we performed in partially evaluating the object. This is contrary to one of the primary properties of lazy evaluation: every thunk is evaluated at most once.

Our solution to these problems is not to revert the black-hole to its original form but to revert the black-hole to (a representation of) its current partially evaluated state.

On the Spineless Tagless Graph-reduction Machine (STG machine) [8], the state of a partially evaluated thunk is stored on the stack; naïve implementations of graph reduction do not use the stack in this way: they store the entire state of a thunk on the "spine" of the thunk. We therefore dub our technique "Putting the Spine back in the Spineless Tagless G-Machine."

2 Updates in the STG machine

The STG machine is described in detail by Simon Peyton-Jones [8]; here we provide an overview of the most important parts of the evaluation and update machinery.

On a naïve implementation of graph reduction, an update is performed on each reduction step. For example, in reducing

```
let x = compose id id 42 in x
```

to `42`, a naïve implementation would update `x` three times with `id (id 42)` then with `id 42` and finally with `42`. This is inefficient because it requires the allocation of many intermediate values and because it requires a large number of writes into the heap.

The STG machine avoids these costs by delaying updates until an expression has been reduced to weak head normal form: each thunk is evaluated at most once.[1] To do this, the STG machine maintains a list of thunks which are in the

[1] The STG machine also allows thunks to be marked as being non-updatable if they are not shared. Black-Holing causes no problems for non-updatable thunks so they are ignored in this paper.

process of being evaluated. This list is threaded through the evaluation stack and is manipulated as follows:

- When an (updatable) thunk is "entered" (i.e., evaluation starts), the STG machine does four things:
  1. a pointer to the thunk is pushed onto the update list (this thunk is known as the "updatee");
  2. the contents of the thunk are pushed onto the stack;
  3. the thunk is overwritten with a black-hole;[2] and
  4. the thunk's code is executed. If the thunk is an application node, this enters the object on top of the stack.
- When evaluation of a thunk completes, the top of the stack contains one of two things:
  - A return address: the evaluator simply jumps to the return address.
  - An entry in the update list: the evaluator overwrites the updatee with the value of the thunk, removes the update frame from the list and tries to return the value again.

3 Reverting Black-Holes

As we noted in the introduction, black-holing causes problems if we interrupt execution because it is neither possible nor entirely desirable to revert a black-hole to its original form. The solution is simple and, with the aid of 20-20 hindsight, very obvious: instead of reverting the black-hole to its original form, we overwrite black-holes with that part of the stack required to complete evaluation of the thunk. That is, we revert each black-hole on the update list as follows:

1. The black-hole is overwritten with a "resumable black-hole" containing the contents of the stack above the update frame. If, as is usually the case, the black-hole is too small to hold the resumable black-hole, a fresh resumable black-hole is created and the black-hole is overwritten with an indirection to the resumable black-hole.
2. The update frame is removed from the head of the update list.
3. A pointer to the black-hole is pushed onto the stack.

When the update list is empty, the remainder of the stack is discarded.

When the STG machine enters a resumable black-hole, it does exactly the same as when it enters an updatable application node. That is:

1. a pointer to the resumable black-hole is pushed onto the update list;
2. the contents of the resumable black-hole are pushed onto the stack;
3. the resumable black-hole is overwritten with a black-hole; and
4. the object on top of the stack is entered.

[2] An optimisation known as "lazy black-holing" allows this step to be delayed until garbage collection time and is discussed in section 5.1.

The only difference between resumable black-holes and application nodes lies in how they are garbage collected: since we create resumable black-holes by copying data off the stack, they have to be garbage collected like miniature stacks.

Figure 1 shows how this works while evaluating this expression

```
let a = enumFromTo 1 100
    b = tail a
    c = head b
in c
```

Initially (figure 1.i), the heap contains three updatable application nodes `a`, `b` and `c` (representing the variables `a`, `b` and `c` respectively), the stack (shown with the "top" towards the bottom of the page) contains some data `D` and the top of the stack contains a pointer to `c`. (One of the strengths of our technique is that it oblivious to what data (if any) occurs between update frames. It is therefore sufficient to label the areas between update frames `A` ... `D`; we need not worry about the contents or sizes of these areas.)

Figures 1.ii to 1.iv show how the spine of the graph is unwound during evaluation of `c`. As each application node is entered, an update frame is pushed onto the stack and added to the head of the update list, the contents of the node are copied onto the stack and the node is black-holed.

Let us suppose that an interrupt occurs just after `a` is entered. The next time a thunk is entered (i.e., when `enumFromTo` is entered), the evaluator detects that the thread is to be killed and start to revert all the black-holes on the update list.

Figures 1.v to 1.viii show how the spine of the graph is reconstructed from the stack while reverting black-holes. As each black-hole is reverted, the black-hole is overwritten with a resumable black-hole containing the contents of the stack above the update frame, the update frame is removed from the head of the update list and a pointer to the black-hole is pushed onto the stack. When the update list is empty, the remainder of the stack is discarded.

Suppose now that we start evaluating something else and, in the course of that expression, we enter thunk `c`. The behaviour of the STG machine on entering a resumable black-hole reverses the sequence of steps from figure 1.viii to figure 1.v. That is:

1. Since `c` is a resumable black-hole, the evaluator adds an update frame to the list, pushes the data `C` on the stack, pushes `b` on the stack, black-holes `c` and enters `b` resulting in figure 1.vii.
2. On entering `b`, the evaluator adds an update frame to the list, pushes the data `B` on the stack, pushes `a` on the stack, black-holes `b` and enters `a` resulting in figure 1.vi.
3. On entering `a`, the evaluator adds an update frame to the list, pushes pointers to `100`, `1` and `enumFromTo` on the stack, black-holes `a` and enters `enumFromTo` resulting in figure 1.v.

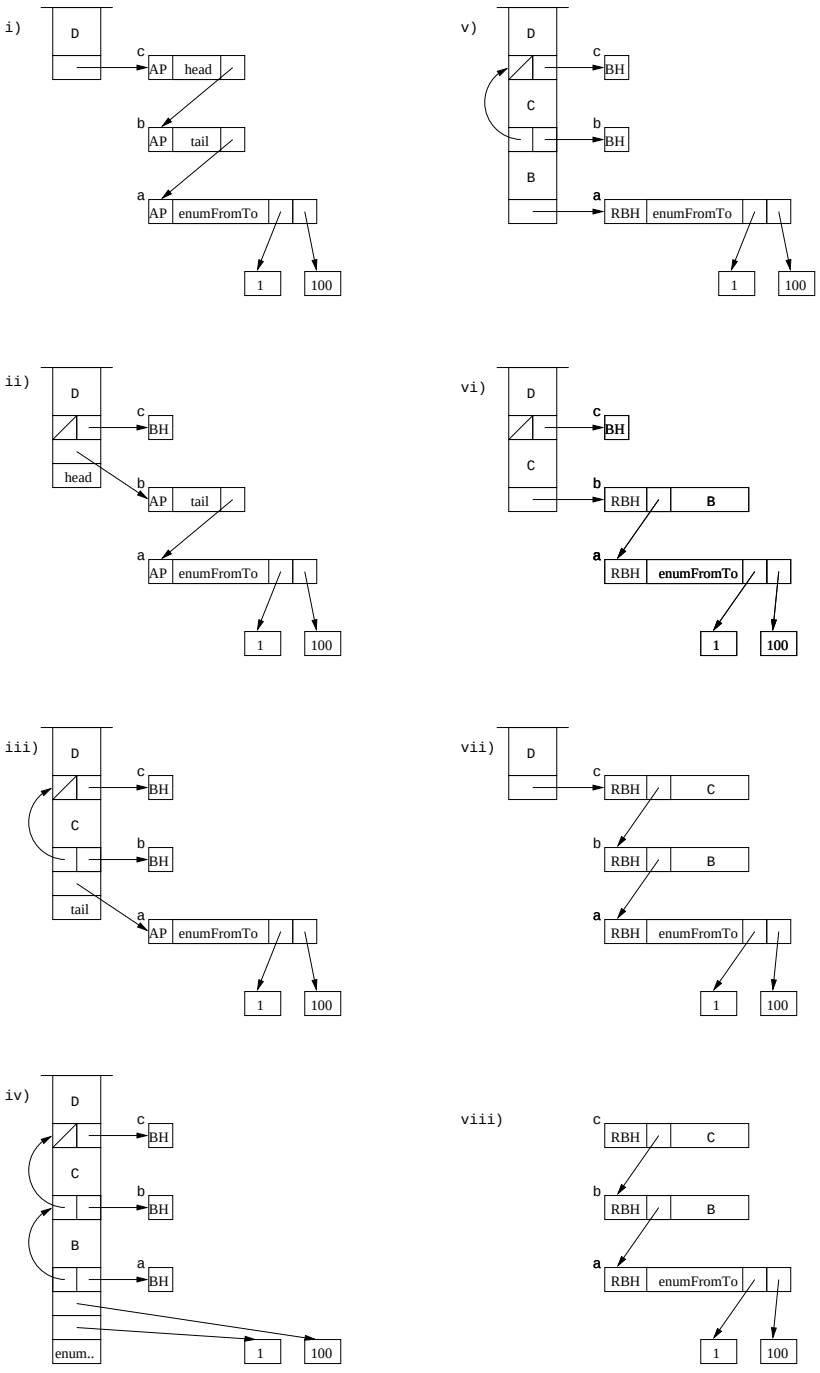

Fig. 1. Reverting Black-Holes

The stack has now been restored to its original state prior to the interrupt and execution continues as before. A similar sequence unfolds if the new evaluation enters `a` or `b`.

An obvious concern is that using this technique will somehow re-introduce the space leak that black-holing is supposed to remove. This clearly does not happen:

- Nothing is changed during normal evaluation. We use exactly the same representation and store exactly the same data as in the original STG machine with black-holing.
- The resumable black-holes generated when discarding a stack require almost exactly the same space as the original stack.
- After an interrupt occurs, every resumable black-hole contains exactly the data needed to evaluate it and so it doesn't leak space unless the original evaluation mechanism leaked space.

Despite this, we might still find that a resumable black-hole takes more space than the original updateable thunk (a thunk may take more space when evaluated). We might also find that a resumable black-hole takes *less* space than the original updateable thunk (a thunk may also take *less* space when evaluated). This is a fundamental property of lazy evaluation rather than a special property of black-holes or resumable black-holes: it also happens in naïve graph reducers which have neither.

Another concern is that the benefits of using this technique may come at a significant cost in performance or in complexity of the runtime system. Again, this does not happen:

- Since nothing is changed during normal evaluation, no overhead is imposed on programs that are not interrupted.
- When a program is interrupted, we copy stack segments into resumable black-hole objects on the heap; when a black-hole is resumed, we copy the stack segments back onto the stack. These costs are typically quite small (smaller than other runtime costs such garbage collection) and are only incurred when interrupts occur.
- The implementation is as simple as our description: it consists of a few hundred lines of C to implement the new object type and to copy stack segments into resumable black holes.

4 Catching Interrupts

The previous section describes how to pause evaluation without leaving black-holes in the heap but says nothing about what to do after evaluation has been paused. This section outlines how to catch interrupts in a programming environment (Hugs) and in the programming language itself. Only the first one has been implemented as yet.

Catching interrupts is absolutely essential in an interactive system such as Gofer [5] or Hugs: we have to be able to terminate long-running programs or

programs that have entered infinite loops and continue. We have written a modified version of Hugs which uses the STG machine for evaluation. When the user interrupts an evaluation, the Hugs user interface sets a flag in the runtime system to indicate that an interrupt occurred. Every time the evaluator enters a node, it tests this flag to see whether it should terminate the current evaluation by reverting all black-holes on the update list.

To catch interrupts in (Sequential) Haskell we need to add a function like Haskell 1.3's `catch` function:

```
catchInterrupt :: IO a -> IO a -> IO a
```

The expression `e `catchInterrupt` h` executes the expression `e`. If `e` terminates before an interrupt occurs, the result of `e` is returned; if an interrupt occurs before `e` terminates, the *handler* `h` is executed and the result of `h` is returned.

To implement this, we define a new type of frame which can be inserted in the update list. These *interrupt handler* frames contain a pointer to a handler thunk; they are added to the list when `catchInterrupt` is executed and removed from the list when `catchInterrupt` completes. When an interrupt occurs, the runtime reverts all black-holes down to the topmost interrupt handler frame, removes the interrupt handler frame and enters the handler thunk.

5 Variations

The STG machine is a very flexible architecture allowing a number of optimisations and extensions. This section describes how reverting black-holes interacts with five such optimisations and extensions.

5.1 Lazy Black-Holing

Section 9.3.3 of the STG paper [8] describes an optimisation of black-holing known as "lazy black-holing" which delays black-holing a thunk until the next garbage collection. When garbage collection occurs, it is a simple matter to run down the update list and black-hole any thunks which are not already black-holed. This does not affect the ability of black-holing to eliminate space leaks because the space leak does not manifest itself until the next garbage collection and so there is no harm in delaying black-holing until then. The benefit of lazy black-holing is that it avoids the extra effort required to black-hole a thunk whose evaluation completes before a garbage collection occurs.

The only thing that changes when reverting black-holes if we use lazy black-holing is that we may have to revert a thunk on the update list which hasn't been black-holed yet. Two questions arise: *should* we revert the thunk; and *can* we revert the thunk. The answer to both questions is "yes":

1. Nothing goes drastically wrong if we don't revert the thunk but we lose some laziness. That is, we discard the result of partially evaluating the thunk and have to repeat that effort if the thunk is re-entered. Worse still, we lose

an *unpredictable* amount of laziness depending on when we last black-holed thunks on the update list. To avoid these problems, we choose to revert all thunks on the update list even if they haven't been black-holed yet.

2. We might worry that a thunk on the update list could be smaller than a black-hole making it impossible to overwrite with either a resumable black-hole or an indirection to a resumable black-hole. Fortunately, this cannot happen: the system already requires that all updatable thunks are big enough to overwrite with a black-hole. This is required since we are able to black-hole all the thunks on the update list before reverting them.

5.2 The `seq` and `strict` functions

Haskell 1.3 added the ability to force evaluation of a thunk using the (equivalent) functions `seq` and `strict` instead of by using a `case` expression. A `case` expression would have pushed a return address onto the stack but, because `seq` can be used on objects with any type (including functions), they require a different implementation. The `seq` function is implemented by pushing a "continuation" onto the stack, and adding a "SEQ frame" to the update list so that the evaluator enters the continuation correctly. If the evaluator finds a "SEQ frame" on the update list when it returns a value, it removes the frame, discards the value and enters the continuation on top of the stack.

This requires the following change to our revertible black-holing mechanism. When we encounter an exception handling frame on the stack, we create a thunk which will push a SEQ frame onto the stack, push the stack contents and resume evaluation. Since the STG machine doesn't have node types that do this already, we have to add `SEQ` nodes to the system. When a `SEQ` node is entered, the evaluator adds a SEQ frame to the update list, pushes the node's contents on the stack and enters the top node.

Figure 2 shows how SEQ frames are reverted when executing the expression

```
let a = 1 + 2
    b = a `seq` x
in b
```

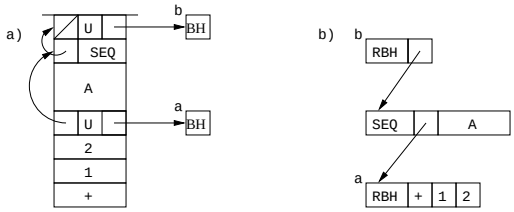

Fig. 2. Reverting SEQ Frames

Figure 2a shows the state of the stack just before entering `+`. Thunks `a` and `b` have been black-holed and the update list consists of an update frame for `a`, a SEQ frame and an update frame for `b`. Frames on the update list are tagged with `U` for update frames and `SEQ` for SEQ frames.

Reverting the black-holes and SEQ frames in figure 2a yields figure 2b. The black-holes are reverted exactly as before and the SEQ frame has been turned into a `SEQ` node containing a pointer to `a`.

This isn't the only possible way of dealing with SEQ frames. An alternative is to allow resumable black-holes to contain lists of SEQ frames and fill in resumable black-holes with everything on the stack that occurs between two update frames: pending arguments, return addresses, environments, SEQ frames, etc. This avoids the cost of introducing `SEQ` nodes at the expense of making resumable black-holes more complex. This extra complexity is most keenly felt in the garbage collector — which is already quite complex enough!

5.3 Exception handling

We recently extended the STG machine with an exception handling mechanism [10, 11] which uses the update list to store exception handlers as well as updatees. When the evaluator finds an exception handler on the update list as it is trying to return a value, it removes the exception handler and tries again.

This requires the following change to our revertible black-holing mechanism. When we encounter an update frame on the update list, we (already) create a thunk to push an update frame onto the stack, push the stack contents and continue evaluation where it left off. Similarily, when we encounter an exception handling frame on the stack, we create a thunk to push an exception handling frame onto the stack, push the stack contents and resume evaluation. Since the STG machine doesn't have node types that do this already, we have to add `CATCH` nodes to the system. When a `CATCH` node is entered, the evaluator adds an exception handler frame to the update list, pushes the node's contents on the stack and enters the top node.

Figure 3 shows how exception handlers are reverted when executing the expression[3]

```
let a = print 1
    b = a `catchException` h
    c = b >> x
in y
```

Figure 3a shows the state of the stack just before entering `print`. Thunks `a`, `b` and `c` have been black-holed and the update list consists of an update frame for `a`, an exception handler frame, an update frame for `b` and an update frame

[3] The expression `a` `` `catchException` `` `h` evaluates `a` and returns its result; if an exception is thrown while evaluating `a`, then the handler `h` is invoked and the result of `h` is returned.

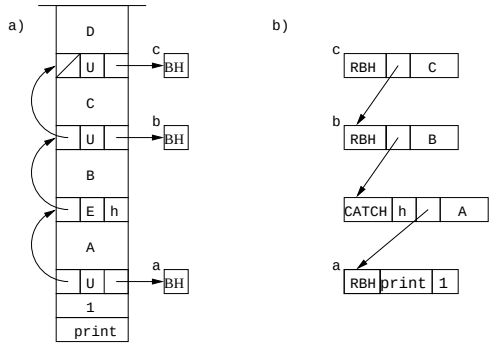

Fig. 3. Reverting Exception Handlers

for `c`. Frames on the update list are tagged with `U` for update frames and `E` for exception handler frames.

Reverting the black-holes and exception handlers in figure 3a, results in figure 3b. The black-holes are reverted exactly as before and the exception handling frame has been turned into a `CATCH` node containing the handler `h`, the application node `a` and the data `A`.

Again, we could have added support for exception handling by enriching the structure of resumable black-holes. The tradeoff here is exactly as it was with SEQ frames but since the choice is now between one complex object and three simple objects the decision to introduce a new node type isn't quite so clear.

5.4 Concurrent Haskell

The STG machine has been extended to support concurrent threads [9]. In a concurrent system, black-holing is modified as follows. We add a queue of threads to every black-hole — the "blocking queue" of the black-hole. The first time a thunk is entered, it is overwritten with a black-hole with an empty queue. If another thread tries to enter a black-hole that thread is suspended and added to the blocking queue. When evaluation of a thunk completes, its black-hole is overwritten with the value of the thunk and all threads in the blocking queue are added to the (global) queue of runnable threads.

This blocking behaviour requires the following change when reverting black-holes: when a black-hole is reverted, all threads in the blocking queue are added to the (global) queue of runnable threads. When these threads next try to execute, the first thread will enter the resumable black-hole and rebuild the stack exactly as it was when the thread was interrupted and all subsequent threads will be added to the blocking queue as before.

To catch interrupts in Concurrent Haskell [9] we need to add two things:[4]

[4] A full threads library might add further features, here we restrict ourselves to the minimum required to catch interrupts.

1. The ability to terminate a thread; and
2. The ability to wait for an interrupt to occur.

It is then straightforward to create threads which wait for interrupts and kill other threads when they occur. This can be combined with functions which wait for a given time period to provide timeouts as well.

To terminate a thread, we need to add thread identifiers and a function to kill a thread. The function `killThread` must revert all black-holes on the thread's update list before killing the thread.

```
data ThreadId  -- abstract
forkIO      :: IO a -> IO ThreadId
getThreadId :: IO ThreadId
killThread  :: ThreadId -> IO ()
```

We also need a way of waiting for an interrupt. This requires a small change to the runtime system to maintain a list of threads waiting for interrupts and add the threads to the runnable queue when an interrupt occurs. This is a (simplified) form of how timers are currently handled.

```
waitForInterrupt :: IO ()
```

5.5 Parallel Haskell

The STG machine has been extended to run on parallel architectures [14]. Black-holes act in the same way as in Concurrent Haskell (i.e., threads block on thunks which are already being evaluated). The big change from Concurrent Haskell is that each processor only has access to a small part of the global heap; if a processor requires an object stored in another part of the graph, it must ask another processor to send it the object. If the object is already being evaluated by a processor, the request blocks until evaluation terminates.

We have not implemented resumable black-holes in Parallel Haskell but we believe that it should be straightforward since reverting the black-holes on a thread's stack is a local operation. When a thread is interrupted, all pending updates are reverted in the same way as in Concurrent Haskell. Just as threads blocked on a black-hole are moved to the queue of runnable threads in Concurrent Haskell, so blocked requests (to move an object to another processor) are moved to the queue of "runnable" requests. Note that it is very important that an object cannot be moved while it is being evaluated since we must be able to overwrite the original object with a resumable black-hole when a thread is interrupted.

Being able to interrupt a thread is particularily important in Parallel Haskell since it makes it possible to control speculative evaluation on idle processors [7]. When resources (CPU and memory) are abundant, speculative threads can be created; and when resources are scarce or poorly balanced between processors, speculative threads can be terminated. Using our revertible black-holes, terminating a thread has the effect of splitting its stack into many small parts allowing unwanted parts to be reclaimed and allowing parts required by other processors to move to the other processor.

6 Related Work

Lazy functional programs can suffer from a variety of space leaks. Whilst many of these problems can only be eliminated by modifying your program, some space leaks can be fixed in the evaluator or in the garbage collector.

One of the first such fixes was the "lazy tuple matching" space leak reported by Hughes [3]. The problem is that extracting a component of a data-structure (aka "tuple matching") is performed lazily and so the runtime system may hang onto a large data structure of which only a small component is required. Wadler showed how this could be fixed by modifying the garbage collector [15] and, more recently, Sparud showed how this could be solved by modifying the evaluator [13].

Another space leak which can be automatically plugged was accidentally introduced by "optimising" tail calls in the G-machine. This problem was identified and fixed by Jones with the introduction of black-holing [6].

Shortly after the introduction of black-holing, Runciman and Wakeling found a baffling space leak using their heap profiling tool [12]. They suspected a problem in their tool until they realised that the problem was the same one reported by Jones. Adding black-holing to their compiler removed this leak and resulted in a factor of four reduction in the cost of running a benchmark. The combination of black-holing, Wadler's fix for the "lazy tuple match" leak and fixing programming problems identified by their tool reduced the space-time cost of their program by two orders of magnitude.

Until now, the major problem with black-holing has been its incompatability with interrupts and with speculative evaluation. Mattson and Griswold [7] use "grey-holes" (a kind of revertible black-hole) to synchronize threads in a Parallel Haskell implementation but, unlike resumable black-holes, terminating a speculative thread reverts grey-holes to their original form. This suffers from the two problems listed in the introduction: it reintroduces the space leak; and it discards work. They suggest that discarding work is beneficial since the unevaluated form of the thunk is often smaller than the evaluated form but Hughes [4] suggests that the opposite is sometimes true.

Trinder et al.[14] use black-holes when moving objects from one processor to another. While the object is in transit, it is overwritten with a "revertible black-hole." If the object is rejected (perhaps because the receiver runs out of heap space), the black-hole is reverted to its original form; otherwise, the revertible black-hole acts like a normal black-hole and is updated with an indirection to the (as yeat still unevaluated) thunk on the remote processor. Like Mattson and Griswold, the black-hole may be reverted to its original form, but this doesn't cause the same problems since revertible black-holes only last long enough to successfully transfer an object from one processor to another. Their system has no need for resumable black-holes since it does not support interrupt catching and it provides task migration in preference to speculative evaluation.

More recently, Chakravarty uses a similar technique to cover communication latencies in his massively parallel STG machine [2]. Besides his different motivation (Chakravarty does not mention interrupts, killing threads or black-holing), there are some important technical differences:

- Chakravarty suspends closures (using objects like our resumable black-holes) while waiting for a value to be received from a remote processor and resumes the closure when the value arrives. Since each closure requires different sets of values from other processors, Chakravarty only suspends the topmost closure instead of reverting all closures currently under evaluation. If the program terminates successfully, all suspended closures will have been restarted.
- In contrast, we are concerned with interrupting normal sequential evaluation: closures are suspended when an interrupt is received and restarted only if and when they are needed by the interrupt handler. Since interrupts affect the entire execution, we suspend all closures which are currently under evaluation by walking down the update list. Most resumable black-holes are not required by the interrupt handler and are quickly garbage collected.

In short, Chakravarty suspends closures which are waiting to be sent input while we suspend closures which are waiting for their output to be (re)demanded.

Looking farther afield, similar problems and similar solutions are found whenever computer scientists want to cancel speculative evaluation or handle exceptions.

- The most important feature of exception and interrupt handling mechanisms is the ability to specify how to clean up shared state. In imperative languages, is necessary to write your own cleanup code since the language cannot be expected to know how to restore your program to a consistent state. This is not necessary in the pure subset of Haskell (i.e., the part of Haskell where black-holing is used) because the the lack of side-effects limits the problems to those introduced by the implementation. In the imperative subset of Haskell, the programmer must write their own cleanup code — we recently added exception-handling to Haskell for this purpose [10, 11].
- Multiscalar processors perform a considerable amount of speculative evaluation and must clean up their internal state when a speculative evaluation is terminated. For example, Breach et al. [1] describe an architecture which tracks dependencies between different stages of the processor. Terminating one stage automatically terminates those stages which have used values produced by the terminating stage. Like our technique, cleanup is performed automatically; unlike our approach work done by the stage is discarded to undo side-effects (and also to conserve resources).

7 Conclusions

The Spineless Tagless G-Machine is an efficient graph-reduction machine which stores the spine of the graph on the stack (rather than storing it on the heap) and which uses black-holing to avoid the resulting space leak. This optimisation comes at a cost: we can't resume interrupted evaluations because black-holing assumes that thunks are only entered once. We have shown that this problem can be resolved efficiently by restoring the spine of the graph to the heap and we have outlined how it interacts with a range of extensions to the language and to the implementation.

Acknowledgements We are grateful to Simon Peyton Jones, Simon Marlow and John Peterson for comments on our approach and on this paper and to Paul Hudak and Greg Hager whose interest in programming robots in Haskell helped motivate this work. We are also grateful to the anonymous referees for their interesting and useful feedback — we found the pointers to related work outside the Haskell community particularily intriguing.

References

1. S. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *27th Annual International Symposium on Microarchitecture (MICRO-27)*, pages 181–190. ACM press, 1994.
2. M. Chakravarty. Lazy thread and task creation in parallel graph-reduction. In *Proceedings of IFL'97*, volume 1467 of *Lecture Notes in Computer Science*, pages 231–249. Springer Verlag, September 1997.
3. R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford University, 1984.
4. R. J. M. Hughes. Parallel functional languages use less space. In *Symposium on Lisp and Functional Programming*, Austin, 1984.
5. M. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, May 1994.
6. R. E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.
7. J. S. Mattson Jr. and W. G. Griswold. Speculative evaluation for parallel graph reduction. In *Parallel Architectures and Compilation Techniques*, pages 331–334. North-Holland, August 1994.
8. S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.
9. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages*, pages 295–308. ACM press, January 1996.
10. S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Programming Languages Design and Implementation*. ACM press, May 1999.
11. A. Reid. Handling Exceptions in Haskell. Research Report YALEU/DCS/RR-1175, Yale University, Department of Computer Science, August 1998.
12. C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–246, April 1993.
13. J. Sparud. Fixing some space leaks without a garbage collector. In *Proc. Conference on Functional Programming Languages and Computer Architecture*. ACM, 1993.
14. P. Trinder, K. Hammond, J. Mattson Jr, A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Programming Languages Design and Implementation*, pages 79–88. ACM press, 1996.
15. P. L. Wadler. Fixing a space leak with a garbage collector. *Software — Practice and Experience*, 17(9):595–608, 1987.