Debugging Reactive Systems in Haskell

Amr Sabry
Department of Computer Science
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Jan Sparud
Department of Computer Science
University of York
Heslington, York Y01 5DD, UK
sparud@cs.york.ac.uk

Abstract

We support the use of Haskell in two classes of applications.

First, we note that although Haskell is well-suited for stream-oriented applications, like reactive systems, most implementations lack tools for locating subtle errors like deadlocks. To support the use of Haskell in such applications, we have designed and implemented a debugger with an intuitive graphical interface. The debugger simplifies the task of locating deadlocks by providing users with a mostly declarative view of programs as systems of recursive equations over streams.

Second, we investigate the use of Haskell in the implementation of the debugger. As expected, this effort requires a variant of Haskell, currently used in the intermediate phases of Haskell compilers, that combines call-by-need semantics with computational effects. To support robust programming in this system-oriented variant, we formalize the semantics of handleable exceptions in a call-by-need language.

1 Streams and Deadlocks

As Kahn showed in 1974 [23], any deterministic network of processes can be modeled using recursive equations over streams. Such networks occur in telephones, missiles, and many other embedded *reactive* programs. The stream model corresponds in spirit to the electrical engineer's concept of a signal-processing system [1] and thus provides a natural formalism for the design, specification, and verification of these networks [11]. Additionally, stream-based specifications are directly executable in a variety of high-level languages. Examples of languages that elegantly support computations over streams include modern lazy functional languages such as Haskell [19], hardware-description languages such as MHDL [7] (an extension of Haskell), data-flow programming languages such as Lucid [4, 5], and synchronous programming languages such as Esterel [9, 10], Lustre [12], and Signal [8].

These languages have been remarkably successful in supporting the rapid development of reliable systems. First, streams (lazy lists) are the fundamental and most widely supported data structure in most of these languages. Second synchronization is dealt with automatically: programmers are only required to specify sets of recursive functions over streams without having to worry about many tedious and error-prone details. Indeed Caspi and Pouzet [13] report that some of the programs running on the Airbus A320 have been obtained in this way. Also Broy *et al.* [11] use this methodology for the development of various distributed systems.

Despite its appeal, programming with streams is quite subtle. A common mistake consists of writing a system of recursive equations over streams whose fixed point is semantically undefined [32]. Operationally this error may manifest itself in terms of an obscure error message from the compiler or an equally obscure error message from the runtime system. For example, when suspecting a deadlock, our liveness analysis [22] returns a large (and incomprehensible) list of unsatisfied constraints. As another example, typical Haskell implementations, *e.g.*, the 'hbc' compiler from Chalmers University, detect and report deadlocks with the message "`Runtime error: Black hole detected in eval zap`," which gives about as much information as the infamous C error message "`Segmentation fault`."

In fact, one can usually locate the sources of segmentation faults using a C debugger but this is not even the case for black holes: users get little debugging support from typical Haskell implementations. Furthermore, conventional debugging techniques for imperative languages such as setting breakpoints, single-stepping, tracing, and watching variables are not particularly suited for locating deadlocks occurring in lazy functional languages [15, 16, 29, 33] or synchronous languages [14]. Indeed our own personal experience in specifying reactive systems in Haskell quickly convinced us of the need for a special-purpose debugging technology specifically tailored towards computations over streams.

This paper reports on the design and implementation of such a debugger in the context of the Haskell compiler 'hbc.' The debugger has an intuitive graphical user interface, and can be used to quickly pinpoint the sources of many deadlocks. Experience with the implementation indicates that the system works remarkably well for small but realistic programs. The debugger nicely complements our formal liveness analysis [22] providing users with a useful set of tools for developing reactive programs.

Like many system-level Haskell programs, the implementation of the debugger relies on expressions with side-effects. Since Haskell bans such expressions, programmers that need this functionality either write low-level routines in C or in an

To appear in: ACM SIGPLAN Haskell Workshop, 1997

ad hoc variant of Haskell that integrates computational effects with the call-by-need semantics. Recently Launchbury and Peyton Jones [24] convincingly argued for the second approach:

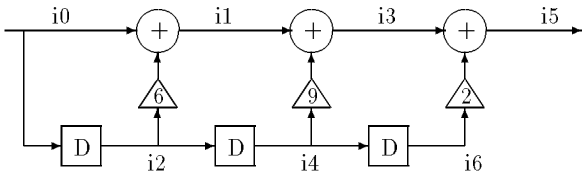
> "Should we outlaw `interleaveST` on the grounds that it is insufficiently well behaved? Not necessarily. Outlawing `interleaveST` would simply drive its functionality underground rather than prevent it happening. For example, we want to have lazy file reading. If it cannot be implemented in Haskell then it will have to be implemented "underground" as a primitive operation written in C or machine code. The same goes for unique-supply trees and lazy arrays.
>
> *Implementing such operations in C does not make them more hygienic or easy to reason about.* On the contrary, it is much easier to understand, modify and construct variants of them if they are implemented in a Haskell library module than if they are embedded in the runtime system"[24, p.45].

We therefore study a variant of Haskell with call-by-need semantics and handleable exceptions. This variant of Haskell is reminiscent of the informal Haskell dialect used by some programmers, at their own risk, to implement generation of new names [6], memoization [20], non-deterministic choice [21] and other paradigms that are *not* expressible in a modular way in a purely functional language. It is also reminiscent of the intermediate language of Haskell compilers like the Glasgow Haskell compiler, that implement monadic state by updates on a global shared store.

The next section motivates the use of streams in programming reactive systems. Section 3 discusses conventional debugging techniques and explains why they are unsuitable for our purpose. Section 4 presents a realistic example that uses Haskell to write the controller for a small industrial production cell. The example deadlocks due to a subtle bug that we unintentionally introduced in our first attempt at writing the code. We illustrate the user's interface of the debugger and show how a simple debugging session fixes the problem. Section 5 provides the details of the implementation of the debugger, and reveals that the implementation requires handleable exceptions. Section 6 presents the semantics of such exceptions in a call-by-need language. Finally Section 7 puts our work in perspective by explaining its weaknesses and charting ideas for future development.

2 Programming Reactive Systems

Reactive systems are programs that must *continuously* react to external stimuli from their surrounding environment [17]. For example, a digital filter is usually specified as a weighted sum of the n last samples of the input signal. The following diagram depicts such a filter:



The block D stands for a delay element. The triangles represent elements that scale every value in the stream by the corresponding number.

It is relatively straightforward to transliterate the filter to working Haskell code. The first step consists in implementing the intended semantics of each block:

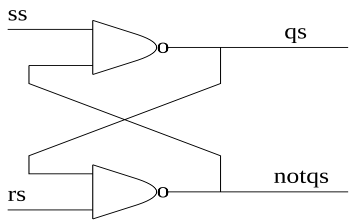
```
sum (x : xs) (y : ys) = (x + y) : sum xs ys
delay x = tail x
weight w = map (w*)
```

Having specified the blocks, it remains to write a set of mutually recursive equations over streams that reflects the interconnections of the network:

```
filter i0 = i5
 where i1 = sum i0 (weight 6 i2)
       i2 = delay i0
       i3 = sum i1 (weight 9 i4)
       i4 = delay i2
       i5 = sum i3 (weight 2 i6)
       i6 = delay i4
```

Note that the generation of this last piece of code can be easily automated given the diagram: it only depends on the interconnections between the components, not on their semantics.

The above development process is simple, natural, and elegant. But one must not get carried away; there are some limitations! Consider the following diagram specifying a basic flip-flop circuit with `nand` gates and the program produced by naïvely following our development process:




```
nand (1 : xs) (1 : ys) = 0 : nand xs ys
nand (_ : xs) (_ : ys) = 1 : nand xs ys

ff rs ss = (qs, notqs)
  where qs = nand ss notqs
        notqs = nand rs qs
```

Although the physical flip-flop device works, our program doesn't. And in fact pure Haskell appears ill-suited for such asynchronous circuits.

However, despite its limitations the stream model is suitable for many applications and is at the core of more expressive formalisms that accommodate asynchronous events, real-time, non-determinism, etc.

3 Related Work

To understand the contribution of our work, it is necessary to review the available debugging tools for Haskell in general and stream computations in particular. But first it is worth noting that most implementations of synchronous languages include debuggers and simulation environments in which users can experiment with the code. The widespread use of such tools reinforces our claim that they are essential in a programming environment that supports computations over streams.

The lazy (demand-driven) evaluation of functional languages schedules computations depending on the demands for their results. This evaluation mechanism is hard to predict and to relate to the source code of the program, and hence is not well suited for conventional debuggers that operate at a low level of abstraction that is closer to the machine code than to the source program. This led researchers to propose *declarative debugging* as a viable strategy for debugging lazy functional programs [27, 28, 29].

One promising declarative debugging technique is *algorithmic debugging* [30, 33, 34]. Algorithmic debugging is a two phase process. First, the program is run and an execution dependence tree is built. Then, this tree is interactively traversed. Each node in the tree corresponds to a function call and its result. While traversing the tree (in a top-down manner), the system asks, for each node, if the result of the corresponding function call is correct or not. Based on the answers the programmer gives, the system chooses which function call to show next. A bug has been found when the result of a function call is erroneous and all children nodes are correct. The method requires that the functions have no side effects, which makes it suitable for pure functional languages.

The ability to view the value of any expression in a program execution, and the expressions on which that value depends, is generally useful for debugging many lazy functional programs. Unfortunately, it is not of much help when debugging stream computations. To understand a computation that deadlocks while manipulating several streams, it is crucial to:

- understand the sequence of *events* that led to the deadlock, and
- have a view of *all* the relevant streams at the point in which the deadlock occurred.

Algorithmic debugging abstracts away from time and does not faithfully report the evaluation steps in the order in which they actually happened. Furthermore, users of an algorithmic debugging tool never have a global view of the computation in which they can keep track of several streams at once. These properties make algorithmic debugging unsuitable for locating deadlocks in stream computations.

4 Controlling An Elevating Table

We present a simplified version of the actual program fragment that spurred the idea of the new debugger.

4.1 The Program

Figure 1 shows a small fragment of a case-study production cell [26]. This fragment is concerned with the implementation of the controller of a simple elevating table and testing it with a simulated hardware environment. The table is initially in a low position. When the controller detects an input blank, it issues an `Up` command to the motor interface. This abstract command is implemented by the interface using a low-level protocol that reads sensors and emits control signals (actuators) to a motor. The block `Env` simulates the actual environment. Assuming the environment operates correctly, the controller eventually receives an acknowledgment, delivers the blank to the next device (a robot arm), and issues a `Down` command to lower the table to its initial position. The `Down` command is implemented by the interface using a protocol similar to the one implementing `Up`; when the `Down` command is acknowledged the controller is ready to accept more input blanks.

We will give the precise implementation of the protocols implementing the commands `Up` and `Down`. First we formally describe the sets of messages that are required by the specification of the table [26]:

```
data MotorCmd  = Up | Down
data Sensor = Low Bool | High Bool
type SensorVector = (Sensor,Sensor)
data Actuator = MotorUp | MotorDown
              | MotorStop | GetStatus
```

The specification of the system requires the existence of two sensors that identify the current location of the table. For example, when the table is in the low position, the sensors are `(Low True, High False)`. The controller can only access the current values of the sensors by issuing a `GetStatus` control signal; changes to the sensors between `GetStatus` signals are undetected by the controller. Finally, the motor that moves the table can be controlled using three self-explanatory control signals: `MotorUp`, `MotorDown`, and `MotorStop`.

The above description yields the following code for an idealized environment:

```
env acs = loop acs (Low True, High False) where
 loop (ac:acs) currentSensors =
   case ac of
    GetStatus  -> currentSensors :
                  loop acs currentSensors
    MotorUp    -> loop acs (Low False, High True)
    MotorDown  -> loop acs (Low True, High False)
    MotorStop  -> loop acs currentSensors
```

The environment code assumes that the table is initially in the low position and repeatedly looks at the current control signal in the actuators stream. If the signal is `GetStatus`, then the current values of the sensors are returned. Otherwise the appropriate action is performed, possibly modifying the current values of the sensors.

It remains to perform the calculation of the `actuators` stream in the module `Interface`:

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

        goU mcs ((Low _, High True) : ss) =
          MotorStop : emit mcs ss
        goU mcs ((Low _, High _) : ss) =
          MotorUp : GetStatus : goU mcs ss

        goD mcs ((Low True, High _) : ss) =
          MotorStop : emit mcs ss
        goD mcs ((Low _, High _) : ss) =
          MotorDown : GetStatus : goD mcs ss
```

First the code issues a `GetStatus` signal to get the initial values of the sensors. Then we start a loop that examines the current abstract command and the current values of the sensors and emits the appropriate actuators. In a realistic environment, the actuators would not be immediately acknowledged and the controller might have to loop several

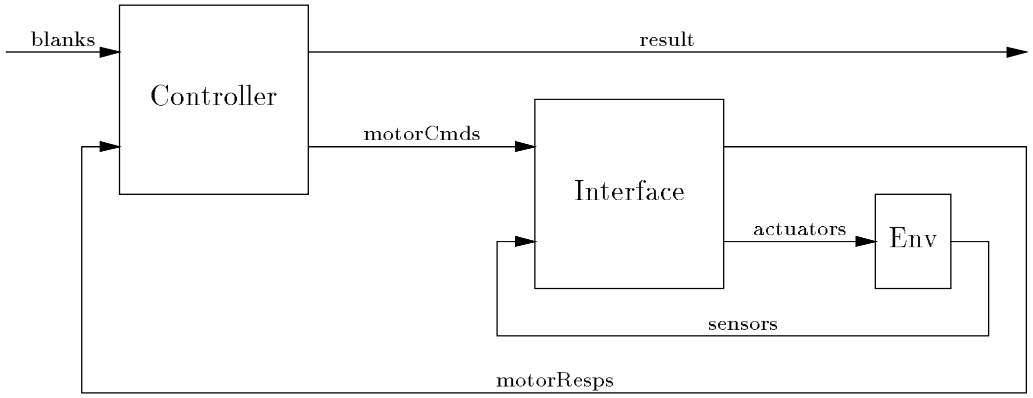


Figure 1. Elevating Table

times until the sensors confirm that the actuator signal has been acknowledged.

The main program reflects the interconnections in Figure 1. To test the program fragment independently, let's assume that the stream of `motorCmds` consists of an infinite sequence of alternating `Up` and `Down` commands:

```
motorCmds = Up : Down : motorCmds
actuators = interface motorCmds sensors
sensors = env actuators
```

Note how the definitions of `sensors` and `actuators` are mutually recursive.

The current code is overly simplified because it ignores the remainder of the production cell, and even ignores some aspects of the elevating table like the fact that it should rotate left and right while moving up and down. Nevertheless, the code captures the essence of our initial implementation of the elevating table including the fact that it contains a subtle error causing the system to deadlock! Because of the simplifications, the bug may be obvious at this time, but it certainly wasn't obvious in the complete program (around 600 lines of Haskell).

4.2 Finding the Deadlock

Our debugger operates at a high-level of abstraction that reflects the source program itself. Instead of getting lost in details about lazy evaluation, our debugging technique maintains the illusion of a mostly declarative computation expressed as a fixed point of recursive equations over streams.

We start by illustrating the user's point of view by going through a possible debugging session for the elevating table code. To find the source of the deadlock the user must identify the relevant streams of interest. This information is used by the debugger to keep track of, and display the sequences of values that belong to these streams. In our case, we only have three streams and we state that we are interested in all of them.

When running the program under the control of the debugger, the window in Figure 2 appears (initially empty). The user then can interact with the computation by clicking on the buttons `Next`. Each click returns the next value on the corresponding stream as well as all values on the other streams on which it depends. In our example, the first click on the stream `actuators` returned the value `GetStatus` without demanding any other values on the other streams. A second click on the same stream returned the values `MotorUp` and `GetStatus`. These two values can only be produced if a command `Up` is issued on the stream `motorCmds` and the sensors indicate that the table is in the low position. After the next click the computation has proceeded to the point where the table is in the high position and the motor is ordered to stop. When attempting to process the `Down` command, the next click displays the symbol $\perp$ in the columns for the streams of sensors and actuators indicating a cyclic dependency between the next two values, *i.e.*, a deadlock.

Even before examining the code for bugs, we can immediately note one important fact: each `GetStatus` signal on the stream of actuators produces the current values of the sensors on the next click. The first two `GetStatus` signals have been processed and acknowledged and no values can appear on the stream of sensors unless a new `GetStatus` signal is issued. It is also clear that the deadlock occurs at a point in the neighborhood of the code that issues the `MotorStop` signal and the error manifests itself immediately after consuming the command `Down`. Looking back at the relevant piece of code we find:

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

(*)     goU mcs ((Low _, High True) : ss) =
          MotorStop : emit mcs ss
        goU mcs ((Low _, High _) : ss) =
          MotorUp : GetStatus : goU mcs ss

        goD mcs ((Low True, High _) : ss) =
          MotorStop : emit mcs ss
        goD mcs ((Low _, High _) : ss) =
          MotorDown : GetStatus : goD mcs ss
```

In the line marked `(*)` we see that after issuing the signal `MotorStop`, we proceed to the `emit` loop which consumes a `Down` command, and then attempts to access the stream of sensors without any intervening `GetStatus` signals. Thus one way to fix the code would be to simply add a `GetStatus` signal after stopping the motor (in both lines that stop the motor):

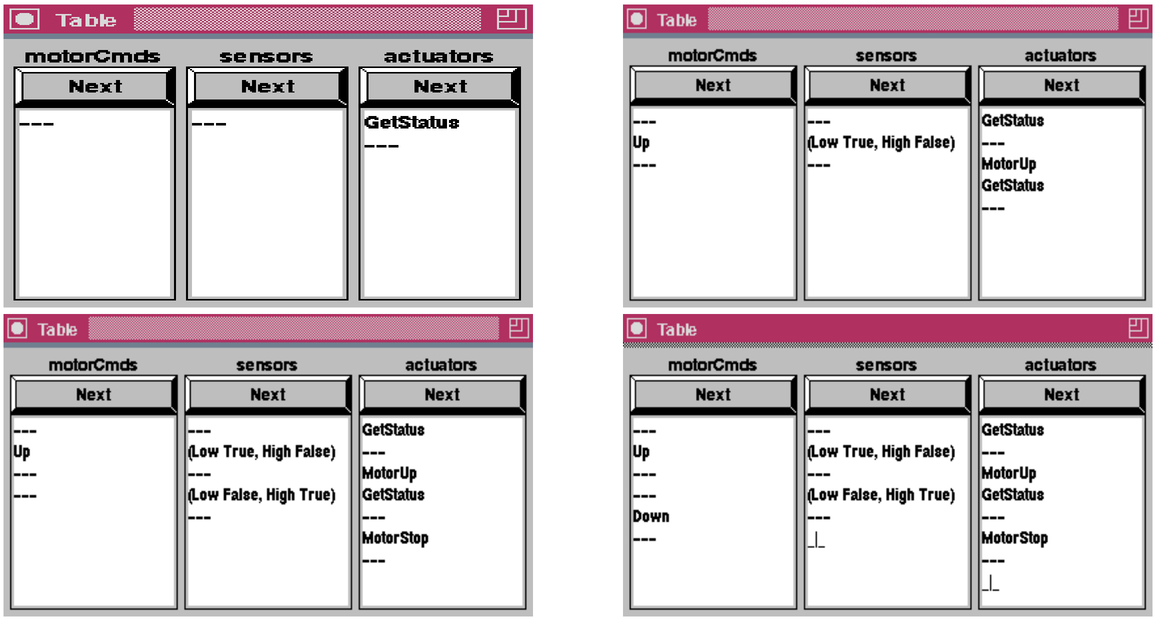


Figure 2. A Sample Debugging Session

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

        goU mcs ((Low _,High True):ss) =
          MotorStop:GetStatus:emit mcs ss
        goU mcs ((Low _,High _):ss) =
          MotorUp:GetStatus:goU mcs ss

        goD mcs ((Low True,High _):ss) =
          MotorStop:GetStatus:emit mcs ss
        goD mcs ((Low _,High _):ss) =
          MotorDown:GetStatus:goD mcs ss
```

And indeed the code now works just fine. A proof that the new code does not produce deadlocks is outside the scope of this paper but could be achieved either by compiling the code to a finite state machine or subjecting it to our liveness analysis [22].

5 Implementing the Debugger

Having argued that the debugger would be useful to include in a programming environment for Haskell, we turn our attention to the details of our current implementation in the context of the Chalmers Haskell compiler 'hbc.'

5.1 Background

The fundamental philosophy behind lazy evaluation [18] is that expressions are not evaluated unless their values are demanded. The initial demand usually starts from the top level routine that seeks to print the answer of the program. During debugging, the initial demand starts when the user clicks on one of the streams. In both cases the demands are propagated until they reach an expression that is already evaluated. Such an expression is said to be in *weak head normal form* (whnf). Examples of expressions in whnf include integers, λ-expressions, and pairs. Generally speaking an expression that consists of a constructor applied to arbitrary (possibly unevaluated) argument expressions is in whnf. For example, the expression `(1+2 : f xs)` is in whnf, since it consists of a top level constructor `(:)`, an unevaluated head `1+2`, and an unevaluated tail `f xs`.

Lazy evaluation is viable because once an expression has been evaluated, there is no need to re-evaluate it again. Since the language is purely functional, all subsequent evaluations of an expression are guaranteed to return the same result as the first, and hence are useless. This property immediately implies that it is impossible to write a Haskell expression that evaluates to `True` the first time, and to `False` the second time.

A deadlock occurs in a lazy functional program when the evaluation of an expression needs its own value to proceed. An obvious way of detecting deadlocks is thus to mark an expression as being under evaluation when its evaluation starts. If the value of such a marked expression is needed, we have a deadlock. Most compilers for lazy functional languages mark expressions in this way and can indeed detect deadlocks at run-time albeit with an obscure error message about black holes.

5.2 Global View

The debugger consists of a number of stream agents and a top level coordinator. A stream agent is responsible for evaluating and displaying the elements of one stream. The graphical representation of each stream agent is as shown in Figure 2.

The top level coordinator and the stream agents interact as follows. When the button Next of a stream agent is clicked, the agent tries to evaluate the next element of the stream. It then sends a notification message to the coordinator. Upon receipt of such a notification, the coordinator sends a broadcast message to all stream agents, to which each agent responds by checking for, and displaying, the evaluated elements on its stream.

5.3 Peeking at Streams

To perform their jobs, the stream agents need to peek at streams to see how many elements are already evaluated to whnf. Unfortunately, in a lazy functional language it is not possible to look at a stream to see how many elements have been produced. In fact, it is by looking at them that they will be produced, since there is demand for them! So when inspecting a stream, the agent would want to see the stream as a read-only data structure.

For this reason we have implemented a new primitive function `peek` that takes an argument and returns a value of type EvalStatus:

```
data EvalStatus = Evaluating | Closure | Evaluated
peek :: a -> EvalStatus
```

Clearly the above function is not pure and hence is not expressible in Haskell. Indeed, it may return different answers when called with the same argument twice. This function is not available to programmers.

Using `peek` we define a function `safeInitialList` that takes a stream as argument, and returns the initial list of already evaluated elements in the stream, a flag saying whether the stream has become deadlocked or not, and the stream itself (except for the initial evaluated segment):

```
safeInitialList :: [a] -> ([a], Bool, [a])
safeInitialList xs =
 case peek xs of
   Evaluating -> ([], True, xs)
   Closure    -> ([], False, xs)
   Evaluated  ->
    case xs of
     y:ys -> let (zs, b, ws) = safeInitialList ys
             in (y:zs, b, ws)
     []   -> ([], False, [])
```

5.4 Recovering from Fatal Errors

A fatal error is one which normally terminates a program with an error message, *e.g.*, a division by zero or a deadlock. Since our debugger is part of the same program as the application, we certainly don't want such errors to abort the debugger. To be able to recover from such faults and get some clues on which part of the source code is responsible for the fault, we introduce a pair of functions `catch` and `throw`:

```
throw :: String -> a
catch :: a -> (String -> a) -> a
```

that provide a simple form of exception handling. The semantics of these extensions is presented in the next section.

Intuitively, they are used as follows. The compiler is modified so that all fatal errors call `throw` with an appropriate error message. These errors are caught (handled) by the debugger. For example, the evaluation of `result` below produces 42:

```
result = catch (f []) handler
f (x:xs) = x + f xs
handler errMsg = 42
```

Otherwise, when not debugging, there will be no active `catch` and the program would terminate with the usual error message: `"No match in f"`.

6 Call-by-Need & Exceptions

Consider a small functional *call-by-need* language extended with exceptions.

Definition 6.1 (Syntax) *The set of terms includes:*

$$\begin{array}{rcl} M, N & ::= & V \mid M + N \mid x \mid MN \\ & & \mid \texttt{throw}\ M \mid \texttt{catch}\ M\ N \\ V & ::= & n \mid \lambda x.M \end{array}$$

In other words, the language extends the pure λ-calculus with a representative set of constants including numbers and addition as well as two expressions for throwing and handling exceptions. The intended semantics of `throw` and `catch` is the usual one [35] adapted to a call-by-need language. Given an expression M, the evaluation of ($\texttt{throw}\ M$) aborts the computation and returns M. The evaluation of ($\texttt{catch}\ M\ N$) where N is a handler proceeds as follows. First evaluate M to whnf; if this evaluation terminates normally then its result is the value of the entire `catch` expression. Otherwise if the evaluation of M throws an exception, the handler is called with the thrown expression as an argument.

Definition 6.2 (Syntax of Types) *The set of types is inductively defined as follows:*

$$\tau \quad ::= \quad \texttt{Int} \mid \tau \rightarrow \tau'$$

The typing rules for our language are in Figure 3. We let Γ scope over type environments (partial mappings from variables to types). A type judgment $\Gamma \vdash e : \tau$ means that under the assumptions in the type environment Γ, expression e has type τ.

The typing rules for the functional core are standard. The typing rules for `throw` and `catch` are similar to the ones found in SML [37] but, for simplicity, we have restricted all thrown expressions to be of type `Int`.

If typechecking guarantees anything, it is that some obviously faulty terms are not typable.

Definition 6.3 (Faulty Terms) *An expression is* faulty *if it is contains one of the following subterms:*

- $n\ M$, *or*
- $(\lambda x.M) + V$, *or*
- $V + (\lambda x.M)$.

It is easy to confirm that faulty terms are not typable.

$$\frac{}{\Gamma \cup \{x:\tau\} \vdash x:\tau} \qquad \frac{\Gamma \cup \{x:\tau\} \vdash M:\tau'}{\Gamma \vdash \lambda x.M:\tau \to \tau'} \qquad \frac{\Gamma \vdash M:\tau' \to \tau \qquad \Gamma \vdash N:\tau'}{\Gamma \vdash M\ N:\tau}$$

$$\frac{}{\Gamma \vdash n:\mathtt{Int}} \qquad \frac{\Gamma \vdash M:\mathtt{Int} \qquad \Gamma \vdash N:\mathtt{Int}}{\Gamma \vdash M+N:\mathtt{Int}}$$

$$\frac{\Gamma \vdash M:\mathtt{Int}}{\Gamma \vdash \mathtt{throw}\ M:\tau} \qquad \frac{\Gamma \vdash M:\tau \qquad \Gamma \vdash N:\mathtt{Int} \to \tau}{\Gamma \vdash \mathtt{catch}\ M\ N:\tau}$$

Figure 3. Typing Rules

6.1 Evaluation Contexts

Because of the *lazy* semantics of the language, the definition of the axioms is intertwined with the definition of evaluation contexts [2, 3]. Intuitively, "needed" variables within subterms correspond to those variables that occur in evaluation context positions. Therefore, we define evaluation contexts first before defining the semantics of the language.

Definition 6.4 (Evaluation Contexts E) *The set is inductively defined as:*

$$\begin{array}{rcl} E & ::= & [\,] \mid EM \mid (\lambda x.E)\ M \mid (\lambda x.E[x])\ E \\ & & \mid E+M \mid n+E \mid \mathtt{catch}\ E\ M \end{array}$$

The first four clauses are identical to the ones for the pure call-by-need calculus. They have the following intuitive explanation. The context EM reflects the fact that the function position of an application is needed. The next two contexts make the call-by-need nature of the language evident: arguments to functions are not evaluated when the function is called. Indeed, we first attempt to evaluate the body of the function without touching the argument. Only if the body of the function "needs" the value of the parameter, do we start processing the argument.

The contexts $E+M$ and $n+E$ state that both arguments to an addition are needed: first the left one, then the right one. In the presence of exceptions and handlers, this order is clearly significant. Finally the context $\mathtt{catch}\ E\ M$ reflect our informal understanding of the semantics of a `catch` expression which first attempts to evaluate the first argument. The second argument (the handler) is only needed if the evaluation of the first argument throws an exception.

6.2 Axioms

We are almost in a position to define the semantics of our language. We begin by identifying a syntactic category of terms that are *answers* and how to observe them.

Definition 6.5 (Answers) *An answer A is one of the following terms:*

$$A \ ::= \ V \mid \mathtt{throw}\ M \mid (\lambda x.A)\ M$$

When observed, answers yield the following output to users:

$$\begin{array}{rcl} obs(n) & = & n \\ obs(\lambda x.M) & = & \text{"Procedure"} \\ obs(\mathtt{throw}\ M) & = & \text{"Uncaught Exception"} \\ obs((\lambda x.A)\ M) & = & obs(A) \end{array}$$

Our definition of answers extends the usual definition for the pure call-by-need calculus by adding an additional kind of answers for uncaught exceptions.

We can now specify the semantics of the language via the set of axioms in Figure 4. The first three axioms are essentially the axioms of the pure call-by-need calculus. The next three axioms specify the semantics of exceptions. A `throw` expression aborts any context other than a `catch`. The value of a `catch` expression is either the value of the first argument or the result of applying the handler to the value thrown by the first argument.

The following proposition guarantees that the definitions of evaluation contexts, faulty expressions, and axioms are consistent and complete.

Proposition 6.1 *Every closed term is either an answer, faulty, or can be uniquely decomposed into an evaluation context and a redex.*

6.3 Evaluator

The above axioms can be used in any context to transform programs (perhaps for optimization purposes). They can also be used to specify an evaluator by orienting them from left to right and always choosing the redex that occurs in the hole of the evaluation context.

Definition 6.6 *The call-by-need evaluator is defined as follows: $eval(M) = B$ if $M \longmapsto^* A$ and $obs(A) = B$, where $M \longmapsto N$ if and only if:*

- *$M = E[R]$,*
- *$R = R'$ is an axiom of Figure 4 (R must be the left hand side), and*
- *$N = E[R']$.*

6.4 Connection to Call-by-Name

Clearly the semantics of a call-by-need with exceptions is *not* the same as the semantics of the call-by-name variant. This is in contrast to the pure functional subset in which call-by-name reasoning principles like β are sound. However this is no worse than the current intermediate languages of Haskell compilers, like the Glasgow Haskell compiler, that use a call-by-need language with assignments to implement monadic state efficiently. Indeed β is also unsound in the intermediate language of the compiler for precisely the same reason [31, 25].

Abbreviations for one-step contexts:

$$\begin{array}{rcl} G & ::= & H \mid \texttt{catch}\,[\,]\,M \\ H & ::= & ([\,]\,M) \mid [\,] + M \mid n + [\,] \mid (\lambda x.E[x])\,[\,] \end{array}$$

Axioms:

$$\begin{array}{rcll} (\lambda x.E[x])\,V & = & (\lambda x.E[V])\,V & \\ G[(\lambda x.A)\,M] & = & (\lambda x.G[A])\,M & \\ n_1 + n_2 & = & n & \text{where } n = n_1 + n_2 \\ \texttt{catch}\,V\,M & = & V & \\ H[\texttt{throw}\,M] & = & \texttt{throw}\,M & \\ \texttt{catch}\,(\texttt{throw}\,M)\,N & = & NM & \end{array}$$

Figure 4. Axioms

7 Current Limitations and Future Work

We have presented a novel debugger for reasoning about the recursive computations over streams usually found in reactive systems. The debugger appears to work remarkably well for small programs, but there is a need for further work to handle larger programs and to formalize the theoretical foundations of the current implementation or to investigate different implementation strategies. We expand on these points in the remainder of the section.

There are three main limitations of the current system. First, the number of streams that can be simultaneously debugged is limited by the size of the screen. In practice we have not found this to be a problem. In large examples, it is relatively easy to use the debugger to narrow down the deadlock to a handful of streams.

Second, our implementation relies on impure primitives that, although confined to the internals of the compiler, could pollute the source language and break some obvious program optimizations. For example, the following two definitions of `ones` are equivalent in pure Haskell but not equivalent during debugging sessions:

```
ones = map id (1 : ones)
ones = 1 : ones
```

To see the difference consider the call (`safeInitialList ones`). Using the first definition of `ones`, the result is a triple of the form (`[], False, ones`). Using the second definition of `ones`, the call (and hence the debugger itself) diverges! To avoid such situations, a fully evaluated stream cannot be used during debugging sessions. Indeed in the elevating table example, the definition that we used for `motorCmds` is actually:

```
motorCmds = map id (Up : Down : motorCmds)
```

This trick of inserting `map id` around trivial stream definitions works fine with the current version of the 'hbc' compiler but is unsatisfactory. A smart compiler that is unaware of the debugger can certainly optimize (`map id e`) to `e` since they are equivalent in pure Haskell. A better solution would be to integrate the debugger and the compiler in a programming environment which would require them to agree on a common representation for streams such as `motorCmds` above.

As another example, addition is commutative in pure Haskell, *i.e.*, $M + N = N + M$. This equivalence no longer holds in the presence of `catch` and `throw`. The evaluation of `result` below produces 5 or 6 depending on the order of evaluation of the arguments to `+`:

```
result = catch (f ()) handler
f () = throw "L" + throw "R"
handler errMsg = if errMsg=="L" then 5 else 6
```

General programmers do not have access to `catch` and `throw` and this situation does not seem to occur in the current debugger. Nevertheless a proof of safety would be interesting. This situation is reminiscent of the use of assignments in the implementation of monadic state without affecting the purity of the source language [24, 25].

Third, the debugger can only cope with streams defined at the top level of the program. Thus if we have a large program in which one function uses locally defined streams, we cannot debug the program until these streams are globally defined at the top level. Such a change may require a major restructuring of the program, and may even be impossible in certain circumstances. This problem is a general Haskell problem: it is impossible to perform any computational effect (*e.g.*, printing) as a side-effect of evaluating an expression. Instead computational effects must be propagated to the top level either explicitly or by using a monad [36]. This is the major limitation of the debugger; we are actively seeking a solution to this problem.

Finally, we have started an investigation of the semantics of computational effects in call-by-needs languages since we believe that this will lead to more robust programs, and will formalize some of the *ad hoc*, informal, and potentially unsafe programming practices in system-level Haskell programs.

References

[1] Abelson, H., and Sussman, G. J. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] Ariola, Z. M., and Felleisen, M. The call-by-need lambda calculus. To appear in the *Journal of Functional Programming*, 1996.

[3] Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages* (1995), pp. 233–246.

[4] Ashcroft, E. A., and Wadge, W. W. Lucid, a nonprocedural language with iteration. *Communications of the ACM 20*, 7 (1977), 519–526.

[5] Ashcroft, E. A., and Wadge, W. W. *Lucid, the Data-Flow Programming Language.* Academic Press, 1985.

[6] Augustsson, L., Rittri, M., and Synek, D. Splitting infinite sets of unique names by hidden state changes. Tech. Rep. 67, Chalmers University of Technology, 1992.

[7] Barton, D. L., and Dunlop, D. D. An introduction to MHDL. In *Proceedings of IEEE 1993 MTT-S Symposium* (San Diego, CA, May 1993), pp. 1487–1490. (Paper UU-1).

[8] Benveniste, A., LeGuernic, P., and Jacquemot, C. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming 16* (1991), 103–149.

[9] Berry, G., and Cosserat, I. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency* (1985), S. Brookes and G. Winskel, Eds., vol. 197 of *Lecture Notes in Computer Science*, Springer.

[10] Berry, G., and Gonthier, G. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[11] Broy, M., et al. The design of distributed systems—an introduction to Focus. Tech. Rep. SFB-Bericht Nr. 342/2-2/92 A, Technische Universität München, 1992.

[12] Caspi, P., Halbwachs, N., Pilaud, D., and Plaice, J. Lustre: A declarative language for programming synchronous systems. In *ACM Symposium on Principles of Programming Languages* (1987).

[13] Caspi, P., and Pouzet, M. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming* (1996), pp. 226–238.

[14] Edwards, S. An Esterel compiler for a synchronous/reactive development system. `ftp://ic.eecs.berkeley.edu/pub/Esterel/esterelcomp.ps.gz`.

[15] Hall, C. V., Hammond, K., and O'Donnell, J. T. An algorithmic and semantic approach to debugging. In *Proceedings of the Glasgow Workshop on Functional Programming* (1990).

[16] Hall, C. V., and O'Donnell, J. T. Debugging in applicative languages. *Lisp and Symbolic Computation* (1988).

[17] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

[18] Henderson, P., and Morris, Jr., J. H. A lazy evaluator. In *ACM Symposium on Principles of Programming Languages* (1976), pp. 95–103.

[19] Hudak, P., Peyton Jones, S. L., and Wadler, P. Report on the programming language Haskell, a nonstrict purely functional language (version 1.2). *SIGPLAN Notices 27*, 5 (1992).

[20] Hughes, J. Lazy memo-functions. Tech. Rep. 21, Chalmers University of Technology, 1985.

[21] Hughes, J., and Moran, A. Making choices lazily. In *Conference on Functional Programming and Computer Architecture* (1995). To appear.

[22] Hughes, J., Pareto, L., and Sabry, A. Proving the correctness of reactive systems using sized types. In *ACM Symposium on Principles of Programming Languages* (1996).

[23] Kahn, G. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress* (1974), J. L. Rosenfeld, Ed., North-Holland, pp. 471–475.

[24] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8* (1995), 193–341.

[25] Launchbury, J., and Sabry, A. Monadic state: Axiomatization and type safety. In *ACM SIGPLAN International Conference on Functional Programming* (1997).

[26] Lewerentz, C., and Lindner, T. *Formal Development of Reactive Systems: Case Study Production Cell*. Lecture Notes in Computer Science 891. Springer-Verlag, 1995.

[27] Naish, L., and Barbour, T. Towards a portable lazy functional declarative debugger. Technical Report 95/27, Department of Computer Science, University of Melbourne, Australia, 1995.

[28] Nilsson, H. A declarative approach to debugging for lazy functional languages. Licentiate Thesis No. 450, Department of Computer and Information Science, Linköping University, Sept. 1994.

[29] Nilsson, H., and Fritzson, P. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming 4*, 3 (July 1994), 337–370.

[30] Nilsson, H., and Sparud, J. The evaluation dependence tree as a basis for lazy functional debugging. *Journal of Automated Software Engineering 4*, 2 (April 1997), 152–205.

[31] Sabry, A. What is a purely functional language? Unpublished manuscript, April 1997.

[32] Sijtsma, B. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems 11*, 4 (1989), 633–649.

[33] Sparud, J. A transformational approach to debugging lazy functional programs. Licentiate Thesis, Department of Computing Science, Chalmers University of Technology, Feb. 1996.

[34] Sparud, J., and Nilsson, H. The architecture of a debugger for lazy functional languages. In *Proceedings of AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging* (Saint-Malo, France, May 1995), M. Ducassé, Ed., IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, Cedex, France.

[35] Steele, G. L. *Common Lisp—The Language*. Digital Press, 1984.

[36] Wadler, P. Comprehending monads. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 61–78.

[37] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 91-160, Rice University, April 1991. Final version in *Information and Computation* **115** (1), 1994, 38–94.