Handling Exceptions in Haskell

Alastair Reid
Yale University
Department of Computer Science
New Haven, CT 06520
reid-alastair@cs.yale.edu

January 19, 1999

Abstract

Using a language without exception handling is like driving a car with no brakes and no seatbelt — things work fine until something goes wrong. You also learn to drive rather carefully.

This paper describes an exception handling extension to the Haskell lazy functional language. The implementation turned out to be very easy but we had problems finding a viable semantics for our system. The resulting semantics is a compromise between theoretical beauty and practical utility.

1 Introduction

Haskell is an ivory tower language: full of lofty ideas, built on solid semantic foundations, praised by grey-bearded professors and about as much use in the real world as a chocolate teapot. For Haskell to emerge from the ivory tower, it must be possible to write the kinds of programs that less idealistic programmers can write in C, Java, Ada and other useful languages: programs that interact with the real world in interesting ways (using graphics, GUIs, databases, etc) and which are robust enough to keep running even when things go wrong.

Recent work on Haskell has dealt with the problem of interacting with the real world: Haskell's IO monad [6, 3] provided an extensible framework for interacting with the real world; GHC's ccall extension [6] made it possible to use libraries written in C; GHC's foreign pointers [11] made it possible to deallocate C objects without compromising laziness; Hugs-GHC's GreenCard [9] made it easy to use standard C libraries; the Hugs-GHC standard libraries [7] added support for fixed size integers facilitating access to C libraries that use them.

This increased ability to interact with the real world is a double-edged sword: our programs may achieve wondrous things when they work correctly; but they can wreak untold havoc when they fail. For example: if a program fails halfway through modifying a database, it might corrupt the database; if a program fails while interacting with the user, it might leave a confused mess of windows on the screen; if a program fails while controlling a robot, the robot might crash into walls or run over the programmer's foot; if a Haskell interpreter fails while executing a user's program, it might abort the interpreter instead of printing an error message and prompting for the next command. Given that any interesting program can go wrong, the only solution is to provide mechanisms for dealing with failure when it happens and to educate programmers to use them.

Haskell's IO monad [3] recognises the importance of exception handling — providing a simple mechanism for raising and catching exceptions within the IO monad. Careful use makes Haskell programs much more robust but exceptions can only be raised within the IO monad — so there is no way to catch exceptions like calls to the error function, division by zero or pattern match failure which occur within "pure" code. This paper describes an extension which lets programs catch "internal exceptions" (e.g. calls to the error function, pattern match failure and division by zero); the essential but different task of catching "external exceptions" (e.g. interrupts and timeouts) is discussed in a companion paper [12] and outlined in Section 5.

The main difficulty in extending Haskell's exception handling capabilities is to avoid compromising Haskell's main strengths: lazy evaluation, type safety, support for equational reasoning and its amenability to both manual and automatic transformation. Section 2 recalls a standard exception handling mechanism (the exception monad and the call-by-name monad translation) and Section 3 describes an efficient implementation of this mechanism. This is the easy part of adding exception handling and is essentially a reprise and update of Dornan and Hammond's work [1]. Section 4 points out a significant flaw in this approach: even though this mechanism preserves laziness, type safety and referential transparency, it renders many common transformations invalid and apparently makes reasoning about Haskell programs difficult. Section 4 shows that this problem can be resolved with a small change in the design and a big change in the way we reason about exception producing programs. The development in the previous sections ignores the interaction between exception handling and two other exception-like features of Haskell: Section 5 describes these features and suggests a unified design.

2 The Exception Monad

Wadler [13] describes how ordinary programmers can add exception handling to a lazy program using the *exception* monad and the *call by name monad translation*. The exception monad and the call by name monad translation are shown in figures 1 and 2. We have extended Wadler's versions in two small ways: we use a `String` to hold error messages; and we cover the full Core Haskell language. In the rules for translating terms, we use x, x_i, ... for variables, e, e_i, ... for expressions, C for a constructor of arity m, k for a constant, *op*

for a strict primitive operation of arity m and $op^{\dagger}$ is the same as op except that it raises an exception whenever op returns $\perp$. The Haskell report [8] gives the necessary rules for transforming Haskell programs into Core Haskell. Figure 3 shows the monad translation in action on a user-defined function `average` and a primitive operation `divide`.

As well as being able to raise exceptions (via `error`, pattern match failure or primitive operations), we need a way to catch exceptions. Wadler provides the *biased-choice operator*

```
? :: E a -> E a -> E a
```

which chooses the first of two possible values that is well defined. Since we distinguish between different exceptions, we change the type slightly to allow the second argument to access the exception raised by the first argument; and we change the name to reflect a similarity to Haskell 1.4's `catch` function.

```
catchException :: E a -> (String -> E a) -> E a
```

(In fact, Section 4 explains that we need to change this type even more.)

This translation has the following desirable properties: it preserves laziness; it preserves type safety; it preserves confluence and termination; and it preserves referential transparency. Wadler's approach has some problems, most of which stem from the fact that we are *encoding* exception handling *in* Haskell rather than making it *part of* the language.

1. While the transformation is simple to apply, it is extremely tedious and error prone — which makes it hard to have any confidence in the (allegedly increased) reliability of the transformed system. The situation is made worse by the fact that one must manually desugar all of Haskell's syntactic extensions (nested patterns, list comprehensions, etc) — thus losing one of Haskell's primary features.

2. This transformation has to be applied to the entire program including the libraries and the standard Prelude. This requires access to the source code of the entire system and, since the Prelude is not just ordinary Haskell code, requires a lot of cooperation from the compiler writer.

3. Adding preconditions to the primitive operations is hard because, for example, it is hard to check for arithmetic overflow without causing overflow yourself and because the preconditions vary from one piece of hardware to the next.

4. Wrapping every data constructor in a `Value` constructor is expensive: almost everything becomes twice as big and twice as slow (we expand on this in the next section).

5. Standard program transformations change the meaning of program which raise exceptions. For example, replacing $a + b$ by $b + a$ changes the result of this expression

   ```
   let { a = error "a"; b = error "b" } in a + b
   ```

```
data E a = Error (E String) | Value a

instance Monad E where
  Error s >>= k = Error s
  Value a >>= k = k a
  return a      = Value a
```

Figure 1: The Exception Monad

$$
\begin{array}{rcl}
x^{\dagger} & = & x \\
(\lambda x \rightarrow e)^{\dagger} & = & \texttt{return}\ (\lambda x \rightarrow e^{\dagger}) \\
(e_1\ e_2)^{\dagger} & = & {e_1}^{\dagger}\ \texttt{`apply`}\ {e_2}^{\dagger} \\
 & & \textit{where}\ \texttt{apply}\ t\ u = t\ \texttt{>>=}\ \lambda f \rightarrow f\ u \\
(C\ e_1 \ldots\ e_m)^{\dagger} & = & \texttt{return}\ (C\ {e_1}^{\dagger} \ldots\ {e_m}^{\dagger}) \\
(\texttt{case}\ e\ \texttt{of}\{C\ x_1 \ldots\ x_m \rightarrow e_1; \_ \rightarrow e_2\})^{\dagger} & = & e^{\dagger}\ \texttt{>>=}\ \lambda x \rightarrow \texttt{case}\ x\ \texttt{of}\{C\ x_1 \ldots\ x_m \rightarrow {e_1}^{\dagger}; \_ \rightarrow {e_2}^{\dagger}\} \\
k^{\dagger} & = & \texttt{return}\ k \\
(op\ e_1\ \ldots\ e_m)^{\dagger} & = & {e_1}^{\dagger}\ \texttt{>>=}\ \lambda x_1 \rightarrow \ldots\ {e_m}^{\dagger}\ \texttt{>>=}\ \lambda x_m \rightarrow op^{\dagger}\ x_1 \ldots\ x_m \\
(\texttt{error}\ s)^{\dagger} & = & \texttt{Error}\ s^{\dagger}
\end{array}
$$

Figure 2: The Call By Name Translation for Core Haskell

```
  average :: [Float] -> Float
  average = \ xs -> (/) (sum xs) (length xs)
==>
  average :: E (E [Float] -> E Float)
  average = return (\xs -> divide (sum `apply` xs) (length `apply` xs))

  divide :: E Float -> E Float -> E Float
  divide x y = x >>= \ x' -> y >>= \ y' -> divide' x y
   where
    divide' x 0 = Error (Value "division by 0")
    divide' x y = return (x / y)
```

Figure 3: The Call by Name Translation in Action

6. This transformation provides no help with infinite loops.

Problems 1, 2 and 3 alone are enough to render this approach infeasible but can be solved by making exception handling part of the language and applying the transformation automatically. Problem 4 can be largely solved by careful implementation and is discussed in Section 3. Problem 5 is a major problem requiring a certain amount of compromise of theoretical beauty for practical utility and is discussed in Section 4. Problem 6 is a major thorn in our side: we're forced to take a pragmatic approach and treat these as resource limits (the limited resource being the patience or lifetime of the user!).

3 An Efficient Implementation

We could implement exception handling as a direct source-to-source transformation (plus some special treatment of primitive operations) using the monad and translation given in Section 2. We chose not to do so because the transformation is very expensive: everything becomes twice as big and twice as slow.

For example, using the STG machine on a 32-bit architecture, a `Cons` cell increases in size from 12 bytes (1 tag word plus 2 pointers) to 20 bytes (a `Cons` cell plus a `Value` cell which contains 1 tag word and 1 pointer) and an `Int` cell increases from 8 bytes to 16 bytes. Worse, all access to the fields of a data constructor requires two case analyses instead of one; all function applications require a case analysis and all primops need an error check.

This overhead can be reduced somewhat by adding a new constructor to every data type. For example, `Bool` and `List` could be defined as follows:

```
data Bool   = Error_Bool Error_String | False | True
data List a = Error_List Error_String | Nil   | Cons a (List a)
```

This eliminates the space overhead on constructors and eliminates the time overhead on case analyses, but the following problems remain:

- It's not possible to have a polymorphic `error` function: we must use a distinct `error` function for each type or we must overload `error` and modify the type of every polymorphic function which raises a polymorphic error.
- This will not work for `Int`s or functions since these are not ordinary datatypes.
- Raising an exception is relatively slow: for every case expression being executed we have to execute something like this:

```
case e of
{ Error_Bool err -> Error_List err
; False          -> ...
; True           -> ...
}
```

These extra case alternatives are particularily galling because they are so trivial: on detecting an error value, they just reraise the *same* error value.

Our solution is to extend the abstract machine with direct support for exception handling. Specifically, the `catchException` function pushes a special "exception handler frame" onto the stack and the `error` function unwinds the stack down to the topmost "exception handler frame" and invokes the associated exception handler.

On a naïve graph reduction machine, our job would now be done but the STG machine (on which we implemented our proposal) delays updating a thunk until after the thunk has been reduced to weak head normal form. Therefore, we must perform all those pending updates as we unwind the stack.

The STG machine maintains a list of pending updates which it threads through the stack. As the STG machine enters an updatable thunk, it adds the thunk to the list and as it returns the value of a thunk, it updates the thunk with its value and removes the thunk from the head of the list. To add exceptions to the STG machine, we add exception handler frames to the update list. This requires the following changes:

- When `catchException e h` is executed, we add an exception handler to the "update list."
- When `error err` is executed, we search down the update list for the topmost exception handler updating each pending update with an error thunk which will reraise `err` if the thunk is reentered. We then apply the topmost exception handler to `err`.
- When returning a constructor or a partially applied function (i.e. a value that is in weak head normal form), the STG machine already tests whether the top of the stack is a return address or an update frame. To this, we add a second test to check for an exception handler frame. If the top of the stack is a return address, the STG machine jumps to that address; if the top of the stack is an update frame, the STG machine performs the update, pops the update frame and tries again; and if the top of the stack is an exception handler, the STG machine pops the exception handler and tries again.

  The second test looks like it might be expensive but, fortunately, we are able to exploit an optimisation already present in the STG machine which is designed to make the first test cheap. The key idea is to make update frames look just like return addresses. That is, the topmost word of every update frame is the address of code which will perform an update when executed. With this small change, there is no need to test whether the top of the stack is a return address or not: we can just jump to the address without a test. This same optimisation works for exception handlers too: so we incur no extra cost when we add exception handling.

Note that we *do not* have to deal with return addresses (which are pushed by case expressions); we only have to deal with pending updates. This avoids most of the overhead associated with the simple source-to-source transformation.

The behaviour of the modified system is illustrated in figure 4 which shows the steps involved in evaluating the expression

```
catchException (error "a" + 1) (const 0)
```

Figures 4i–4v show how the STG machine unwinds the spine of the graph onto the stack and constructs the update list. Figure 4i shows the initial state of the machine: the stack contains a pointer to the representation of the expression to be evaluated (which is stored on the heap) and a `STOP` frame (which is the head of the update list). (The STG paper does not mention `STOP` frames but they were present in the implementation — after all, the last "real" update frame on the update list had to point to something!) Figure 4ii shows how an update frame is added to the update list when the first thunk is entered. Figure 4iii shows how an exception handler frame is added to the update list when `catchException` is executed. Note that the exception handler frame contains a pointer to the exception handler whereas an update frame contains a pointer to the updatee. Figure 4iv shows another update frame being added to the update list. Figure 4v shows the `+` primitive operation pushing a return address `ret_+` on the stack and evaluating its first argument. Since its first argument is an error, this triggers the exception handling mechanism.

Figures 4vi–4x show how the STG machine propagates and recovers from errors. Figure 4vi shows the topmost pending update being updated with an indirection to an error thunk. Figure 4vii shows the exception handler frame being popped off the stack in preparation for applying the handler to the error message `"a"`. Figures 4viii and 4ix show the exception handler `const 0` being applied to the error message `"a"`. Finally, figure 4x shows the final updatee being updated with the result of the exception handler leaving just the result `0` and the `STOP` frame on the stack.

4 A Problem and Two Solutions

In the previous section, we observed that the translation broke simple transformations. This section expands on the problem and describes two solutions: the first one is obvious but doesn't quite work, the second is less obvious but works.

The problem is that standard identities like the following hold in an untransformed program but do not hold in a transformed program. (This problem is easily verified using `a = error "a"` and `b = error "b"`.)

Arithmetic identities:

$$\begin{aligned} a + b &= b + a \\ a * b &= b * a \\ a * 1 &= a \end{aligned}$$

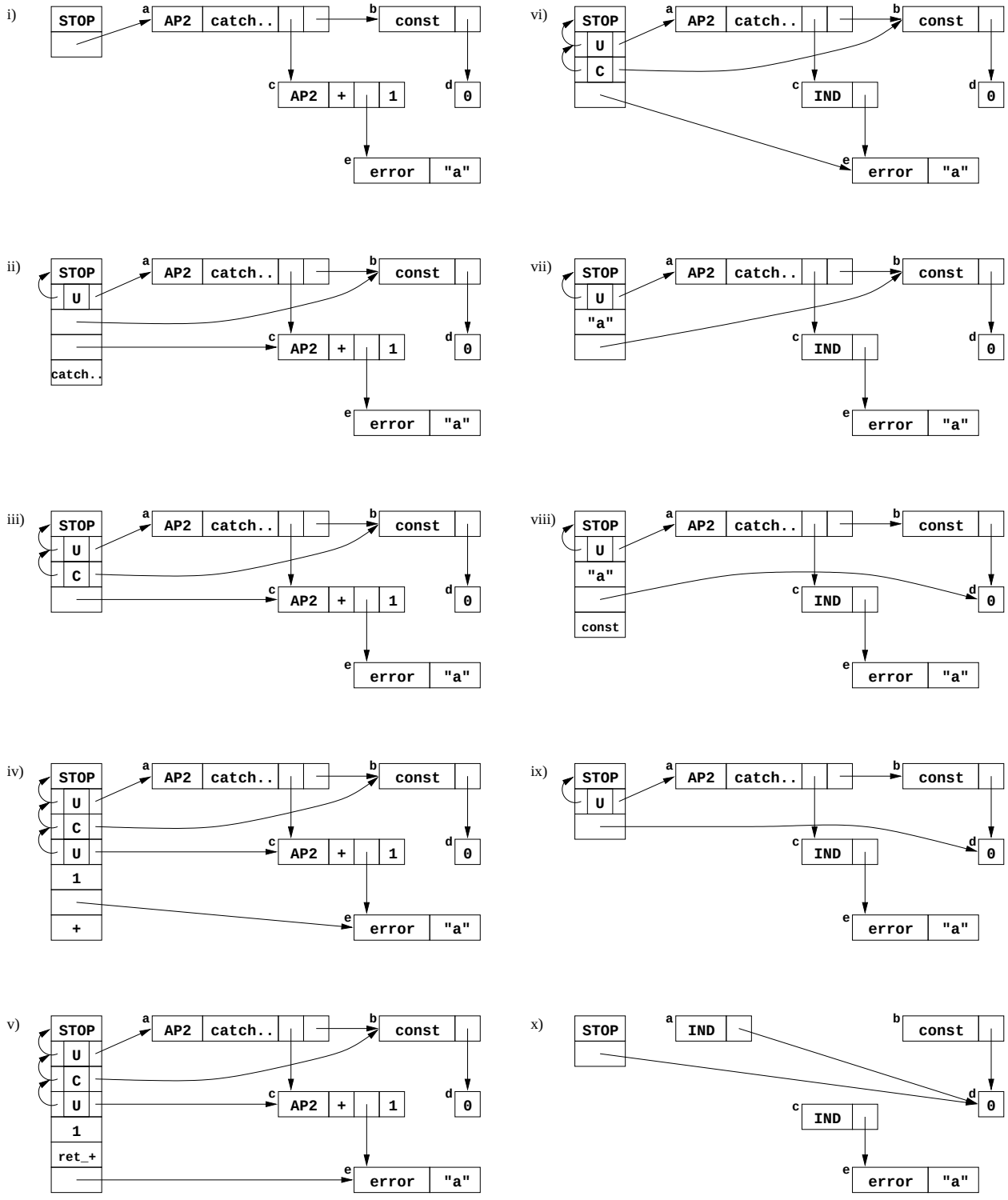

Figure 4: Catching Exceptions in the STG machine

Rearrangement of non-failing case:

$$\begin{aligned}&\texttt{case } a \texttt{ of } (a1, a2) \texttt{ -> case } b \texttt{ of } (b1, b2) \texttt{ -> } (a1 + b1, a2 + b2)\\ =\ &\texttt{case } b \texttt{ of } (b1, b2) \texttt{ -> case } a \texttt{ of } (a1, a2) \texttt{ -> } (a1 + b1, a2 + b2)\end{aligned}$$

The problem with these "reordering transformations" is that they change the dependencies within the program and so change which exception a program will raise. Since reliability (and, hence, exception handling) is essential for real world use, the obvious solution is to outlaw such transformations. This is unpalatable for several reasons:

1. One of the principal arguments for using lazy evaluation is that it supports transformations such as those above. This allows programs to be developed, explained or even proved correct by transforming an inefficient specification into an efficient implementation. Losing this ability to freely transform programs would throw away one of Haskell's main strengths.

2. Simply outlawing reordering transformations is not enough to make exception handling predictable. We also have to choose and clearly document the present order of evaluation in primitive operations such as `(+) :: Int -> Int -> Int`, the Haskell Prelude and standard libraries, and any non-standard libraries we may obtain from a third party.

   It seems unreasonable to insist on this level of documentation or to expect normal programmers to make use of it. Indeed, while we know of many libraries (for other languages) which list which exceptions a function may raise, none provide detailed documentation of exactly what circumstances cause each exception to be raised and how these exceptions are prioritised.

3. Optimising Haskell compilers use transformations like the above to automatically improve the performance of Haskell programs. If we forbid these transformations, or attach side conditions to their use, optimising compilers become much more limited in scope. In particular, they would have great trouble exploiting the effects of strictness analysis — the worker-wrapper transformation is no longer valid.

A second solution is to accept that programmers will not be able to reason about precisely which exception a program will raise and provide a new semantics (or new reasoning tools — there is little practical difference) which accepts a certain degree of *non-determinism*. We believe this is acceptable to programmers because programmers using languages which support exception handling seem willing to accept imprecise statements as to which exceptions a function might raise in return for more concise documentation and more implementation freedom.

The problem in making exception handling non-deterministic is in controlling the amount of non-determinism: if we allow too much non-determinism, the semantics will confuse programs that the programmer wishes to keep distinct; if we allow too little non-determinism,

then we must severely restrict exception handling and/or limit the set of transformations which we consider valid. To resolve this problem, we borrow an idea from Hughes and O'Donnell's seminal paper [5] on reasoning about non-deterministic functional programs. Their main idea was to separate deterministic parts of their programs from non-deterministic parts of their programs and to restrict non-determinism to the top-level of their programs.

Applying this idea to exception handling, we take care to keep (non-deterministic) exception handling code separate from normal (deterministic) code. This requires just one change to the implementation described in Section 3: we restrict `catchException` to the IO monad by giving it the more restrictive type

```
catchException :: IO a -> (String -> IO a) -> IO a
```

By limiting exception catching to the IO monad, we are able to use non-determinism in describing exception handling without the non-determinism contaminating the semantics of "pure" parts of the program. The next two sections describe how we use non-determinism when reasoning about exception handling.

4.1 Non-deterministic exceptions: a first attempt

For a long time, we thought the way to make exception handling non-deterministic was to take a second idea from Hughes and O'Donnell [5]:

1. They introduce a new abstract data type $\{\alpha\}$ whose elements are sets of values of type α but whose intended implementation is a single representative element chosen non-deterministically from the set it represents.

2. Non-deterministic expressions are clearly distinguished by their type: a non-deterministic `Int` expression is given type $\{$`Int`$\}$.

3. The operations on sets are carefully designed so that non-determinism cannot *leak out* into deterministic parts of the program. All operations on non-deterministic sets generate non-deterministic sets as results. In particular, they explicitly *do not* provide a function like

   ```
   choose :: {a} -> a
   ```

   Rather, non-deterministic programs (i.e. expressions of type $\{a\}$) can only be run at the "top-level" of the program.

Applying this idea to our semantics, we replace the error string with a *set* of error strings. That is, we change the exception type `E` described in Section 2 to

```
data E a = Errors {E String} | Value a
```

and changed the exception monad accordingly. In particular, we change primitive operations to return the union of all exceptional arguments instead of just returning the first exceptional argument.

This change restores the commutativity of integer addition but it does not restore the validity of all the other transformations. In particular, the *case of unfailing case* transformation given above still does not hold. With a little ingenuity and a lot of changes we were able to restore the validity of the *case of unfailing case* transformation as well, but the resulting system suffered from two fatal flaws:

1. it is hard to understand the resulting system; and
2. it is harder yet to imagine proving the resulting system correct w.r.t. a set of transformations.

We therefore reject this approach as being too hard to understand and too hard to validate whether it could account for all the non-determinism associated with a set of transformations.

4.2 Non-deterministic exceptions: a second attempt

The fundamental problem with the previous approach is that it does not directly mention the transformations that we want to preserve. So how are we meant to prove that they are preserved; and how are we meant to tweak the system if we want new transformations to hold? We fix this problem by making the transformations used in the compiler (and by library writers) explicit in the semantics.

Let us suppose that we have a relation $\bot\!\!\rightarrow$ which captures all the transformations that the compiler might apply (that is, $e1 \bot\!\!\rightarrow e2$ if the compiler might transform $e1$ into $e2$ during compilation). Then the set of values that an expression may return is $\mathcal{ND}[\![e]\!]$.

$$\mathcal{ND}[\![e]\!] = \{\mathcal{D}[\![e'^{\dagger}]\!] \mid e \bot\!\!\rightarrow^* e'\}$$

where $\mathcal{D}[\![e]\!]$ is the normal (deterministic!) value of e and $\bot\!\!\rightarrow^*$ is the reflexive, transitive closure of $\bot\!\!\rightarrow$. (The application of the monad translation $\dagger$ to the transformed expression e' reflects the fact that we implement the monad translation in our abstract machine and so it is applied *after* the compiler has done its job.)

If $\mathcal{D}[\![e]\!] \neq \bot$, then $\mathcal{ND}[\![e]\!]$ will, of course, contain a single value (assuming that $\bot\!\!\rightarrow$ respects the Haskell semantics). But if $\mathcal{D}[\![e]\!] = \bot$, then $\mathcal{ND}[\![e]\!]$ may contain multiple values depending on e and $\bot\!\!\rightarrow$. To see how the choice of $\bot\!\!\rightarrow$ affects the semantics, we consider three possible choices of transformation.

1. If the compiler does no optimisation, then $\bot\!\!\rightarrow$ is the identity relation and $\mathcal{ND}[\![e]\!]$ reduces to

   $$\mathcal{ND}[\![e]\!] = \{\mathcal{D}[\![e^{\dagger}]\!]\}$$

   We can reason exactly about what exceptions will be raised but we have to be careful when transforming programs.

2. At the other extreme, if we have no idea what transformations the compiler (or library writers) perform, we have to assume they perform any valid transformation. That is, $e1 \perp\!\!\!\rightarrow e2$ iff $\mathcal{D}[\![e1]\!] = \mathcal{D}[\![e2]\!]$. This is always a safe choice, but it includes such dubious transformations as:

$$\texttt{error "a"} \perp\!\!\!\rightarrow \texttt{error "b"}$$
$$\texttt{error "a"} \perp\!\!\!\rightarrow \texttt{let x = x in x}$$

   which real compilers are unlikely to use.

3. Finally, if we know that the compiler performs (only) the unfailing case of case transformation given earlier, we choose $\perp\!\!\!\rightarrow$ accordingly and we have

$$\begin{aligned}&\mathcal{NDE}[\![\texttt{case } a \texttt{ of } (a1, a2) \texttt{ -> case } b \texttt{ of } (b1, b2) \texttt{ -> } (a1 + b1, a2 + b2)]\!]\\ =\ &\mathcal{NDE}[\![\texttt{case } b \texttt{ of } (b1, b2) \texttt{ -> case } a \texttt{ of } (a1, a2) \texttt{ -> } (a1 + b1, a2 + b2)]\!]\end{aligned}$$

The idea then is to choose a relation $\perp\!\!\!\rightarrow$ which includes the transformations that the compiler and library writers typically use but excludes those that are valid but unlikely, such as changing error messages or replacing error messages with infinite loops.

There is just one fly in the ointment: many standard transformations allow a program which raises an error to be transformed into a program which does not terminate and so we are forced to confuse non-termination with raising an exception in our semantics. This is not particularily satisfactory but it seems to be the best we can do — and it can be dealt with by adding facilities to catch interrupts or timeouts as discussed in the next section.

5 Unification

The development in the previous sections ignored the interaction between exception handling and two other exception-like features of Haskell. This section describes these features and suggests a unified design which combines all three. Only the first has been implemented so far.

Haskell 1.4 [8] introduced a restricted form of exception handling which was labelled "error catching". This was a very conservative design which restricted both *raising* and *catching* of exceptions to the IO monad. One can view our exception handling features as an extension of "error catching" in which exceptions can be raised outside of the IO monad.

The "error catching" operations provided in Haskell 1.4 are as follows:

```
catch :: IO a -> (IOError -> IO a) -> IO a
fail  :: IOError -> IO a
```

In addition, many input/output operations in the IO monad call `fail` in response to error situations in the execution environment. For example, `writeFile` "fails" if the named file does not exist or is not writable.

To write robust programs, one must catch both Haskell 1.4 errors and our exceptions by writing something like:

```
catchException (catch e h1) h2
```

If we assume that most programmers will want to catch both kinds of exceptions, it makes sense to combine `catch` and `catchException` into a single operation which catches either kind of error. The resulting system looks like this:

```
catch :: IO a -> (IOError -> IO a) -> IO a
fail  :: IOError -> IO a
raise :: IOError -> a
```

(We also need to extend the `IOError` data type; this is discussed later in this section.) Merging these operations doesn't just simplify life for the programmer, it also simplifies the implementation since our exception handling mechanism can be used to efficiently implement Haskell 1.4's error catching operations.

We recently extended the STG machine with an interrupt catching mechanism [12]. In a sequential Haskell system, we add this function

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

The semantics is as follows: `catchInterrupt e h` executes `e`; if `e` returns a value without being interrupted, `catchInterrupt e h` returns the value returned by `e`; if an interrupt occurs while executing `e`, then `h` is executed.

Again, programmers are likely to want to catch both exceptions and interrupts and so we extend `catch` and `IOError` accordingly. There is just one subtlety: when propagating exceptions, we overwrite pending updatees with error values; when interrupting programs, we overwrite pending updates with reverted blackholes (this is the main subject of our other paper [12]). The reason for this difference is that if an *internal* exception is raised when executing an expression `e`, that expression will always raise an exception but if an *external* exception (such as an interrupt) is raised when executing `e`, it is entirely possible that no external exception would be raised the next time `e` is evaluated.

Finally, Haskell 1.4's `IOError` type needs to modified to let us encode errors, internal exceptions and external exceptions to the programmer. We have not explored this change as yet because it is not yet clear what programmers will want to do with `IOError`s. If all they want to do is print them on the screen, it is sufficient to provide a function to convert `IOError`s to `String`s; if they want to detect distinct exceptions and respond to them in different ways, will they want to make a clear distinction between errors, internal exceptions and external exceptions or would such a distinction merely get in the programmer's way?

6 Related Work

There have been three previous attempts to add exception handling to lazy functional languages. Gerald [10] was an early attempt to add exception handling to a lazy language — but it has no clear semantics and seems to be limited to untyped languages. Wadler's exception monad and call by name translation [13] is semantically sound (indeed, it is the basis for our semantics!) and requires no language extensions but, as we discussed in Section 2, it is tedious to apply and renders programs almost unreadable. Dornan and Hammond [1, 2] proposed the same semantics that we describe in Section 2, implemented their proposal and proved that the semantics is sound (confluent and consistent). The primary difference between their work and ours is our observation that soundness is not sufficient: adding exception handling breaks a large number of transformations. Our solution is to limit exception catching to the IO monad (where less transformations are valid) and to use non-determinism to describe the semantics of programs that use exception handling.

Finally, Henderson [4] independently proposed using Hughes and O'Donnell's non-deterministic sets when catching exceptions. A key difference is that instead of providing `catchException` in the `IO` monad, he provides the following function.

```
ndset_catch :: a -> Either {String} a
```

The problem with this (more flexible) proposal is that it allows non-determinism to contaminate the pure parts of the system with non-determinism as well: we cannot predict whether this pure expression will terminate or not.

```
let { a = a; b = error "b" } in (seq (ndset_catch (a+b)) "Mystery")
```

7 Discussion

It is relatively straightforward to add exception handling to a Haskell implementation; it is much harder to design a language extension which preserves the essential properties of a lazy language. This paper describes how to do both. The implementation is simple, efficient and obvious; the design is rather subtle and requires some care to produce a design which balances pragmatic concerns (we have to be able to catch exceptions) with more theoretical concerns (we have to be able to reason about our programs).

Acknowledgments: This work was carried out while working with Simon Peyton Jones, Simon Marlow and Sigbjorn Finne (all at Glasgow University) to reimplement the STG machine. We benefited from many conversations with them and from being able to implement our ideas in a state of the art Haskell implementation. Thanks too, to Paul Hudak and John Peterson (both at Yale) for comments on this paper and to Paul especially for the Dark Shadows conversation which inspired section 4.2.

References

[1] C. Dornan and K. Hammond. Exception handling in lazy functional languages. Research Report CSC 90/R5, Glasgow University, Department of Computing Science, January 1990.

[2] K. Hammond. Exception handling in a parallel functional language. Research Report CSC 89/R17, Glasgow University, Department of Computing Science, August 1989.

[3] K. Hammond and A. Gordon. Monadic I/O in Haskell 1.3. In *Proceedings of the 1995 Haskell Workshop*, pages 50–68, La Jolla, California, June 1995.

[4] F. Henderson. electronic mail to the Haskell mailing list. June 1998.

[5] R. Hughes and J. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In K. Davis and R. Hughes, editors, *Glasgow Functional Programming Workshop*, Workshops in Computing, pages 308–328. Springer Verlag, 1989.

[6] S. P. Jones and P. Wadler. Imperative functional programming. In *20th POPL*, pages 71–84, Charleston, Jan 1993. ACM.

[7] S. Marlow and A. Reid. The Hugs-GHC libraries. Hugs compiler documentation, June 1998.

[8] J. Peterson and K. Hammond (editors). Report on the Programming Language Haskell 1.4, A Non-strict Purely Functional Language. Research Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, April 1997.

[9] S. Peyton Jones, T. Nordin, and A. Reid. Greencard: a foreign-language interface for Haskell. In *Proc Haskell Workshop*, Amsterdam, June 1997.

[10] A. Reeves, D. Harrison, A. Sinclair, and P. Williamson. Gerald: An exceptional lazy functional programming language. In K. Davis and R. Hughes, editors, *Glasgow Functional Programming Workshop*, Workshops in Computing, pages 371–390. Springer Verlag, 1989.

[11] A. Reid. Malloc pointers and stable pointers: Improving Haskell's foreign language interface. draft proceedings of Glasgow Functional Programming Workshop, July 1994.

[12] A. Reid. Putting the Spine back in the Spineless Tagless G-machine: an implementation of revertible blackholes. Submitted to IFL'98, August 1998.

[13] P. Wadler. Comprehending monads. In *Proc ACM Conference on Lisp and Functional Programming*, Nice, June 1990. ACM.