A semantics for imprecise exceptions

Simon Peyton Jones
Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

Alastair Reid
Yale University
reid-alastair@cs.yale.edu

Tony Hoare[α]
Cambridge University Computer Laboratory
carh@comlab.ox.ac.uk

Simon Marlow
Microsoft Research Ltd, Cambridge
t-simonm@microsoft.com

Fergus Henderson
The University of Melbourne
fjh@cs.mu.oz.au

Abstract

Some modern superscalar microprocessors provide only *imprecise exceptions*. That is, they do not guarantee to report the same exception that would be encountered by a straightforward sequential execution of the program. In exchange, they offer increased performance or decreased chip area (which amount to much the same thing).

This performance/precision tradeoff has not so far been much explored at the programming language level. In this paper we propose a design for imprecise exceptions in the lazy functional programming language Haskell. We discuss several designs, and conclude that imprecision is essential if the language is still to enjoy its current rich algebra of transformations. We sketch a precise semantics for the language extended with exceptions.

The paper shows how to extend Haskell with exceptions without crippling the language or its compilers. We do not yet have enough experience of using the new mechanism to know whether it strikes an appropriate balance between expressiveness and performance.

1 Introduction

All current programming languages that support exceptions take it for granted that the language definition should specify, for a given program, what exception, if any, is raised when the program is executed. That used to be the case in microprocessor architecture too, but it is no longer so. Some processors, notably the Alpha, provide so-called *imprecise exceptions*. These CPUs execute many instructions in parallel, and perhaps out of order; it follows that the first exception (divide-by-zero, say) that is encountered is not necessarily the first that would be encountered in simple sequential execution. One approach is to provide lots of hardware to sort the mess out, and maintain the programmer's illusion of a simple sequential execution engine; this is what the Pentium does. Another, taken by the Alpha, is to give a less precise indication of the whereabouts of the exception.

In this paper we explore this same idea at the level of the programming language. The compiler, or the programmer, might want to improve performance by changing the program's evaluation order. But changing the evaluation order may change which exception is encountered first. One solution is to ban such transformations, or to restrict them to evaluations that provably cannot raise exceptions [19]. The alternative we propose here is to trade precision for performance: permit richer transformations, and make the language semantics less precise with respect to which exception is raised.

Note that the use of imprecise exceptions at the programming language level is not due to the use of imprecise exceptions at the hardware level. (Indeed, the latter may well prove ephemeral.) Rather, both of these arise from the same motivation: permitting better optimization. It's quite possible to have imprecise exceptions at the programming language level but not at the hardware level, or vice versa. However, the use of imprecise exceptions at the programming language level may make it much easier for implementations to generate efficient code on hardware that has imprecise exceptions.

We make all this concrete by considering a particular programming language, Haskell, that currently lacks exceptions. Our contributions are as follows:

- We review and critique the folk-lore on exception-handling in a lazy language like Haskell (Section 2). Non-functional programmers may find the idea of exceptions-as-values, as opposed to exceptions-as-control-flow, interesting.
- We present a new design, based on *sets* of exceptions, to model imprecision about which exceptions can occur (Section 3).
- We sketch a semantics for the resulting language, using

[α] On study leave from the Oxford University Computing Laboratory

This paper appears in Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'99), Atlanta

two layers: a denotational semantics for pure expressions (including exception-raising ones), and an operational semantics "on top" that deals with exception handling, as well as input/output (Section 4).

- Informed by this semantics, we show that various extensions of the basic idea, such as resource-exhaustion interrupts, can readily be accommodated; while others, such as a "pure" exception handler, are more troublesome (Section 5).

There has been a small flurry of recent proposals and papers on exception-handling in Haskell [3, 13, 12]. The distinctive feature of this paper is its focus on the semantics of the resulting language. The trick lies in getting the nice features of exceptions (efficiency, implicit propagation, and the like) without throwing the baby out with the bath-water and crippling the language design.

Those less interested in functional programming *per se* may nevertheless find interesting our development of the (old) idea of exceptions-as-values, and the trade-off between precision and performance.

2 The status quo ante

Haskell has managed without exceptions for a long time, so it is natural to ask whether they are either necessary or appropriate. We briefly explore this question, as a way of setting the scene for the rest of the paper.

Before we begin, it is worth identifying three different ways in which exceptions are typically used in languages that support them:

Disaster recovery uses an exception to signal a (hopefully rare) error condition, such as division by zero or an assertion failure. In a language like ML or Haskell we may add pattern-match failure, when a function is applied to a value for which it does not have a defining equation (e.g. `head` of the empty list). The programmer can usually also raise an exception, using a primitive such as `raise`.

The exception handler typically catches exceptions from a large chunk of code, and performs some kind of recovery action.

Exception handling used in this way provides a degree of modularity: one part of a system can protect itself against failure in another part of the system.

Alternative return. Exceptions are sometimes used as an alternative way to return a value from a function, where no error condition is necessarily implied. An example might be looking up a key in a finite map: it's not necessarily an error if the key isn't in the map, but in languages that support exceptions it's not unusual to see them used in this way.

The exception handler typically catches exceptions from a relatively circumscribed chunk of code, and serves mainly as an alternative continuation for a call.

Asynchronous events. In some languages, an asynchronous external event, such as the programmer typing "`^C`" or a timeout, are reflected into the programmer's model as an exception. We call such things *asynchronous exceptions*, to distinguish them from the two previous categories, which are both *synchronous exceptions.*

2.1 Exceptions as values

No lazy functional programming language has so far supported exceptions, for two apparently persuasive reasons.

Firstly, *lazy evaluation scrambles control flow.* Evaluation is demand-driven; that is, an expression is evaluated only when its value is required [14]. As a result, programs don't have a readily-predictable control flow; the only productive way to think about an expression is to consider the *value it computes*, not the *way in which the value is computed.* Since exceptions are typically explained in terms of changes in control flow, exceptions and lazy evaluation do not appear very compatible.

Secondly, *exceptions can be explicitly encoded in values*, in the existing language, so perhaps exceptions are in any case unnecessary. For example, consider a function, `f`, that takes an integer argument, and either returns an integer or raises an exception. We can encode it in Haskell thus:

```
data ExVal a = OK a
             | Bad Exception

f :: Int -> ExVal Int
f x = ...defn of f...
```

The `data` declaration says that a value of type `ExVal t` is either of the form `(Bad ex)`, where `ex` has type `Exception`, or is of the form `(OK val)`, where `val` has type `t`. The type signature of `f` declares that `f` returns a result of type `ExVal Int`; that is, either an `Int` or an exception value. In short, *the exception is encoded into the value returned by* `f`.

Any consumer of `f`'s result is forced, willy nilly, to first perform a case analysis on it:

```
case (f 3) of
  OK val -> ...normal case...
  Bad ex -> ...handle exception...
```

There are good things about this approach: no extension to the language is necessary; the type of a function makes it clear whether it can raise an exception; and the type system makes it impossible to forget to handle an exception.

The idea of exceptions as values is very old [10, 18]. Subsequently it was realised that the exception type constructor, `ExVal`, forms a *monad* [6, 9]. Rather than having lots of *ad hoc* pattern matches on `OK` and `Bad`, standard monadic machinery such as Haskell's `do` notation, can hide away much of the plumbing.

2.2 Inadequacies of exceptions as values

Encoding exceptions explicitly in an un-modified language works beautifully for the alternative-return usage of exceptions, but badly for the disaster-recovery use, and not at all for asynchronous events. There are several distinct problems:

- *Increased strictness.* When adding exception handling to a lazy program, it is very easy to accidentally make the program strict, by testing a function argument for errors when it is *passed* instead of when it is *used.*
- *Excessive clutter.* The principal feature of an exception mechanism is that exceptions propagate implicitly, without requiring extra clutter in the code between the place the exception is raised and where it is handled. In stark contrast, the explicit-encoding approach forces all the intermediate code to deal explicitly (or monadically) with exceptional values. The resulting clutter is absolutely intolerable for those situations where exceptions are used to signal disaster, because in these cases propagation is almost always required. For example, where we would originally have written:

  ```
  (f x) + (g y)
  ```

  we are now forced to write[1]:

  ```
  case (f x) of
    Bad ex -> Bad ex
    OK  xv -> case (g y) of
                Bad ex -> Bad ex
                OK  yv -> OK (xv+yv)
  ```

  These strictures do not apply where exceptions are used as an alternative return mechanism. In this case, the approach works beautifully because propagation isn't nearly so important.
- *Built-in exceptions are un-catchable.* In Haskell, all the causes of failure recognised by the language itself (such as divide by zero, and pattern-match failure) are treated semantically as bottom ($\perp$), and are treated in practice by bringing the program to a halt. Haskell allows the program to trigger a similar failure by calling the standard function `error`, whose type is:

  ```
  error :: String -> a
  ```

  So, evaluating the call `(error "Urk")` halts execution, printing "`Urk`" on standard error. The language offers no way to catch and recover from any of these (synchronous) events. This is a serious problem when writing programs composed out of large pieces over which one has little control; there is just no way to recover from failure in any sub-component.
- *Loss of modularity and code re-use,* especially for higher-order functions. For example, a sorting function that takes a comparison function as an argument would need to be modified to be used with an exception-raising comparison function.

[1] The monadic version is nearly as bad.

- *Poor efficiency.* Exceptions should cost very little if they don't actually occur. Alas, an explicit encoding into Haskell values forces a test-and-propagate at every call site, with a substantial cost in code size and speed.
- *Loss of transformations.* Programs written in a monadic style have many fewer transformations than their pure counterparts. We elaborate on this problem in Section 3.
- *No asynchronous exceptions.* Asynchronous exceptions, by their nature, have nothing to do with the value of the unfortunate expression that happens to be under evaluation when the external event occurs. Since they arise from external sources, they clearly cannot be dealt with as an explicitly-encoded value.

2.3 Goals

With these thoughts in mind, we have the following goals:

- Haskell programs that don't invoke exceptions should have unchanged semantics (no clutter), and run with unchanged efficiency.
- All transformations that are valid for ordinary Haskell programs should be valid for the language extended with exceptions. It turns out that we do not quite achieve this goal, for good reasons (Section 4.5).
- It should be possible to reason about which exceptions a program might raise. For example, we might hope to be able to prove that non-recursive programs will terminate, and programs that don't use arithmetic can't raise division by zero.
- In so far as non-determinism arises, it should be possible for the programmer to confine the non-determinism to a clearly-delineated part of the program.

These properties may seem obvious, but they are a little tricky to achieve. In existing languages that support exceptions, such as ML or Ada, the need to maintain the exception semantics noticeably constrains the valid set of transformations and optimisations that a programmer or compiler can perform. Compilers often attempt to infer the set of possible exceptions with a view to lifting these restrictions, but their power of inference is limited; for example, they must be pessimistic across module boundaries in the presence of separate compilation. We claim that our design retains almost all useful opportunities for transformation, using only the monadic type system built into Haskell. No separate effect analysis is required.

3 A new design

Adding exceptions to a lazy language, as opposed to encoding exceptions in the un-extended language, has received relatively little attention until recently. Dornan and Hammond discussed adding exceptions to the pure (non-I/O) part of a lazy language [2], and there has been a flurry of recent activity [3, 13, 12]. Drawing on this work, we propose a programming interface for an exceptions mechanism. This sets the scene for the core of our paper, the semantics for the resulting language.

3.1 The basic idea

As discussed in Section 2.1, our first design decision is more or less forced by the fact that Haskell is a lazy language: *exceptions are associated with data values, rather than with control flow.* This differs fundamentally from the standard approach to exceptions taken for imperative, or strict functional, languages, where exceptions are associated with control flow rather than with data flow. One place that exceptions-as-values does show up in the imperative world is the NaNs (not-a-number) and infinities of the IEEE floating point standard, where certain bit-patterns encode exceptional values, which are propagated by the floating point operations [20].

We extend this exceptions-as-values idea uniformly to values of *any* type. A value (of any type) is either a "normal" value, or it is an "exceptional" value. An "exceptional" value contains an exception, and we must say what that is. The data type `Exception` is the type of exceptions. It is a new algebraic data type, supplied as part of the Haskell Prelude, defined something like this:

```
data Exception = DivideByZero
               | Overflow
               | UserError String
               ...
```

One could imagine a simpler type (e.g. encoding an exception as an integer, or a string), or a richer type (e.g. a user-extensible data type, such as is provided by ML), but this one is a useful compromise for this paper. Nothing we say depends on the exact choice of constructors in the data type; hence the "`...`".

For each type a, the new, primitive function `raise` maps an `Exception` into an exceptional value of type a:

```
raise :: Exception -> a
```

Here, immediately, we see a difference from the explicit-encoding approach. *Every* type in the language contains exeptional values — previously only the type `ExVal t` had that possibility. We can also see that the same `Exception` type serves to represent an exception, regardless of the type into which the exception is embedded.

The previously-primitive function `error` can now readily be defined using `raise`:

```
error :: String -> a
error str = raise (UserError str)
```

Next, we need to be able to catch exceptions. The new, primitive function `getException` takes a value, and determines whether or not it is an exceptional value[2]:

```
getException :: a -> ExVal a
```

[2] We will see later that there is a fundamental problem with giving `getException` this type, but we defer discussion of this point to Section 3.5.

In effect, `getException` reifies the implicit presence or absence of an exception in its argument to an explicit discriminated union, represented by the new Prelude data type `ExVal`:

```
data ExVal a = OK a | Bad Exception
```

Here is an example of how `getException` might be used:

```
case getException (goop x) of
      OK  val -> normal_case val
      Bad exn -> recovery_case exn
```

Here, `getException` catches any exception raised while `goop` is evaluated, and presents the result as a value of type `ExVal`. The `case` expression scrutinises that value and takes appropriate action.

3.2 Propagation

The whole point of exceptions is, of course, that they propagate automatically. So integer addition, for example, should deliver an exceptional value if either of its arguments is an exceptional value.

In a lazy language, however, we have to re-examine our notion of propagation. In particular, an exceptional value might lurk inside an unevaluated function argument or data structure. For example, consider the `zipWith` function:

```
zipWith f []     []     = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs     ys     = error "Unequal lists"
```

A call to `zipWith` may return an exception value directly — for example, `zipWith (+) [] [1]`. A call to `zipWith` may also return a list with an exception value at the end — for example, `zipWith (+) [1] [1,2]`. Finally, it may deliver a list whose spine is fully defined, but some of whose elements are exceptional values — for example `zipWith (/) [1,2] [1,0]`.

To repeat: it is *values* not *calls* that may be exceptional, and exceptional values may, for example, hide inside lazy data structures. To be sure that a data structure contains no exceptional values one must force evaluation of all the elements of that structure (this can be done using Haskell's built-in `seq` function).

3.3 Implementation

One advantage of the story so far is that it is readily, and cheaply, implementable. We certainly do not want the space and time cost of explicitly tagging every value with an indication of whether it is "normal" or "exceptional". Fortunately, the standard exception-handling mechanisms from procedural languages work perfectly well:

- `getException` forces the evaluation of its argument to head normal form; before it begins this evaluation, it marks the evaluation stack in some way.
- `raise ex` simply trims the stack to the top-most `getException` mark, and returns `Bad ex` as the result

of `getException`.

- If the evaluation of the argument to `getException` completes without provoking a call to `raise`, then `getException` returns `OK val`, where `val` is the value of the argument.

Actually, matters are not quite as simple as we suggest here. In particular, trimming the stack after a call `(raise ex)` we must be careful to overwrite each thunk that is under evaluation with `(raise ex)`. That way, if the thunk is evaluated again, the same exception will be raised again, which is as it should be[3]. The details are described by [12], and need not concern us here.

The main point is that the efficiency of programs that do not invoke exceptions is unaffected. Indeed, the efficiency of any function that does not invoke exceptions explicitly is unaffected. Notice that an exceptional value *behaves* as a first class value, but it is never *explicitly represented* as such. When an exception occurs, instead of building a value that represents it, we look for the exception handler right away. The semantic model (exceptional values) is quite different from the implementation (evaluation stacks and stack trimming). The situation is similar to that with lazy evaluation: a value may *behave* as an infinite list, but it is certainly never *explicitly represented* as such.

3.4 A problem and its solution

There is a well-known difficulty with the approach we have just described: it invalidates many useful transformations. For example, integer addition should be commutative; that is, e_1+e_2 = e_2+e_1. But what are we to make of this expression?

```
getException ((1/0) + (error "Urk"))
```

Does it deliver `DivideByZero` or `UserError "Urk"`? Urk indeed! There are two well known ways to address this problem, and one more cunning one which we shall adopt:

- Fix the evaluation order, as part of the language semantics. For example, the semantics could state that `+` evaluates its first argument first, so that if its first argument is exceptional then that's the exception that is returned. This is the most common approach, adopted by (among others) ML, FL, and some proposals for Haskell [2]. It gives rise to a simple semantics, but has the Very Bad Feature that it invalidates many useful transformations — in particular, ones that alter the order of evaluation.

  This loss of transformations is a serious weakness. Williams, Aiken, and Wimmers give numerous examples of how the presence of exceptions can seriously weaken the transformation algebra of the (strict) language FL [19]. For a lazy language, the loss of transformations would be even more of a catastrophe. In particular, Haskell compilers perform strictness analysis to to turn call-by-need into call-by-value. This crucial transformation changes the evaluation order, by evaluating a function argument when the function is called, rather than when the argument is demanded.

  Rather than remove such transformations altogether, optimising compilers often perform some variant of effect analysis, to identify the common case where exceptions cannot occur (e.g. [8]). They use this information to enable the otherwise-invalid transformations. Williams, Aiken, and Wimmers describe a calculus for the language FL that expresses the absence of exceptions as a special program annotation; they can then give a precise characterisation of the transformation algebra of this augmented language [19].

  What all these approaches have in common is that useful transformations are disabled if the sub-expressions are not provably exception-free.

- Go non-deterministic. That is, declare that `+` makes a non-deterministic choice of which argument to evaluate first. Then the compiler is free to make that choice however it likes. Alas, this approach exposes non-determinism in the source language, which also invalidates useful laws. In particular, β reduction is not valid any more. For example, consider:

  ```
  let x = (1/0) + (error "Urk")
  in getException x == getException x
  ```

  As it stands, the value of this expression is presumably `True`. But if the two occurrences of `x` are each replaced by `x`'s right hand side, then the non-deterministic `+` might (in principle) make a different choice at its two occurrences, so the expression could be `False`. We count this too high a price to pay.

- The more cunning choice is to return *both* exceptions! That is, we redefine an exceptional value to contain a *set* of exceptions, instead of just one; and `+` takes the union of the exception sets of its two arguments. Now `(1/0) + (error "Urk")` returns an exceptional value including both `DivideByZero` and `UserError "Urk"`, and (semantically) it will do so regardless of the order in which `+` evaluates its arguments.

  The beauty of this approach is that *almost all transformations remain valid*, even in the presence of exceptions (Section 4.5 discusses the "almost"). No analysis required!

3.5 Fixing `getException`

The allegedly cunning choice may have fixed the commutativity of `+`, but, now that an exceptional value can contain a set of exceptions, we must revisit the question of what `getException` should do. There are two possibilities.

One alternative is for `getException` to return the complete set of exceptions (if any) in its argument value. This would be an absolute disaster from an implementation point of view! It would mean that the implementation would really have to maintain a set of exceptions; if the first argument to `+` failed, then the second would have to be evaluated anyway so that any exceptions in it could be gathered up.

[3] Real implementations overwrite a thunk with a "black hole" when its evaluation is begun to avoid a celebrated space leak [5]. That is why, when an exception causes their evaluation to be abandoned, they must be overwritten with something more informative.

The cunning choice is only cunning because there is another alternative: `getException` can choose just one member of the set of exceptions to return. Of course, that simply exposes the non-determinism again, but we can employ a now-standard trick [1]: put `getException` in the `IO` monad. Thus, we give `getException` the following type:

```
getException :: a -> IO (ExVal a)
```

To make sense of this new definition, we digress briefly to introduce Haskell's `IO` monad. In Haskell, a value of type `IO t` is a *computation* that might perform some input/output, before eventually returning a value of type `t`. A value of type `IO t` is a first-class value — it can be passed as an arguments, stored in a data structure — and evaluating it has no side effects. Only when it is *performed* does it have an effect. An entire Haskell program is a single value of type `IO ()`; to run the program is to perform the specified computation. For example, here is a complete Haskell program that gets one character from standard input and echoes it to standard output[4]:

```
main :: IO ()
main = getChar        >>= (\ch ->
       putChar ch     >>= (\() ->
       return ()
       ))
```

The types of the various functions involved are as follows:

```
(>>=)   :: IO a -> (a -> IO b) -> IO b
return  :: a -> IO a
getChar :: IO Char
putChar :: Char -> IO ()
```

The combinators `>>=` glues together two `IO` computations in sequence, passing the result from the first to the second. `return` does no input/output, simply returning its argument. `getChar` gets a character from the standard input and returns it; `putChar` does the reverse. When `main` is performed, it performs `getChar`, reading a character from standard input, and then performs the computation obtained by applying the `\ch -> ...` abstraction to the character, in this case `putChar ch`. A more complete discussion of monadic I/O can be found in [17].

Now we return to the type of `getException`. By giving it an `IO` type we allow `getException` to perform input/output. Hence, when choosing which of the exceptions in the set to choose, `getException` is free (although absolutely not required) to consult some external oracle (the FT Share Index, say). Each call to `getException` can make a different choice; the same call to `getException` in different runs of the same program can make a different choice; and so on.

Beta reduction remains valid. For example the meaning of:

```
let
    x = (1/0) + error "Urk"
in
getException x >>=   (\v1 ->
getException x >>=   (\v2 ->
return (v1==v2)))
```

is unaffected if both occurrences of `x` are replaced by `x`'s right hand side, thus:

[4] The "\" is Haskell's notation for λ

```
getException ((1/0) + error "Urk") >>= (\v1 ->
getException ((1/0) + error "Urk") >>= (\v2 ->
return (v1==v2)))
```

Why? Because whether or not this substitution is made, `getException` will be performed twice, making an independent non-deterministic choice each time. Like any `IO` computation, `(getException e)` can be shared, and even evaluated, without actually performing the nondeterministic choice. That only happens when the computation is *performed.*

The really nice thing about this approach is that the stack-trimming implementation does not have to change. *The set of exceptions associated with an exceptional value is represented by a single member*, namely the exception that happens to be encountered first. `getException` works just as before: mark the evaluation stack, and evaluate its argument. Successive runs of a program, using the same compiler optimisation level, will in practice give the same behaviour; but if the program is recompiled with different optimisation settings, then indeed the order of evaluation might change, so a different exception might be encountered first, and hence the exception returned by `getException` might change.

The idea of using a single representative to stand for a set of values, from which a non-deterministic choice is made, is based on an old paper by Hughes and O'Donnell [15]. Our contribution is to apply this idea in the setting of exception handling. The key observation is that non-determinism in the *exceptions* can be kept separate from non-determinism in the normal *values* of a program.

4 Semantics

So far we have reasoned informally. In this section we give a precise semantics to (a fragment of) Haskell augmented with exceptions. Here are two difficulties.

- Consider

  ```
  loop + error "Urk"
  ```

  Here, `loop` is any expression whose evaluation diverges. It might be declared like this:

  ```
  loop = f True
       where
         f x = f (not x)
  ```

  So, does `(loop + error "Urk")` loop forever, or does it return an exceptional value? Answer: it all depends on the evaluation order of `+`. As is often the case, bottom muddies the waters.

- Is the following equation true?

  ```
    case x of
      (a,b) -> case y of
                 (p,q) -> e
  =
    case y of
      (p,q) -> case x of
                 (a,b) -> e
  ```

In Haskell the answer is "yes"; since we are going to evaluate both `x` and `y`, it doesn't matter which order we evaluate them in. Indeed, the whole point of strictness analysis is to figure out which things are sure to be evaluated in the end, so that they can be evaluated in advance [7]. But if `x` and `y` are both bound to exceptional values, then the order of the `case`s clearly determines which exception will be encountered. Unlike the `+` case, it is far from obvious how to combine the exceptional value sets for `x` and `y`: in general the right hand side of a `case` alternative might depend on the variables bound in the pattern, and it would be unpleasant for the semantics to depend on that.

The rest of this section gives a denotational semantics for Haskell extended with exceptions, that addresses both of these problems. We solve the first by identifying $\perp$ with the set of all possible exceptions; we solve the latter by (semantically) evaluating the `case` alternatives in "exception-finding mode".

4.1 Domains

First we describe the domain $[\![\tau]\!]$ that is associated with each Haskell type τ. We use a rather standard monadic translation, for a monad $\mathcal{M}$, defined thus:

$$\begin{array}{rcl} \mathcal{M}\ t & = & t_\perp + \mathcal{P}(\mathcal{E})_\perp \\ \mathcal{E} & = & \{\texttt{DivideByZero}, \texttt{Overflow}, \texttt{UserError}, \ldots\} \end{array}$$

The "+" in this equation is coalesced sum; that is, the bottom element of $[\![\tau]\!]_\perp$ is coalesced with the bottom element of $\mathcal{P}(\mathcal{E})_\perp$. The set $\mathcal{E}$ is the set of all the possible synchronous exceptions; to simplify the semantics we neglect the `String` argument to `UserError`. $\mathcal{P}(\mathcal{E})$ is the lattice of all subsets of $\mathcal{E}$, under the ordering

$$s_1 \sqsubseteq s_2 \quad \equiv \quad s_1 \supseteq s_2$$

That is, the bottom element is the set $\mathcal{E}$, and the top element is the empty set. This corresponds to the idea that the fewer exceptions that are in the exceptional value, the more information the value contains. The least informative value contains all exceptions. This entire lattice is lifted, by adding an extra bottom element, which we also identify with a set of exceptions:

$$\perp \quad = \quad \mathcal{E} \cup \{\texttt{NonTermination}\}$$

At first we distinguished $\perp$ from the set of all exceptions, but that turns out not to work. Instead, we identify $\perp$ with the set of all exceptions, adding one new constructor, `NonTermination`, to the `Exception` type:

```
data Exception = ...    -- (as before)
               | NonTermination
```

This construction of $\mathcal{P}(\mathcal{E})_\perp$ is a very standard semantic coding trick; it is closely analogous to a canonical representation of the Smyth powerdomain over a flat domain, given by [11].

Here is an alternative, and perhaps more perspicuous, way to define $\mathcal{M}$, in which we tag "normal" values with Ok, and "exceptional" values (including $\perp$) with Bad:

$$\begin{array}{rcl} \mathcal{M}\ t & = & \{Ok\ v \mid v \in t\} \cup \\ & & \{Bad\ s \mid s \subseteq \mathcal{E}\} \cup \\ & & \{Bad\ (\mathcal{E} \cup \{\texttt{NonTermination}\})\} \end{array}$$

One might wonder what sort of a value $Bad\ \{\}$ is: what is an exceptional value containing the empty set of exceptions? Indeed, such a value cannot be the denotation of any term, but we will see shortly that it is nevertheless a very useful value for defining the semantics of `case` and for reasoning about it (Section 4.3).

Now that we have constructed the exception monad, we can translate Haskell types into domains in the usual way:

$$\begin{array}{rcl} [\![\texttt{Int}]\!] & = & \mathcal{M}\ \mathcal{Z} \\ [\![\tau_1\texttt{->}\tau_2]\!] & = & \mathcal{M}\ ([\![\tau_1]\!] \rightarrow [\![\tau_2]\!]) \\ [\![\texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}]\!] & = & \mathcal{M}\ ([\![\tau_1]\!] \times [\![\tau_2]\!]) \\ & \ldots\text{etc}\ldots & \end{array}$$

We refrain from giving the complete encoding for arbitrary recursive data types, which is complicated. The point is that we simply replace the normal Haskell monad, namely lifting, with our new monad $\mathcal{M}$.

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & k & \text{constant} \\ & | & e_1\ e_2 & \text{application} \\ & | & \lambda x.e & \text{abstraction} \\ & | & C\ e_1 \ldots e_n & \text{constructors} \\ & | & \texttt{case}\ e\ \texttt{of}\ \texttt{\{}\ldots p_i\texttt{->}r_i\texttt{;}\ \ldots\texttt{\}} & \text{matching} \\ & | & \texttt{raise}\ e & \text{raise exception} \\ & | & e_1\ \texttt{+}\ e_2 & \text{primitives} \\ & | & \texttt{fix}\ e & \text{fixpoint} \\ \\ p & ::= & C\ x_1 \ldots x_n & \text{pattern} \end{array}$$

Figure 1: Syntax of a tiny language

4.2 Combinators

Next, we must give the denotation, or meaning, of each form of language expression. Figure 1 gives the syntax of the small language we treat here. The denotation of an expression e in an environment ρ is written $[\![e]\!]\rho$.

We start with `+`:

$$\begin{array}{ll} [\![e_1\ \texttt{+}\ e_2]\!]\rho & \\ \quad = v_1 \oplus v_2 & \text{if} \quad Ok\ v_1 = [\![e_1]\!]\rho \\ & \text{and} \quad Ok\ v_2 = [\![e_2]\!]\rho \\ \quad = Bad\ (\mathcal{S}([\![e_1]\!]\rho) \cup \mathcal{S}([\![e_2]\!]\rho)) & \text{otherwise} \end{array}$$

The first equation is used if both arguments are normal values. The second is used if either argument is an exceptional value, in which case the exceptions from the two arguments are unioned. We use the auxiliary function $\mathcal{S}()$, which returns the empty set for a normal value, and the set of exceptions for an exceptional value:

$$\begin{array}{rcl} \mathcal{S}(Ok\ v) & = & \emptyset \\ \mathcal{S}(Bad\ s) & = & s \end{array}$$

The auxiliary function $\oplus$ simply does addition, checking for overflow:

$$\begin{array}{lcll} v_1 \oplus v_2 & = & Ok\ (v_1+v_2) & \text{if } -2^{31} \langle (v_1+v_2) \langle 2^{31} \\ & = & Bad\ \{\texttt{Overflow}\} & \text{otherwise} \end{array}$$

The definition of $[\![\texttt{+}]\!]$ is monotonic with respect to $\sqsubseteq$, as it must be. The fact that `+` is strict in both arguments is a consequence of the fact that $\bot$ is the set of all exceptions; a moment's thought should convince you that if either argument is $\bot$ then so is the result.

Next, we deal with `raise`:

$$\begin{array}{lcll} [\![\texttt{raise}\ e]\!]\rho & = & Bad\ s & \text{if } Bad\ s = [\![e]\!]\rho \\ & = & Bad\ \{C\} & \text{if } Ok\ C = [\![e]\!]\rho \end{array}$$

Thus equipped, we can now understand the semantics of the problematic expression given above:

```
loop + error "Urk"
```

Its meaning is the union of the set of all exceptions (which is the value of `loop`), and the singleton set `UserError "Urk"`, which is of course just $\bot$, the set of all exceptions.

The rules for function abstraction and application are:

$$\begin{array}{rcll} [\![\lambda x.e]\!]\rho & = & Ok\ (\lambda y.[\![e]\!]\rho[y/x]) & \\ [\![e_1\ e_2]\!]\rho & = & f\ ([\![e_2]\!]\rho) & \text{if } Ok\ f = [\![e_1]\!]\rho \\ & = & Bad(s \cup \mathcal{S}([\![e_2]\!]\rho) & \text{if } Bad\ s = [\![e_1]\!]\rho \end{array}$$

A lambda abstraction is a normal value; that is $\lambda x.\bot \neq \bot$. The (more purist) identification of these two values is impossible to implement: how can `getException` distinguish $\lambda x.\bot$ from $\lambda x.v$, where $v \neq \bot$? Fortunately, in Haskell $\lambda x.\bot$ and $\bot$ are indeed distinct values.

Applying a normal function to a value is straightforward, but matters are more interesting if the function is an exceptional value. In this case *we must union its exception set with that of its argument*, because under some circumstances (notably if the function is strict) we might legitimately evaluate the argument first; if we neglected to union in the argument's exceptions, the semantics would not allow this standard optimisation. That is why we do not use the simpler definition:

$$\begin{array}{rcll} [\![e_1\ e_2]\!]\rho & = & f\ ([\![e_2]\!]\rho) & \text{if } Ok\ f = [\![e_1]\!]\rho \\ & = & Bad\ s & \text{if } Bad\ s = [\![e_1]\!]\rho \end{array}$$

We have traded transformations for precision. Notice, however, that we must *not* union in the argument's exceptions if the function is a normal value, or else we would lose β reduction; consider `(λx.3)(1/0)`

The rules for constants and constructor applications are simple; they both return normal values. Constructors are non-strict, and hence do not propagate exceptions in their arguments. Variables and fixpoints are also easy.

$$\begin{array}{rcl} [\![k]\!]\rho & = & Ok\ k \\ [\![C\ e_1 \ldots e_n]\!]\rho & = & Ok\ (C\ ([\![e_1]\!]\rho)\ldots([\![e_n]\!]\rho)) \\ [\![x]\!]\rho & = & \rho(x) \\ [\![\texttt{fix}\ e]\!]\rho & = & \bigsqcup_{k=0}^{\infty} ([\![e]\!]\rho)^k(\bot) \end{array}$$

4.3 `case` expressions

Haskell contains `case` expressions, so we must give them a semantics. Here is the slightly surprising rule:

$$\begin{array}{ll} [\![\texttt{case}\ e\ \texttt{of}\ \{p_i\texttt{->}r_i\}]\!]\rho & \\ \quad = [\![r_i]\!]\rho[v/p_i] & \text{if } Ok\ v = [\![e]\!]\rho \\ & \text{and } v \text{ matches } p_i \\ \quad = Bad\ (s \cup (\bigcup_i \mathcal{S}([\![r_i]\!]\rho[Bad\ \{\}/p_i]))) & \\ & \text{if } Bad\ s = [\![e]\!]\rho \end{array}$$

The first case is the usual one: if the case scrutinee evaluates to a "normal" value v, then select the appropriate case alternative. The notation is a little informal: $\rho[v/p_i]$ means the environment ρ with the free variables of the pattern p_i bound to the appropriate components of v.

The second equation is the interesting one. If the scrutinee turns out to be a set of exceptions (which, recall, includes $\bot$), the obvious thing to do is to return just that set — but doing so would invalidate the `case`-switching transformation. Intuitively, the semantics must explore all the ways in which the implementation might deliver an exception, so it must "evaluate" all the branches anyway, in "exception-finding mode". We model this by taking the denotations of all the right hand sides, binding each of the pattern-bound variables to the strange value $Bad\ \{\}$. Then we union together all the exception sets that result, along with the exception set from the scrutinee. The idea is exactly the same as in the special case of `+`, and function application: if the first argument of `+` raises an exception we still union in the exceptions from the second argument. Here, if the case scrutinee raises an exception, we still union in the exceptions from the alternatives.

Remember that there is no implication that an implementation will do anything other than return the first exception that happens to be encountered. The rather curious semantics is necesary, though, to validate transformations that change the order of evaluation, such as that given at the beginning of Section 4.

4.4 Semantics of `getException`

So far we have not mentioned `getException`. The semantics of operations in the `IO` monad, such as `getException`, may involve input/output or non-determinism. The most straightforward way of modelling these aspects is by giving an *operational* semantics for the `IO` layer, in contrast to the *denotational* semantics we have given for the purely-functional layer.

We give the operational semantics as follows. From a semantic point of view we regard `IO` as an algebraic data type with constructors `return`, `>>=`, `putChar`, `getChar`, `getException`. The behaviour of a program is the set of traces obtained from the following labelled transition system, which acts on the *denotation* of the program. One advantage of this presentation is that it scales to other extensions, such as adding concurrency to the language [16].

Here are the structural transition rules:

$$\frac{v_1 \rightarrow v_2}{(v_1 \text{ >>= } k) \rightarrow (v_2 \text{ >>=} k)}$$

$$((\texttt{return}\ v) \text{ >>= } k) \rightarrow (k\ v)$$

The first ensures that transitions can occur inside the first operand of the `>>=` constructor; the second explains that a `return` constructor just passes its value to the second argument of the enclosing `>>=`. The rules for input/output are now quite simple:

$$\begin{array}{rcl} \texttt{getChar} & \stackrel{?c}{\rightarrow} & \texttt{return}\ c \\ \texttt{putChar}\ c & \stackrel{!c}{\rightarrow} & \texttt{return}\ () \end{array}$$

The "$?c$" on top of the arrow indicates that the transition takes place by reading a character c from the environment; and inversely for "$!c$".

Now we can get to the semantics of exceptions. The rules are:

$$\begin{array}{rcl} \texttt{getException}\ (Ok\ v) & \rightarrow & \texttt{return}\ (\texttt{OK}\ (Ok\ v)) \\ \texttt{getException}\ (Bad\ s) & \rightarrow & \texttt{return}\ (\texttt{Bad}\ x) \\ & \text{if} & x \in s \\ \texttt{getException}\ (Bad\ s) & \rightarrow & \texttt{getException}\ (Bad\ s) \\ & \text{if} & \texttt{NonTermination} \in s \end{array}$$

If `getException` scrutinises a "normal" value, it just returns it, wrapped in an `OK` constructor.

For "exceptional" values, there are two choices: either

- pick an arbitrary member of the set of exceptions and return it, or
- if `NonTermination` is in the set of exceptions, then make a transition to the same state.

The transition rules for `getException` are deliberately non-deterministic. In particular, if the argument to getException is $\perp$, then `getException` may diverge, or it may return an arbitrary exception.

To execute a Haskell program, one performs the computation `main`, which has type `IO ()`. In the presence of exceptions, the value returned might now be $Bad\ x$, rather than $Ok\ ()$. This simply corresponds to an uncaught exception, which the implementation should report.

4.5 Transformations

Our overall goal is to add exceptions to Haskell without losing useful transformations. Yet it cannot be true that we lose *no* transformations. For example, in Haskell as it stands, the following equation holds:

```
error "This" = error "That"
```

Why? Because both are semantically equal to $\perp$. In our semantics this equality no longer holds — and rightly not! So our semantics correctly distinguishes some expressions that Haskell currently identifies.

Some transformations that are identities in Haskell become refinements in our new system. Consider:

$$\begin{array}{rcl} lhs & = & \texttt{(case e of \{ True -> f; False -> g \}) x} \\ rhs & = & \texttt{case e of \{ True -> (f x); False -> (g x) \}} \end{array}$$

Using `e` = `raise E`, `x` = `raise X`, and `f` = `g` = λ`v.1`, we get $[\![lhs]\!]\rho = Bad\ \{E, X\}$ but $[\![rhs]\!]\rho = Bad\ \{E\}$. Hence, $lhs \sqsubseteq rhs$, but not $lhs = rhs$. We argue that it is legitimate to perform a transformation that increases information — in this case, changing lhs to rhs reduces uncertainty about which exceptions can be raised.

We currently lack a systematic way to say which identities continue to hold, which turn into refinements, and which no longer hold. We conjecture that the lost laws deserve to be lost, and that optimising transformations are either identities or refinements. It would be interesting to try to formalise and prove this conjecture.

5 Variations on the theme

5.1 Asynchronous exceptions

All the exceptions we have discussed so far are synchronous exceptions (Section 2). If the evaluation of an expression yields a set of synchronous exceptions, then another evaluation of the same expression will yield the same set. But what about asynchronous exceptions, such as interrupts and resource-related failures (e.g. timeout, stack overflow, heap exhaustion)? They differ from synchronous exceptions in that they perhaps will not recur (at all) if the same program is run again. It is obviously inappropriate to regard such exceptions as part of the denotation of an expression.

Fortunately, they can fit in the same general framework. We have to enrich the `Exception` type with constructors indicating the cause of the exception. Then we simply add to `getException`'s abilities. Since `getException` is in the `IO` monad, it can easily say "if the evaluation of my argument goes on for too long, I will terminate evaluation and return `Bad TimeOut`", and similarly for interrupts and so on. We express this formally as follows:

$$\begin{array}{rcl} \texttt{getException}\ v & \stackrel{\clubsuit x}{\rightarrow} & \texttt{return}\ (\texttt{Bad}\ x) \\ & & \text{if } x \text{ is an asynchronous exception} \end{array}$$

The $\clubsuit x$ above the arrow indicates that the transition may take place only when an asynchronous event x is received by the evaluator. Notice that v might not be an exceptional value — it might be say, `42` — but if the event x is received, `getException` is nevertheless free to discard v and return the asynchronous exception instead. In the case of a keyboard interrupt, the event `ControlC` is injected; in the case of timeout, some presumed external monitoring system injects the event `TimeOut` if evaluation takes too long; and so on.

There is a fascinating wrinkle in the implementation of asynchronous exceptions: when trimming the stack, we must overwrite each thunk under evaluation with a kind of "resumable continuation", rather than a computation which raises the exception again. The details are in [13].

5.2 Detectable bottoms

There are some sorts of divergence that are detectable by a compiler or its runtime system. For example, suppose that `black` was declared like this:

```
black = black + 1
```

Here, `black` is readily detected as a so-called "black hole" by many graph reduction implementations. Under these circumstances, `getException black` is permitted, but not required, to return `Bad NonTermination` instead of going into a loop! Whether or not it does so is an implementation choice — perhaps implementations will compete on the skill with which they detect such errors.

5.3 Fictitious exceptions

There is actually a continuum between our semantics and the "fixed evaluation order" semantics, which fully determines which exception is raised. As one moves along the spectrum towards our proposal, more compiler transformations become valid — but there is a price to pay. That price is that the semantics becomes vaguer about which exceptions can be raised, and about when non-termination can occur. Our view is that we should optimise for the no-exception case, accepting that if something does go wrong in the program, then the semantics does not guarantee very precisely what exception will show up. An extreme, and slightly troubling, case is this:

```
getException loop
```

Since `loop` has value $\perp$, `getException` is, according to our semantics, justified in returning `Bad DivideByZero`, or some other quite fictitious exception — and in principle a compiler refinement might do the same.

We sought a way to give `loop` the denotation

$$Bad\ \{\texttt{NonTermination}\}$$

rather than (the less informative) $\perp$, but we know of no consistent way to do so. The modelling of non-termination to include all other behaviours is characteristic of the denotational semantics of non-determinism. It means that set inclusion gives a simple interpretation of program correctness, encompassing both safety and liveness properties. It ensures that recursion can be defined as the weakest fixed point of a monotonic function, and that this fixed point can be computed as the limit of a (set-wise) descending chain of approximations. But what is more important for our purposes is that *it gives maximal freedom to the compiler, by assuming that non-termination is never what the programmer intends*[5].

An operational semantics would model more precisely what happens, and hence would not suffer from the problem of fictitious exceptions. Arguably, for reasoning about divergent programs, the programmer should use an operational semantics anyway. Because, in the end, it seems unlikely that a compiler will gratuitously report a fictitious exception when the program gets into a loop, so this semantic technicality is unlikely to have practical consequences.

[5] Indeed, there are a number of situations in which it is useful to be able to assume that a value is not $\perp$. For example, if `v` is not $\perp$, then the following law holds:

```
case v of { True -> e; False -> e }   =   e
```

Our compiler has a flag `-fno-pedantic-bottoms` that enables such transformations, in exchange for the programmer undertaking the proof obligation that no sub-expression in the program has value $\perp$.

5.4 Pure functions on exceptional values

Is it possible to do anything with an exceptional value other than choose an exception from it with `getException`? Following [15], one possibility suggests itself as a new primitive function (i.e. one not definable with the primitives so far described):

```
mapException :: (Exception -> Exception) -> a -> a
```

Semantically, `mapException` applies its functional argument to each member of the set of exceptions (if any) in its second argument; it does nothing to normal values. From an implementation point of view, it applies the function to the sole representative (if any) of that set. Here's an example of using `mapException` to catch all exceptions in `e` and raise `UserError "Urk"` instead:

```
mapException (\x -> UserError "Urk") e
```

Notice that `mapException` does not need to be in the `IO` monad to preserve determinism. In short, `mapException` raises no new technical dificulties, although its usefulness and desirability might be debatable.

`mapException` maps one kind of exception to another, but it doesn't let us get from exceptions back into normal values. Is it possible to go further? Is it possible, for example, to ask "is this an exceptional value"?

```
isException :: a -> Bool
```

(It would be easy to define `isException` with a monadic type `a -> IO Bool`; the question is whether it can have a pure, non-monadic, type.) At first `isException` looks reasonable, because it hides just which exception is being raised — but it turns out to be rather problematic. What is the value of the following expression?

```
isException ((1/0) + loop)
```

If the compiler evaluates the first argument of `+` first, the result will be `True`; but if the compiler evaluates the second argument of `+` first, the computation will not terminate. Two different implementations have delivered two different values!

It is quite possible to give a perfectly respectable denotational semantics for `isException` — in fact there are two different such semantics that we might use, the "optimistic" one

$$\begin{array}{lcl} \texttt{isException}\ (Bad\ s) & = & \texttt{True} \\ \texttt{isException}\ (Ok\ v) & = & \texttt{False} \end{array}$$

or the "pessimistic" one

$$\begin{array}{lcll} \texttt{isException}\ (Bad\ s) & = & \perp & \text{if } \texttt{NonTermination} \in s \\ \texttt{isException}\ (Bad\ s) & = & \texttt{True} & \text{if } \texttt{NonTermination} \notin s \\ \texttt{isException}\ (Ok\ v) & = & \texttt{False} & \end{array}$$

The trouble is that neither of these semantics are efficiently implementable, because they require the implementation to detect nontermination. Consider our example

```
isException ((1/0) + loop)
```

An implementation that evaluates the arguments of `+` right-to-left would evaluate `loop` before `1/0`; hence, the call to `isException` would loop, i.e. evaluate to $\perp$, rather than returning `True` as the "optimistic" semantics requires. But conversely, an implementation that evaluates the arguments of `+` left-to-right would evaluate `1/0` before `loop`; hence, the call to `isException` would return `True`, rather than $\perp$ as the "pessimistic" semantics requires. Since we want implementations to be able to evaluate arguments in any order, neither the optimistic nor the pessimistic semantics will work.

There are a number of possible things we could say:

1. Because `isException` is unimplementable, it should be banned.

2. Programmers may use `isException`, but when they do so they undertake a proof obligation that its argument is not $\perp$. If this can be assumed, the implementation is in no difficulty (c.f. Section 5.3).

3. The denotational semantics for `isException` should be the pessimistic one; to make it implementable, the language semantics should be changed so that result of the program is defined to be any value that is the same as or *more defined* than the program's denotation. If the program yields $\perp$, then any value at all could be delivered.

   This alternative has the undesirable property that a program that goes into an infinite loop would be justified in returning an `IO` computation that (say) deleted your entire filestore.

4. The denotational semantics for `isException` should be the optimistic one; to make it implementable, the language semantics should be changed so that result of the program is defined to be any value that is the same as or *less defined* than the program's denotation. $\perp$ would *always* be a valid result.

   This alternative has the undesirable property that an implementation could, in theory, abort with an error message or fail to terminate for any program at all, including programs that do not use `isException`. Still, in comparison to the previous alternative, at least the failure mode is much less severe: the semantics would only allow the implementation would to loop or abort, not to perform arbitrary I/O operations.

The latter two options would both require a significant global change to Haskell's semantics, and even then, neither of them really captures the intended behaviour with sufficient precision. It would be possible to refine these approaches to give more precision, but only at the cost of some additional semantic complexity. Therefore we prefer the second option, renaming `isException` to `unsafeIsException` to highlight the proof obligation.

Other declarative languages, particularly logic programming languages such as Gödel and Mercury already make a distinction between the declarative (i.e. denotational) semantics and the operational semantics similar to that mentioned in the fourth possibility above [4]. In Mercury, for example, the operational semantics allows non-termination in some situations even though the declarative semantics specifies that the program should have a result other than non-termination. So if our proposal for Haskell were to be adopted to other languages for which the operational semantics is already incomplete (in the above sense) with respect to the declarative semantics, then a refinement of the fourth alternative might well be the best approach.

6 Other languages

We have described a design for incorporating exceptions into Haskell. In this section we briefly relate our design to that in other languages.

First, it is clear that our design is somewhat less expressive than that in other languages; we will take ML as a typical example. In ML it is posssible to completely encapsulate a function that makes use of exceptions: one can declare an exception locally, raise it, and handle it, all without this implementation becoming visible to the function's caller. In our design, one cannot handle an exception without using the `IO` monad. Furthermore, the `IO` monad is (by design) like a trap door: you cannot encapsulate an I/O performing computation inside a pure function — and rightly not!

Though we do not yet have much experience of using exceptions in Haskell, we speculate that the fact that `getException` is in the `IO` monad will not prove awkward in practice, for several reasons:

- Only exception *handling*, using `getException`, is affected. One can *raise* an exception without involving the `IO` monad at all.
- Most disaster-recovery exception handling is done near the top of the program, where all other input/output is in any case performed.
- Much local exception handling can be done by encoding exceptions as explicit values (Section 2.1).

No doubt there will remain situations where the lack of a "pure" `getException` will prove annoying. One alternative would be to provide an `unsafeGetException` (analogous to `unsafeIsException`; Section 5.4), with associated proof obligations for the programmer.

Second, the big payoff of our approach is that we lose no (useful) transformations compared to a guaranteed-exception-free program. Could the same technique be used in other languages, such as ML or Java? It is hard to see how it could apply directly; our approach depends crucially on distinguishing computations in the `IO` monad (whose transformations are restricted by the possibility of side effects and non-determinism) from purely-functional expressions (whose transformations are unrestricted).

Nevertheless, standard effect analyses for ML and Java seek to find which portions of the program cannot *raise* an exception, whereas in our system transformations are limited only for those parts of the program that *handle* exceptions. We speculate that an effect system that focused instead on the latter instead of the former might yield more scope for optimisation.

Our work does not directly address the question of how the exception-raising behaviour of a function should be manifested in its type. Java requires methods to declare which (checked) exceptions they may throw, but this approach does not seem to scale well to higher-order languages [8]. In our design, explicitly-encoded exceptions are certainly manifested in the function's type, but exceptions generated by `raise` are not.

7 Conclusion

As usual, implementation is ahead of theory: the Glasgow Haskell Compiler (4.0 and later) implements `raise` and `getException` just as described above. If nothing else, this reassures us that there are no hidden implementation traps. A useful practical outcome of writing this paper was a clear idea about what is, and what is not, semantically justifiable in the programming interface. For example, we originally implemented a version of `isException`, without fully understanding the impact on the semantics. We now know that this feature would require a significant liberalisation of Haskell's semantics, one that may not be acceptable to all Haskell programmers, so it should not be added without due consideration.

Incidentally, exceptions in the `IO` monad itself are also now handled in the same way, which makes the implementation of the `IO` monad very much more efficient, and very much less greedy on code space. Previously, every `>>=` operation had to test for, and propagate, exceptions.

We do not yet have much experience with using exceptions in Haskell. The proof of the pudding is in the eating. Bon appetit.

Acknowledgement. We gratefully acknowledge helpful feedback from Cedric Fournet, Corin Pitcher, Nick Benton, and the PLDI referees.

References

[1] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4:117–123, January 1994.

[2] C. Dornan and K. Hammond. Exception handling in lazy functional languages. Technical Report CS90/R5, Department of Computing Science, University of Glasgow, Jan 1990.

[3] F. Henderson. Non-deterministic exceptions. Electronic mail to the `haskell` mailing list, June 1998.

[4] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996. A more recent version is available via `<http://www.cs.mu.oz.au/mercury/information/documentation.html>`.

[5] R. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, Jan. 1992.

[6] E. Moggi. Computational lambda calculus and monads. In *IEEE Symposium on Logic in Computer Science*, June 1989.

[7] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc 4th International Symposium on Programming*, pages 269–281. Springer Verlag LNCS 83, 1981.

[8] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proc Principles of Programming Languages (POPL'99), San Antonio*, Jan 1999.

[9] PL Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[10] PL Wadler. How to replace failure by a list of successes. In *Proc Functional Programming Languages and Computer Architecture, La Jolla*. ACM, June 1995.

[11] G. Plotkin. Domains. Technical report, Department of Computer Science, University of Edinburgh, 1983.

[12] A. Reid. Handling Exceptions in Haskell. Research Report YALEU/DCS/RR–1175, Yale University, August 1998.

[13] A. Reid. Putting the spine back in the Spineless Tagless G-machine: an implementation of resumable black holes. In *Proc Implementation of Functional Languages Workshop 1998 (IFL'98)*. Springer Verlag LNCS (to appear 1999), Sept 1998.

[14] RJM Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989.

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

[16] SL Peyton Jones, AJ Gordon, and SO Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, pages 295–308. ACM, Jan 1996.

[17] SL Peyton Jones and PL Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston*, pages 71–84. ACM, Jan 1993.

[18] J. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–43, Jul 1990.

[19] J. William, A. Aiken, and E. Wimmers. Program transformation in the presence of errors. In *Proc Principles of Programming Languages (POPL'90), San Francisco*, pages 210–217. ACM, Jan 1990.

[20] WJ Cody *et al.* A proposed radix- and word-length independent standard for floating point arithmetic. *IEEE Micro*, 4(4):86–100, Aug. 1984.