

1st Year Transfer Dissertation

Joel Wright
Foundations of Programming Group
University of Nottingham

January 14, 2003

This document is a review of the first year of my PhD, working on exceptions and concurrency in Haskell, funded jointly by the University of Nottingham and Microsoft Research Ltd, Cambridge. I shall discuss background information on topics relating to exceptions and concurrency in Haskell as well as relevant papers. I shall also describe the work that I have undertaken in the last year, and outline the research I intend to undertake for my thesis.

1 Background

In order to understand and explore concurrency and exceptions in Haskell it was necessary to do background research into a number of topics. The semantics of Concurrent Haskell is defined as an outer monadic transition semantics “wrapped around” an inner denotational semantics which describes the behaviour of pure terms. Simple exceptions may also be modelled in terms of a monad, so research into semantics and monads was required. Concurrency and exception handling in general were also clearly important, and allowed me to better understand how these concepts are tackled in Haskell. All the above topics are described in this section, as well as a tool for testing Haskell programs.

1.1 Semantics

The semantics of a programming language are usually defined either denotationally or operationally. In this section we will introduce the idea of an operational semantics, a denotational semantics, and the relationship between the two. In order to accomplish this, we shall consider a simple language of arithmetic expressions consisting of the set \mathbb{Z} of integers and the addition operator $+$, whose syntax is defined by the following BNF grammar:

$$expr ::= \mathbb{Z} \mid expr + expr \mid (expr)$$

Operational Semantics

Operational semantics are defined in terms of transitions of program states. This describes the execution of a program step-by-step as it would be run by a machine. A simple formal definition of an operational semantics for a language comprises a set of states S , and a relation $\rightarrow \subseteq S \times S$, which relates each state to all the states which can be reached by performing a single transition [Hut98]. The transition relation \rightarrow is usually defined by inference rules. Here are the inference rules for our simple language of arithmetic expressions:

$$\frac{}{\overline{\overline{n + m}} \rightarrow \overline{\overline{n + m}}} \text{Add1}$$

$$\frac{x \rightarrow x'}{x + y \rightarrow x' + y} \text{Add2} \quad \frac{y \rightarrow y'}{x + y \rightarrow x + y'} \text{Add3}$$

$$\frac{}{\overline{(\overline{n})} \rightarrow \overline{\overline{n}}} \text{Bracket1} \quad \frac{x \rightarrow x'}{\overline{(x)} \rightarrow \overline{(x')}} \text{Bracket2}$$

In the above rules an expression represented as \overline{n} is an integer, a fully evaluated expression. An integer can make no transition, and therefore has no transition rule. The rule Add1 shows that an addition is complete if both sides of the addition are integers, the result is just their sum. The rules Add2 and Add3 show that an addition expression can make a transition if either side of the addition can make a transition. Finally the Bracket rules show that an integer in a bracket is simply an integer, and that expressions within brackets may make transitions. Using these rules it is possible to trace the evaluation of any legal expression in the language. For example, the following proof tree verifies the transition $\overline{(\overline{1 + \overline{2}}) + \overline{3}} \rightarrow \overline{(\overline{3}) + \overline{3}}$:

$$\frac{\frac{\overline{\overline{1 + \overline{2}} \rightarrow \overline{\overline{3}}}}{\overline{(\overline{1 + \overline{2}}) \rightarrow \overline{(\overline{3})}}} \text{Bracket1}}{\overline{(\overline{1 + \overline{2}}) + \overline{3}} \rightarrow \overline{(\overline{3}) + \overline{3}}} \text{Add1}$$

The rules given above are non-deterministic, as there is a choice as to which side of an addition should proceed. However the calculation will always give the same answer, regardless of the evaluation order of the $+$. If desired, we could fix the evaluation order of the $+$ operator, by modifying the Add3 rule as follows:

$$\frac{y \rightarrow y'}{\overline{\overline{n}} + y \rightarrow \overline{\overline{n}} + y'} \text{Add3}$$

This new rule means that the right hand side of an addition can only make a transition if the left hand side is fully evaluated, thus forcing the left hand side to evaluate first.

Denotational Semantics

Denotational semantics are defined by a valuation function mapping terms to values. Formally, a denotational semantics for a language T of syntactic terms comprises two components: a set V of semantic values, and a compositional valuation function $\llbracket \cdot \rrbracket : T \rightarrow V$ that maps terms to

their meanings as values [Hut98]. A denotational semantics for the above language would be defined as follows:

$$\begin{aligned} \llbracket n \rrbracket &= n \\ \llbracket (x) \rrbracket &= \llbracket x \rrbracket \\ \llbracket x + y \rrbracket &= \llbracket x \rrbracket + \llbracket y \rrbracket \end{aligned}$$

That is, a number is fully evaluated and is therefore not rewritten, while an addition is the sum of the evaluation of both sides. Notice that this form of semantics does not define an evaluation order for the expressions.

This leads us to the relationship between the two forms of semantics. We should be able to define whether two semantics are equivalent. In this case the operational and denotational semantics are equivalent if every possible operational execution of an expression gives the same result as its denotational evaluation. We will formalise this idea later on.

1.2 Monads

Monads come from category theory, and their use for structuring denotational semantics was first discovered by Moggi [Mog89]. Wadler later realised that this work could also be applied to structuring Haskell programs [Wad90]. Monads provide a uniform framework for modelling computational features such as input/output, sequencing, state, concurrency and error handling.

Every monad has the same basic structure, consisting of a type constructor M , a return function and a bind function, usually written as $\gg=$. The return function allows us to move from ordinary values into monadic values, and the bind function allows us to sequence monadic operations.

$$\begin{aligned} \text{return} &:: a \rightarrow M a \\ \gg= &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \end{aligned}$$

Let us consider the Maybe monad for simple error handling.

Maybe

The values of type *Maybe a* are *Nothing*, together with values of the form *Just x* for a value x of type a .

$$\begin{aligned} \text{data Maybe } a &= \text{Just } a \\ &| \text{Nothing} \end{aligned}$$

This gives us the capacity for simple error handling, as a successful computation results in *Just a* and a failure results in *Nothing*. Since we now have a type for failure, we can now implement simple error recovery, however we are limited to only being able to safely handle exceptions within a monad.

We can now define `return` and `bind` ($\gg=$) so that we can move from ordinary values to monadic values, and exceptions are propagated (i.e. if f fails then so will $f \gg= g$)

```

return    :: a -> M a
return x  = Just x
( $\gg=$ )    :: M a -> (a -> M b) -> M b
f  $\gg=$  g = case f of
           Nothing -> Nothing
           Just a  -> g a

```

For more information on *Maybe* for error handling, see *A Functional Theory of Exceptions* [Spi88].

State

The state monad allows us to model programs which may contain a computational state. First we must define a new type *State*, which can be made into an instance of a monad.

```

newtype State s a = ST (s -> (a, s))

```

That is, a value of type *State s a* is simply a function that takes an initial state of type *s*, and returns a pair comprising a value of type *a* and a possibly new state *s'*.

```

instance Monad (State s) where
return    :: a -> State s a
return x  = ST (\s -> (x, s))
 $\gg=$      :: State s a -> (a -> State s b) -> State s b
(ST f)  $\gg=$  g = ST (\s -> let
                        (x, s') = f s
                        ST g' = g x
                        in
                        g' s')

```

The `let` declaration enables the threading of state, and is the only part of the above definition which may be unclear. I shall therefore break the statement down into its three constituent parts:

- (x, s') is the value and new state obtained by applying the state transformer function f to s .
- $ST\ g' = g\ x$ simply removes the *ST* tags.
- $g'\ s'$ is the final state computation.

For more information, see the paper *State in Haskell* [LJ95].

Input/Output

I/O in Haskell was problematic for many years until the introduction of monads for structuring programs. Now I/O is modelled as a special kind of state monad, where the state is that of the world. An action of type *IO a* conceptually has the following type:

$$\text{type IO } a = \text{World} \rightarrow (a, \text{World})$$

That is, a value of type *IO a* takes the state of the world as input and returns a, possibly new, world state and a value of type *a*. In practice, of course, the state of the world is not actually threaded through the program. For information on how *IO* is handled in Haskell, see *Functional Programming and Input/Output* [Gor92].

The ‘do’ notation

All monadic calculations may be sequenced using the do notation. This is particularly useful in I/O operations where it is natural to specify a number of operations to be performed in order, for example getting input from a user, performing some calculations and writing the result to a file.

The do notation is simply syntactic sugar for the bind operator, for example

```
readWrite :: IO()
readWrite = putStrLn " Write something" >>= \- →
            getLine >>= \line →
            putStrLn (" You Wrote " ++ line)
```

which writes a string to the standard output, reads in a line of input then prints it out, would be written as

```
readWrite :: IO()
readWrite = do
    putStrLn " Write Something"
    line ← getLine
    putStrLn (" You Wrote " ++ line)
```

The longer and more complex the sequence of monadic operations, the clearer the do notation syntax becomes compared to the bind syntax. For more information, see *A system of constructor classes: overloading and implicit higher-order polymorphism* [Jon95].

1.3 Concurrency

Concurrency is a programming technique which allows a system to be made up of many threads, or sub-programs, which can synchronise and communicate with each other. This allows more intuitive programming, as programs which conduct many separate tasks may be made up of a

thread for each task, communicating with a controller, rather than a single loop ordering each task in sequence.

This power comes at a price however, as it is much more complex to reason about and effectively design concurrent systems. Due to the complexity of concurrent programming, methods have been developed to model concurrent programs. These include CCS and more recently π -calculus [Mil99] which deals with mobility as well as concurrency.

Concurrency in Haskell

Concurrency in Haskell [JGF96] is based around a function *forkIO*, and the *MVar* data type. These two primitives alone allow us to write programs containing multiple threads which may communicate and synchronise. *MVar*'s have the advantage of being simple, whilst also being powerful enough to be used to build higher level concurrent abstractions such as Semaphores and Channels.

forkIO

At the core of a concurrent implementation is the ability to fork new threads which operate concurrently with the parent thread. This ability is provided by the *forkIO* function, which takes an *IO* action as input, forks that action as a new thread running concurrently with the existing computation, and returns the new thread's *ThreadID*, which can be used, for example, to kill the thread later on.

$$\text{forkIO} :: IO\ a \rightarrow IO\ ThreadID$$

MVars

Forking new threads without providing for communication and synchronisation would not enable programmers to solve many concurrent problems. In Haskell communication and synchronisation are both achieved with a single low level data type, the *MVar*.

$$\text{type } MVar\ a \quad \text{--- } A\ \text{synchronised\ mutable\ variable}$$

The basic functions for manipulating *MVars* allow for creating, writing and retrieving values. The function *newMVar* takes an input of type *a* as input, and returns a new *MVar a* containing the value of the input. The action *newEmptyMVar* simply returns an *MVar a* containing no data:

$$\begin{aligned} \text{newMVar} &:: a \rightarrow IO\ (MVar\ a) \\ \text{newEmptyMVar} &:: IO\ (MVar\ a) \end{aligned}$$

The interesting functions from a concurrent point of view are *takeMVar* and *putMVar*, as they deal with communication and synchronisation. The types and behaviour of these functions are as follows. Firstly,

$takeMVar :: MVar a \rightarrow IO a$

takes an *MVar* *a* as input, and behaves as follows:

- If the *MVar* contains data, *takeMVar* returns the value of the data and leaves the *MVar* empty.
- If the *MVar* is empty, *takeMVar* blocks the process, waiting for the *MVar* to be filled by another process calling *putMVar*.

Secondly

$putMVar :: MVar a \rightarrow a \rightarrow IO ()$

takes an *MVar* *a* and a value of type *a*, and behaves as follows:

- If the *MVar* contains data, *putMVar* blocks the process, waiting for the *MVar* to be emptied by another process calling *takeMVar*.
- If the *MVar* is empty, *putMVar* fills the *MVar* with the input value, and restarts one on the processes which may be blocked trying to take from the empty *MVar*.

The *MVar* and its associated functions provide all the functionality needed to write effective concurrent programs. They are, however, too low level to be generally useful to programmers as the code will be difficult to read and maintain. It is therefore useful to define higher level primitives which are more useful to programmers, examples of which are the semaphore and the channel.

Semaphores

The semaphore is the simplest of the higher level primitives. It is somewhat similar in behaviour to the *MVar*, however it is only used for synchronisation, and unlike the *MVar*, processes blocked on a semaphore are restarted in the order they were blocked.

$newtype QSem = QSem (MVar (Int, [MVar ()]))$

The semaphore is modelled as an *MVar*, which itself contains an integer and a list of *MVars*. The integer represents the number of available resources protected by the semaphore, and the list of *MVars* is used to keep track of the processes blocked, waiting for access. Each time a process blocks on a semaphore it is added to the back of the list of blocked processes. The functions available for manipulating semaphores are:

$newQSem :: Int \rightarrow IO QSem$ — Create a new semaphore
 $waitQSem :: QSem \rightarrow IO ()$ — Wait for access to the shared resource
 $signalQSem :: QSem \rightarrow IO ()$ — Release access to the shared resource

For implementation details see the paper Concurrent Haskell [JGF96], or the *QSem.lhs* file provided with your Haskell compiler.

Channels

The channel type is more complex, and is represented by two *MVars* keeping track of the two ends of the channel contents, the read and write ends. Empty *MVars* are used to block consumers trying to read from an empty channel.

```
data Chan a = Chan
    (MVar (Stream a))
    (MVar (Stream a))
```

```
type Stream a = MVar (ChItem a)
```

```
data ChItem a = ChItem a (Stream a)
```

Due to the complexity of the channel type we shall not explain it in depth here. For more information see the paper *Concurrent Haskell* [JGF96]. The following functions are available for manipulating channels:

```
newChan  :: IO (Chan a)
writeChan :: Chan a → a → IO ()
readChan :: Chan a → IO a
```

The function *newChan* sets up the read and write end of a channel by initialising them as two empty *MVars*. To put an element on a channel, a new hole at the write end is created. What was previously the empty *MVar* at the back of the channel is then filled in with a new stream element holding the entered value and the new hole. To read an element from a channel, the value at the read end is read, and the read end is moved to the next stream element.

1.4 Exceptions

Any programming language used for producing serious applications must have the capacity for error detection and recovery, thereby allowing for more robust systems. Some errors in a program can be avoided by careful programming and thorough testing, however not all errors can be caught in this way. Unpredictable errors, such as attempting to write to a file on a full disc or a program thread going into an infinite loop should not cause an entire program to fail. We shall now discuss exceptions in Haskell and the various techniques used to implement them.

Haskell98 Exceptions

Haskell's *I/O* monad offers a simple form of exception handling. *I/O* operations may raise an exception if something goes wrong, and that exception can be caught by a handler. The exception primitives offered by Haskell98 are:

```

userError :: String → IOError
ioError   :: IOError → IO a
catch     :: IO a → (IOError → IO a) → IO a

```

You can construct an *IOError* using the function *userError* and passing it a string, or by using one of the built in *IOErrors*, such as *DivByZero*. These errors can be raised by calling *ioError*. The action *catch a h* attempts to perform the action *a*, if an error is encountered it returns the value of the handler function *h* applied to the error encountered, otherwise it behaves simply as the action *a*. There are two major limitations of this approach to exception handling:

- Exceptions may only be raised and handled in *IO* code. The consequence of this is that errors in purely functional code, such as a pattern match failure or division by zero, immediately bring the entire program to a halt.
- It cannot handle asynchronous exceptions. A synchronous exception arises as a direct result of executing some piece of code, for example, opening a non-existent file, and as such can only be raised at well defined places. An asynchronous exception, on the other hand, is raised by something in the thread's environment. Timeouts and user interrupts are examples of asynchronous exceptions. Asynchronous exceptions occur at any time, which makes them far more difficult to deal with than synchronous exceptions [Jon00].

Imprecise Exceptions

Imprecise exceptions are an extension to Haskell's exception handling capabilities, which allow exceptions to be raised in purely functional code. Two major obstacles, however, stand in the way of this goal:

- Laziness - It is difficult to determine whether or not any particular piece of code is evaluated, therefore it is difficult to determine if an exception is raised.
- Evaluation Order - Haskell's denotational semantics deliberately avoids fixing evaluation order, so what exception does the expression $e1 + e2$ raise? (Where $e1$ and $e2$ are exceptions)

The paper *A Semantics for Imprecise Exceptions* [JRH⁺99] discusses an approach to solving these problems. This paper describes the following extensions to Haskell98's exception handling mechanism:

```

type IOError = Exception

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

```

The type *IOError* no longer seems to make sense, since exceptions are no longer limited to the *IO* monad, therefore the type *IOError* is represented as a synonym for the *Exception* data type. We also have the following extra functions for exception handling:

$$\begin{aligned} \text{raise} &:: \text{Exception} \rightarrow a \\ \text{evaluate} &:: a \rightarrow \text{IO } a \end{aligned}$$

The function *raise* takes an exception and constructs an exceptional value. This means that code only raises an exception if the call to *raise* is actually evaluated, solving the first problem of raising exceptions in purely functional code. The ‘evaluation order’ problem is solved by the trick of evaluating code in which we wish to *catch* exceptions in the *IO* monad. This allows us to safely ‘choose’ an exception to raise (if any are encountered) whilst preserving the semantics of the language. So in order to safely handle the example given above we use the following Haskell code:

$$\text{catch } (\text{evaluate } (\text{raise } e1 + \text{raise } e2)) h$$

Where *h* is a handler which will be applied to the chosen exception. This exception handling mechanism allows us to use the existing *catch* function to recover from exceptions raised anywhere in a program.

Asynchronous Exceptions

Asynchronous exceptions are an extension to Concurrent Haskell, which allows threads to asynchronously raise exceptions in other threads. A mechanism for safely locking a thread is also provided, which allows asynchronous exceptions to be handled without leaving the shared mutable state inconsistent. A thread may only be interrupted by an asynchronous exception if it is unlocked, or it is blocked waiting for access to a shared resource. The functions provided by this extension to Concurrent Haskell are:

$$\begin{aligned} \text{throwTo} &:: \text{ThreadID} \rightarrow \text{Exception} \rightarrow \text{IO } () \\ \text{block} &:: \text{IO } a \rightarrow \text{IO } a \\ \text{unblock} &:: \text{IO } a \rightarrow \text{IO } a \end{aligned}$$

The *throwTo* function takes a ThreadID *t* and an exception *e*, and raises *e* in the target thread *t*, just as if *t* itself had called *ioError e*. The *block* and *unblock* functions each take an *IO* action and execute it in a locked or unlocked state respectively. This approach to asynchronous exceptions, as well as a full semantics, is explored in the paper *Asynchronous Exceptions in Haskell* [MJMR01].

We can use asynchronous exceptions to implement a variety of concurrent abstractions which would otherwise be inaccessible. One example is a function *ParIO* which runs its two arguments in parallel, taking the result from the first one to finish, and killing the other:

$$\text{parIO} :: \text{IO } a \rightarrow \text{IO } a \rightarrow \text{IO } a$$

To implement this we can spawn two concurrent threads, each racing to fill a single *MVar* which will contain the result. The first thread to succeed will fill the *MVar*, whilst the other will block, trying to write to a full *MVar*. The parent thread then takes the result from the *MVar* and kills both children:

```
parIO      :: IO a → IO a → IO a  
parIO a1 a2 = do  
    m ← newEmptyMVar  
    c1 ← forkIO (child m a1)  
    c2 ← forkIO (child m a2)  
    r ← takeMVar m  
    throwTo c1 Kill  
    throwTo c2 Kill  
    return r  
  where child m a = do  
        r ← a  
        putMVar m r
```

1.5 Quickcheck

Quickcheck is a random testing tool for Haskell programs, developed by Classen and Hughes [CH00]. It allows the testing of functions using random test sets, but it is not a formal proof. This testing is, however, acceptable in some situations, and by carefully designing the test generators, many errors can be caught. Quickcheck can also be used for testing monadic code, but it is limited in its support of external libraries. It has been very useful to me as it has saved me many times from attempting to prove results which seemed reasonable, but were in fact false.

A Simple Example - Reverse

We wish to define a function which takes a list as input, and produces the list reversed as output. The Haskell function below implements this idea.

```
reverse      :: [a] → [a]
reverse []   = []
reverse (x : xs) = (reverse xs) ++ [x]
```

We know this function should have a number of simple properties, so in order to test this function, we define a number of laws which the function should obey:

```
reverse [x]      = [x]
reverse (xs ++ ys) = (reverse ys) ++ (reverse xs)
reverse (reverse xs) = xs
```

In order to check these laws using quickcheck we must represent them as Haskell functions, i.e the Haskell function for each law should return *True* in all cases.

```
prop_RevUnit x    = reverse [x] == [x]
prop_RevApp xs ys = reverse (xs ++ ys) == (reverse ys) ++ (reverse xs)
prop_RevRev xs    = reverse (reverse xs) == xs
```

Each law may be tested by issuing the command `quickCheck prop_xxx`. These laws alone are, however, insufficient. The function `quickcheck` is overloaded to allow it to handle laws with a varying number of inputs, and the overloading cannot be resolved if the law itself has a polymorphic type. The programmer must therefore specify a fixed type for which the law is to be tested. For this example a type signature could be given for each law, e.g.

```
prop_RevApp :: [Int] → [Int] → Bool
```

For more detailed information on testing using quickcheck, you can look at the papers *Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs* [CH00], and *Testing Monadic Code with Quickcheck* [CH]. These papers give far more detail about testing programs with quickcheck, including how to test conditional laws, and how to generate random test data.

1.6 Reading

This section contains an overview of the papers I consider most relevant to my research, as well as some background papers I found particularly useful.

- *Tackling the Awkward Squad* [Jon00] gives an overview of the techniques which have been developed by the Haskell community to address the real world problems of input/output, robustness, concurrency and interfacing to programs written in other languages.
- *Concurrent Haskell* [JGF96] describes the a proposed concurrent extension to Haskell98, allowing users to fork communicating program threads.
- *Handling Exceptions in Haskell* [Rei98] describes an exception handling extension to Haskell, as an alternative to the exception monad.
- *A Semantics for Imprecise Exceptions* [JRH⁺99] describes a proposed extension to Haskell98's exception handling mechanisms, which allow exceptions to be raised in purely functional code.
- *Asynchronous Exceptions in Haskell* [MJMR01] describes an extension of Concurrent Haskell to allow for concurrent threads to raise exceptions in other threads.
- *Writing a high performance web server in Haskell* [Mar00] brings together the concepts of concurrency, I/O and exception handling with a case study showing the development of a web server in Haskell.
- *Comprehending Monads, Essence of Functional Programming* and *Monads for functional programming* [Wad90] [Wad92] [Wad93] show how Moggi's work on structuring denotational semantics can be applied to structuring Haskell programs.
- *State in Haskell* [LJ95] presents a way of securely encapsulating stateful computations that manipulate multiple, named, mutable objects, in Haskell.

Relating the Semantics

In order to prove the two semantics equivalent we must prove that for every expression e both semantics produce the same result. By tracing every possible operational execution of the expression, and comparing each result to the denotational evaluation of the expression, we can determine whether the semantics differ. This notion is captured in the list comprehension below:

$and [e' == Val (eval e) | e' \leftarrow leaves (iterate e)]$ — *Theorem 1*

The function *iterate* fully evaluates the operational semantics into a tree, storing final results as leaves, and the function *leaves* retrieves the results from the tree:

```
data Tree      = Node Expr [Tree]

iterate        :: Expr -> Tree
iterate e     = Node e [iterate e' | e' \leftarrow trans e]

leaves         :: Tree -> [Expr]
leaves Node e [] = [e]
leaves Node e xs = concat (map leaves xs)
```

To prove *Theorem 1* we first prove that a transition in the operational semantics does not alter the denotational meaning of the expression. Again this notion is captured as a list comprehension, which states that every expression obtained from a legal transition in the operational semantics evaluates to the same value as the original expression:

$and [eval e = eval e' | e' \leftarrow trans e]$ — *Lemma 1*

Before we move to formally proving this result, we can test it informally using `quickCheck`. Since we defined the equivalence of the two semantics in Haskell, this is a simple case of running *Lemma 1* and *Theorem 1* as `quickcheck` laws;

```
prop_Lemma1    :: Expr -> Bool
prop_Lemma1 e = and [eval e' = eval e | e' \leftarrow trans e]

prop_Theorem1  :: Expr -> Bool
prop_Theorem1 e = and [e' = Val (eval e) | e' \leftarrow leaves (iterate e)]
```

Note that it is also necessary to write a random expression generator for `quickcheck`, but we will not discuss it here. For further details see *QuickCheck: a lightweight tool for random testing of Haskell programs* [CH00]. We can now test our semantics equivalent using `quickCheck`. When run in `hugs` we see that they both pass 100 random tests:

```
Main> quickCheck prop_Lemma1
OK, passed 100 tests.

Main> quickCheck prop_Theorem1
OK, passed 100 tests.
```

Having gained confidence from testing our semantics using quickcheck, we can now move on to proving the relationship formally. We start by proving *Lemma 1* by induction on e . The base cases of $Val\ n$ and $Add\ (Val\ n)\ (Val\ m)$ are trivial, so we only prove the interesting case, $Add\ x\ y$, which can be done as follows:

$$\begin{aligned}
& \text{and } [eval\ (Add\ x\ y) = eval\ e' \mid e' \leftarrow trans\ (Add\ x\ y)] \\
\Leftrightarrow & \{ \text{definition of eval, trans} \} \\
& \text{and } [eval\ x + eval\ y = eval\ e' \mid e' \leftarrow [Add\ x'\ y \mid x' \leftarrow trans\ x] \\
& \quad ++ [Add\ x\ y' \mid y' \leftarrow trans\ y]] \\
\Leftrightarrow & \{ \text{both cases are similar} \} \\
& \text{and } [eval\ x + eval\ y = eval\ e' \mid e' \leftarrow [Add\ x'\ y \mid x' \leftarrow trans\ x]] \\
\Leftrightarrow & \{ \text{substituting for } e' \} \\
& \text{and } [eval\ x + eval\ y = eval\ (Add\ x'\ y) \mid x' \leftarrow trans\ x] \\
\Leftrightarrow & \{ \text{definition of eval} \} \\
& \text{and } [eval\ x + eval\ y = eval\ x' + eval\ y \mid x' \leftarrow trans\ x] \\
\Leftrightarrow & \{ \text{cancelling the eval y} \} \\
& \text{and } [eval\ x = eval\ x' \mid x' \leftarrow trans\ x] \\
\Leftrightarrow & \{ \text{induction hypothesis} \} \\
& True
\end{aligned}$$

In order to prove *Theorem 1*, we use induction over the size of the expression e , defined simply as the number of constructors in e . Again we do not show the proof for the trivial cases and proceed straight to the interesting case, $Add\ x\ y$:

$$\begin{aligned}
& \text{and } [e' = Val\ (eval\ e) \mid e' \leftarrow leaves\ (iterate\ e)] \\
\Leftrightarrow & \{ \text{definition of iterate} \} \\
& \text{and } [e' = Val\ (eval\ e) \mid e' \leftarrow leaves\ (Node\ e\ [iterate\ e'' \mid e'' \leftarrow trans\ e])] \\
\Leftrightarrow & \{ \text{definition of leaves} \} \\
& \text{and } [e' = Val\ (eval\ e) \mid e' \leftarrow concat\ [leaves\ (iterate\ e'') \mid e'' \leftarrow trans\ e]] \\
\Leftrightarrow & \{ \text{list comprehensions} \} \\
& \text{and } [e' = Val\ (eval\ e) \mid e'' \leftarrow trans\ e, e' \leftarrow leaves\ (iterate\ e'')] \\
\Leftrightarrow & \{ \text{list comprehensions} \} \\
& \text{and } [and\ [e' = Val\ (eval\ e) \mid e' \leftarrow leaves\ (iterate\ e'')] \mid e'' \leftarrow trans\ e] \\
\Leftrightarrow & \{ \text{Lemma 1} \} \\
& \text{and } [and\ [e' = Val\ (eval\ e'') \mid e' \leftarrow leaves\ (iterate\ e'')] \mid e'' \leftarrow trans\ e] \\
\Leftrightarrow & \{ \text{induction hypothesis, Lemma 2} \} \\
& \text{and } [True \mid e'' \leftarrow trans\ e]
\end{aligned}$$

Writing a Compiler

We now proceed to produce a compiler for our arithmetic expression language, and an interpreter which will run the produced code. The interpreter is a simple stack machine:

$$\begin{aligned} \text{data Op} &= \text{Push Int} \mid \text{DoAdd} \\ \\ \text{type Stack} &= [\text{Int}] \\ \text{type Code} &= [\text{Op}] \\ \\ \text{compile} &:: \text{Expr} \rightarrow [\text{Op}] \\ \text{compile (Val } n) &= [\text{Push } n] \\ \text{compile (Add } x \ y) &= \text{compile } x \ ++ \ \text{compile } y \ ++ \ [\text{DoAdd}] \\ \\ \text{interpret} &:: \text{Stack} \rightarrow \text{Code} \rightarrow \text{Stack} \\ \text{interpret } xs \ [] &= xs \\ \text{interpret } xs \ (\text{Push } x) : os &= \text{interpret } (x : xs) \ os \\ \text{interpret } (y : x : xs) \ (\text{DoAdd} : os) &= \text{interpret } (x + y) : xs \ os \\ \text{interpret } _ _ &= \text{error "Stack Error"} \end{aligned}$$

The arithmetic expressions are compiled to stack instructions. The machine can either *Push* an integer onto the stack, or add the top two stack elements with *DoAdd*. The interpreter simply takes a stack and a compiled expression (a list of operations), and runs the compiled code.

Relating the Compiler and Semantics

Since we have produced a compiler for our arithmetic expression language, we should proceed to show it correct with respect to our semantics. We need only prove one of the semantics equivalent to the interpreted code since we have proved both semantics equivalent. This idea is captured in the following quickCheck property, in which we relate the denotational semantics to the interpreter:

$$\begin{aligned} \text{prop_Compile} &:: \text{Expr} \rightarrow \text{Bool} \\ \text{prop_Compile } e &= \text{interpret } [] \ (\text{compile } e) = [\text{eval } e] \end{aligned}$$

That is, if we compile an expression e and interpret it with an initially empty stack, we should get the result of evaluating that expression pushed onto an empty stack. This test passes the 100 random cases generated by quickCheck, so we now formally prove the compiler and interpreter equivalent to the semantics. This idea is captured in the Haskell code below:

$$\text{interpret } [] \ (\text{compile } e) = [\text{eval } e]$$

However we shall consider a more general case, where the code is executed with an arbitrary stack, and not the empty stack. This more general case is needed to successfully prove the theorem by induction.

$interpret\ s\ (compile\ e) = (eval\ e) : s$ — Theorem 2

We now prove *Theorem 2* by induction over e . Since an expression may be a value or an addition of two expressions, we must prove equivalence for both cases. We begin by proving the case $Val\ n$:

$$\begin{aligned}
 & interpret\ s\ (compile\ (Val\ n)) \\
 = & \{ \text{definition of compile} \} \\
 & interpret\ s\ [Push\ n] \\
 = & \{ \text{definition of interpret} \} \\
 & interpret\ (n : s)\ [] \\
 = & \{ \text{definition of interpret} \} \\
 & n : s \\
 = & \{ \text{definition of eval} \} \\
 & eval\ (Val\ n) : s
 \end{aligned}$$

Now we need to prove for the case $(Add\ x\ y)$:

$$\begin{aligned}
 & interpret\ s\ (compile\ (Add\ x\ y)) \\
 = & \{ \text{definition of compile} \} \\
 & interpret\ s\ (compile\ x\ ++\ compile\ y\ ++\ [DoAdd]) \\
 = & \{ \text{Lemma 3 (see below)} \} \\
 & interpret\ (interpret\ (interpret\ s\ (compile\ x))\ (compile\ y))\ [DoAdd] \\
 = & \{ \text{induction hypothesis} \} \\
 & interpret\ (interpret\ ((eval\ x) : s)\ (compile\ y))\ [DoAdd] \\
 = & \{ \text{induction hypothesis} \} \\
 & interpret\ ((eval\ y) : (eval\ x) : s)\ [DoAdd] \\
 = & \{ \text{definition of interpret} \} \\
 & interpret\ (((eval\ x) + (eval\ y)) : s)\ [] \\
 = & \{ \text{definition of interpret} \} \\
 & ((eval\ x) + (eval\ y)) : s \\
 = & \{ \text{definition of eval} \} \\
 & eval\ (Add\ x\ y) : s
 \end{aligned}$$

Lemma 3 states that in order to interpret the concatenation of two pieces of code, we can first interpret the first piece of code, then use the resulting stack to interpret the second piece of code. This can be translated in Haskell code as follows:

$$\text{interpret } s (xs \ ++ \ ys) = \text{interpret } (\text{interpret } s \ xs) \ ys$$

The proof of *Lemma 3* is shown below, which we do by induction over xs . Note that we do not show the proof for the trivial case, $xs = []$. We start by proving the case where xs is of the form $(\text{Push } n) : xs$:

$$\begin{aligned} & \text{interpret } s ((\text{Push } n) : xs \ ++ \ ys) \\ = & \{ \text{definition of interpret } \} \\ & \text{interpret } (n : s) (xs \ ++ \ ys) \\ = & \{ \text{induction hypothesis } \} \\ & \text{interpret } (\text{interpret } (n : s) \ xs) \ ys \\ = & \{ \text{definition of interpret } \} \\ & \text{interpret } (\text{interpret } s ((\text{Push } n) : xs)) \ ys \end{aligned}$$

We now prove *Lemma 3* for the final case, where xs is of the form $((\text{DoAdd}) : xs)$. Note that we also assume the stack is of the correct form, $(x : y : xs)$. The stack not having the required form corresponds to a “stack underflow” error. In this case, the equation to be proved is trivially true, because the result of both sides is undefined (\perp), provided that we assume that *interpret* is strict in its stack argument ($\text{interpret } \perp \ ops = \perp$). This extra strictness assumption could be avoided by representing and managing stack underflow explicitly, rather than doing so implicitly using \perp .

$$\begin{aligned} & \text{interpret } (x : y : s) (\text{DoAdd} : xs \ ++ \ ys) \\ = & \{ \text{definition of interpret } \} \\ & \text{interpret } ((x + y) : s) (xs \ ++ \ ys) \\ = & \{ \text{induction hypothesis } \} \\ & \text{interpret } (\text{interpret } ((x + y) : s) \ xs) \ ys \\ = & \{ \text{definition of interpret } \} \\ & \text{interpret } (\text{interpret } (y : x : s) (\text{DoAdd} : xs)) \ ys \end{aligned}$$

We have now shown the compiler and interpreter equivalent to the semantics, and can consider adding exceptions to our language.

2.2 Adding exceptions

In order to add exceptions to the language, we added two new operations, *Throw*, which simply raises an arbitrary exception, and *Catch* which takes two expressions, the expression to be evaluated and the expression to use in the case of an exception.

$$\begin{aligned} \text{data } Expr &= Val Int \\ &| Add Expr Expr \\ &| Throw \\ &| Catch Expr Expr \end{aligned}$$

Notice that we do not distinguish between exceptions, we simply have a *Throw* expression representing all exceptions, and that exceptions are “caught” by running an alternative expression. We also altered the denotational semantics of our language to propagate exceptions, meaning that any evaluation containing an uncaught *Throw* becomes exceptional.

$$\begin{aligned} eval &:: Expr \rightarrow Maybe Int \\ eval (Val n) &= return n \\ eval (Add x y) &= \mathbf{do} \\ &\quad a \leftarrow eval x \\ &\quad b \leftarrow eval y \\ &\quad return (a + b) \\ eval (Throw) &= mzero \\ eval (Catch x h) &= eval x \text{ 'mplus' } eval h \end{aligned}$$

Here we use the behaviour of the *Maybe* monad, and its membership of the *MonadPlus* class to implement the *Catch* function. The *MonadPlus* class adds two new monad operations, *mzero* and *mplus*. Using these in combination we can model a raised exception as *mzero*, which always fails, and catch exceptions using *mplus*. The function *mplus* takes two monadic operations, if the first succeeds the result is simply returned, otherwise the second is evaluated. We also produce a new operational semantics which allows exceptions to be propagated and caught:

$$\begin{aligned} trans &:: Expr \rightarrow [Expr] \\ trans (Val n) &= [] \\ trans (Add x y) &= \mathbf{case } (x, y) \mathbf{ of} \\ &\quad (Val n, Val m) \rightarrow [Val (n + m)] \\ &\quad (Throw, _) \rightarrow [Throw] \\ &\quad (_, Throw) \rightarrow [Throw] \\ &\quad (_, _) \rightarrow [Add x' y \mid x' \leftarrow trans x] ++ [Add x y' \mid y' \leftarrow trans y] \\ trans (Throw) &= [] \\ trans (Catch x h) &= \mathbf{case } x \mathbf{ of} \\ &\quad Val n \rightarrow [Val n] \\ &\quad Throw \rightarrow [h] \\ &\quad _ \rightarrow [Catch x' h \mid x' \leftarrow trans x] \end{aligned}$$

Here we do not provide proofs for the new language of arithmetic expressions with exceptions. We do however test a number of key behaviours using quickcheck. In order to test whether

the new denotational and operational semantics are equivalent we use the two quickcheck tests below. Again these tests check whether an operational transition alters the denotational meaning of the expression, and whether all possible operational executions of an expression result in the denotational evaluation of the expression:

$$\begin{aligned}
prop_Trans &:: Expr \rightarrow Bool \\
prop_Trans\ e &= and\ [eval\ e' = eval\ e \mid e' \leftarrow trans\ e] \\
\\
prop_Exec &:: Expr \rightarrow Bool \\
prop_Exec\ e &= and\ [match\ e' (eval\ e) \mid e' \leftarrow leaves\ (exec\ e)] \\
&\quad where \\
&\quad match\ (Val\ n)\ (Just\ m) = n == m \\
&\quad match\ (Throw)\ (Nothing) = True \\
&\quad match\ _ = False
\end{aligned}$$

These tests both pass 100 random expressions generated by quickcheck. Formal proofs will be given in a forthcoming paper. We now move on to compiling the expressions to stack code. When running code with a *Catch* we *Mark* the stack with the compiled handler, and place an *UnMark* operation at the end of the compiled *Catch* code. These *Mark* and *Unmark* operations are used to define the scope of a particular *Catch* block, and are used, in the case of *Mark* *h* to recover from a raised exception, or *UnMark* to remove unused handler code from the stack.

$$\begin{aligned}
compile &:: Expr \rightarrow Code \\
compile\ (Val\ n) &= [Push\ n] \\
compile\ (Add\ x\ y) &= compile\ x ++ compile\ y ++ [AddOp] \\
compile\ (Throw) &= [ThrowOp] \\
compile\ (Catch\ x\ h) &= Mark\ (compile\ h) : compile\ x ++ [Unmark]
\end{aligned}$$

We also want to run the code. In order to accomplish this we need a new stack type which will allow us to ‘*Push*’ handler code, and a machine capable of running the new stack code;

$$\begin{aligned}
type\ Stack &= [Item] \\
\\
data\ Item &= Value\ Int \mid Handler\ Code
\end{aligned}$$

The new interpreter, *run*, should recover from a *throw* by ‘unwinding’ the stack to the handler, and executing the handler code, before continuing running the operations after the current *catch* block. The extra work done by our interpreter compared to the exception free version was all added to handle recovering from exceptions.

$$\begin{aligned}
run &:: Stack \rightarrow Code \rightarrow Stack \\
run\ s\ [] &= s \\
run\ s\ (Push\ n : ops) &= run\ (Value\ n : s)\ ops \\
run\ s\ (AddOp : ops) &= \mathbf{case\ } s \mathbf{ of} \\
&\quad (Value\ y : Value\ x : s') \rightarrow run\ (Value\ (x + y) : s')\ ops
\end{aligned}$$

$$\begin{aligned}
run\ s\ (ThrowOp\ :\ ops) &= \mathbf{case}\ s\ \mathbf{of} \\
&\quad [] \rightarrow [] \\
&\quad (Value\ _ : s') \rightarrow run\ s'\ (ThrowOp\ :\ ops) \\
&\quad (Handler\ ops' : s') \rightarrow run\ s'\ (ops' \ ++\ skip\ ops) \\
run\ s\ (Mark\ ops' : ops) &= run\ (Handler\ ops' : s)\ ops \\
run\ s\ (Unmark : ops) &= \mathbf{case}\ s\ \mathbf{of} \\
&\quad (x : Handler\ _ : s') \rightarrow run\ (x : s')\ ops
\end{aligned}$$

The main part of exception handling is done when we try to execute code whose first instruction is *ThrowOp*. In this case we remove information from the stack until we find a handler, which was placed onto the stack using the *Mark ops* operation. We then skip the pending operations, *ops*, until we find the end of the catch block we are in, giving us the set of operations which would have been executed after the catch block. With this information we can run the handler, and carry on executing the expression. The code to skip operations is given below. If a *ThrowOp* is not encountered and we reach an *UnMark* operation the, therefore unused, handler is removed from the stack.

$$\begin{aligned}
skip &:: Code \rightarrow Code \\
skip\ (Unmark\ :\ ops) &= ops \\
skip\ (Mark\ _ : ops) &= skip\ (skip\ ops) \\
skip\ (_ : ops) &= skip\ ops
\end{aligned}$$

Implementing skip must take into account that *catch* blocks may be nested. Therefore when the start of a new catch block is encountered whilst ‘skipping’ we must entirely skip through that catch block before continuing. We can test that our approach to compiling expressions with exceptions is equivalent to our semantics using the following quickCheck test. Again we only test against our denotational semantics as the two semantics are equivalent:

$$\begin{aligned}
exec &:: Expr \rightarrow Stack \\
exec\ e &= run\ []\ (compile\ e) \\
prop_Compile &:: Expr \rightarrow Bool \\
prop_Compile\ e &= \mathbf{case}\ eval\ e\ \mathbf{of} \\
&\quad Just\ n \rightarrow exec\ e = [Value\ n] \\
&\quad Nothing \rightarrow exec\ e = []
\end{aligned}$$

We are currently in the process of writing an article called *The Essence of Compiling Exceptions* that explains the ideas discussed above in more detail, and provides a formal proof of the correctness of the compiler with respect to the denotational semantics.

2.3 Other Work

We briefly looked at a number of other extensions to our language:

- Separating the pure and exceptional expressions in the language. We can also force the programmer to explicitly sequence exceptional code so that any exceptions raised can be handled using a catch operator. This is similar to the ideas expressed in *A Semantics for Imprecise Exceptions* [JRH⁺99].
- Adding more than one exception and a functional handler. This extension would allow different exceptions to be handled individually rather than just running an alternative piece of code.

We now feel that the best way to proceed is to start looking at the ‘real’ language as described in the paper *Asynchronous Exceptions in Haskell* [MJMR01]. This decision will mean that we will focus on the semantics and not compilation issues.

3 Research Plan

I will now discuss some possible directions for further work. My sponsorship from Microsoft dictates that I should do something with the semantics of concurrency and asynchronous exceptions, so I shall consider only options related to these subjects. Possible directions are a higher level concurrency and exception library, or a more theoretical look at the uses of the asynchronous exception semantics.

3.1 High Level, Provably Correct Libraries

- Prove important properties of QSem's and Channels, i.e. no more than n processes can access a quantity semaphore with value n , and messages put onto a channel are recieved in the same order.
- Higher level Concurrency abstractions, look at ways to translate Monitors or Critical Regions into a functional setting.
- Higher level Exception abstractions, look for interesting and useful exception models, i.e. timeout and choice operators.

3.2 Towards An Algebra For Exceptions

- What does it mean for a program fragment to be safe? How can this be used to prove things about programs with exceptions?
- Higher level library than just *forkIO* and *throwTo*, provably correct using above information.
- Optimising rewrites of programs with exceptions using the previous results.

References

- [CH] Koen Claessen and John Hughes. Testing monadic code with quickcheck.
- [CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
- [Gor92] Andrew Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, Aug 92.
- [Hut98] Graham Hutton. Fold and Unfold for Program Semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.
- [JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
- [Jon95] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.
- [Jon00] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell, 2000.
- [JRH⁺99] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.
- [LJ95] John Launchbury and Simon L. Peyton Jones. State in haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
- [Mar00] S. Marlow. Writing high-performance server applications in haskell, 2000.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
- [MJMR01] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.
- [Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asilomar, CA*, pages 14–23, 1989.
- [Rei98] A. Reid. Handling exceptions in haskell, 1998.
- [Spi88] Mike Spivey. A functional theory of exceptions, 1988.

- [Wad90] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.
- [Wad92] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 19 – 22, 1992. ACM Press.
- [Wad93] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

Appendix A - Code (No Exceptions)

```
import QuickCheck

-- Arithmetic expressions:

data Expr          = Val Int | Add Expr Expr
                  deriving (Show,Eq)

esize              :: Expr -> Int
esize (Val _)     = 1
esize (Add x y)   = 1 + esize x + esize y

-- Test case generator:

instance Arbitrary Expr where
  arbitrary = sized arbExpr

arbExpr          :: Int -> Gen Expr
arbExpr 0        = do n <- arbitrary
                  return (Val n)
arbExpr n        = if (n < 8) then
                    frequency [(1, do n <- arbitrary
                                      return (Val n))
                                ,(4, do e1 <- arbExpr (n `div` 2)
                                      e2 <- arbExpr (n `div` 2)
                                      return (Add e1 e2))]
                    else arbExpr (n `div` 2)

-- Denotational semantics:

eval              :: Expr -> Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y

-- Operational semantics:

trans             :: Expr -> [Expr]
trans (Val n)    = []
trans (Add (Val n) (Val m)) = [Val (n+m)]
trans (Add x y)  = [Add x' y | x' <- trans x] ++
                  [Add x y' | y' <- trans y]

data Tree        = Node Expr [Tree]
```

```

exec          :: Expr -> Tree
exec e       = Node e [exec e' | e' <- trans e]

leaves       :: Tree -> [Expr]
leaves (Node e []) = [e]
leaves (Node e ts) = concat (map leaves ts)

-- Relating the two semantics:

prop_Lemma1  :: Expr -> Bool
prop_Lemma1 e = and [eval e' == eval e |
                    e' <- trans e]

prop_Theorem1 :: Expr -> Property
prop_Theorem1 e = collect (esize e) $
    and [e' == Val (eval e) |
        e' <- leaves (exec e)]

-- Stack-based compiler:

data Op      = Push Int | DoAdd

type Stack   = [Int]
type Code    = [Op]

compile     :: Expr -> [Op]
compile (Val n)      = [Push n]
compile (Add x y)    = compile x ++ compile y ++ [DoAdd]

interpret   :: Stack -> Code -> Stack
interpret s []      = s
interpret s (Push n : ops) = interpret (n : s) ops
interpret (y:x:s) (DoAdd : ops) = interpret (x+y : s) ops

-- Correctness of the compiler:

prop_Compile :: Expr -> Bool
prop_Compile e = interpret [] (compile e) == [eval e]

main :: IO()
main = quickCheck prop_Theorem1

```

Appendix B - Code (Exceptions Added)

```
import QuickCheck
import Monad

-- Types:

data Expr          = Val Int
                  | Add Expr Expr
                  | Throw
                  | Catch Expr Expr
                  deriving Show

type Code          = [Op]

data Op            = Push Int
                  | AddOp
                  | ThrowOp
                  | Mark Code
                  | Unmark
                  deriving (Show,Eq)

-- Test case generator:

instance Arbitrary Expr where
  arbitrary        = sized arbExpr

arbExpr            :: Int -> Gen Expr
arbExpr 0          = frequency [(4, do n <- arbitrary
                                     return (Val n))
                                ,(1, return Throw)]
arbExpr n          = frequency [(1, arbExpr 0)
                                ,(4, do x <- arbExpr (n `div` 2)
                                     y <- arbExpr (n `div` 2)
                                     return (Add x y))
                                ,(1, do x <- arbExpr (n `div` 2)
                                     y <- arbExpr (n `div` 2)
                                     return (Catch x y))]
```

-- Denotational semantics:

```

eval                :: Expr -> Maybe Int
eval (Val n)        = return n
eval (Add x y)      = do a <- eval x
                      b <- eval y
                      return (a+b)
eval (Throw)        = mzero
eval (Catch x h)    = eval x 'mplus' eval h

-- Compiler:

compile             :: Expr -> Code
compile (Val n)     = [Push n]
compile (Add x y)   = compile x ++ compile y ++ [AddOp]
compile (Throw)     = [ThrowOp]
compile (Catch x h) = Mark (compile h) : compile x ++ [Unmark]

-- Stack machine:

type Stack          = [Item]

data Item           = Value Int | Handler Code
                    deriving (Show,Eq)

run                 :: Stack -> Code -> Stack
run s []            = s
run s (Push n : ops) = run (Value n : s) ops
run s (AddOp : ops)  = case s of
                        (Value y : Value x : s') ->
                          run (Value (x+y) : s') ops
run s (ThrowOp : ops) = case s of
                        [] -> []
                        (Value _ : s') -> run s' (ThrowOp : ops)
                        (Handler ops' : s') -> run s' (ops' ++ skip ops)
run s (Mark ops' : ops) = run (Handler ops' : s) ops
run s (Unmark : ops)    = case s of
                        (x : Handler _ : s') -> run (x:s') ops

-- Skip to next unmark:

skip                :: Code -> Code
skip (Unmark : ops) = ops
skip (Mark _ : ops) = skip (skip ops)

```

```
skip (_      : ops) = skip ops
```

```
-- Main program:
```

```
exec          :: Expr -> Stack
exec e        = run [] (compile e)
```

```
-- Correctness of the compiler:
```

```
prop Compile      :: Expr -> Bool
prop Compile e    = case eval e of
    Just n  -> exec e == [Value n]
    Nothing -> exec e == []
```

```
-- Examples:
```

```
test1          = Add (Val 1) (Val 2)
test2          = Add (Catch Throw (Val 1)) (Val 2)
test3          = Add (Val 1) (Catch Throw (Val 2))
test4          = Catch (Catch Throw (Val 1)) (Val 2)
```