Oscar Key

Implementing and evaluating algebraic effect based code migration

Computer Science Tripos – Part II

Corpus Christi College

2016

Proforma

Name:	**Oscar Key**
College:	**Corpus Christi College**
Project Title:	**Implementing and evaluating algebraic effect based code migration.**
Examination:	**Computer Science Tripos – Part II, June 2016**
Word Count:	**11377**[1]
Project Originator:	Ohad Kammar
Supervisor:	Ohad Kammar

Original Aims of the Project

The original aim of the project was to evaluate the use of algebraic effects and handlers for implementing transparent code migration. Code migration has many advantages, but I expected this approach to be limited. Thus, I planned to implement a migration library for Haskell using this technique, and compare writing a distributed program using the library in standard Haskell to evaluate the limitations.

Work Completed

I have written an algebraic effect handlers based migration library for Haskell. I have also written two implementations of an example program, one using the migration library and the other using standard client/server design, but both providing the same functionality to the user. I have compared these two implementations to evaluate the advantages and disadvantages of the library.

[1]This word count was computed using *texcount*. It includes tables and footnotes, and excludes appendices, bibliography, photographs and diagrams.

Special Difficulties

None.

Declaration of originality

I Oscar Key of Corpus Christi College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

My project investigates the feasibility of algebraic effect based code migration. I have successfully implemented a limited transparent code migration library for Haskell using this technique, and evaluated it in terms of usability for the programmer and performance.

1.1 Motivation and Previous work

For some applications it may be convenient to have a portion of the computation run on a different host from where it started. There are several methods of implementing this, many of which involve several instances of a program running across different hosts, communicating using messages. However, having several instances running in parallel is confusing for the programmer, resulting in lower productivity and a greater likelihood of bugs. A different approach is code migration. Here the programmer issues a command such as `move [host]`, which requests that the computation migrate to another host and continue there. This is substantially easier for the programmer to reason about as there is now just one single instance of the program.

There are several possible approaches to code migration which have been previously implemented:

Write a new language.
: For example, Nomadic Pict is a statically typed language with native support for migration using mobile agents. [1].

Modify an existing language.
: The compiler and/or runtime of an existing language could be modified to add support for migration.

Write a library.
: For example, Sumii describes an implementation transparent code migration

as a library for Scheme [2]. This makes use of Scheme's support for code-as-data and dynamic typing to serialise the computation and transport it to another host.

Writing a new language or modifying an existing language is likely to be more complicated than writing a library. Additionally, writing a new language makes adoption more difficult for programmers, as they will have to learn new syntax and paradigms. If modifying an existing runtime, modifications are likely to be tightly coupled to the runtime, and so may need to be rewritten for a new version. In comparison, a library is far more likely to work, without modification, with different versions of the underlying language. Sumii's library has these advantages, but it relies on dynamic typing. Ideally, we would like to be able to implement migration as a library which can used with an existing statically typed language. However, this approach is likely to be more limited. I have written a partial code migration library for Haskell to investigate to what extent migration can be supported, and if the limitations of the implementation outweigh the benefits of migration.

1.2 Overview of approach

The library is based on the concept of representing a computation by its interactions with the environment, its side effects, such as memory accesses and random number generation. A pure language such as Haskell allows the type system to be used to restrict the side effects that a given computation may emit. If a computation may not perform any side effects then it must be referentially transparent. This means that for the same input it will always produce the same output in all contexts. Thus, we could represent the computation not as a set of instructions or mathematical operations, but instead as a mapping from input values to output values. This technique can be extended to include side effects by treating each side effect as a potential branching point in the path that the computation takes. Thus, we can build a tree of effects which represents every possible path the computation can take from an input value to a result. Take, for example, the code fragment below which might be taken from a Unix-style *wget* program:

```
let connected = isNetworkConnected ()
in if connected
    then ([download and print file]; SUCCESS_CODE)
    else (print "Error: no connection"; FAILURE_CODE)
```

This could be represented by a tree as in Figure 1.1. Here each effect is represented by a rectangle, with the labelled edges representing every possible return value from the effect. The result values from the computation are represented by the rounded rectangles.

Once a computation tree has been produced for a given program, we can perform the computation by traversing the tree. At each node we perform the effect it represents, and then recursively traverse the subtree associated with the return

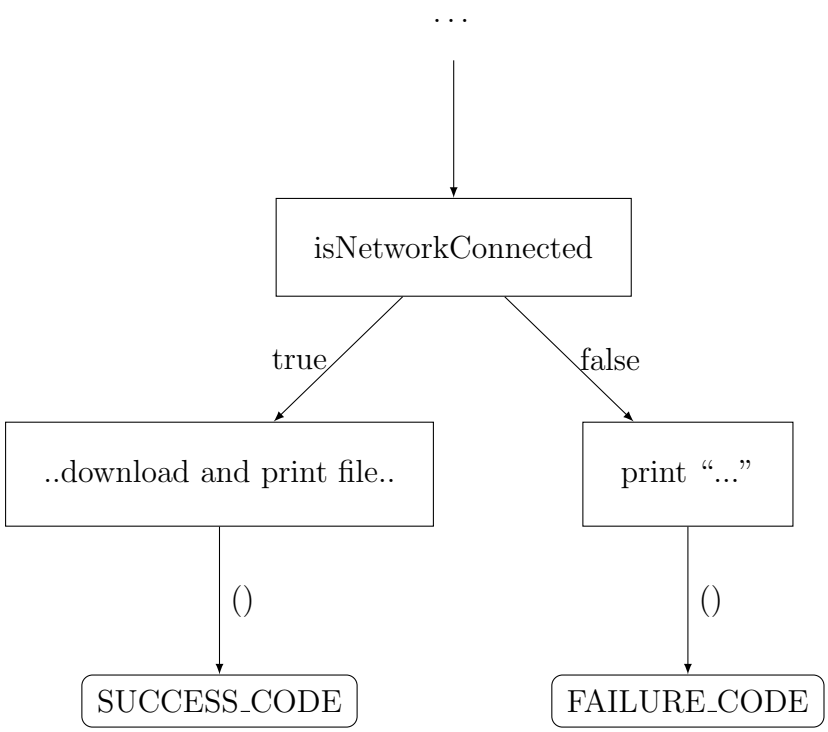

Figure 1.1: An example computation tree.

value of the effect. Thus, by moving the computation tree from one host to another we can effectively migrate the computation.

It is evident that there are several limitations to the approach. Firstly, the computation tree will exhibit exponential growth with respect to the number of effects with branching degree greater than one, so it is infeasible to represent large programs naively. Additionally, it will not be possible to directly represent effects which can return more than a handful of possible values as the branching factor will be unfeasibly high. For example, if an effect returns an integer it will not be possible to explore the computation for every possible value. A further limitation is that recursion must be fully unrolled, the computation tree must contain an instance of each effect produced by the function for every recursive call. This means that bounded recursion is represented inefficiently, and potentially infinite recursion, even if it terminates at runtime, cannot be represented at all. Finally, side effects normally modify some form of external state which may need to be migrated with the program. For example, a program might open a file, write to the file and then close it. If the computation migrates between opening and writing to the file, then either a new file must be opened, or the write must fail or be redirected. This approach does not solve this problem.

While these limitations were known from the start, I believed that the usability benefits for the programmer might outweigh them. I also thought that it might be possible to work around some of the problems described above in a practical implementation. By writing the migration library and an example application, I have investigated to what extent it is possible to solve the problems, and determined if this approach to migration is useful.

Chapter 2

Preparation

2.1 Technical background

The project used a number of advanced functional programming techniques. I had to familiarise myself with typeclasses, monads and effect handlers in Haskell. In the following sections I describe these.

2.1.1 Haskell

I chose to implement the project in the programming language Haskell [3]. As I will describe in the implementation, the project required a language which was both pure and also had an implementation of algebraic effect handlers available. Haskell satisfies both these criteria, and is reasonably popular.

In order to learn about Haskell, and the important language features such as typeclasses and monads, I read the book *Learn You A Haskell* [4]. Below I describe typeclasses and monads, which help to understand the design and implementation.

Typeclasses

The project makes extensive use of Haskell typeclasses in order to write more generic, polymorphic code. Typeclasses are an overloading mechanism, which allow several instances of the same function to be created with different implementations. A typeclass declaration contains the types of several named functions, and instances of the typeclass for specific types provide the implementation of the functions. Thus, the set of typeclasses for which a type has instances defines the operations which can be performed with the type. For example, the built-in `Eq` typeclass defines the infix function == which can be used to test if two values are equal to each other [5]:

```
class Eq a where
```

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
x /= y = not (x == y)
x == y = not (x /= y)
```

The brackets simply indicate that == is an infix operator. The typeclass declaration provides a definition of /= using == and vice versa, so that only one of the two functions needs to be defined by an instance.

We could then write an instance for booleans:

```
instance Eq Bool where
  (==) True  True  = True
  (==) False False = True
  (==) _     _     = False
```

When the function `(==)` is used, the Haskell compiler examines the types of the parameters and the return types and uses this information to choose the correct implementation of `(==)` to use. Importantly, this happens at compile time, not at run time. This means sufficient type information must be known at compile time to pick the correct instance.

When writing a type, the programmer can restrict type variables to members of a given class, and thus make assumptions about the operations available. For example, we could write a function which checks that the first two items in a list are equal:

```
firstwo :: (Eq a) => [a] -> Bool
firstwo (x:y:_) = x == y
firstwo _ = False
```

Here, `(Eq a) =>` specifies that any type which is substituted for `a` must be a member of the `Eq` typeclass, and so have `(==)` and `(/=)` functions associated with it.

The implementation of polymorphic equality with typeclasses appears similar to that in ML, which uses equality types [6]. ML specifies a set of equality types, for example integers and reals, which are types which can be tested for equality. It also provides the equality type variables `''a`, `''b`,... which range over the equality types. For example, the type of `firstwo` in ML is:

```
val firstwo = fn: ''a list -> bool
```

However, Haskell typeclasses are far more powerful because they allow the programmer to specify the exact implementation of equality for a new data type. For example, if we defined a set type which internally used a list, then in ML the built-in equality function would compare the sets as lists and thus take order into account. In Haskell, you could provide an implementation of `(==)` which ignores the order. Typeclasses are also more general as they can be used for many purposes, not just equality.

Haskell has many typeclasses built-in, along with instances for most of the common types. However, in many cases the programmer may want to define instances for new abstract data types (ADTs). Manually writing new instances is time consuming. Thus, Haskell provides an automatic derivation mechanism for some of the built-in typeclasses which automatically generates an instance of the class for an ADT, provided every type contained in the ADT is also a member of the class. For example, the following tree ADT derives an instance for `Eq`:

```
data Tree = Leaf Int | Branch Tree Tree deriving (Eq)
```

`Tree` values can now be tested for equality using `(==)`.

Monads

Below I give only a brief overview of the syntax of monads, which is sufficient to understand the implementation chapter. Appendix A contains a fuller description.

Haskell uses monads to abstract over computations formed of a sequence of steps. Normally, such computations are written using `do` notation. In a `do` block, each line is an action, and the entire block takes the value of the final line. Without extra syntax, the value of each action is discarded, so the actions in a `do` block behave much like a sequence of statements in an imperative language. In Haskell, a common use of monads is to represent IO computations which have side effects, without disrupting the overall purity of the language. For example, the following code prints to the standard output and writes to the given file:

```
writeMessage :: IO ()
writeMessage filename message = do
  putStrLn ( "Writing to file " ++ filename )
  writeFile filename message
```

Monadic actions produce monadic values — notice that the type of `writeMessage` is `IO ()` not `()`. Thus, we need to be able to convert to and from monadic values. The `return` function defined for each monad, completely unrelated to return statements in imperative languages, wraps a value as a monadic value:

```
class Monad m where
  return :: a -> m a
  ...
```

If we wish to unwrap a monadic value, we can use the <- operator in a `do` block. This is an operator which takes a monadic value on its right-hand side, and makes the value available as the value given on the left-hand side. Both `return` and <- are used in the following example which asks the given question, and returns true if the answer is "y", or false otherwise.

```
ask :: String -> IO Bool
ask question = do
  putStrLn question        -- Print question
  answer <- getLine        -- Read answer.
```

```
  return ( answer == "y" ) -- Compare answer.
```

The type of `getLine` is `IO String`, but the equals operator, in this case, expects a `String`. Thus, it is necessary to use `<-` to unwrap the `IO String` value to `String`. Conversely, the function should result in a value of type `IO Bool`, but the equals operator produces a `Bool`, thus it is necessary to use `return` to wrap the boolean as a monadic value.

2.1.2 Algebraic effect handlers

Algebraic effect handlers are an advanced functional programming technique which allows side effects to be raised and handled separately from the main program flow, much like exceptions in many programming languages [7]. We define a set of side effects that a computation may raise, and then separately define handlers for each of the effects.

Take, for example, an implementation of a program similar to the Unix *cat* program, which reads characters from the standard input, and writes them to the standard output. To support this we implement effects and handlers for reading and writing single characters. We define the effects themselves:

```
operation PutChar :: Char -> ()
operation GetChar :: Char
```

We define a computation type `CharIOComp`, for computations which can cause `PutChar` and `GetChar` effects, but no others.

```
type CharIOComp a = Comp a handles {PutChar, GetChar}
```

Using these effects, we can then implement the Unix-like *cat* program:

```
cat :: CharIOComp ()
cat = do
  c <- getChar
  if c == EOF then return ()
              else do
                putChar c
                cat
```

Here, the handlers library generates the functions `getChar` and `putChar` which raise the associated effect.

`cat` is a computation that raises `PutChar` and `GetChar` effects, but does not provide a concrete implementation of these effects. Thus, it is not possible to run `cat` by itself. Instead, we separately define one or more effect handlers to provide the effect implementations. We then apply the effect handlers to the computation in order to run it.

This is similar to the behaviour of exceptions and exception handlers. However,

when an exception is raised the execution of the code wrapped by the exception handler is terminated, whereas when an effect is raised the computation may continue after the effect has been executed. We achieve this using delimited continuations. A delimited, or partial, continuation stores the control state of a computation at a specific point. Evaluating the continuation is equivalent to executing the portion of the computation remaining from that point. When an effect is raised, the handlers library computes the continuation, a function from the return value of the effect up to the enclosing handler. The library passes the continuation to the handler for the effect. The handler then performs the effect, potentially applying the continuation function to the result value of the effect.

A basic handler which simply reads and writes characters using the IO monad is below:

```
1 handler CharIO :: IO ()
2   handles {GetChar, PutChar} where
3     Return    x -> return ()
4     PutChar c k -> do
5       writeChar c
6       k ()
7     GetChar   k -> do
8       c <- readChar
9       k c
```

Line 1 declares the name of the handler, and its type.

Line 2 declares the effects which the handler can handle.

Line 3 expresses how to return the final value of the computation. In this case the final value of the computation is ignored, and the unit value is returned. `return` is used to convert this to a monadic value to match the handlers `IO ()` type.

The remainder of the handler specifies the implementations for the two effects.

Lines 4-6 specify the implementation of the `PutChar` effect. `c` is the parameter of the effect, the character to be written, and `k` is the delimited continuation. The implementation uses the built-in function `writeChar` to write the character to the standard output. The continuation is applied to the unit value, the value of the effect.

Lines 7-9 are similar for the `GetChar` effect. This has no parameter. The continuation is passed the character read from the standard input.

The handlers library generates the function `charIO`, which applies the handler to a computation, running the computation:

```
charIO cat
```

cat could have been implemented without effect handlers using just the Haskell standard libraries. However, with effects and handlers we can change the behaviour of the computation by replacing the simple handler with a more complex one. For example, we could write a logging handler, which records each character written in an internal buffer instead of writing it immediately to the standard output. The final value of the computation is the buffer:

```
handler CharLogIO :: String -> IO String
  handles {GetChar, PutChar} where
    Return    x s -> return (reverse s)
    PutChar x k s -> k () (c:s)
    GetChar   k s ->
      c <- readChar
      k c s
```

This handler carries some state from effect to effect, represented here by the value `s`. In this case, this state is the list of characters put so far. The `PutChar` implementation calls the continuation with the unit value, and also appends the character put to the list. The `Return` implementation returns the buffer of characters when the computation ends. The type of the handler is `String -> IO String`, so when it is applied to a computation it must also be provided with an initial state, in this case just an empty string:

```
charLogIO "" cat
```

This demonstrates that, by changing the handler, we can control the behaviour of side effects emitted by a computation. In the migration library, I use a handler which captures each side effect rather than performing it immediately, and adds it to the computation tree.

2.2 Planning

2.2.1 Development model

As the project was an investigation into how much support for code migration could be achieved using algebraic effects, it was not clear at the start of the project exactly what would be possible. To mitigate the risk of uncertainty, I developed the project under the iterative development model, as described by Pressman [8]. I split the project into several phases, and specified what features the library should support, and the example program should demonstrate, at the end of each phase. This meant that, should unexpected difficulties have arisen, I could still have performed the evaluation and achieved the success criteria. Additionally, at the end of each phase I was able to revise my estimate of the final capabilities of the library, based on my new experience of what was possible, and so update the planned phases.

2.2.2 Phases

The project proposal split the project into three phases:

Phase 1 Support the migrating of computations from one host to another, but only effects with unit or boolean types.

Phase 2 Support effects for common operations, such as exceptions, nondeterministic backtracking and storing/retrieving a single bit of state.

Extension Support additional types, such as integers and strings.

While I was writing the example program to demonstrate the features of phase 1, I realised that it would not be possible to write an interesting program using the proposed features of phase 1 and phase 2 of the library. I thought that supporting integers and strings, along with efficient recursion, would be useful for the example applications, so I altered the phases to bring this work forward:

Phase 1 Support the migrating of computations from one host to another, but only effects with unit or boolean types.

Phase 2 Support additional types, such as integers and strings.

Extension Support efficient recursion and iteration.

2.3 Tools

I used the library for Haskell by Kammar, Lindley and Oury for the implementation of effect handlers [9] which required me to use Haskell 7.8.
I used Git for source control, and cloned the repository on a second machine as a backup. I used Dropbox as a secondary backup.

Chapter 3

Implementation

The library is a Haskell module which exports two primary functions, along with many other data types and functions which are useful when writing migrating computations. The two primary functions are:

- `listenForComp :: Port -> IO ()`
  Hosts which the computation may migrate to run this function. It continuously listens for incoming connections from other instances of the library, and executes any computations it receives.
- `runMigrationComp :: Port -> MigrationComp () -> IO ()`
  The host on which the computation starts runs this function. It first reifies the computation to produce a computation tree, and then executes this tree. Should the computation migrate away from the initial host, that host will begin listening for incoming computations.

The programmer might use the library as follows:

```
import Migration

myProgram :: MigrationComp ()
myProgram = ...

main :: IO ()
main = runMigrationComp "8001" myProgram
```

In addition to the main library, I produced a convenience program `Listen` which just runs `listenForComp`. Altogether, the library is about 500 lines of Haskell.

As I described in the preparation chapter, I implemented the library in three phases, building on its capabilities in each. Firstly I implemented the core migration system, which suffers from the limitations mentioned previously. I then added the abstract values system, which allows effects to return many possible values without causing the computation tree to grow exponentially. Finally, I

implemented and investigated some optimisations to efficiently migrate recursive functions. I have described the implementation in these three phases.

3.1 Phase 1: Basic migration

Stage 1 of the library supports migrating computations from one host to another, but only effects with boolean values.

The library transforms the Haskell program into a computation tree in a process called reification, and then executes the tree. If it encounters a migrate effect, it serialises the tree and transports it to another host. The instance of the library running on the second host deserialises the tree and continues to execute it. This process requires the following components:

Reification: The Haskell program is transformed into the computation tree.

Serialisation, deserialisation & transport: The computation tree is transported over the network from one host to another.

Execution: The computation tree is executed by traversing it and performing the effects.

3.1.1 Effects

The migration library does not support migration of arbitrary effects. This is due firstly to practical reasons, as the effect handlers library requires that each effect a computation can perform is explicitly defined. Additionally, I decided to avoid side effects which have dependencies on resources. Most side effects alter the state of the machine. For example, in order to write to a file you might first open the file, then perform the write operation and finally close the file. The second two effects depend on the first having already been performed, as the file resource must have been opened. Thus, if the computation migrates between any of the effects, the migration library would have to manage the machine state, for example by opening a file of the same name on the new host. We can imagine a system where each effect declares the resources that it either opens or closes, and the migration library keeps track of the set of currently open resources. Upon migration, the library closes the resources on the current host and opens them on the new host. I decided to avoid this complexity by not implementing effects with dependencies on resources. For example, an effect which writes to a file will atomically open, write to and close the file as one operation.

One of the reasons I chose to use Haskell to implement the library is because it is a pure language, so computations cannot perform arbitrary side effects. Instead, I allowed computations to perform a fixed set of algebraic effects, defined by the computation type `MigrationComp`. A subset of the effects defined is:

```
operation Migrate   :: (HostName, Port) -> ()
operation PrintStr  :: String -> ()
operation Ask       :: String -> Bool
...
```

`Migrate` moves the computation to another instance of the library listening on the host name and port specified. `PrintStr` prints the given string to the standard output. `Ask` prints the given question to the standard output and asks the user for yes/no response. If the user enters 'y' then it returns true, otherwise it returns false.

I also implemented many other effects which I will not describe in detail. These support reading and writing files, listing the files in a directory and reading strings and integers from the standard input.

3.1.2 Computation tree

The computation tree is a tree of side effects. Every path through the tree from root to leaf represents a possible execution of the computation.

Leaves represent the possible final values of the computation.

Branches represent effects. Each may have several parameters, and one or more children.

I represented the computation tree using an ADT:

```
data CompTree a = Result a
                | Migrate (HostName, Port) (CompTree a)
                | PrintStrEffect String (CompTree a)
                | AskEffect String (CompTree a) (CompTree a)
                ...
```

`Result a` is a leaf, the result value of the computation. `PrintStrEffect` represents the `PrintStr` effect above. This effect has a single possible value, `()`, thus it has a single child tree as there is only one possible path through the computation after this effect. In contrast, `AskEffect` has a boolean value, so has two child trees, one for the true case and one for the false.

3.1.3 Reification

The reification function transforms a compatible Haskell program into a computation tree:

```
reifyComp :: MigrationComp a -> CompTree a
```

I achieved this using a handler that reifies each effect as it is raised, rather than performing it immediately, and adds the reified representation to the computation tree. This is much like the logging handler described in the preparation chapter, only now we are dealing with a tree rather than a list.

However, as the function reifies effects rather than performing them immediately, in some cases we do not know their result value. Some effects have only one possible value, for example `PrintStr` always has the unit value. For these effects, we know the run time value of the effect at reification time, so we can pass it to the continuation. However, other effects such as `Ask` can return more than one possible value, and we do not know which at reification time. In this case, we must reify the computation for every possible value of the effect to produce multiple subtrees, and then choose the correct child at runtime. This is possible because the delimited continuation is just a function, so we can apply it as many times as necessary to different values.

A portion of the reification handler is below:

```
1 handler ReifyComp :: CompTree a
2   handles {PrintStr, Equal, ...}
3     Return        x -> Result x
4     PrintStr  str k -> PrintStrEffect str (k ())
5     Ask         q k -> AskEffect q (k True) (k False)
6     ...
```

Line 3 is the return case. We construct a leaf containing the final value of the computation and reification terminates.

Line 4 demonstrates an effect with only a single possible value, `()`. We apply the continuation `k`, the rest of the computation which has already been reified, to `()` to produce the subtree. We then construct the branch which is the effect along with its parameter and subtree.

Line 5 demonstrates an effect with two possible values. We apply the continuation twice, once for the true case and once for the false case.

This method of reification is limited because it relies on iterating over every possible return value for an effect. Thus, depending on the types of the effects, the reification function can have poor space and time complexity, as the computation tree can grow very large and take a long time to reify. For effects with only a single possible value, the computation tree will grow linearly with respect to the number of effects so reification is feasible. However, for effects with more than one value, the tree will grow exponentially. If each effect has only two or three possible values, for example booleans, then for small programs this may not be a problem. However, for effect returning types such as integers or strings the set of possible values is vast, so even reifying just one invocation of these effects becomes infeasible. The abstract value system which I will introduce in section 3.2 works around this problem. A further limitation of this reification system is that it does not support unbounded recursion. I will explore this in the evaluation.

3.1.4 Serialisation

The reification process produces a computation tree in memory, but it is not possible to directly transport this over the network. Serialisation is the process of converting a data structure in memory into a sequence of bytes from which the original structure can later be reconstructed. During migration, I transport the serialised tree from one host to another, deserialising it at the destination. The computation can then continue.

I investigated several serialisation systems which are available for Haskell. I decided to use the built-in `Read` and `Show` system which can convert data structures to and from character strings. This system is available in the standard libraries, simple to use and produces human readable serialised data, which is convenient for debugging. While it is slow, it is sufficient to demonstrate the library, and could easily be replaced with a faster library at a later date.

The `Read` and `Show` typeclasses are defined as follows:

```
class Read a where
  read :: String -> a
  ...

class Show a where
  show :: a -> String
  ...
```

`Show` performs serialisation by producing the character string which represents a value, while `Read` performs deserialisation by parsing a character string to find the value it represents.

To allow the computation tree to be serialised and deserialised, it must have an instance for both typeclasses. I used `deriving` notation to automatically generate both instances. However, a challenge arises because `CompTree a` contains the type variable `a`. If we enforce that `a` is only instantiated with types with an instance for `Show` then we can serialise the computation tree without an issue. However, deserialisation is more difficult. When applying `read` to a string, the compiler chooses the particular instance to use based on the inferred type. Unfortunately, we do not know the type of a computation received from another host at compile time, so it is not possible to select an instance of read. More generally, deserialising and executing a computation of an unknown type is at odds with static typing, as we do not know all the types at compile time.

I investigated a work around which is possible if we limit the possible types to a predefined set. The Haskell `Typeable` class provides the function `typeOf :: a -> TypeRep` which produces an object representing the type of a value [10, Data.Typeable]. I tried implementing a serialisation function that packaged the full type of the computation tree along with the tree itself:

```
show (tree, typeOf tree)
```

The deserialisation function performed a case match over the type, allowing the compiler to choose the correct instance of `read`.

As the set of possible types is still limited, I decided the benefit of this solution was not great enough to justify the extra complexity in the code. Additionally, I noted that the main function in Haskell programs must be of type `IO ()`, thus I thought it was reasonable to fix the type of migrating computations to `MigrationComp ()`. This avoids the issues described above.

3.1.5 Network

The requirement of the networking system is to transfer character strings, the serialised computation tree, from one host to another. I implemented this using the `Network.Simple.TCP` library which provides a simple API for reading and writing character strings over TCP sockets [11].

3.1.6 Execution

The execution function traverses the computation tree and performs each effect. Each effect is performed locally on the host which the computation is currently executing on, rather than the host which initiated the computation. For effects which perform console input and output, this will be connected to the console which is running the instance of the migration library currently executing the computation.

Starting at the root of the tree, the effect represented by the current node is performed. The subtree corresponding to the return value of the effect is then chosen, and this tree is executed recursively. A subtlety is how the execution terminates. In some cases, the execution may continue until it reaches a leaf, a result value. However, in other cases the computation may migrate before it reaches a leaf, so there is no result on that host. Thus, I used the `Maybe` datatype to allow the execution to terminate without reaching a value.

```
1  runCompTree :: (Show a, Read a) => CompTree a -> IO (Maybe a)
2  runCompTree effect = case effect of
3    Result x -> return $ Just x
4    MigrateEffect (host, port) comp -> do
5      sendComp (host, port) comp
6      return Nothing
7    PrintStrEffect str comp -> do
8      putStrLn str
9      runCompTree comp
10   AskEffect q compT compF ->
11     let response = ...
12     in if response then runCompTree compT
13                    else runCompTree compF
14   ...
```

Line 3: The computation terminates with a result value, so the execution function terminates with the value.

Line 4-6: On encountering a migrate effect the execution function calls a helper function, which serialises and transfers the computation tree, and then terminates with no value.

Line 7-9: This demonstrates an effect with only one possible value, and so only one child computation. The execution function writes the string to the standard output, and then recursively executes the rest of the computation.

Line 10-13: This demonstrates an effect with two possible values, true or false, and so two child computations. The execution function recursively executes the subtree for either the true case or the false case based on the response given by the user.

The implementation so far is sufficient to migrate computations from one host to another. However, it is severely limited because it only supports effects with boolean values. As most computations use other types, a useful library must support them. Thus, I implemented the abstract value system.

3.2 Phase 2: Abstract values

Abstract values allow effects to have a large number of possible values without causing the computation tree to grow in size exponentially.

The idea is that an abstract value can be returned by one effect and consumed by another, but cannot be directly inspected by the computation. This means that the path the computation takes between effects is independent of the value of the abstract value, so it is not necessary to branch over every value it may take. At reification time, an effect returns a fresh variable rather than a standard value.

When the execution function performs the effect, it substitutes the actual value returned for all occurrences of the variable in the rest of the computation.

Implementing the substitution method would be inefficient because it would require recursing over, and thus rebuilding, the entire computation tree to perform the substitution during execution. Instead, I simulated substitution using a store. The store is a mapping from keys to values, where the type of the key indicates the type of the value at that location:

```
save :: Store -> StoreKey a -> a -> Store
retrieve :: Store -> StoreKey a -> a
```

Variables are now associated with a key, which can be used to retrieve the value of the variable from the store. For example, I defined abstract integers:

```
data AbsInt = IntVal Int | IntVar (StoreKey Int)
```

Here an abstract integer either wraps a concrete integer, which is useful when converting concrete types to abstract types, or is a variable.

At reification time, the reification function generates a key for an unused location in the store for any effect which returns an abstract value. At execution time, the execution function saves the actual value of the effect at this location. Later during execution, an effect which consumes an abstract value evaluates it, retrieving any variables it contains from the store. If the computation migrates, the library transports the store from one host to another with the computation tree.

The keys for the store are integers. However they are contained by the `StoreKey` datatype which is parameterised by a phantom type parameter that does not appear in the definition. I describe the reasoning behind this in section 3.2.3.

```
data StoreKey a = StoreKey Int
```

As the keys are integers, I generated fresh keys by incrementing the integer value. In order to to this, I modified the reification handler to maintain an integer state `i`, the next unused key. When reifying an effect which returns an abstract value, the handler uses the state to construct a new key before incrementing it. The state is initially set to zero.

Take, for example, the effect `ReadInt :: AbsInt`, which reads and integer from the standard input:

```
1 handler ReifyComp a :: GenericStoreKey -> CompTree a
2   handles {...} where
3     ReadInt  k i ->
4       let key = StoreKey i
5       in ReadIntEffect key (k (IntVar key) (i+1))
6     ...
```

`i` is the state, the next unused key. Line 4 constructs a `StoreKey` using the state. Line 5, constructs the branch in the computation tree. The branch contains the

key where the value of the effect should be saved during execution. I construct a new abstract integer, `IntVar`, using the key and pass this to the continuation, along with the incremented state.

It is necessary to only use each store key once to avoid collisions during execution. We can see that this is the case by examining the computation tree. When the effect at the root of the tree was reified the state was `i`. Should the effect at the root of the tree not generate a fresh key, then the state during the reification of each of its children will still be `i`. This key is unused. Should the effect at the root of the tree generate a fresh key, then the state during the reification of all of its children will be `i'=i+1`. This state is also unused. Inductively, this is true for every node and its children. Thus, for any path down the tree from the root to a leaf we can see that we will never encounter the same key twice. Keys are reused in different paths, however only one path is ever executed so this does not create a collision.

For example, on encountering `ReadIntEffect` at execution time, the execution function saves the value read from the standard input to the store:

```
1 runCompTree (store, effect) = case effect of
2   ...
3   ReadIntEffect key comp -> do
4     value <- readLn
5     let store' = save store key value
6     runCompTree (store', comp)
7   ...
```

Line 4 reads and parses an integer from the standard input.

Line 5 saves this to the store at the key generated at reification time to generate a new, modified store.

Line 6 recursively continues the execution of the computation with this new store.

When the execution function executes an effect which consumes an abstract value, the abstract value must be evaluated to a concrete value. I define relationship between abstract and concrete types using the `Abstract` typeclass:

```
class Abstract a c | a -> c where
  eval :: Store -> a -> c
  toAbs :: c -> a
```

`eval` evaluates an abstract type to a concrete type, using the store to retrieve the values of any variables. `toAbs` wraps a concrete type in an abstract type. The instance for `AbsInt` is given later.

`Abstract` is a multi-parameter typeclass, a typeclass which can take more than one argument. The functional dependency `a -> c` specifies that the abstract type

fully determines the concrete type, i.e. that each abstract type represents exactly one concrete type [12]. This avoids any ambiguity when applying `eval` to an abstract type, as there can only be one associated concrete type. Thus, Haskell can choose the correct instance of `Abstract` without it being specified. Without the functional dependency, multiple concrete types could be associated with a single abstract type. This would mean that when using `eval` the expected concrete type would have to be specified so that the correct instance of `Abstract` could be chosen, which clutters the code. Both multi-parameter typeclasses and functional dependencies are supported by GHC by a language extension [13, section 7.6.1.1, 7.6.2].

3.2.1 Operations on abstract values

In addition to simply having abstract values that can be returned by one effect and consumed by another, it would be useful if it were possible to perform operations on abstract values as you can with normal values. For example, if the user enters two numbers, you may wish to print the sum of these numbers. Abstract values support this by representing not only a value or variable, but also a sequence of operations. For example, if we have numbers 3 and 5 and add them together, we can represent this either by the result 8, or by the sum $3 + 5$. The second representation is possible even if we have variables instead of values, for example $z = x + y$. Clearly it is not possible to compute the actual value of z at this point, but we can represent it as the sum of the future values of x and y. Once the values of x and y are known, we can then evaluate the sum to find z.

To support this, I extended the definition of `AbsInt`:

```
data AbsInt = IntVal Int
            | IntVar (StoreKey Int) -- A variable
            | OpPlus AbsInt AbsInt  -- x + y
            | OpMinus AbsInt AbsInt -- x - y
            | OpMult AbsInt AbsInt  -- x * y
            | OpSig AbsInt          -- -1,0,1 depending on sign
    deriving (Show, Read)
```

An abstract integer may now also represent various operations. For example, `z = OpPlus x y` represents $z = x + y$.

The `eval` function must now also perform operations when evaluating abstract values, in addition to substituting values for variables. For example, for integers:

```
instance Abstract AbsInt Int where
  eval store (IntVal    x) = x
  eval store (IntVar    k) = retrieve store k
  eval store (OpPlus  x y) = (eval store x) + (eval store y)
  ...
  toAbs x = IntVal x
```

Here we can see that `eval` evaluates concrete values to themselves, variables by retrieving the value from the store, and the plus operation by recursively evaluating the two operands and then adding the results. `toAbs` converts a concrete integer into an abstract integer by wrapping it in the constructor.

3.2.2 Standard library compatibility

Ideally, abstract values would be compatible with their standard counterparts. For example, if we have an existing function operating on integers, it would be convenient if it could also operate on abstract integers without modification.

A Haskell design pattern for achieving this behaviour is to define functions which operate on a typeclass rather than on a specific type. For example, a function might operate on any member of the more abstract class `Num`, rather than on `Int` specifically. `Num` defines basic numerical operations:

```
class  (Eq a, Show a) => Num a  where
  (+), (-)        :: a -> a -> a
  negate          :: a -> a
  abs, signum     :: a -> a
  fromInteger     :: Integer -> a
```

I took advantage of this in the migration library by writing instances for several abstract values. This allows them to be used in place of the standard values. For example, `AbsInt` has the following instance for `Num`:

```
1  instance Num AbsInt where
2    x + y = OpPlus x y
3    ...
4    signum x = OpSig x
5    abs x = x * (signum x)
6    fromInteger x = IntVal (fromInteger x)
```

Line 3 is the definition of `(+)`. It constructs a new abstract integer from the two operands which specifies that they should be added together.

To allow the programmer to use abstract values naturally, the library automatically converts literals, for example `3` or `"A string"`, into abstract values. This enables expressions such as `x + 3`, where `x` is an `AbsInt`. I implemented this behaviour using Haskell's support for overloaded numeric literals. Literals are not tied to a specific concrete type, but instead any type with a `Num` instance [5, section 6.4.1]. For example, `5 :: Num a => a` states that `5` can be of any type which has an instance of `Num`. At compile time Haskell uses the context to select the correct instance of `Num`, and applies the `fromInteger` of this instance to convert the literal into an value of the type. Line 6 above shows the implementation for abstract integers. It first converts the literal into an `Int`, and then wraps this in an `IntVal` constructor to produce an `AbsInt`. Similar behaviour

is available for lists and strings using language extensions available in GHC [13, section 7.6.5] [13, section 7.6.4].

This use of Haskell's standard classes makes writing migrating computations substantially easier. The programmer can use familiar syntax from standard Haskell, and even reuse existing code.

3.2.3 Heterogeneous store

As I described above, the store is a mapping from keys to values. The interface is given again below:

```
save :: Store -> StoreKey a -> a -> Store
retrieve :: Store -> StoreKey a -> a
```

Importantly, the store can save and retrieve any type `a`. Thus the store is a heterogenous mapping, containing values of different types. Unfortunately, there are no heterogenous collection implementations available in the Haskell standard libraries, they are all homogenous. I investigated the several possible solutions suggested on the Haskell wiki [14].

One option is to use existential types. This allows a collection to store values of any type, provided they are members of a given set of typeclasses. Unfortunately, once a value has been packed as an existential value it is not possible to unpack it to retrieve a value of the original type. This means that only the functions defined by the classes the value is a member on can operate on it. This is too restrictive for the library.

A second option is to use dynamic typing. Haskell provides some support for dynamic typing through the type `Dynamic`, which encapsulates an object of any type along with its type [10, Data.Dynamic]. This allows us to store values of different types in the store, and then inspect their values at runtime. Unfortunately, I found that `Dynamic` values are difficult to serialise. The built-in instance of `Show` serialises only the type of the object, not the object itself, and there is no instance of `Read`. Serialisation would require unwrapping the value from the `Dynamic` object, and then serialising it along with its type as I described in section 3.1.4. This is unwieldy.

If we limit the set of types supported then we can simulate the behaviour of `Dynamic` using a sum algebraic data type, which is easily serialisable by deriving instances for `Read` and `Show`. Every value in the store then has the same type so the map is homogenous, but the accessor functions wrap and unwrap the values, so to the user it appears to be heterogenous. This will only work for a fixed and finite (and relatively small) set of types, but its simplicity makes it a strong solution.

I implemented the store as a map from `Int` to `StoreValue`, with the constructors of `StoreValue` enumerating every type of value which can be stored.

```
data StoreKey a = StoreKey Int deriving (Show, Read)
data StoreValue = StoreIntValue Int
            | StoreBoolValue Bool
            | StoreStringValue String
            | ...
    deriving (Show, Read)
type Store = Map.Map Int StoreValue
```

As described, the interface to the store appears to be a map `StoreKey a -> a`. This is achieved by using the accessor functions `save` and `retrieve` to disguise the actual type of the map. I defined the typeclass `Storeable`, with an instance for every type which can be saved in the store. This defines the `save` and `retrieve` functions:

```
class Storeable a where
  save     :: Store -> StoreKey a -> a -> Store
  retrieve :: Store -> StoreKey a -> a

instance Storeable Int where
  save store k x = genericSave store k (StoreIntValue x)
  retrieve store k =
    let v = genericRetrieve store k
    in case v of StoreIntValue x -> x
                 _ -> error "Wrong type, expected Int"
```

Instances wrap the value in the appropriate `StoreValue` constructor when saving, and unwrap it when retrieving. The helper functions `genericSave` and `genericRetrieve` unwrap the `StoreKey` to an `Int` and perform the access operation on the `Map`.

Note that `retrieve` must perform a case split over the constructors of `StoreValue`. In the case where the constructor is not the one expected it must raise an error, a dynamic type error. Clearly it would be preferable if the Haskell compiler could catch such errors at compile time. While every key in the store is in fact an integer, the `StoreKey a` datatype is not simply an alias for `Int` but instead is a phantom type. This is an ADT parameterised by a type variable, but the variable does not appear on the right-hand side of the definition [15]. This means that, for example, a `StoreKey String` is a different type from a `StoreKey Char`, despite them both just wrapping an `Int`. The `save` function enforces that the type of the value being saved matches the type of the key, thus, unless the key or store is tampered with, the type of the value at the location accessed by retrieve is guaranteed to also match the type of the key. Should the programmer attempt to save a value of the wrong type, this will be caught by the static type checker. Hence, a dynamic type error cannot occur.

3.2.4 Other abstract values

In addition to integers, I also provided abstract implementations for booleans, chars and lists.

Abstract booleans and chars are implemented in much the same way as `AbsInt`. However, the equivalent of the `Num` typeclass does not exist, so there is no compatibility with existing code. In order to override existing functions such as `||` and `&&` I hide these from the prelude, and then redefine them.

The implementation of abstract lists is more complicated. Operations such as `cons`, `append` and `tail` can be implemented in a similar fashion to operations on the other abstract types:

```
data AbsList a = Nil
               | ListVal [a]
               | ListVar (StoreKey [a])
               | Cons a (AbsList a)
               | Append (AbsList a) (AbsList a)
               | Tail (AbsList a)
               ...
```

Clearly this implementation of tail is inefficient, as the list grows for every call, rather than shrinking as would be expected. The effect this has on the performance of the library is explored in the evaluation.

The implementation of `head` is more complex, because it is necessary to return a value rather than another list. At reification time, the head of the list is not known. Indeed, the list may be empty and not have a head. Thus, I implemented `head` as an effect:

```
operation Hd :: AbsList a -> Maybe a
```

This is a polymorphic effect. Unfortunately, the effect handlers library does not support polymorphic effects, so I had to restrict it to `AbsList AbsString`. This is sufficient as a demonstration.

The effect returns a value of type `Maybe` to reflect that the list may or may not have a head. This forces the programmer to test the maybe and so account for both possibilities. At reification time, both cases are explored by calling the continuation twice. In one, the handler returns a fresh variable to represent the future head of the list. In the other, the handler returns `Nothing` to indicate that the list was empty.

```
handler ReifyComp a :: GenericStoreKey -> CompTree a
  handles {...} where
    Hd  xs k i ->
      let key = StoreKey i
          head = Just (ListVar key)
      in HdEffect xs key (k head (i+1)) (k Nothing i)
```

The branch in the tree for the `Hd` effect contains the abstract list to take the head of, and the key of the variable for which the head should be substituted.

The portion of the execution function for abstract lists is shown below:

```
1 runCompTree (store, effect) = case effect of
2   HdEffect xs key headComp emptyComp ->
3     let xs' = eval store xs
4     in case xs' of
5       [] -> runCompTree (store, emptyComp)
6       (x:xs) -> do
7         let x' = eval store x
8             store' = save store key x'
9         runCompTree (store', headComp)
```

Line 3 evaluates the abstract list.

Line 5 is the case for when the evaluated list is empty. The handler recursively executes subtree for the empty case.

Line 6-9 is the case for when the list is not empty, and has a head. The head is an abstract value itself, and so must be evaluated. Line 8 saves the concrete value in the store.

The abstract values implementation allows useful computations to be written using the migration library. However, compatibility with standard Haskell programs is lost, and performance suffers. I explore the impact of this on library usage in the evaluation.

3.3 Extension: Recursion optimisations

As I described in the introduction, a limitation of representing computations as effect trees is that it is not possible to represent any recursion efficiently, and not possible to represent infinite recursion at all. This is because recursive calls must be fully unrolled. When representing recursion, the same set of effects will be repeated many times, thus increasing the size of the computation tree along with reification time. Additionally, if a computation recurses or loops infinitely, the reification function will never terminate. However, recursion is very commonly used in computations, thus the library should have some efficient support for it. I successfully implemented limited efficient recursion over lists, and also examined a possible technique for implementing full recursion efficiently.

3.3.1 Iteration over lists

While experimenting with the library, I found that iteration over lists was a common operation. Implemented naively, this quickly produces an unfeasibly large

computation tree, as explored in the evaluation. The Haskell standard library contains functions such as `map` and `traverse`, which move over every element in the list performing some function. For example, if we wanted to print every element in a list we could use `traverse_`, which maps each element of a list onto a monadic action, in this case writing the element to the standard output:

```
let greetings = ["Hey", "What's up?", "Hi"]
in traverse_ (\greeting -> putStrLn greeting) greetings
```

This will print:

```
Hey
What's up?
Hi
```

Based on this idea, I implemented an effect which iterates over an abstract list, applying a function to each element:

```
operation forEach ::
  (Abstract a) => (a -> MigrationComp ()) -> AbsList a -> ()
```

I had to restrict this type to `AbsString` because the effect handlers library does not support polymorphic effects.

`forEach` creates a new fresh variable which represents any particular element in the list, and applies the given function to it. This produces a computation applied to a general abstract value. `forEach` then reifies this computation to produce a computation tree `rf`. When the execution function reaches the `ForEach` effect, it evaluates the list and iterates over it. For every element, it saves the element to the store at key generated, and then executes `rf`. Thus, the reified function is executed on every element in the list.

3.3.2 Unbounded recursion

While I could have implemented more recursion schemes such as counterparts for `map` and the various folds, it would be better to support unbounded recursion. It would then be possible to implement those functions on top of this support. One method for fully supporting recursion or iteration would be to insert loops into the computation tree. Here, the subtree of an effect is in fact a reference back to an earlier effect in the tree. This would reduce the size of the computation tree by avoiding the sequence of effects in the loop being included several times, and also support infinite recursion in a finite computation tree. However, this requires detecting recursion during reification in order to inset the loops into the tree. This is not possible as the reification function sees only effects, not function calls. Thus, it would be necessary to perform recursion using an effect:

```
recurse :: FuncId -> (a -> b) -> ()
```

This takes the function to call, `a`, and also an id. The library would then maintain a mapping from ids to nodes in the computation tree. When `recurse` is raised

the reification handler looks up the id in the mapping. Initially the id will not be present, so the handler reifies the function as usual, and inserts an entry into the map from the id to the first effect of the function. On subsequent calls the id will be present, so the reification handler inserts a loop into the tree from this effect to the effect which the map points to.

As I will explore in the evaluation, fully unrolling recursive calls has a major negative impact on the performance of the library. The optimisations above solve this problem.

Chapter 4

Evaluation

In this chapter I evaluate both the library and the overall success of the project. We know that migration provides many advantages when writing many distributed applications, but I determine if these advantages are outweighed by the limitations of this particular implementation. I compare two implementations of an example program, one using the migration library and the other using standard client/server Haskell, but both providing the same functionality to the user.

4.1 Methodology

4.1.1 Usability

The usability of a programming language or library is a qualitative assessment, and there are many aspects that could be examined. I used the following, which are all closely related, as points of comparison:

Expressive power: The number of concepts which the language has syntax to represent.
A more expressive language allows complex ideas to be represented succinctly, leading to programs which are more concise and readable.

Abstraction: High-level data and control structures are used.
Higher-level constructs enhance conciseness and readability, and avoid bugs in implementing common operations.

Conciseness: The programmer can express what they mean without excessive text.
The more concise a program, the less needs to be read to understand what it does. This will enhance readability and help avoid bugs. However, being overly concise may obfuscate the meaning of the code.

Readability: The programmer can quickly and easily understand what a program does by reading it.
This makes it easier to maintain existing code, particularly for programmers new to the project, and makes it more likely bugs will be spotted.

Portability: The ability to run the same program on multiple platforms without significant extra work. Additionally, the development environment is easy to set up.

4.1.2 Performance

I used the following as quantitative measures of performance:

Network traffic: The quantity of data transferred over the network.
I used networking functions which represent data as character strings, so I measured traffic by recording the number of characters written to the socket.

Execution time: The time taken for the program to execute.
I modified the programs to record the time at the start of execution and at the end, and took the difference between these two values.

I expected that recursion would cause the computation tree to grow exponentially. I investigated this by altering the depth of the recursion performed by the example program, and comparing this to the network traffic, and thus the size of the computation tree.

4.2 Example program

In order to evaluate the points above, I compared two implementations of an example program, *getfile*. One of the implementations uses the migration library, while the other is written in a standard client/server style. *getfile* is a basic system for retrieving text files from a set of remote servers, where access to the servers is protected by a username and password. The system contains three entities: the client machine, the authorisation server and the file servers. The file servers are protected behind a firewall which blocks all inbound connections, except those from the authorisation server. This arrangement is shown in Figure 4.1.

The migrating application starts on the client and asks for the authentication details. The firewall prevents it from migrating directly to a file server, so it first migrates to the authentication server. Here, the library has been configured to only allow migration to the file servers for authenticated users. If the authentication succeeds, the program then migrates to the file server, reads the file from the disk, migrates back to the client and displays the file.

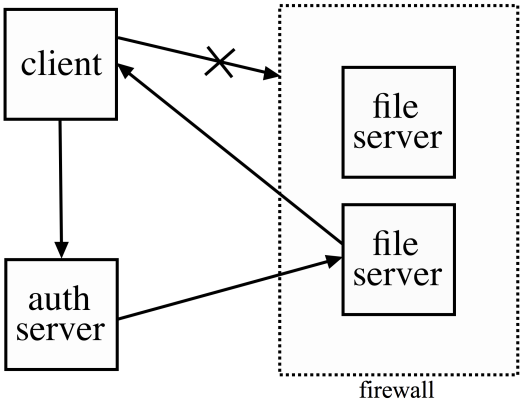

Figure 4.1: The architecture of *getfile*

The client/server implementation is made up of a client application, an authorisation server and a file server. The client sends a request to the authentication server, which contains both the authentication details and the file(s) requested. The authentication server then verifies the username and password. If they are correct, it forwards the request to the relevant file server. The file is then transferred directly back to the client.

The program operates in two modes. In the first, where it is just supplied with a server name, it retrieves a list of the files on that server. In the second, where it is supplied with a server name and a list of files, it retrieves each of the files and prints them to the standard output.

4.3 Results

4.3.1 Usability

Expressive power

Computations compatible with the migration library are less expressive than those written in standard Haskell in two aspects:

Recursion Without the recursion optimisations, migrating computations cannot express infinite effectful recursions.
I knew this was a limitation of the approach, and the evaluation emphasised the impact on the programmer. The example program must iterate over a list of filenames, and read each file. In standard Haskell we can express recursion as shown in Fragment 1, which terminates when the list of file names is empty. However, this kind of function would cause the reification function in the migration library to fail to terminate as the recursive calls producing effects must be fully unrolled. Thus, the programmer must write a recursive function which is explicitly limited

to a given depth. This is shown in Fragment 2.

```
readFiles :: [FileName] -> IO [String]
readFiles [] = return []
readFiles (filename : filenames) = do
  file <- readFile filename
  files <- readFiles filenames
  return (file : files)
```

Fragment 1: Client/server version: Explicit recursion. (This is for comparison, you would normally use a map to implement this.)

```
readFiles :: AbsList AbsFileName -> MigrationComp
                                      (AbsList AbsString)
readFiles fileNames =
  let limited 0 _ = return Nil
    limited n files = do
      file <- hd files
      case file of Nothing -> return Nil
                   Just f -> do
                     text <- readFl f
                     rest <- limited (n-1) (tl files)
                     return $ acons text rest
  in limited 10 fileNames
```

Fragment 2: Migrating version: Explicit recursion limited to a depth of 10 files.

Clearly this is a major limitation of the expressiveness of computations supported by the library. It will prevent them from operating on unbounded data, for example an application such as a text editor which needs to continuously accept user input. Additionally, limiting the recursion depth clutters the code, again harming usability. However, pure parts of the computation which do not use abstract values can still express recursion as normal, as these sections do not have to be represented in the computation tree.

Extensibility The migration library is not extensible.
As computations using the migration library must perform all side effects via effects supported by the library, the expressiveness is limited by this set of effects. Thus, ideally it would be possible for a programmer to extend the library by writing their own effects, possibly using additional modules or plugins. However, this is not supported by the library. Constructors for new effects cannot be added to the computation tree ADT, and the reification and execution functions cannot be modified with cases for these effects. Additionally, the store can only contain a fixed set of types, so abstract values which need to store types other than these would be difficult to implement.

Abstraction

```
-- Build and send the request.
let authRequest = AuthRequest {username = username,
                               password = password,
                               fileHost = fileHost,
                               dataRequest = request}
connect authHost authPort $ \(socket, remoteAddress) -> do
  send socket $ show authRequest
```

Fragment 3: Client/server verison: Create an authentication request message, serialise and send it.

```
migrate (authServer, authPort)
```

Fragment 4: Migrating version: Migrate from the client to the authorisation server.

High-level network interaction In regard to networking, the migration library provides a significantly higher level of abstraction than standard Haskell. Fragment 3 shows the creation of instance of an authorisation message, and its serialisation in the client/server implementation. In the migrating version, the programmer uses the single line show in Fragment 4 to migrate to the authorisation server. The transparent nature of the migration means that the programmer does not need to explicitly transport any data, as effects executed on the server can automatically make use of any values defined on the client. This high level of abstraction greatly simplifies the model that the programmer is exposed to.

Loss of compatibility with the standard library Migrating computations cannot make use of the majority of the standard library high-level functions. For example, both programs must read the contents of multiple files. In standard Haskell this is written conveniently using the monadic map function over the list of filenames:

```
files <- mapM (\file -> readFile file) fileNames
```

However, an equivalent function does not exist for abstract lists, so the migrating version uses explicit, depth-limited recursion, as in Fragment 2. Explicit recursion is generally discouraged by the Haskell community [16].

Conciseness

Migrating computations are more concise.
I performed a line count on both of the programs using the program *cloc* [17]. The count excluded blank and comment lines.

	lines
migrating	59
client/server	124

As both versions of the program implement the same functionality, this demonstrates that the migrating code is more concise. This is largely due to the high level of abstraction provided by the migration library, which means that no extra code has to be written to handle the distributed nature of the computation.

Readability

```
File Server:
  files <- getDirectoryContents "."
  let response = ListResponse files
  connect clientHost clientPort $ \(socket, _) -> do
    send socket $ show response

Client:
  recvLine port $ \line -> do
    let response = read line
    case response of ListResponse files ->
      traverse_ (\file -> putStrLn file) files
```

Fragment 5: Client/server Haskell: Read the list of files on the server, print it on the client.

```
files <- listFls
migrate (clientHost, clientPort)
forEach (\fileName -> printStr fileName) files
```

Fragment 6: Migrating Haskell: Read the list of files on the server, print it on the client.

Migrating computations are substantially more readable.
Fragments 5 and 6 show reading the list of the files in the current directory on the file server, and then printing this information on the client. The migrating code is a simple linear sequence of instructions, which will be executed in order. This is easy for the programmer to understand and reason about as they can simply follow the path of execution through the program.

In contrast, the asynchronous nature of the client/server version adversely affects the readability. The client sends a message to the server, but the points at which the message is sent and received are neither associated with nor close to each other in the code, making it difficult to follow the execution flow. Callbacks make this more difficult still. A callback is a function passed to another function which is called when a result becomes available. For example, the function passed to `recvLine` in Fragment 5 is called when a line has been received. However, this is done asynchronously, which means that lines below the callback may be executed before the callback itself, again making it less intuitive to follow the execution path. In some cases, a callback may make a further asynchronous request which will result in a further callback. These nested callbacks can be particularly difficult for the programmer to follow.

Portability

The portability of both versions is roughly equal.
In both cases, the program must be compiled individually for the target platform, though no changes are required to the source code. Additionally, in both cases it is possible for the different hosts involved to be running different operating systems.

The migration library has stricter requirements of the development environment than standard Haskell. The effect handlers library requires GHC version 7.8, whereas the client/server version has no specific requirement. This dependence on GHC version impacts the usability of developing with the migration library, as the user may not have this specific version installed, and it may conflict with any existing installation.

4.3.2 Performance

Network traffic

I measured the number of characters transferred during each communication phase depicted in Figure 4.1.

This analysis ignores the initial overhead of transferring the migration library and the client/server implementation to each machine. However, I would expect the program to be executed many times over its lifetime, thus this initial overhead will be dwarfed by the data transferred during execution.

When retrieving the list of files present on a server, 140 characters long, the migrating version does transfer more data, though the difference is not problematic:

	migrating	client/server
client to auth server	439	150
auth server to file server	261	59
file server to client	439	182

However, when retrieving files the migrating version transfers substantially more data. For a 1224 character file, the characters transferred were:

	migrating	client/server
client to auth server	5056	188
auth server to file server	4873	97
file server to client	1523	1292

This difference was due to the type of recursion used. When retrieving the list of files on the server, the only recursion performed by the migrating version is using the optimised `forEach` to print every file name in the list. In contrast,

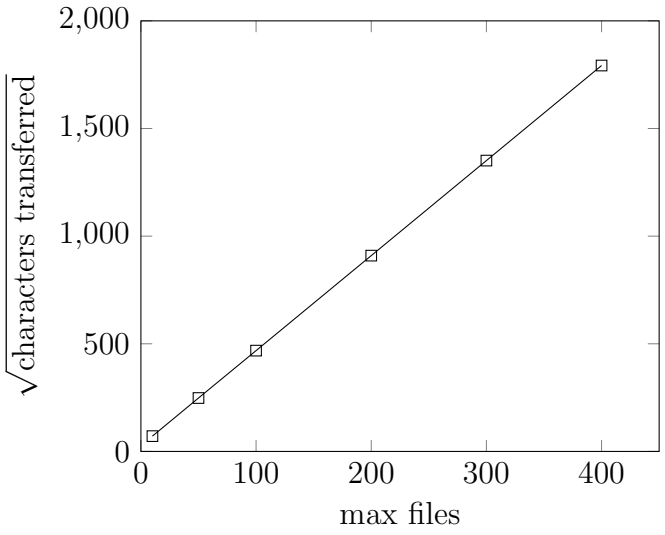

Figure 4.2: Plot showing a quadratic increase in network transfer as the maximum number of files is increased.

when retrieving the file contents the program must iterate over each of the files and read it. This iteration is not possible using `forEach`, so the example has to perform explicit, depth-limited recursion. In the first two migrations, where the reified iteration code is part of the computation tree, the migrating version transfers an order of magnitude more data. However, for the final stage, where the iteration has been executed and so is no longer in the computation tree, the number of characters transferred by both versions is more similar.

To investigate further, I varied the limit on the depth of the recursive function which loads the files, and thus the maximum number of files supported. I measured the effect on the characters transferred in the first migration:

max files	characters transferred
10	5056
50	61544
100	219269
200	827219
300	1825169
400	3213119

I plotted $\sqrt{\text{characters transferred}}$ against max files, as shown in Figure 4.2. The straight trend line shows that:

$$\text{characters transferred} \propto \text{max files}^2$$

If we examine the code for the bounded recursion in Fragment 2 the reason becomes clear. Every recursion produces a head effect. Here, the computation tree splits in two, one branch for if the list has a head, and a second if the list is empty.

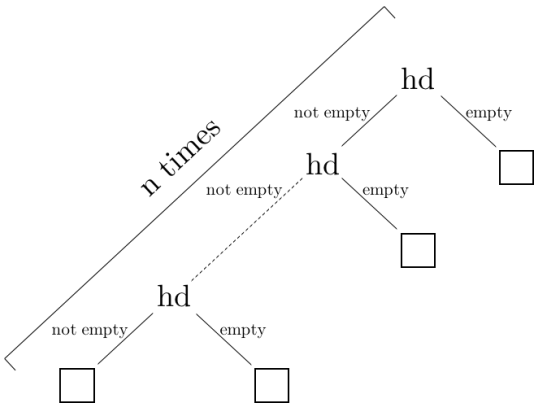

Figure 4.3: The computation tree produced by recursing over a list. The boxes represent the rest of the computation, which is has a constant size.

However, in the empty case the branch is of a constant size as it just contains the set of effects to migrate back to the client and print the files. This happens n times, if the recursion is limited to this depth, see figure 4.3. Each `hd` effect contains a copy of the list which it is taking the head of, and this has maximum length n. Thus, the number of characters transferred grows proportionally to $n \times n = n^2$.

Execution time

I modified the example programs to record the interval between the program starting, and the results being printed on the screen. I removed the username and password inputs and replaced them with hardcoded values to produce consistent timings. I then recorded the time to fetch the same 1224 character file used above. As before, I expected the performance of the migrating example to deteriorate as the maximum number of supported files is increased. Thus, I took measurements increasing the maximum number of files:

max files	migrating (s)	client/server (s)
10	0.19	0.12
50	0.53	0.12
100	1.30	0.12
200	4.16	0.12
300	9.14	0.12

For low values of max files the performance of the migrating version is within the same order of magnitude as the client/server version, and the difference is not significant enough for the user to notice. However, the performance gap increases quickly as the number of files increases, quickly making the migrating version infeasible.

4.4 Overall result

The usability evaluation shows that migrating applications in general offer a higher level of abstraction and have a simple control flow, and so are more readable. However, my implementation of migration has many usability compromises. As effectful recursive functions are fully unrolled at reification time, expressing recursion is difficult. Additionally, many standard library functions cannot be used.

The performance evaluation showed that while, as expected, the migrating version of the sample application always has a higher network usage and longer execution time than the client/server implementation, this difference is not necessarily that large. Recursion has a large influence on the performance, and this largely determines the feasibility of programs using the library. Explicit recursion performs very poorly, and programs using it are not feasible, however optimised recursion performs well. This suggests that if I had implemented the full recursion optimisation, the performance of the migrating computation would be much closer to the client/server version.

Overall, the evaluation demonstrates that the library is not a suitable choice for real-world applications in its current state, the programmer would be better off using standard Haskell. However, the usability advantages of migration itself are clear, and these are complemented by the ease with which this implementation can be added to a project as a library. Thus, with further enhancements it may be a good be a good choice in some specific scenarios. The main advantage of migrating programs is that they are easier for the programmer to reason about, and this advantage grows with the number of hosts involved. Thus, the migration library is best suited to programs which must run over a large number of hosts and where performance is not critical. For example, we could imagine using a migrating program as an alternative method to configure a large number of hosts in a cluster.

4.5 Project evaluation

The original aim of the project was to build a library for Haskell which supports limited transparent code migration in order to investigate the extent to which this is possible, and evaluate its utility. More specifically, the success criterion was to implement the two phases of the library specified in the proposal. I have not implemented all of the effects specified in phase 2, as I revised the phases in order to build a more interesting example program. However, the overall capabilities of the revised phases well exceed the original ones, and I have completed these along with a portion of the extension work on recursion.

The iterative development model worked well. As the project was an investigation, defining the development phases upfront helped to constrain its scope, and helped me to remain on time by fixing dates by which each phase should be completed. Additionally, I chose the iterative model specifically because it allowed regular

evaluation as the project was completed, and thus reassessment of what work should be completed next. This turned out to be vital during the project, as it allowed me to realise that the initial plan did not focus on the most interesting areas during the evaluation of phase 1, and thus adjust phase 2 and the extension.

Chapter 5

Conclusions

The evidence in the evaluation confirms that code migration, when possible, is a useful technique which has several advantages when writing distributed programs. Compared to other techniques such as message passing, code migration results in programs which are more concise and readable, increasing developer productivity and reducing the likelihood of bugs. As I expected, the algebraic effect handlers approach to code migration is also convenient because it allows migration to be added using a library to an existing, statically typed language. I knew that this approach would have several limitations, which might mean it was not feasible to use in practice, but I have successfully to worked around these to produce a working library. The abstract value system allows effects to return types with many possible values, which is essential for writing more realistic systems. I also implemented one recursion optimisation, which allows efficient iteration over a list, and investigated how full recursion might be optimised. This goes some way to solving the limitations of the approach.

While the current implementation of the library is not ready for real-world use, as I suggested in the evaluation I believe with a little more work it could satisfy a niche. I would like to implement the full recursion optimisation and change to a faster serialisation mechanism to see how close this brings the performance of the migrating example to the client/server version. It would then be possible to evaluate which applications the library is suitable for.

The project was challenging because it involved learning about several advanced functional programming techniques and then integrating them with the mechanics of a real-world distributed system. I have successfully achieved this to produce a library which exceeds the success criteria and extension work set out in the original proposal. It also satisfies the revised success criteria, and implements a portion of the extension work on recursion. This allowed me to evaluate algebraic effect based migration as a technique. I enjoyed learning about effect handlers and the other advanced functional programming techniques which I used, and I hope to be able to apply them to other work in the future.

Bibliography

[1] *Nomadic Pict*. URL: http://www.cs.put.poznan.pl/pawelw/npict/ (visited on 02/20/2016).

[2] Eijiro Sumii. "An implementation of transparent migration on standard Scheme". In: *Proceedings of the Workshop on Scheme and Functional Programming, Technical Report 00-368, Rice University*. 2000, pp. 61–64. URL: http://repository.readscheme.org/ftp/papers/sw2000/sumii.pdf.

[3] *Haskell Language*. URL: https://www.haskell.org (visited on 02/29/2016).

[4] Miran Lipovača. *Learn you a Haskell for great good! : a beginners guide*. San Francisco, Calif: No Starch Press, 2011. ISBN: 9781593272838 (pbk).

[5] Simon Marlow et al. *Haskell 2010 language report*. URL: https://www.haskell.org/definition/haskell2010.pdf (visited on 03/09/2016).

[6] Lawrence C Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[7] Matija Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper". In: *Electronic Notes in Theoretical Computer Science* 319 (2015), pp. 19–35.

[8] Roger S Pressman. *Software engineering : a practitioners approach*. Eight edition / Roger S. Pressman, Ph.D., Bruce R. Maxim, Ph.D. New York, NY: McGraw Hill Education, 2015. ISBN: 9781259253157.

[9] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in action". In: *ACM SIGPLAN Notices*. Vol. 48. 9. ACM. 2013, pp. 145–158.

[10] *Haskell Base package documentation*. URL: https://hackage.haskell.org/package/base-4.8.2.0/docs/ (visited on 04/19/2016).

[11] *Network.Simple.TCP*. URL: https://hackage.haskell.org/package/network-simple-0.4.0.4/docs/Network-Simple-TCP.html (visited on 03/11/2016).

[12] Mark P Jones. "Type classes with functional dependencies". In: *Programming Languages and Systems*. Springer, 2000, pp. 230–244.

[13] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.4*. URL: https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/index.html (visited on 03/15/2016).

[14] *Haskell Wiki: Heterogenous collections*. URL: https://wiki.haskell.org/Heterogenous_collections (visited on 01/19/2016).

[15] *Haskell Wiki: Phantom type*. URL: https://wiki.haskell.org/Phantom_type (visited on 01/19/2016).

[16] *Haskell programming tips*. URL: https://wiki.haskell.org/Haskell_programming_tips (visited on 03/09/2016).

[17] *cloc*. URL: https://github.com/AlDanial/cloc (visited on 03/08/2016).

Appendix A

Monads

Monads are a category theoretic concept which are used in functional programming to represent sequential computations. Monads allow a sequence of actions to be chained together, where the output from one step may be modified in some way before being fed into the next step.

For example [4], this is useful if we are performing a sequence of actions where any of the actions may fail. If an action succeeds, its output should be the input to the next action, if it fails the entire sequence should fail. The concept of a value either containing some data, or representing failure, can be represented in Haskell using the built-in `Maybe` data type:

```
data Maybe a = Just a | Nothing
```

A function returning `Maybe` can return `Just x`, where `x` is the data, if the operation succeeded, or `Nothing` if the operation failed. However, if we perform two operations is sequence, where the output of the first operation is fed into the second, it is necessary for the second to check if the first failed, and if so fail also. This quickly becomes unwieldy. For example, we might have functions `credit` and `debit` which act on a bank account, where `debit` fails if the balance falls below zero:

```
type Balance = Int

credit :: Int -> Balance -> Int -> Maybe Balance
credit x balance = Just (balance + x)

debit :: Int -> Balance -> Maybe Balance
debit x balance =
  let balance' = balance - x
  in if balance' >= 0 then Just balance'
                      else Nothing
```

We could perform several transactions in a sequence:

```
doTransactions Balance -> Maybe Balance
doTransactions balance =
  let balance' = debit 10 balance
  in case balance' of
    Nothing -> Nothing
    Just x ->
      let balance'' = debit 5 x
      in case balance'' of
        Nothing -> Nothing
        Just x -> credit 3 x
```

Clearly, this quickly becomes difficult to read.

The code above has the form of a sequence of actions, in each case either a credit or a debit. The actions are joined together by some logic, the case matches, which either calls the next action, or aborts the sequence if an action has failed. Similar sequences of actions often occur. For example, a computation might want to maintain some state, perhaps a random number generator. In this case, each action receives the state and is able to modify it. The modified state is then passed onto the next action.

Monads generalise this idea of sequences of actions bound together. In Haskell, monads are defined by a typeclass:

```
class Monad m where
  return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

>>= is pronounced "bind". This is the function which allows us to chain operations together as above. Bind takes a monadic value, performs the given action on it and produces a new monadic value. The action is represented by the function `(a -> m b)`. We can use bind to replace the explicit logic chaining actions together, as demonstrated below.

In addition to bind, the other important function defined by the monad typeclass is `return`. Despite its name, `return` is not related to the return statements in imperative languages. As actions in a sequence will operate on monadic values, `return` is used to convert a value to the corresponding monadic value.

As suggested above, with an appropriate implementation of >>= it would be possible to sequence the set of bank account actions much more succinctly. The `Maybe` monad in the standard library provides this:

```
instance Monad Maybe where
  return           = Just

  (Just x) >>= k   = k x
  Nothing  >>= _   = Nothing
```

If the previous action produces a `Just` value, then bind unwraps the inner value and passes this to the next action. Otherwise, if the previous action failed with `Nothing` then bind ignores the next action and just fails. `return` wraps any value `x` as `Just x`. Using the `Maybe` monad we can rewrite `doTransactions` as follows:

```
doTransactions balance =
  let balance' = return balance
  in balance' >>= debit 10 >>= debit 5 >>= credit 3
```

This is much easier to read, as the bind operator clearly sequences the actions.

However, explicitly using bind can still become difficult to read, especially when monadic values must be frequently unwrapped using pattern matches. For this reason, Haskell provides syntactic sugar, `do` notation. In a `do` block, every line is a monadic action, and the block takes the value of the expression on the final line. Once de-sugared, the actions are sequenced together using >>=, but the value of the previous action is discarded. If the value of an action is required in a following action, we use the operator <-. This unwraps a monadic value on the right of the operator, and makes it available to following actions under as the value given on the left.

Rewriting `doTransactions` using `do` notation does not result in better readability. However, take the following example using the `IO` monad, which allows effectful input and output to be performed. It asks the user to enter their first and second name, and then greets them:

```
greet =
  putStrLn "Enter first" >>
  getLine >>=
  (\first -> putStrLn "Enter second" >>
  getLine >>=
  (\second -> putStrLn $ "Hi " ++ first ++ " " ++ second))
```

This is much easier to read when written using `do` notation:

```
greet = do
  putStrLn "Enter your first name:"
  first <- getLine
  putStrLn "Enter your second name:"
  second <- getLine
  putStrLn $ "Hi " ++ first ++ " " ++ second
```

Monad laws

For a data structure to be considered a monad, in addition to having an instance of the monad typeclass it must also satisfy the monad laws [5, section 13.1]. These

laws allow us to reason about how monads will behave.

Left identity `return a >>= k  ==  k a`
: Applying bind to a value wrapped using `return` and a function is the equivalent to applying the function directly to the value.

Right identity `m >>= return  ==  m`
: Applying bind to a monadic value and `return` is equivalent to the monadic value on its own.

Associativity `m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h`
: The nesting of bind operations does not affect the result.

Appendix B

Migration library

Migration.hs

```
1  {-# LANGUAGE TypeFamilies,
2      GADTs,
3      RankNTypes,
4      MultiParamTypeClasses,
5      QuasiQuotes,
6      FlexibleInstances,
7      FlexibleContexts,
8      OverlappingInstances,
9      UndecidableInstances,
10     ConstraintKinds,
11     OverloadedLists,
12     OverloadedStrings,
13     StandaloneDeriving,
14     FunctionalDependencies #-}
15
16 module Migration where
17
18 import Prelude hiding ((&&), (||), not, iterate)
19 import Handlers
20 import DesugarHandlers
21 import Network.Simple.TCP (connect, serve, HostPreference(Host),
       ↪ HostName)
22 import Network.Socket (send, socketToHandle)
23 import System.IO
24 import System.Directory (getDirectoryContents)
25 import GHC.IO.Handle
26 import qualified Data.Map.Strict as Map
27 import Data.List hiding (iterate)
28 import Data.String
29 import Data.Boolean
30 import Data.Foldable (traverse_)
31 import GHC.Exts (IsList(Item, fromList, toList))
32
33
34 -- Store.
35 data StoreKey a = StoreKey Int deriving (Show, Read)
36 type GenericStoreKey = Int
```

```
37 data StoreValue = StoreIntValue Int
38                 | StoreBoolValue Bool
39                 | StoreCharValue Char
40                 | StoreBoolListValue [Bool]
41                 | StoreStringValue String
42                 | StoreAbsStringValue AbsString
43                 | StoreStringListValue [String]
44                 | StoreAbsStringListValue [AbsString]
45     deriving (Show, Read)
46 type Store = Map.Map GenericStoreKey StoreValue
47
48 emptyStore :: Store
49 emptyStore = Map.empty
50
51 genericSave :: Store -> StoreKey a -> StoreValue -> Store
52 genericSave store (StoreKey k) x = Map.insert k x store
53
54 genericRetrieve :: Store -> StoreKey a -> StoreValue
55 genericRetrieve store (StoreKey k) =
56     let v = Map.lookup k store in
57         case v of Just u  -> u
58                   Nothing -> error "No value associated with store
                       ↪ location"
59
60 class Storeable a where
61     save :: Store -> StoreKey a -> a -> Store
62     retrieve :: Store -> StoreKey a -> a
63
64 instance Storeable Int where
65     save store k x = genericSave store k (StoreIntValue x)
66     retrieve store k =
67         let v = genericRetrieve store k
68         in case v of StoreIntValue x -> x
69                      _ -> error "Wrong type in store, expected Int"
70
71 instance Storeable Bool where
72     save store k x = genericSave store k (StoreBoolValue x)
73     retrieve store k =
74         let v = genericRetrieve store k
75         in case v of StoreBoolValue x -> x
76                      _ -> error "Wrong type in store, expected Bool"
77
78 instance Storeable Char where
79     save store k x = genericSave store k (StoreCharValue x)
80     retrieve store k =
81         let v = genericRetrieve store k
82         in case v of StoreCharValue x -> x
83                      _ -> error "Wrong type in store, expected Bool"
84
85 instance Storeable [Bool] where
86     save store k x = genericSave store k (StoreBoolListValue x)
87     retrieve store k =
88         let v = genericRetrieve store k
89         in case v of StoreBoolListValue x -> x
90                      _ -> error "Wrong type in store, expected [Bool]
                          ↪ "
91
92 instance Storeable String where
```

```
93      save store k x = genericSave store k (StoreStringValue x)
94      retrieve store k =
95          let v = genericRetrieve store k
96          in case v of StoreStringValue x -> x
97                       _ -> error "Wrong type in store, expected String
                             ↪ "
98
99  instance Storeable AbsString where
100     save store k x = genericSave store k (StoreAbsStringValue x)
101     retrieve store k =
102         let v = genericRetrieve store k
103         in case v of StoreAbsStringValue x -> x
104                      _ -> error "Wrong type in store, expected
                             ↪ AbsString"
105
106 instance Storeable [String] where
107     save store k x = genericSave store k (StoreStringListValue x)
108     retrieve store k =
109         let v = genericRetrieve store k
110         in case v of StoreStringListValue x -> x
111                      _ -> error "Wrong type in store, expected [
                             ↪ String]"
112
113 instance Storeable [AbsString] where
114     save store k x = genericSave store k (StoreAbsStringListValue x)
115     retrieve store k =
116         let v = genericRetrieve store k
117         in case v of StoreAbsStringListValue x -> x
118                      _ -> error "Wrong type in store, expected [
                             ↪ AbsString]"
119
120 -- Abstract values.
121 class Abstract a c | a -> c where
122     eval :: Store -> a -> c
123     toAbs :: c -> a
124
125 -- While these types are not abstract, it is helpful to map them to
    ↪ themselves so we can eval them.
126 instance Abstract Int Int where
127     eval _ x = x
128     toAbs x = x
129
130
131 -- Abstract Int.
132 data AbsInt = IntVal Int
133             | IntVar (StoreKey Int)
134             | OpPlus AbsInt AbsInt
135             | OpMinus AbsInt AbsInt
136             | OpMult AbsInt AbsInt
137             | OpSig AbsInt
138     deriving (Show, Read)
139
140 instance Num AbsInt where
141     (IntVal x) + (IntVal y) = IntVal (x + y)
142     x + y = OpPlus x y
143     (IntVal x) - (IntVal y) = IntVal (x - y)
144     x - y = OpMinus x y
145     (IntVal x) * (IntVal y) = IntVal (x * y)
```

```
146     x * y = OpMult x y
147     negate x = (IntVal 0) - x
148     signum x = OpSig x
149     abs x = x * (signum x)
150     fromInteger x = IntVal (fromInteger x)
151
152 instance Abstract AbsInt Int where
153     eval store (IntVal    x) = x
154     eval store (IntVar    k) = retrieve store k
155     eval store (OpPlus  x y) = (eval store x) + (eval store y)
156     eval store (OpMinus x y) = (eval store x) - (eval store y)
157     eval store (OpMult  x y) = (eval store x) * (eval store y)
158     eval store (OpSig     x) = signum $ eval store x
159     toAbs x = IntVal x
160
161
162 -- Abstract Bool.
163 data AbsBool = BoolVal Bool
164              | BoolVar (StoreKey Bool)
165              | And AbsBool AbsBool
166              | Or AbsBool AbsBool
167              | Not AbsBool
168     deriving (Show, Read)
169
170 instance BoolValue AbsBool where
171     true = BoolVal True
172     false = BoolVal False
173
174 instance Boolean AbsBool where
175     (&&) x y = And x y
176     (||) x y = Or x y
177     not x = Not x
178
179 instance Abstract AbsBool Bool where
180     eval store (BoolVal x) = x
181     eval store (BoolVar k) = retrieve store k
182     eval store (And   x y) = (eval store x) && (eval store y)
183     eval store (Or    x y) = (eval store x) || (eval store y)
184     eval store (Not     x) = not (eval store x)
185     toAbs x = BoolVal x
186
187
188 -- Abstract Char.
189 data AbsChar = CharVal Char
190              | CharVar (StoreKey Char)
191     deriving (Show, Read)
192
193 instance Abstract AbsChar Char where
194     eval store (CharVal x) = x
195     eval store (CharVar k) = retrieve store k
196     toAbs x = CharVal x
197
198
199 -- Abstract lists.
200 data AbsList a = Nil
201                | ListVal [a]
202                | ListVar (StoreKey [a])
203                | Cons a (AbsList a)
```

```
                | Append (AbsList a) (AbsList a)
                | Take Int (AbsList a)
                | Drop Int (AbsList a)
                | Tail (AbsList a)
deriving instance Show a => Show (AbsList a)
deriving instance Read a => Read (AbsList a)

type AbsString = AbsList Char

instance IsList (AbsList a) where
    type Item (AbsList a) = a
    fromList [] = Nil
    fromList xs = ListVal xs
    toList xs = error "Cannot convert from AbsList to [] without
        ↪ store."

instance IsString AbsString where
    fromString s = fromList s

instance (Storeable [a]) => Abstract (AbsList a) [a] where
    eval store           Nil = []
    eval store (ListVal  xs) = xs
    eval store (ListVar   k) = retrieve store k
    eval store (Cons   x xs) = x : (eval store xs)
    eval store (Append  x y) = (eval store x) ++ (eval store y)
    eval store (Take   n xs) = take n (eval store xs)
    eval store (Drop   n xs) = drop n (eval store xs)
    eval store (Tail     xs) = let xs' = eval store xs
                                  in case xs' of [] -> []
                                                 (y:ys) -> ys
    toAbs x = ListVal x

acons :: a -> AbsList a -> AbsList a
acons x xs = Cons x xs

(+++) :: AbsList a -> AbsList a -> AbsList a
(+++) x y = Append x y

atake :: Int -> AbsList a -> AbsList a
atake n xs = Take n xs

adrop :: Int -> AbsList a -> AbsList a
adrop n xs = Drop n xs

tl :: AbsList a -> AbsList a
tl (ListVal []) = ListVal []
tl (ListVal (x:xs)) = ListVal xs
tl xs = Tail xs


-- Abstract iteration
-- Ideally forEach would be an effect of type
-- (a -> MigrationComp b) -> AbsList a -> MigrationComp b
-- However, the effect handlers system does not allow polymorphic
    ↪ effect types, nor can you pass an
-- effect a function. Hence forEach reifies the function, and passes
    ↪ the computation tree to the
-- Iterate effect.
```

```
259 forEach :: (AbsString -> MigrationComp ()) -> AbsList AbsString ->
      ↪ MigrationComp ()
260 forEach f xs = do
261     fresh <- freshVar
262     let k = StoreKey fresh
263         rf = reifyComp (fresh*10000) (f (ListVar k))
264     iterate (rf, xs, k)
265
266 -- forEvery is a helper function called by the execution function to
      ↪ execute the Iterate effect.
267 forEvery :: UnitCompTree -> [AbsString] -> Store -> StoreKey String
      ↪ -> IO ()
268 forEvery f [] store k = return ()
269 forEvery f (x:xs) store k = do
270     let str = eval store x
271         store' = save store k str
272     executeCompTree NoAuth (store', f)
273     forEvery f xs store k
274
275
276 -- Abstract Show.
277 class AbsShow a where
278     ashow :: Store -> a -> String
279
280 instance AbsShow AbsInt where
281     ashow store x = show $ eval store x
282
283 instance AbsShow AbsBool where
284     ashow store x = show $ eval store x
285
286 instance AbsShow AbsString where
287     ashow store x = eval store x
288
289 instance (Storeable [a], Show a) => AbsShow (AbsList a) where
290     ashow store x = show $ eval store x
291
292 instance AbsShow Int where
293     ashow store x = show x
294
295
296 -- Abstract Eq.
297 class AbsEq a where
298     (===) :: a -> a -> MigrationComp Bool
299
300 instance AbsEq AbsInt where
301     (===) x y = equal (EqableAbsInt x, EqableAbsInt y)
302
303 instance AbsEq AbsBool where
304     (===) x y = equal (EqableAbsBool x, EqableAbsBool y)
305
306 instance AbsEq AbsChar where
307     (===) x y = equal (EqableAbsChar x, EqableAbsChar y)
308
309 instance AbsEq Int where
310     (===) x y = return $ x==y
311
312 instance AbsEq AbsString where
313     (===) x y = equal (EqableAbsString x, EqableAbsString y)
```

```
314
315 data AbsEqable = EqableAbsInt AbsInt
316                | EqableAbsBool AbsBool
317                | EqableAbsChar AbsChar
318                | EqableAbsString AbsString
319     deriving (Show, Read)
320
321 evalAbsEqable :: Store -> (AbsEqable,AbsEqable) -> Bool
322 evalAbsEqable store (EqableAbsInt x, EqableAbsInt y) = (eval store x)
    ↪  == (eval store y)
323 evalAbsEqable store (EqableAbsBool x, EqableAbsBool y) = (eval store
    ↪ x) == (eval store y)
324 evalAbsEqable store (EqableAbsChar x, EqableAbsChar y) = (eval store
    ↪ x) == (eval store y)
325 evalAbsEqable store (EqableAbsString x, EqableAbsString y) = (eval
    ↪ store x) == (eval store y)
326 evalAbsEqable store (_,_) = error "Mismatched Eqable constructors"
327
328
329 -- Computation tree.
330 data CompTree a = Result a
331     | MigrateEffect (AbsHostName, AbsPort) (CompTree a)
332     | AMigrateEffect (AbsString, AbsHostName, AbsPort) (CompTree a)
333     | PrintStrEffect AbsString (CompTree a)
334     | PrintStrListEffect (AbsList AbsString) (CompTree a)
335     | PrintIntEffect AbsInt (CompTree a)
336     | ReadStrEffect (StoreKey [Char]) (CompTree a)
337     | ReadIntEffect (StoreKey Int) (CompTree a)
338     | AskEffect AbsString (CompTree a) (CompTree a)
339     | ReadFlEffect AbsString (StoreKey [Char]) (CompTree a)
340     | ListFlsEffect (StoreKey [AbsString]) (CompTree a)
341     | EqualEffect (AbsEqable, AbsEqable) (CompTree a) (CompTree a)
342     | IterateEffect (CompTree (), AbsList AbsString, StoreKey String)
        ↪  (CompTree a)
343     | HdEffect (AbsList AbsString) (StoreKey String) (CompTree a) (
        ↪ CompTree a)
344     deriving (Show, Read)
345 -- UnitCompTree sometimes has to be used in place of CompTree (),
    ↪ this needs to be investigated.
346 type UnitCompTree = CompTree ()
347
348
349 -- Handlers and reification.
350 [operation|Migrate      :: (AbsHostName, AbsPort) -> ()|]
351 [operation|AMigrate     :: (AbsString, AbsHostName, AbsPort) -> ()|]
352 [operation|PrintStr     :: AbsString -> ()|]
353 [operation|PrintStrList :: AbsList AbsString -> ()|]
354 [operation|PrintInt     :: AbsInt -> ()|]
355 [operation|ReadStr      :: AbsString|]
356 [operation|ReadInt      :: AbsInt|]
357 [operation|ReadFl       :: AbsString -> AbsString|]
358 [operation|Ask          :: AbsString -> Bool|]
359 [operation|ListFls      :: AbsList AbsString|]
360 [operation|Equal        :: (AbsEqable, AbsEqable) -> Bool|]
361 -- See the note next to forEach.
362 [operation|Iterate      :: (UnitCompTree, AbsList AbsString, StoreKey
    ↪  String) -> ()|]
363 [operation|Hd           :: AbsList AbsString -> Maybe AbsString|]
```

```
[operation|FreshVar     :: GenericStoreKey|]

type MigrationComp a = ([handles|h {Migrate, AMigrate, PrintStr,
    ↪ PrintStrList, PrintInt, ReadStr,
                                    ReadInt, Ask, ReadFl, ListFls,
                                        ↪ Equal, Iterate, Hd,
                                    FreshVar}|])
                         => Comp h a

[handler|
    ReifyComp a :: GenericStoreKey -> CompTree a
        handles {Migrate, AMigrate, PrintStr, PrintStrList, PrintInt,
            ↪  ReadStr, ReadInt, Ask,
                 ReadFl, ListFls, Equal, Iterate, Hd, FreshVar} where
            Return            x i -> Result x
            Migrate    (h, p) k i -> MigrateEffect (h, p) (k () i)
            AMigrate  (a,h,p) k i -> AMigrateEffect (a,h,p) (k () i)
            PrintStr      str k i -> PrintStrEffect str (k () i)
            PrintStrList strs k i -> PrintStrListEffect strs (k () i)
            PrintInt        x k i -> PrintIntEffect x (k () i)
            ReadStr           k i ->
                let key = StoreKey i
                in ReadStrEffect key (k (ListVar key) (i+1))
            ReadInt           k i ->
                let key = StoreKey i
                in ReadIntEffect key (k (IntVar key) (i+1))
            Ask             q k i -> AskEffect q (k True i) (k False
                ↪ i)
            ReadFl       file k i ->
                let key = StoreKey i
                in ReadFlEffect file key (k (ListVar key) (i+1))
            ListFls           k i ->
                let key = StoreKey i
                in ListFlsEffect key (k (ListVar key) (i+1))
            Equal       (x,y) k i -> EqualEffect (x,y) (k True i) (k
                ↪ False i)
            Iterate  (f,xs,x) k i -> IterateEffect (f,xs,x) (k () i)
            Hd             xs k i ->
                let key = StoreKey i
                    x = Just (ListVar key)
                in HdEffect xs key (k x (i+1)) (k Nothing i)
            FreshVar          k i -> k i (i+1)
|]


-- Networking.
type AbsHostName = AbsString
type Port = String
type AbsPort = AbsString
data AuthPhrase = NoAuth | Auth String

-- If authPhrase is provided, normal migration will fail, only
    ↪ aMigrate will work.
listenForComp :: Port -> AuthPhrase -> IO ()
listenForComp port authPhrase = do
    putStrLn "Listening for incoming connections..."
    serve (Host "127.0.0.1") port $ \(socket, remoteAddress) -> do
        handle <- socketToHandle socket ReadMode
```

```
416         hSetBuffering handle LineBuffering
417         str <- hGetLine handle
418         hClose handle
419         putStrLn "Received computation, running it"
420         let (store, comp) = (read str :: (Store, CompTree ()))
421         executeCompTree authPhrase (store, comp)
422         return ()
423
424 sendComp :: (Show a, Read a) => (HostName, Port) -> (Store, CompTree
    ↪ a) -> IO Int
425 sendComp (hostName, port) (store, comp) = do
426     connect hostName port $ \(socket, remoteAddress) -> do
427         putStrLn $ "Sending computation to " ++ hostName
428         send socket $ show (store, comp)
429
430
431 -- Interpreter.
432 executeCompTree :: (Show a, Read a) => AuthPhrase -> (Store, CompTree
    ↪  a) -> IO (Maybe (Store, a))
433 executeCompTree auth (store, effect) = case effect of
434     Result x -> return $ Just (store, x)
435     MigrateEffect (host, port) comp ->
436         -- If authentication has been provided, then normal migration
            ↪  is not allowed.
437         case auth of Auth _ -> return Nothing
438                      NoAuth -> do
439                         let host' = eval store host
440                             port' = eval store port
441                         sendComp (host', port') (store, comp)
442                         return Nothing
443     AMigrateEffect (authTry, host, port) comp ->
444         -- Don't allow migration if the two auth phrases don't match.
445         case auth of NoAuth -> return Nothing
446                      Auth phrase -> do
447                             let authTry' = eval store authTry
448                                 host' = eval store host
449                                 port' = eval store port
450                             if phrase == authTry' then do
451                                 sendComp (host', port') (store, comp)
452                                 return Nothing
453                             else return Nothing
454     PrintStrEffect str comp -> do
455         putStrLn $ ashow store str
456         executeCompTree auth (store, comp)
457     PrintStrListEffect strs comp -> do
458         let strs' = eval store strs
459         traverse_ (\str -> putStrLn $ ashow store str) strs'
460         executeCompTree auth (store, comp)
461     PrintIntEffect x comp -> do
462         putStrLn $ ashow store x
463         executeCompTree auth (store, comp)
464     ReadStrEffect k comp -> do
465         line <- getLine
466         let store' = save store k line
467         executeCompTree auth (store', comp)
468     ReadIntEffect k comp -> do
469         value <- readLn
470         let store' = save store k value
```

```
471         executeCompTree auth (store', comp)
472     ReadFlEffect file k comp -> do
473         let file' = eval store file
474         text <- readFile file'
475         let store' = save store k text
476         executeCompTree auth (store', comp)
477     ListFlsEffect k comp -> do
478         files <- getDirectoryContents "."
479         let absFiles = map (\s -> toAbs s) files
480         let store' = save store k absFiles
481         executeCompTree auth (store', comp)
482     EqualEffect (x,y) compt compf ->
483         if evalAbsEqable store (x,y) then executeCompTree auth (store
            ↪ , compt)
484                                      else executeCompTree auth (store
                                             ↪ , compf)
485     IterateEffect (f,xs,k) comp -> do
486         let xs' = eval store xs
487         forEvery f xs' store k
488         executeCompTree auth (store, comp)
489     HdEffect xs k compt compf ->
490         let xs' = eval store xs
491         in case xs' of [] -> executeCompTree auth (store, compf)
492                        (x:xs) -> do
493                            let x' = eval store x
494                                store' = save store k x'
495                            executeCompTree auth (store', compt)
496
497 runMigrationComp :: Port -> MigrationComp () -> IO ()
498 runMigrationComp port comp = do
499     let comp' = reifyComp 0 comp
500     executeCompTree NoAuth (emptyStore, comp')
501     listenForComp port NoAuth
502     return ()
```

Computer Science Part II Project Proposal

Implementing and evaluating algebraic effect based code migration.

Oscar Key, Corpus Christi College

Project supervisor: Ohad Kammar
Project overseers: Peter Sewell, Markus Kuhn
Director of Studies: David Greaves

[1481 words]

Introduction

For some applications it may be convenient to have a portion of the computation run on a different host from where it started. One way this can be achieved is by using transparent code migration, which allows the computation to move from one machine to another while it is executing. The programmer may issue statements such as “move [host]” to request that the execution of the computation move to the given host and continue there. Compared to other methods of distributed computing this may be easier for the programmer to understand and reduce the complexity of the source code. For example, many applications often involve a split between client and server: one program runs on the client machine and interacts with the user, while another runs on the server and interacts with the database. Instead, a single application could be built which could move between the client and the server, thus reducing the complexity for the programmer.

Full transparent code migration requires modification of the programming language runtime, but this is a difficult process due to the size and complexity of the runtime and may affect compatibility with existing programs. A more convenient approach might be to implement a partial migration using a library. My proposal is to investigate how much can be implemented without runtime support using *algebraic effects*, and evaluate the utility of such a library. I will build a limited code migration library for Haskell, and then demonstrate it by implementing an example distributed application. I will also write the same application using a client/server design, and the two will then be compared to evaluate the library.

Starting point

There are already several implementations of transparent code migration. In particular Sumii describes an implementation of a library using delimited continuations and TDPE to add support for limited

migration to Scheme.[1] This implementation relies on code-as-data, which is not possible in Haskell, but it will be used as a guide.

The project itself will be built using the standard Haskell hierarchical libraries. These will be augmented by an implementation of algebraic effect handlers in Haskell.[2]

I already have some experience with algebraic effects in OCaml. I will use Haskell instead for the project because the implementation of effects is more powerful. The project also requires that the language is pure in order to limit the interaction with the OS to a set of predefined effects.

Work to be done

Preparation

Read the papers listed as the starting point:

- "Handlers in Action" - O Kammar, S Lindley, N Oury
- "An Implementation of Transparent Migration on Standard Scheme" - Eijiro Sumii

Setup the development environment by installing Haskell and the needed libraries.

Become familiar with Haskell and the techniques required by reading the following:

- "Haskell for Ocaml Programmers" - Raphael 'kena' Poss[3]
- "Type-Directed Partial Evaluation" - Olivier Danvy[4]
- "An Introduction to Algebraic Effects and Handlers" - Matija Pretnar[5]

I will also write short practice programs using handlers.

Library

The library will use algebraic effects, which allow side effects to be emitted and handled outside of the main program flow, much like the behaviour of exceptions in many languages. As Haskell is a pure language, its type system can prevent programs from having any interaction with the OS other than through a set of predefined effects in the library. This means that the control flow of the program can only branch depending on the result of an algebraic effect, and the only observable effects of the program will be performed as algebraic effects, so the computation can be represented as a tree of effects. Each node in the tree is an effect that the program may raise and each child of the node corresponds to the return value of the effect. The leaves of the tree are the final return values of the program. A path down the tree represents the sequence of effects raised by the program, eventually leading to a particular return value. The library will create this tree, serialize it and move it between hosts.

The following will be required in the library:

Stage 1

- A custom "move" effect which can be raised by a program during its execution to request that the computation migrate to another host.

---

[1] "An Implementation of Transparent Migration on Standard Scheme" - Eijiro Sumii - http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/go.pdf

[2] Handlers in Action - O Kammar, S Lindley, N Oury - http://homepages.inf.ed.ac.uk/slindley/papers/handlers.pdf

[3] http://science.raphael.poss.name/haskell-for-ocaml-programmers.html

[4] http://www.cs.au.dk/~danvy/tdpe-ln.pdf

[5] http://events.cs.bham.ac.uk/mfps31/mfpsproc.pdf - p.272

When this is raised the effect handlers implementation will pass the handler in the library the delimited continuation of the program, which represents the remaining part of the computation yet to be executed. This will then be passed to the reification function.

- A handler to reify the continuation.
  This will explore every path the computation can take using type-directed partial evaluation, implemented in Haskell with type classes. When an effect is raised the continuation will be called several times, once for every possible return value of the effect. This will build the tree described above.
  In order to make this computationally feasible the library will be limited, at least initially, to types which can be represented with a few bits. These will be the unit, boolean and pair types.
- Serialization and deserialization functions for the reified continuation.
  This will allow the continuation tree to be passed through a socket.
- A function to continue the reified computation on the second host.
  This will move down the tree performing the effects, and choosing the appropriate subtree based on the return value of each effect.
- Network infrastructure to move the serialized data from one host to another.

Stage 2

- Other effects and handlers for common operations useful for applications using the library.
  To simplify the implementation I will fix the set of effects that may be used by the migrating program to those provided by the library. I will implement effects for exceptions, non-deterministic backtracking and storing/retrieving a single bit of state.

Extension

- Limited support for additional types, such as strings.

Example application

The example application is an attendance register, as might be used in a school, which will record which pupils are currently present. The attendance data is stored on a central server, and teachers can use the machines in their classrooms to view and update the data.

I will implement the application both using the migration library and also using a client/server design.

I will extend the application as I extend the capabilities of the library:

Stage 1

In the first instance the library will only support the move effect, so the application will pass a number from the classroom machine to the server, to indicate how many people are present in the class.

Stage 2

The application will have a predefined list of students, and will represent who is present as a list of boolean values which the teacher can update.

Extension

The application will be updated to support adding and removing students by name.

Comparison of approaches

The two implementations of the application will be compared, exploring:

- A qualitative comparison of the experience of writing each application, as recorded in the project logbook, for example the bugs encountered.
- The length of the source code of the two implementations.
- The performance of the two implementations.

Success criterion

The success criterion is to implement stage 1 of the library and example, and to compare the two applications.
Once this is completed I will implement stage 2 and the extension.

Timetable of work

1. **Michaelmas weeks 3 - 4**
   Preparatory reading, setting up the environment and familiarization with tools
2. **Michaelmas weeks 5 - 6**
   Implement stage 1 of the library
3. **Michaelmas weeks 7 - 8**
   Begin implementing stage 1 of the example applications
4. **Christmas break**
   Continue implementing stage 1 of the example applications
   Compare stage 1 of the example applications
   Plan the detail of how stage 2 of the example program will behave, and how it will be tested
   Learn about type classes
5. **Lent week 1 - 2**
   Write progress report, due Friday start of week 2 (29th Jan)
   Start raw draft material for implementation and background sections of the dissertation
6. **Lent weeks 3 - 4**
   Implement stage 2 of the library
7. **Lent week 5 - 6**
   Implement stage 2 of the example applications
8. **Lent weeks 7 - 8**
   Compare the applications
   Safety buffer or extension work
9. **Easter break**
   Start writing formal dissertation
10. **Easter weeks 1 - 3**
    Finalize dissertation, due Friday start of Easter week 4 (13th May)

Resources Required

I plan to use my own machine for the project. It is a dual core machine which runs Ubuntu Linux. I will use Git as a source control system, and backup the local copy of the repository to GitHub. The project will also be backed up to Dropbox. Should my computer fail I will easily be able to download the project from one of these services and continue it on MCS machines in either my college or the Computer Lab. No other special resources are required.