Distributing Computation in Haskell

Ian Connolly
B.A.(Mod.) Computer Science
Final Year Project April 2015
Supervisor: Dr. Glenn Strong

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

*Dublin, April 21, 2015*

______________________________
Ian Connolly, April 21, 2015
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_Dublin, April 21, 2015_

Ian Connolly, April 21, 2015
Abstract

Cloud Haskell is a framework for writing distributed concurrent applications in Haskell. I have built an Amazon Web Services transport backend library that allows for the running of Cloud Haskell applications on Amazon Web Services virtual machines. In addition, these libraries expose scaling primitives for the first time, allowing for changes in the size of virtual machine clusters in a Cloud Haskell system at run-time.
Acknowledgements

Firstly, thank you to my supervisor, Dr. Glenn Strong, for his unbounded help, advice, and support throughout the project.

Thank you to my family, and Lara, for their constant love, patience, and support.

Thank you to Miles McGuire, Andrew Anderson, Matthew Donnelly, Conor Brennan, Eoin Houlihan, and everyone else who helped with any aspect of this project and report.

Thank you to the Cloud Haskell team, especially Duncan Coutts, Tim Watson, Nicholas Wu, Edsko De Vries, and Jeff Epstein for their stellar work in advancing Haskell in the cloud.

Finally thank you, dear reader, for giving your time to read the culmination of many sleepless nights and tired mornings.
# Contents

1 Introduction .................................................. 3

2 Background .................................................. 5
   2.1 Haskell .................................................. 5
       2.1.1 Types .............................................. 6
       2.1.2 Purity .............................................. 9
       2.1.3 Laziness ........................................... 11
       2.1.4 Concurrency ....................................... 12
       2.1.5 Ecosystem and Tooling ......................... 13
   2.2 The Cloud ............................................... 15
       2.2.1 Amazon Web Services ............................ 15
       2.2.2 Alternatives ..................................... 16
   2.3 Cloud Haskell ........................................... 17
       2.3.1 Overview .......................................... 18
       2.3.2 Implementation ................................... 19

3 Design ....................................................... 21
   3.1 Existing Systems ....................................... 21
       3.1.1 distributed-process-azure ....................... 21
       3.1.2 azure-service-api ................................ 23
       3.1.3 Conclusions ....................................... 24
   3.2 Design Goals ........................................... 25
       3.2.1 Usability .......................................... 25
       3.2.2 Scalability ....................................... 25
       3.2.3 Ease of Installation ............................ 26
       3.2.4 Ease of Configuration ........................... 26

4 Implementation ............................................... 28
   4.1 Overview ............................................... 28
       4.1.1 aws-service-api .................................. 28
       4.1.2 distributed-process-aws ......................... 30
   4.2 Selected Details ....................................... 31
       4.2.1 AWS Interaction .................................. 31
Chapter 1

Introduction

Haskell is a pure, lazy, strongly-typed functional programming language. Haskell’s types and purity allow application constraints to be statically checked and allow many common classes of errors to be eliminated at compile-time. Its laziness and abstractions encourage a highly modular programming style that scales well with application complexity.

Cloud Haskell is a suite of Application Programming Interfaces (APIs) for distributed programming that brings Erlang/OTP-inspired message-passing concurrency to Haskell. The Cloud Haskell project aims to take the work the Erlang community has done in concurrency and combine it with the benefits given by Haskell’s types, purity, and abstractions.

Modern public clouds are democratising computational power by allowing many more people than before to run applications in a highly distributed environment. Amazon Web Services is the largest public cloud provider in the world, and their Elastic Compute Cloud allows programmers to launch and run many virtual machine instances programmatically.

Thus I propose to build the following:

- An Amazon Web Services (AWS) Service Management library that allows for the creation, scaling, and management of Cloud Services and Virtual Machines on AWS Elastic Compute Cloud (EC2) in Cloud Haskell applications.
- A Cloud Haskell wrapper of this service management library that exposes its functionality to Cloud Haskell applications.

The goal for this project is to build the required libraries to allow current Cloud Haskell applications on AWS, and to expose new functionality over the existing state-of-the-art that allows Cloud Haskell programmers to create and alter the Cloud Service their application is executing on programmatically, at run-time.

The rest of this report is structured as follows:
Distributing Computation in Haskell

Background This chapter will explain the technologies (Haskell, the Cloud, and Cloud Haskell) that this project will work with, specifically their characteristics that are relevant to this project that may be unfamiliar to the reader.

Design This chapter will examine the design of the existing state-of-the-art in an attempt to identify elements that are important to the design of the project. It will also set out a number of design goals that the project aims to accomplish.

Implementation This chapter will provide an overview of the completed project. It will examine certain interesting aspects of the project in further detail. Finally it will highlight some challenges faced in the completion of the project.

Results This chapter will show some examples of the project in use, and assess the project’s success in meeting its design goals and its usefulness to the wider Cloud Haskell ecosystem.

Conclusion Finally, we will conclude with some reflection on the project as a whole.
Chapter 2

Background

This chapter will explain the technologies (Haskell, the Cloud, and Cloud Haskell) that this project will work with, specifically their characteristics that are relevant to this project that may be unfamiliar to the reader.

2.1 Haskell

Haskell is a lazy, polymorphically statically typed, purely functional programming language. It is named for Haskell Curry, whose work in mathematical logic is foundational in functional programming languages. Haskell has an expressive syntax, an innovative type system, and is commonly regarded as both concise and readable.

In the Haskell community, the example of choice when introducing the language is an implementation of the sorting algorithm QuickSort[16].

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort smaller ++ [x] ++ quicksort bigger
    where smaller = [a | a <- xs, a <= x]
         bigger  = [a | a <- xs, a > x]
```

This example showcases a number of features of Haskell including its polymorphic type system, pattern-matching, and list comprehensions.

The type signature of the function

```haskell
quicksort :: (Ord a) => [a] -> [a]
```

tells us that this is a function from a list of type a to a new list of type a, where the type a is a member of the Ord type-class. We can think of type-classes as being similar to Interfaces in the Java programming language (indeed, they are the work of the same people). The Ord type-class contains all built-in types that can be ordered.
The rest of the function’s implementation isn’t important to understanding this project, so we will go no further in working through it. But having seen how we can express a function’s constraints succinctly at the type-level, let us now discuss Haskell’s type-system more thoroughly.

2.1.1 Types

We said earlier that Haskell is a polymorphically statically typed programming language. In Haskell, ‘polymorphism’ refers to the idea of parametric polymorphism. This means functions or data-types in Haskell can be written generically so that it can handle values of any type identically. The QuickSort implementation we saw earlier was an example of this. It worked on values of type \( a \), constrained to members of the type-class Ord. This means that that function will work on any list of type \( a \) that is a member of Ord. Concretely, we can imagine it working on both a [Int] and a [Char].

When we say Haskell is statically typed, we mean that the compiler will check our program for type errors at compile-time. Static typing has a reputation for being verbose, indeed much of the reason for the rise in popularity of languages such as Ruby and Python was the speed of development their dynamic typing apparently allowed.

As we see above, Haskell’s parametric polymorphism helps minimise this verbosity by allowing us to write the most generally typed functions possible. Another Haskell feature, Type Inference, also helps in this regard. The Haskell compiler implements an extended version of Hindley-Milner type inference. This allows the compiler to work out the most general type for a function or value that hasn’t been annotated by the programmer.

To re-use our QuickSort example again, we could actually write its implementation without its type signature, and the compiler would be able to work out its type signature without issue. Haskell’s type inference algorithm is quite advanced, and only in very rare cases will a programmer need to annotate a value with its types. It is, however, good practice to provide the types of the top-level functions in a program.

New datatypes in Haskell can be created with the data keyword.

```
data Bool = True | False
```

The above can be read as: ‘the type Bool is constructed by the data constructors True or False’.

Both the type constructors like Bool and the data constructors like True can have arguments, and these can be polymorphic. Here’s an example of the Maybe type which is used for modelling the idea of a value existing or not existing.

```
data Maybe a = Just a | Nothing
```
We can imagine using the above to model a function that does division by zero:

```haskell
safeDiv :: (Integral a) => a -> a -> Maybe a
safeDiv _ 0 = Nothing
safeDiv x y = Just (div x y)
```

Now we model failure at the type-level rather than throwing a run-time exception.

Our data constructors can have lots of parameters if we want a type to contain lots of information:

```haskell
data Person = Person
  String -- First name
  String -- Surname
  Int -- Age
  String -- Street
  String -- City
```

But this can be unwieldy to use because we have to do something like this to get at the age of the `Person`.

```haskell
getAge :: Person -> Int
Person _ _ _ _ age _ _ = age
```

And so on for each field. Fortunately, Haskell has a special syntax that will create getters like this for us, called `record syntax`:

```haskell
data Person = Person {
  firstname :: String,
  surname :: String,
  age :: Int,
  street :: String,
  city :: String
}
```

Now we have accessor functions like `firstname :: Person -> String` available.

Haskell programs commonly use record types like the above, just as C programs use structs.[7]

This advanced type-system provides a couple of advantages to the programmer. First and foremost, the compiler is able to eliminate a whole class of errors at compile-time. In fact, much of programming in Haskell boils down to teasing out the types the functions in your program will work on, to try and maximise the amount of logic you express at the type-level.
This leads to the type-system’s second advantage, its expressiveness. Much of a Haskell program’s functionality can be deduced from its type definitions. Throughout this report we will rarely show code that isn’t either a data-type definition or a function type-signature. We can do this, because functions in Haskell do not have any side-effects. That is, a function \( f :: a \rightarrow b \) is a mapping from type \( a \) to \( b \) and has no observable side-effect in the program or exterior to the program (ie. no input/output). This is called \textbf{purity}, and is enforced by the Haskell compiler.
2.1.2 Purity

Let us now discuss what we mean when we say Haskell is a pure programming language.

We can define the notion of a function being pure as follows:

- Given the same arguments, the function will always evaluate to the same value. This means the function cannot read any state not explicitly passed to it as an argument, nor can it obtain input to the program from external sources.

- All the function does is evaluate to its value. Only that value is visible outside of that function. No global state is updated by the function, and no external output is performed.

These constraints lead us to determine that functions in Haskell are mathematical functions, a pure mapping from input to output. This is distinct from what imperative languages such as C have defined as functions, which are not mappings but instead scoped subroutines. As noted before, this leads us to structure Haskell programs into types, and mappings between those types. Haskell programs tend to be structured as a pipeline of these mappings from type to type. These programs are easy to reason about, as all transformations are explicit and there is no secret state being held.[12]

However, the world is a stateful place. Moreover, a programming language cannot actually do anything if it can not perform input or output. This problem led to the novel solution of Monads.[28]

A monad is a structure that abstracts the sequencing of steps of computation. The State monad abstracts the boilerplate of threading input and output state parameters through a pipeline of functions. There is special syntactic sugar called do-notation that allows us to write code that looks almost imperative, but is still purely functional. It simply desugars down to a long chain of anonymous functions combined with the monadic bind operator >>=.

There is a special monad in Haskell called IO. The IO monad allows us to sequence IO actions. IO actions are given special treatment by the Haskell run-time and are allowed interact with the world. No input/output is allowed except for within an IO action, and so pure and impure code is separated and this separation enforced at the type-level by the compiler. The Haskell run-time looks for a special IO action called main when it runs a Haskell program, and it is the entry point for a Haskell program. Thus a Haskell program consists of a small IO monadic imperative core which calls a number of pure functions before doing some output and exiting. An example of this would be:

```haskell
import Data.Map
```
Distributing Computation in Haskell

```haskell
main :: IO ()
main = do
    data <- readFile "dataset.txt"
    let results = analyseWordFrequency data -- pure function
    print $ toList results -- list of (word, frequency) tuples

analyseWordFrequency :: String -> Map String Float -- pure function
analyseWordFrequency =

    We can imagine that `analyseWordFrequency` is the entry and exit point of a large pipeline of pure data transformation functions. We can see that Haskell enforces the separation of pure and impure code.

    To understand this project, it is sufficient to understand that Haskell’s purity encourages a modular, top-down program style and enforces a separation between pure and impure code. Haskell’s monads are also more widely useful than for just I/O, and are a useful abstraction for more generally managing state, or for wrapping and sequencing any sort of computation.

    Haskell did not set out to be a pure programming language. Instead it set out to be a lazy programming language, and the idea of laziness kept the language pure, out of necessity.[15]
```
2.1.3 Laziness

Earlier we describe Haskell as a lazy programming language. Moreover, we said that Haskell’s purity is because of its laziness. We shall now discuss this briefly in more detail.

Haskell is more correctly described as a non-strict programming language, of which lazy evaluation is just one implementation\[15\]. Lazy evaluation is a combination of call-by-need semantics and sharing. What call-by-need means specifically is that functions are only evaluated when they are needed, and only to the extent they are needed. This means that we can create infinite data structures, and work on them as if they were finite. Consider the following case of a function that returns a list of the first $N$ numbers:

```haskell
firstN :: Int -> [Int]
firstN n = take n [1..] -- [1, 2, 3, 4, 5, 6 ,...]
```

Haskell’s laziness allows us to simply generate an infinite list of integers and take the first $N$ of them. It should be noted here that we are not only generating an infinite list, but we are passing that infinite list to a function - take. take is a recursive function that collects the head of the list it is passed, and recurses with the list less its head, and its passed $n$ minus 1. As long as take defines an appropriate base case to its recursion, it will evaluate normally.

More generally, this means we can pass multiple large data structures around as parameters to our functions, and Haskell will only evaluate those data structures that we need, only to the extent we need them, and only when we need them.

What sharing means is that while there may be many references to data structures throughout the program, they will all point to the same internal data structure. This means that if one of those references requires an evaluation of the data structure then all the references will point to the evaluated value. This means that large computations are re-used and costly re-computation is kept to a minimum.

Laziness requires purity for somewhat obvious reasons. If we do not know when our program will evaluate any value fully, then we do not know what order the values will be evaluated in. This means that in a lazy language, the ordering of side-effects is indeterminable. A programmer can not evaluate a side-effectful program using lazy semantics and know what the program will do. Thus, Haskell banishes I/O to the IO monad.

Laziness is a huge boon when writing programs, first and foremost for its optimising properties. Sharing gives all the benefits of memoization in an imperative language, without any of the code annotation. Secondly, laziness allows us to reason about large and even infinite data structures, and not worry about the evaluation cost until the final moment when we actually
need values. Being able to map, filter, and transform huge data structures efficiently allows our code to be almost uniquely succinct.

2.1.4 Concurrency

Concurrency in Haskell takes the form of a function \texttt{forkIO} and a synchronisation primitive called an \texttt{MVar}. \texttt{forkIO} has type \texttt{forkIO :: IO () -> IO ()}, it takes an \texttt{IO} action and evaluates to another \texttt{IO} action.[19] Semantically, it takes the \texttt{IO} action it is passed and executes it in another process. This forking (and subsequent scheduling and running) is done within the Haskell run-time, such that various spawned processes do not map to OS-level threads. Another function \texttt{forkOS} is available to those users who need that functionality. As the Haskell run-time handles the concurrency, \texttt{forkIO} processes are lightweight, and the cost of creation and switching is very low.[4]

Here’s an example of forking a process in Haskell:

\begin{verbatim}
main :: IO ()
main = do
  forkIO $ do -- fork a process
    loop 'a'
  loop 'z'

loop :: Char -> IO () -- print an infinite sequence
loop ch = do
  putChar ch
  loop ch
\end{verbatim}

The above code forks a loop the prints an infinite sequence of the character ‘a’, before looping infinitely printing ‘z’. The Haskell run-time will sequence these processes in such a way that their output will be interleaved. However, the ordering of the interleaving is non-deterministic.[4]

Haskell also provides synchronisation primitive called an \texttt{MVar}. \texttt{MVars} can be thought of as a box that can be empty or full.[19] We can create a new \texttt{MVar} with \texttt{newMVar :: IO (MVar a)}. We can read from a full \texttt{MVar} (leaving it empty), or block until an empty one is full, with \texttt{takeMVar :: MVar a -> IO a}. We can put a value into an empty \texttt{MVar} (thus filling it), or blocking until it is empty, with \texttt{putMVar :: MVar a -> a -> IO ()}. The blocking semantics of the \texttt{MVar} allow us to synchronise our concurrent processes by inserting and reading values from it. An \texttt{MVar} is a shared-memory location that all processes with a reference to can read or write from.

For this project, it should be noted that whenever we discuss shared-memory concurrency in Haskell we are referring to \texttt{forkIO} and \texttt{MVars}.

12
The lightweight nature of Haskell’s forkIO processes allow us to create, schedule, and execute many thousands of lightweight processes simultaneously.[19] This makes Haskell an ideal language for writing highly performant concurrent applications such as web servers. Indeed, Warp, a Haskell HTTP server, is capable of processing over 80,000 requests per second.[22]

2.1.5 Ecosystem and Tooling

Haskell is a standardised language with a number of implementations. The de facto standard compiler is the Glasgow Haskell Compiler (GHC).[8] GHC supports a number of extensions to the base Haskell language, many of which are common in Haskell programs. One of these is Template Haskell, an extension which allows compile-time metaprogramming. This is can be used to auto-generate boilerplate code for the programmer through macro-like expansion.[26]

GHC comes with a robust Read-Eval-Print-Loop application called GHCi. This allows for interactive testing of Haskell programs in manner not normally seen in natively compiled languages.

Haskell’s build system is called cabal.[25]. It helps people to configure, build and install Haskell software and to distribute it easily to other users and developers. A .cabal file allows a programmer to declare dependencies, dependency version constraints, included modules, and installation directives. It can be considered an advanced, fully-declarative, Haskell-native Make.

Cabal also comes built-in with an extension called cabal-install. This allows Cabal to fetch and install remote packages and dependencies. By default it pulls packages from a repository called Hackage[25]. Developers can publish their Haskell libraries onto Hackage for easy use by other developers.

Cabal is somewhat infamous in the Haskell community for something called cabal hell.[5][6]. By default it, and until recently only, installed packages system-wide (or user-wide if it was installed into the user’s home directory). When many different libraries, each requiring to be linked to specific versions of their own dependencies, were installed, users regularly found themselves in a position where they would be unable to install a package because cabal would be unable to resolve a path through the dependency tree.

This was somewhat alleviated by the creation of cabal-sandbox, which allows packages and their dependencies to be installed into an isolated sandbox. However, aspects of cabal hell still remain as all dependencies (and the dependencies’ dependencies and so on) are all installed in the same sandbox, so they can still conflict. This can be exacerbated by developers who over-specify the versions of the dependencies that they will work with when in fact, their project would still build with more versions. It is also exacerbated by developers who under-specify the versions of their dependencies,
or when their dependencies do not properly follow semantic versioning, and
their project no longer compiles with a version of a dependency that it says
it supports.

As can be deduced from the existence of *cabal hell*, Haskell’s ecosystem
is still in its developing stage. Despite this, there are a multitude of
high-quality packages that a Haskell programmer can leverage to build their
applications.
2.2 The Cloud

‘The Cloud’ is sometimes considered something of a modern buzzword. It will be useful for our purpose to define what various terms specifically mean in the context of this report before we continue.

**Cloud** We use the definition provided by Epstein et al. in *Towards Haskell in the Cloud*[^13]. They define ‘cloud’ to mean “a large number of processors with separate memories that are connected by a network and have independent failure modes”.

**Public Cloud** A public cloud is a service that allows for provisioning of clouds by the public. Generally, much, if not all, of its functionality is exposed programmatically which allows for quick and automated scaling[^1][^20].

**(Public) Cloud Provider** An organisation that makes a cloud available to the public. Providers often also have a number of services that an application running on its cloud can access, such as databases and batch processing. Amazon[^1] and Microsoft[^20] are examples of cloud providers.

**Horizontal Scaling** We define this as the addition of extra machines to an application cluster in order to increase performance. Horizontal scaling is useful for I/O bound tasks[^17] and computationally intensive tasks that are parallelisable.

**Vertical Scaling** We define this as the upgrading of the hardware (or virtual hardware) of a machine or machines in an application cluster to increase performance. Vertical scaling is useful for computationally intensive tasks that are not easily parallelisable[^17].

This project builds on top of Amazon Web Services’ Elastic Compute Cloud, so we will now give an overview of what Amazon Web Services is before briefly addressing some alternative public cloud providers.

2.2.1 Amazon Web Services

*Amazon Web Services (AWS)* is the term given to the suite of remote computing services that make up Amazon’s public cloud computing platform. AWS is centred around two key services: AWS Elastic Compute Cloud (EC2), which allows users to rent virtual machine instances[^1], and AWS Simple Storage Service (S3), which gives users access to cheap blob storage. AWS also contains a number of other services, including DNS, various flavours of databases, monitoring, a content-delivery network, and many
more.[14]. In fact, AWS offers so many services that you can build an entire application's technology stack out of interconnected AWS services.[3].

As we said earlier, EC2 allows us to rent virtual machine instances. There is a huge array of configuration options. EC2 has also various APIs that allow programmatic control over the service.

As we use EC2 in this project, the following is a brief glossary of terms used when discussing EC2[1]:

**Instance** A discrete virtual machine in EC2.

**Instance Type** The type of virtual hardware an instance runs on. There are different types (optimised for general/CPU-heavy/RAM-heavy/GPU-heavy etc. workloads), and different sizes (with more resources). Each type comes in a variety of sizes.

**Region** Amazon has multiple data-centres across many locations in the world. A user can choose where to host their instance on creation.

**Image** Each instance is created from an image. Amazon provides multiple base images of a variety of Linux distributions. An end-user can also generate a new image from the current state of an already created instance. This allows us to create new instances which are identical to the instance that the image was generated from.

**Instance State** Once created, an instance can be started, stopped, or destroyed by the user.

**Tags** Instances have a set of tags, which are key/value pairs of strings. These are useful for annotating your instances with metadata eg. for naming a particular instance.

We will now briefly look at how integrating with EC2 compares versus some alternatives.

### 2.2.2 Alternatives

We will first look at a competitor to AWS, Microsoft Azure. We will then look at OpenStack, an abstraction layer over Cloud Providers.

**Microsoft Azure**

Microsoft Azure is another public cloud provider.[20] In their virtual machine instance solution (ie. the EC2 rival), there is little difference between the two (despite what Microsoft says[21]). AWS EC2 has more instance types and sizes [1] whereas Azure is notably cheaper at publicly available rates [20].
OpenStack

OpenStack is a software abstraction layer over the provisioning of computing resources\[11\]. It allows users to interact with computing resources without knowledge of where said resources are located or provisioned. In essence, it allows for the backend to be anywhere, or for a heterogenous set of backends to be utilised in one system. A company’s own private cloud could be one such backend, and a backend for AWS EC2 exists also. \[11\].

This allows companies to take advantage of the cloud, without worrying about their code being locked in to one cloud vendor. However, this comes at the cost of being unable to take advantage of the unique features of a particular cloud provider.

Comparison

EC2 is not the only virtual machine instance provider, but it has a number of advantages over competing services:

- Because it is the de facto standard in the public cloud computing world, AWS has a distinct advantage when it comes to available tooling and scripting. For example, the Python AWS library \textit{boto} is considered standard tooling for system admistrators and automators. \[23\]

- EC2 provides a number of instance types and sizes that other providers do not, including I/O and GPU optimised instances.\[1\]

- AWS is more than just EC2, and going ‘all-in’ on AWS’s various services can save a huge amount of time and complexity.\[3\] Many of these services are unique to AWS, and cannot be controlled through an abstraction layer like OpenStack. To take full advantage of AWS one must integrate with it directly.\[3\]

2.3 Cloud Haskell

Cloud Haskell is a suite of Application Programming Interfaces (APIs) for distributed programming that brings Erlang/OTP-inspired message-passing concurrency to Haskell, first proposed by Epstein et al. in “Towards Haskell in the Cloud”.\[13\].

This project integrates Cloud Haskell with Amazon Web Services for the first time. To understand this, we must understand Cloud Haskell in a general sense. As such, we will first examine Cloud Haskell as laid out by Epstein et al. Following that, we will briefly look at the current state of its implementation and ecosystem.
2.3.1 Overview

Cloud Haskell is a shallowly embedded Domain Specific Language for cloud computing in Haskell. It is designed to present the programmer with a Erlang-inspired message-passing communication model, designed to make explicit the cost of communication, intended for communicating between Haskell processes without sharing memory. Processors in a cloud have independent failure modes, as we defined earlier, and Cloud Haskell is tolerant of partial failure, again in the Erlang model. Cloud Haskell provides an innovative method of serialising function closures, without any modification to the Haskell compiler or run-time. Cloud Haskell aims to innovate on the Erlang model by leveraging Haskell’s advantages: purity, types, and monadic programming. As Haskell is pure and immutable by default, we do not miss shared, mutable state. Also, pure functions are idempotent, meaning that we can restart functions that have failed due to hardware elsewhere without worry.

Because this project builds below Cloud Haskell’s interface, we shall only discuss the Cloud Haskell programming model itself, as its method of implementation is unimportant to us.

In Cloud Haskell, the basic unit of concurrency is the process. Epstein et al. define a process as “a concurrent activity that has been “blessed” with the ability to send and receive messages”. Process creation is lightweight, just as in forkIO and in Erlang. It should be noted that processes can utilise traditional Haskell shared-memory concurrency within the process itself. Processes communicate via messages. However, MVars cannot be sent as messages.

Messages can be untyped, as in Erlang, or typed, in order to take advantage of Haskell’s strong type system. Any type that is an instance of Serializable can be sent as a message between processes. Basic untyped messaging is available via send and expect. More advanced primitives such as matchIf allow the programmer to wait on a message based on a predicate.

An example of this messaging is a process that sends and receives pings and pongs corresponding to the Process ID of the Process it is communicating with.[13]

```haskell
data Ping = Ping ProcessId
data Pong = Pong ProcessId

ping :: ProcessM ()
ping = do
  Pong partnerPid <- expect
  ownPid <- getSelfPid
  send partnerPid (Ping ownPid)
ping
```
Messages can be sent directly to a process, as above, or they can be sent via Channels. Channels are typed First-In-First-Out (FIFO) queues with a send port and a receive port. Channels allows senders and receivers to only have relevant code interacting with particular messages (for example, rather than your remote calculator code matching on every Int the process receives, there’s a specific channel for Ints the sender wishes to send to the calculator).

A channel-ised version of the ping example looks as follows:[13]

```haskell
ping2 :: SendPort Ping -> ReceivePort Pong -> ProcessM ()
ping2 pingOutPort pongInPort = do
    (Pong partnerPort) <- receiveChan pongInPort
    sendChan partnerPort (Ping pongInPort)
    ping2 pingOutPort pongInPort
```

In Cloud Haskell, processes can monitor other processes for changes or failure. This can be unidirectional or bidirectional. In the event of failure, the monitoring process is notified and the function it attached to the listener as a callback is invoked. This allows Cloud Haskell systems to monitor and tolerate partial failure of processes.

Functions cannot be directly serialized in Cloud Haskell due to limitations in the run-time. This means that in order to pass a function to a remote process, the two processes must be running matching binaries. This allows us to simply pass the function pointer between the two processes. Functions with no free variables are readily passable this way. Cloud Haskell also defines a method for passing function closures (that is, a function pointer and an environment). However, only functions with an available symbol at compile-time can be passed as messages between processes.

Where a Cloud Haskell Process runs is not part of the Cloud Haskell programming model as such. The programming model deals only with processes and how they communicate. Instead, it is part of the implementation which we will now discuss.

### 2.3.2 Implementation

Jeff Epstein originally designed and implemented a prototype of Cloud Haskell as part of his Masters thesis.[13] Following this, Cloud Haskell was re-implemented by programmers at Well Typed, LLP – a Haskell consultancy company[24]. Jeff Epstein, Tim Watson, and a number of people from Well Typed are the current maintainers of this re-implementation. It is this implementation that we will now discuss.

The current implementation consists mainly of the following libraries:[24]

- distributed-process: Base concurrency and distribution support.
• distributed-process-platform: The Cloud Haskell Platform.
• distributed-static: Support for static values.
• network-transport: Generic Network.Transport API.
• distributed-process-simplelocalnet: Simple backend for local networks.
• distributed-process-azure: Azure backend for Cloud Haskell (proof of concept).

The first 5 of these libraries are packaged together as the cloud-haskell package on Hackage, maintained by Tim Watson, which allows for easy installation of the main Cloud Haskell system.

distributed-process and distributed-process-platform form the implementation of the Cloud Haskell programming model that we have already discussed. The Network.Transport libraries define how Nodes communicate with each other in the real world. distributed-process-simplelocalnet and distributed-process-azure are proof-of-concept implementations of what a Node is.

We have just mentioned a Node without defining what it is, let us now do so. A Node is where a Process runs. We can think of a Node in real terms as a distinct Haskell process (process is used here in the Operating System sense of the word). In practice, a Node is treated as both the Haskell process and the computer itself.

Thus in order to run Processes, we need to describe computers as Nodes. distributed-process-azure is a library for describing Azure instances as Nodes.

Cloud Haskell is a new way of thinking about concurrency in Haskell. It allows for Haskell programmers to create and control massive systems within a solid theoretical foundation. Cloud Haskell, and Haskell in the cloud in general, has the potential for unlocking a new frontier for real-world functional programming.

In order to understand the motivation for this project, it is important to stop here to recognise that the current Cloud Haskell backends are proof-of-concept quality at best. As outlined in the introduction, this project endeavours to build an AWS backend for Cloud Haskell that not only has feature parity with the current state-of-the-art, but innovates beyond that. We will now examine distributed-process-azure in detail in order to understand how it is designed, and then use that knowledge to set out design goals for our own library, distributed-process-aws.
Chapter 3

Design

This chapter will examine the design of the existing state-of-the-art Cloud Haskell transport backend in an attempt to identify elements that are important to the design of the project. It will then set out the design goals that the project aims to accomplish.

3.1 Existing Systems

As previously mentioned, there exists a Microsoft Azure backend for Cloud Haskell, `distributed-process-azure`. It was built by Duncan Coutts, Nicolas Wu, and Edsko de Vries who describe it as a “proof of concept Azure backend for Cloud Haskell” that “provides just enough functionality to run Cloud Haskell applications on Azure virtual machines”[10]. The library is built on top of another library by the same authors, `azure-service-api`[9], that performs the actual interfacing with Microsoft Azure.

3.1.1 distributed-process-azure

distributed-process-azure is a library that exposes a Microsoft Azure Transport to Cloud Haskell applications, allowing them to run on Azure virtual machines.

As it is the only publicly available backend for Cloud Haskell, distributed-process-azure’s API is the de facto standard used in example applications in Cloud Haskell literature[24]. It has three demonstration applications that can be compiled along with the library, Echo, Fib, and Ping, that demonstrate how to use the library to build Cloud Haskell applications. For this reason, it is important that we briefly discuss the design decisions made in this library in order to inform the later discussion on the design of my project.

In distributed-process-azure, the local machine (which has initiated the application) talks to the remote machines via SSH. The remote machines
then talk to each other via TCP/IP. The remote machines talk to each other using standard Cloud Haskell primitives, but the communication between the local machine and the remote machines is done via special primitives provided by this library. This is enforced upon the library because Cloud Haskell does not yet support the use of multiple transports (eg. TCP/IP and SSH) in the same system.

These special primitives look as follows:

```haskell
data ProcessPair a = ProcessPair {
    ppairRemote :: RemoteProcess ()
    , ppairLocal :: LocalProcess a
}

localSend :: Serializable a -> a -> LocalProcess ()
localExpect :: Serializable a -> LocalProcess a
remoteSend :: Serializable a -> a -> Process ()
```

We are not concerned with their implementation, but we should briefly note that the Process type comes from Cloud Haskell itself\[13\][27], whereas RemoteProcess and LocalProcess and the associated messaging primitives are provided by this library.

A large portion of the exposed API space of the library deals with these special primitives. In fact, if we ignore (for the moment, we’ll come back to it in a moment) the configuration and utility functions as well as these new primitives, the export list from the library looks as follows:

```haskell
    ( -- * On-VM main
      onVmMain
      , onVmMain
        , CloudService(..)
        , VirtualMachine(..)
        , Endpoint(..)
        , AzureSetup
        , Azure.cloudServices
          -- * High-level API
          , spawnNodeOnVM
          , terminateNode
    )
```

We can see that we have quite a simple API for interacting with this library, with only two functions marked as “high-level API”.  

22
Distributing Computation in Haskell

`onVmMain` is the function that users of this library should call when
the executable is run on a remote machine. `spawnNodeOnVM` creates a
Cloud Haskell node on a given virtual machine, and `terminateNode` will terminate
a given node. These are the main functions utilised by

The rest of the functions come from the underlying `azure-service-api`
library. This library’s architectural decisions (specifically its types) form the
fulcrum around which `distributed-process-azure` is built.

### 3.1.2 azure-service-api

`azure-service-api` integrates Microsoft Azure and Cloud Haskell. It de-
defines types that Cloud Haskell can work with, and interacts with the Mi-
crosoft Azure API to get the data to form values of that type.

The types it defines are quite simple abstractions over raw virtual ma-
chine instance metadata and look as follows:

```haskell
-- A cloud service is a bunch of virtual machines
-- that are part of the same network
data CloudService = CloudService {
  cloudServiceName :: String
,  cloudServiceVMs :: [VirtualMachine]
}

-- Virtual machine
data VirtualMachine = VirtualMachine {
  vmName :: String
,  vmIpAddress :: String
,  vmInputEndpoints :: [Endpoint]
}

-- Globally accessible endpoint for
-- a virtual machine
data Endpoint = Endpoint {
  endpointName :: String
,  endpointPort :: String
,  endpointVip :: String
}
```

These types together form the abstraction over the underlying service
provider. A `CloudService` has many `VirtualMachines` and a `VirtualMachine`
has many `Endpoints`. `azure-service-api` integrates Cloud Haskell and
Microsoft Azure by making an API call to Azure to get information on a
user’s virtual instances, and parses the response into `Endpoints`, `VirtualMachines`,
and `CloudServices`. Its important to note that these types are not tied to
Distributing Computation in Haskell

Microsoft Azure in any way, thus the authors have separated the type-level contract from the implementation. It exposes this functionality via the following function:

```haskell
cloudServices :: AzureSetup -> IO [CloudService]
```

We should now tackle the issue of configuration. In the above function, we can see that it takes a value of type `AzureSetup` as a parameter. `AzureSetup` is defined as follows:

```haskell
data AzureSetup = AzureSetup
  { -- | Azure subscription ID
    subscriptionId :: String
    -- | SSL client certificate
    , certificate :: X509
    -- | RSA private key
    , privateKey :: PrivateKey
    -- | Base URL (generally <https://management.core.windows.net>)
    , baseUrl :: String
  }
```

Note the `X509` and `PrivateKey` values. These are used for authentication with Azure itself, and communication with Azure is done via the SOAP protocol. The `distributed-process-azure` library also requires Azure generated SSH keys to be the `id_rsa` SSH identity for the user.

`AzureSetup` types are constructed by the `azureSetup` function:

```haskell
azureSetup :: String -- ^ Subscription ID
            -> String -- ^ Path to certificate
            -> String -- ^ Path to private key
            -> IO AzureSetup
```

This function is called by the configuration primitives in `distributed-process-azure` that we glossed over earlier.

### 3.1.3 Conclusions

From this examination of the two libraries we can identify some key design points:

- Separation of Cloud Haskell-related primitives (`distributed-process-azure`) and Azure-related primitives (`azure-service-api`)
- An Azure independent contract for the abstractions over virtual machines (`CloudService, VirtualMachine, Endpoint`)
• Cloud Haskell enforced separation of \texttt{LocalProcess} and \texttt{RemoteProcess} and associated primitives.

• Use of the SOAP protocol for communication and X509 certificates for authentication.

3.2 Design Goals

Having now examined the main existing API, let me discuss the design goals of the project.

When building a library (or in this case, libraries), an important aspect of its design is its interface, or how the user uses the library. This can be separated into a couple of key points which we will now tackle in sequence.

3.2.1 Usability

When designing a library interface, usability is of foremost important. ISO/IEC 25010 includes usability as one of its metrics of determining a software product’s quality. Included in its definition of usability is learnability.\cite{18}

Thus, a key design goal of this project is that its interface is learnable for a user. In user interface design theory, this is called the Principle of Least Astonishment\cite{18}. Or, in more practical terms, the project’s interface should be understandable and familiar to a user with domain experience. In this project’s case specifically, this means we should adopt the API conventions set out by the current state-of-the-art, \texttt{distributed-process-azure} and \texttt{azure-service-api}. While API compatibility is a worthy goal too, we stop short of labelling it a design goal for this project. This is because Cloud Haskell is still in the early stages of development, and we should not feel strongly bound to proof-of-concept APIs if they would prevent us from iterating and advancing.

Thus, we aim to find a balance between following the existing conventions while not being bound to their exact specifics. In practice this should mean that applications that depend on the existing state-of-the-art may not work with this project out of the box, but they should only require minimal changes.

3.2.2 Scalability

Earlier we saw that cloud providers allow for the launching and running of many virtual instances cheaply and easily. Unfortunately, Cloud Haskell in its current state does not take advantage of this. Indeed, the existing state-of-the-art backend, the Microsoft Azure integration, only supports the programmatic description of a manually created \texttt{CloudService}.\cite{9}
We want to expose the creation and termination of `VirtualMachines` to the user of our library. As a `CloudService` is defined as a collection of `VirtualMachines`, we also wish to expose the creation and scaling of `CloudServices` to the user. These scaling primitives, which would allow the end-user to change the hardware the application is executing on at run-time, would represent a step forward in the ecosystem’s ability to take advantage of the cloud.

### 3.2.3 Ease of Installation

As we have seen previously, the Haskell community uses the `cabal` (specifically `cabal` and `cabal-install`) tool to download and build packages. The `distributed-process-azure` and `azure-service-api` libraries are installable from the Haskell package repository `Hackage` via `cabal` and this project should be no different.

Another important aspect of installation is ensuring that the package can be installed on a modern version of the Haskell compiler `GHC`. The Azure libraries only build on older versions of Haskell, so providing a backend for Cloud Haskell that allows users to take advantage of modern advances in the Haskell language is a practical and useful design goal.

### 3.2.4 Ease of Configuration

When working with cloud technologies, there are a huge array of options available. Many cloud services provide a variety of different authentication and APIs for the end-user. Some of these require more knowledge of cryptography than others, eg. the Simple Object Access Protocol (SOAP) API requires the user to create properly formatted X509 Certificates and an X509 RSA Private Key. If usability is a goal, then it would be nice if a user need only concern themselves with the minimum of security configuration, and thus this project should provide the simplest possible security option to the user.

One of the main advantages of public clouds as discussed earlier is their ability to scale horizontally (ie. the addition of more machines to an application cluster). But we saw that public clouds also allow applications to scale vertically, that is by increasing the power of the machines, rather than their number. Amazon Web Services has a multitude of different instance types, each optimised for a different workload[1]. Similarly, Amazon has dozens of data centres, called regions, throughout the world and allows users to select which region they would like their virtual instances created in. This can be useful for taking advantage of geographical locality to reduce latency between the application computers and the end-user of the application. Both regions and instance types and other options can all be configured when making API requests to Amazon Web Services, and this
configuration should be presented to the user of our library in the easiest way possible. This would be a great advantage over the existing libraries, which do not allow for this sort of configuration (or indeed, the management of virtual machines at all, see: Scalability).

Thus in summary, we want to create a library that builds on the modern Haskell toolchain, that integrates Cloud Haskell and Amazon Web Services for the first time, that is easily configured, that follows of existing API idioms as far as possible, and that allows its user to take full advantage of running their application in a public cloud.
Chapter 4

Implementation

This chapter will provide an overview of the completed project. It will examine certain interesting aspects of the project in further detail. Finally it will highlight some challenges faced in the completion of the project.

4.1 Overview

The project is separated into two separate Haskell libraries:

- **aws-service-api** is integrates Amazon Web Services as a Cloud Haskell backend. It handles the modelling of AWS EC2 instances as Cloud Haskell nodes and the aggregation of instances into CloudServices. It exposes functions for describing the current state of an EC2 account’s instances as Cloud Haskell CloudServices, and also exposes functions for creating and modifying existing CloudServices and Virtual Machines.

- **distributed-process-aws** wraps aws-service-api and integrates the backend into Cloud Haskell, exposing the Cloud Haskell-level functions that work with the backend under the hood. Specifically, it provides the functionality that allows CloudHaskell Processes to be run on the Backend that aws-service-api provides.

We will now give an overview of each of these libraries in turn.

4.1.1 aws-service-api

aws-service-api is built on three types: CloudService, VirtualMachine, and Endpoint. This are almost identical to their cousins in azure-service-api, as this allows us the leverage a significant amount of the code in distributed-process-azure, as we will see later. Their type definitions are as follows:

```haskell
data CloudService = CloudService {
    cloudServiceName :: String
```
Distributing Computation in Haskell

```haskell
data VirtualMachine = VirtualMachine {
    vmName :: String,
    vmInstanceId :: String,
    vmIpAddress :: String,
    vmInputEndpoints :: [Endpoint]
} deriving (Show)

data Endpoint = Endpoint {
    endpointName :: String,
    endpointPort :: String,
    endpointVip :: String
} deriving (Show)
```

The only difference here is the addition of `vmInstanceId` to `VirtualMachine`. This allows us to easily uniquely address a `VirtualMachine` without having to fetch all of the instances and filter by name locally. This allows us to give the user the option of using `vmName` for unique and memorable names.

The rest of the library consists of worker functions that interact with AWS EC2 to create and alter values of these types, and higher-level user-facing functions that are compositions of these worker functions. The type signatures of these higher-level functions are as follow:

```haskell
cloudServices :: AWSSetup -> IO [CloudService]
addVM :: AWSSetup -> String -> CloudService -> IO CloudService
destroyVM :: AWSSetup -> String -> CloudService -> IO CloudService
scaleUpService :: AWSSetup -> CloudService -> Int -> IO CloudService
scaleDownService :: AWSSetup -> CloudService -> Int -> IO CloudService
createService :: AWSSetup -> String -> Int -> IO CloudService
```

Together, these functions and the data types form the library’s exposed API. Their functionality is as follows:

- **cloudServices** returns a list of all the `CloudServices` on a user’s AWS EC2 account.
- **addVM** takes a name as a `String` and a `CloudService` and creates a `VirtualMachine` of with that name, configured by the `AWSConfig`, and adds it to the `CloudService` and returns the altered `CloudService`.
- **destroyVM** takes the name of a `VirtualMachine` and a `CloudService` and deletes the `VirtualMachine` with that name, and returns the altered `CloudService`. 

• `scaleUpService` takes a `CloudService` and a number as an `Int` and adds that many new `VirtualMachines` in that `CloudService`.

• `scaleDownService` takes a `CloudService` and a number as an `Int` and destroys that many new `VirtualMachines` in that `CloudService`.

• `createService` takes a name as a `String` and a number as an `Int` and creates a new `CloudService` with that name with that number of `VirtualMachines`.

Configuration is handled by the `Config` submodule which exposes the `AWSSetup` type and a `loadSetupFromFile :: FilePath -> IO AWSSetup` function. We will discuss this configuration in more detail later.

These higher-level functions use lower-level functions to actually communicate with the AWS API. These lower-level functions use a Haskell library, `aws-sdk`, to manage their communications with the AWS API. This library wraps the raw AWS HTTP API with Haskell functions that interact with various endpoints, parse the responses into Haskell data-types, and provides a monadic interface for interacting with the API. We will discuss exactly how we do this interaction later.

We will now give an overview of the library that builds on top of this, `distributed-process-aws`, before we dive deeply into how `aws-service-api` is implemented.

### 4.1.2 distributed-process-aws

Earlier, we gave an overview of how `distributed-process-azure` was designed and implemented. Luckily for us, because our definitions of `CloudService`, `VirtualMachine`, and `Endpoint` are virtually identical to the definitions in `azure-service-api`, the vast majority of the code in `distributed-process-aws` is line-for-line identical to its Azure counterpart. There are some changes related to how configuration is passed around, which we will detail when we discuss configuration later, and some new re-exporting of new `aws-service-api` functionality, but beyond that the code is the same.

Unfortunately, the realisation that much of this code would be identical except for type names being changed and the configuration came late in the development of this project, so I was unable to pull out the similarities into a common base library. (`distributed-process-cloud` with `distributed-process-cloud-azure` and `distributed-process-cloud-aws`, perhaps?).

The configuration changes result in a slight change of API, which we will discuss later.
4.2 Selected Details

Having given an overview of the libraries’ interfaces, we will now take a closer look at some of the more interesting aspects of its implementation.

4.2.1 AWS Interaction

As mentioned before, our exposed API functions are written in terms of lower-level worker functions that utilise a library called `aws-sdk`. `aws-sdk` exposes types and functions for interacting with the AWS API, saving us the trouble of crafting and parsing the raw HTTP requests and responses ourselves.

In this section, we will show Haskell code in greater detail than we have before, to illustrate how this implementation is structured as compositions and sequencings of lower-level worker functions.

**Instance Type**

One of the types it exposes is the `Instance` type. The `Instance` type is a huge record-type. It models everything the AWS API exposes about an EC2 instance. However, Cloud Haskell knows nothing about `Instances`, so we must parse the `Instance` into a `VirtualMachine`.

```haskell
parseInstance :: Instance -> VirtualMachine
parseInstance ins = VirtualMachine {
    vmName = getTagValue "Name" ins
    , vmInstanceId = unpack $ instanceId ins
    , vmIpAddress = unpack $ fromJust $ instanceDnsName ins
    , vmInputEndpoints = [Endpoint {
        endpointName = "SSH"
        , endpointVip = unpack $ fromJust $ instanceDnsName ins
        , endpointPort = "22"
    }
    ]
}
```

This parsing is a quite straightforward example of constructing a new record-type, but it does demonstrate how we are using AWS tags. To ‘name’ our `Instances` in AWS, we simply create a tag called ‘Name’. Tags are structured oddly by `aws-sdk` so we have a helper function `getTagValue` to pull it out of an `Instance` for us.

```haskell
getTagValue :: String -> Instance -> String
getTagValue key ins = head [unpack . fromJust $ resourceTagValue resTag
    | resTag <- instanceTagSet ins
    , unpack (resourceTagKey resTag) == key]
```
Services

We will now look at how the function \texttt{cloudServices :: AWSSetup \rightarrow IO [CloudService]} is implemented. As mentioned earlier, it is a composition of lower-level worker functions. It’s own implementation looks as follows:

\[
\text{cloudServices :: AWSConfig \rightarrow IO [CloudService]}
\]

\[
\text{cloudServices conf = do}
\]

\[\text{ins} \leftarrow \text{getInstances conf}
\]

\[\text{let ins’} = \text{filter (x \rightarrow \text{instanceState} x == \text{InstanceStateRunning}) ins}
\]

\[\text{let services} = \text{groupServices ins’}
\]

\[\text{return $ parseServices services}
\]

Let us tackle this line-by-line.

\[
\text{ins} \leftarrow \text{getInstances conf}
\]

\[\text{calls the function getInstances :: AWSConfig \rightarrow IO [Instance] and puts the resulting [Instance] in ins. As the reader can probably deduce, getInstances returns a list of all the instances in a given EC2 account by making a call to the AWS API. We then do a preliminary filter on this list on the next line, let ins’ = filter (x \rightarrow \text{instanceState} x == \text{InstanceStateRunning}) ins, and filter out all of the Instances which are not currently running. This is done as a CloudService can only contain currently running VirtualMachines. At this point, ins’ contains all running instances.}
\]

\[
\text{We then wish to group these Instances by their CloudService. The way we denote membership of a CloudService is via another Tag on the Instance, this time called “service”. So we wish to group Instances with the same “service” tag. We can also take this opportunity to parse the Instance into a VirtualMachine. We do this by creating a Map of the service name to a [VirtualMachine].}
\]

\[
\text{groupServices :: [Instance] \rightarrow Map String [VirtualMachine]}
\]

\[
\text{groupServices instances = fromListWith (++) serviceTuplesWithLists}
\]

\[\text{where serviceTuplesWithLists = map makeLists serviceTuples}
\]

\[\text{makeLists (service, ins) = (service, [ins])}
\]

\[\text{serviceTuples = map makeServiceTuple instances}
\]

\[\text{makeServiceTuple ins = (getTagValue "service" ins, parseInstance ins)}
\]

Finally, we want to convert this Map String [VirtualMachine] into a [CloudService] by first turning it into a [(String, [VirtualMachine])] and then mapping a parse function down the list.

\[
\text{parseServices :: Map String [VirtualMachine] \rightarrow [CloudService]}
\]

\[
\text{parseServices = map parseTuple . toList}
\]

\[\text{where parseTuple (name, vms) = CloudService {}
\]

\[\text{cloudServiceName = name}
\]
Distributing Computation in Haskell

, cloudServiceVMs = vms
}

The usage of the **aws-sdk** library to handle the parsing of the XML/JSON from the AWS API allows us to structure our code as idiomatic Haskell transformations between types.

The **cloudServices** function is the main inward integration with AWS in the sense that it requires the greatest transformation of the response. There is a significant amount of outward integration too (ie. functionality that alters the state of AWS), but before discussing that we will first look at how configuration is handled in this project.
4.2.2 Configuration

This project uses a library called `aws-sdk` for its integration with Amazon Web Services. We discuss this decision later in Section 4.3.1 when discussing challenges. `aws-sdk` uses the HTTP REST API to communicate with AWS. In order to authenticate with this API, a user needs to generate an `accessKey` and a `secretAccessKey` via the AWS management portal, and use it to sign all requests to AWS. Thus, in order to use `aws-sdk` and thus our library, we need to provide some way for the use to provide these keys to us.

In addition, there are dozens of other configuration options a user can provide when making AWS requests. Pertinently, it is important for a user to be explicit about which AWS region they wish their request to be handled by. Similarly, if we are to support the creation of new Instances, a user needs to be able to supply an identifier for an image that their Instance will be created from. Furthermore, it would be useful if the user was not locked into using the default `t2.micro` instance type, as while it is free, it is small and not generally suitable for production[1].

For ease of configuration, and to discourage users from hardcoding their security credentials into their source code, we export a `awsSetup :: FilePath -> IO AWSSetup` function that parses a configuration file into a data type. This type, and the parser itself, are automatically generated, via Template Haskell, by the `simple-config` and `parsec` libraries. Their implementation looks like the below:

```haskell
mkConfig "configParser" [config]
AWSSetup
   accessKey ByteString
   secretAccessKey ByteString
   region String
   image String
   instanceType String
|
loadSetupFromFile :: FilePath -> IO AWSSetup
loadSetupFromFile path = do
   str <- BS.readFile path
   case parse configParser "" str of
      Left err -> fail $ show err
      Right conf -> return conf
```

[1] `awsSetup` is simply `loadSetupFromFile` renamed for API compatibility.
This auto-generates a parser and a data-type that parse a file that looks like this:

```plaintext
accessKey: rogpberbg4
secretAccessKey: glnerjglekjrbgjkerbg
region: eu-west-1
image: ami-c7a73db0
instanceType: t2.micro
```

`aws-sdk` itself defines a `Credential` type which it requires to be passed to a `runEC2` block. We create a `Credential` type from our `AWSSetup` like so:

```haskell
let cred = newCredential (accessKey conf) (secretAccessKey conf)
```

We mentioned earlier that the Azure libraries used SOAP to communicate with Microsoft Azure, and that our configuration changes necessitated some breaking API changes in `distributed-process-aws`. There are actually two separate changes. Firstly, and quite simply, we only need to pass one `FilePath` as a parameter to `defaultAWSParameters` rather than two for `defaultAzureParameters`. Secondly, it is important to copy the configuration file to the `VirtualMachine` when we copy the binary to it. This is actually an additional feature that allows a remote `VirtualMachine` to take over as the master in a cluster, as `distributed-process-azure` never copied the X509 certificates or Private Key to the remote machine.

Many of the above changes seem straightforward, and implementation-wise they are, but their ergonomics are much easier for the user. Previously users of `distributed-process-azure` had to fiddle with the OpenSSL command-line tool to properly format the Private Key from the X509 cert. We will now see what the `region`, `image`, and `instanceType` keys in our configuration file are useful for.

### 4.2.3 Scaling

We will now tackle the new scaling primitives in `aws-service-api`. These represent the largest innovation over the existing state-of-the-art, and thus deserves careful examination.

We will be discussing the implementation of these functions, which we looked at earlier:

```haskell
addVM :: AWSSetup -> String -> CloudService -> IO CloudService
destroyVM :: AWSSetup -> String -> CloudService -> IO CloudService
scaleUpService :: AWSSetup -> CloudService -> Int -> IO CloudService
scaleDownService :: AWSSetup -> CloudService -> Int -> IO CloudService
createService :: AWSSetup -> String -> Int -> IO CloudService
```
At their heart, these functions consist of creating and terminating AWS Instances. As such, they all utilise wrappers around this, entitled createVM and terminateVM. Let us look at terminateVM first as it is simpler. Its implementation looks as follows:

```haskell
terminateVM :: AWSSetup -> VirtualMachine -> IO ()
terminateVM conf vm = do
  let cred = newCredential (accessKey conf) (secretAccessKey conf)
  let id = [pack $ vmInstanceId vm]
  runResourceT $ runEC2 cred $ do
    setRegion $ pack $ region conf
    Util.list $ terminateInstances id
  return ()
```

Here we can see the usefulness of adding vmInstanceId to the VirtualMachine type, as it allows us to directly target it for termination. As this is the first code that actually interacts with AWS we have seen, we should look at what runEC2 and runResourceT do, as they are elegant solutions to structuring our AWS code. runEC2 allows us to ‘run’ an EC2 monad action, which setRegion and terminateInstances are. Thus it lets us build up an EC2 request as a sequence of actions before running the whole thing at once. The ResourceT monad allows us to wrap another monad, in this case EC2, which lets us ensure all of the resources allocated in an EC2 monad block (ie. sockets) are freed.

From this terminateVM function, our destroyVM function’s implementation falls out for free, as it simply pulls the VirtualMachine out of the CloudService, calls terminateVM, and ensure the CloudService is cleaned up.

```haskell
destroyVM :: AWSSetup -> String -> CloudService -> IO CloudService
destroyVM conf name cs = do
  let vm = head $ filter (\v -> vmName v == name) (cloudServiceVMs cs)
  terminateVM conf vm
  return $ cs {
    cloudServiceVMs = filter (\v -> vmName v /= name) (cloudServiceVMs cs)
  }
```

The lower-level function, createVM, is more involved than terminateVM, and indeed we won’t show it here for that reason, but it is simply a collection of AWS API requests just like terminateVM. Its type looks as follows:

```haskell
createVM :: AWSSetup -- config
  -> String -- vmName
  -> String -- name of cloud service
  -> IO VirtualMachine
```
What’s interesting here is that the name of the CloudService is passed as a String. This allows us to create a machine in a CloudService that may not exist yet, because as far as AWS is concerned our CloudService only exists as a ‘service’ tag on our Instances. Internally, createVM also pulls the region, image, and instanceType out of our AWSSetup and sets up the Instance appropriately. It also requests the Instance data back from AWS so it can parse and return a VirtualMachine, as we want to keep our code decoupled from aws-sdk.

On top of this, addVM simply calls createVM, with the added feature that it does a waitForInstanceState so that the caller is guaranteed that all of the VirtualMachines in their CloudService are running when control cedes back to them.

addVM and destroyVM add a layer of scalability over the existing status quo, but they require the programmer to deal in specific VirtualMachines. As we are building Haskell for the cloud, we want to expose this scalability over CloudServices.

Firstly, we want to be about to create a new CloudService, we can do with the createService function. We saw already how createVM allows us to create VirtualMachines with a String for the CloudService name. So we simply use createVM to create bootstrap our CloudService with the first VirtualMachine, and then call scaleUpService to create the rest.

scaleUpService is an interesting function. Let’s look at its type again:

scaleUpService :: AWSSetup -> CloudService -> Int -> IO CloudService
scaleUpService conf cserv 0 = return cserv
scaleUpService conf cserv n = do
  name <- newInstanceName $ cloudServiceName cserv
  cserv' <- addVM conf name cserv
  scaleUpService conf cserv' (n - 1)

It recursively uses addVM to add new VirtualMachines to a CloudService. Eagle-eyed readers may notice that we previously skimmed over what createService used as the vmName for the first VirtualMachine it added to the new CloudService before it called scaleUpService.

Both scaleUpService and createService use a helper function called newInstanceName. It’s quite simple:

newInstanceName :: String -> IO String
newInstanceName cserv = do
  g <- newStdGen
  let (uuid, _) = random g
  return $ cserv ++ "-" ++ toString uuid

\[A\] A helper function.
\[3\] Or, in AWS terms, the ‘Name’ tag for the instance.
Distributing Computation in Haskell

It simply generates a string with the CloudService’s name, again passed as a String to allow us to bootstrap a new CloudService, prepended to a randomly generated Universally Unique Identifier (UUID).

scaleDownService has a similar structure to scaleUpService, except it recursively calls destroyVM. As destroyVM requires us to pass a value of type VirtualMachine to it, we have a helper which plucks out a random one from the CloudService.

```haskell
randomVM :: CloudService -> IO VirtualMachine
randomVM cserv = do
    g <- newStdGen
    let vms = cloudServiceVMs cserv
    let (index, _) = randomR (0, length vms - 1) g
    return $ vms !! index
```

These CloudService-oriented scaling primitives allow a user of the library to create and alter very large CloudServices, without prior setup or configuration of individual VirtualMachines.

4.3 Challenges

Every project encounters challenges, and this one was no different. We shall now briefly outline some of the more relevant and interesting of them.

4.3.1 API Compatibility

One of the design goals of this project was to maintain API compatibility with the existing Azure library. Specifically that means that any overlap in functionality would be exposed by the same data types and functions. The idea being that a user of the libraries could switch out their import from Azure to AWS and their code would continue to work. I set it as a challenge to get the demonstration applications in distributed-process-azure compiling and running transparently on top of distributed-process-aws.

Unfortunately, perfect API compatibility proved too challenging. This was due to the use of SOAP in azure-service-api. To make aws-service-api API identical to azure-service-api I would also need to use SOAP, because the configuration functions and types expected to be provided with an X509 certificate and an RSA Private Key.

I initially started aws-service-api using the SOAP protocol to communicate with AWS. In addition to the HTTP solution that has already been introduced, AWS also supports SOAP. Unfortunately, this led to much time lost trying to get the necessary cryptography libraries building on a modern Haskell compiler (see the next section). Even once they were integrated, writing custom SOAP code was brittle. Every time I wanted to add a new
feature I needed to spend an inordinate amount of time parsing the resulting XML and crafting SOAP requests.

The final nail in the coffin for SOAP came when I decided I wanted to support custom images. Cloud Haskell requires `libssh2-1` to be installed on the `VirtualMachines` it runs on, so the only way to support run-time scalability primitives was to ensure the instance had it installed. So the choice was either to use SOAP to create the instance, and then programmatically SSH into the new instance and run some shell commands to install `libssh2-1`. This was untenable as it would mean only supporting certain Linux package managers, which would be a regression compared to the Azure library.

With the decision to switch to HTTP, this problem was solved by allowing for an image to be specified in the same configuration file as the access keys. I also no longer had to craft custom API requests as I could leverage `aws-sdk` to authenticate, make the requests, and clean up after itself. This came at the cost of a small amount of API changes related to configuration. However, this are very minor compared to how elegant the resulting code is, and how easy the library is to configure for the user.

### 4.3.2 Cabal

As mentioned above, `cabal hell` was a long-time companion during this project. We detailed what `cabal hell` was in 2.1.5. Much of it was to do with the the versions of packages relating to cryptography and networking. Cloud Haskell is intended to be built with GHC version 7.4.2, and I was attempting to write this project using GHC version 7.8.3. However, there were a sizable amount of conflicts between the cryptography dependencies of the HTTP and SOAP libraries I wanted to use, and the cryptography dependencies of Cloud Haskell itself. Fixing these involves stepping through the dependents of every library that is erroring out and seeing if they’re over-specifying or under-specifying their version contraints. Eventually, after a large amount of pain and torture and manual intervention, I managed to get a skeleton project building.

These issues were mostly fixed by the previously discussed move away from SOAP. These removed the need for any cryptographic packages in my libraries, which removed any opportunity for conflicts. However, the version of `aws-sdk` available from cabal has an open-ended dependency for the `monad-control` library, but it made a breaking change in its API when it jumped major version numbers. This means that a user of the project has to pull down the source of `aws-sdk`, edit its `.cabal` file to include a constraint `monad-control < 1.0.0.0` and then add this local folder as a source to their cabal sandbox. There is unfortunately no way around this until the `aws-sdk` team push the latest version of their library to Hackage.

This is quite a simple fix, but it is a good example of the process involved
in getting many of the aforementioned cryptography libraries working the first time round. It is slow and exhausting work tracing through compiler and package manager errors for code you are completely unfamiliar with.

Having now looked at the project’s implementation and some of the challenges faced, we will now look at some practical examples of the project in use, and then assess the project against the design goals we set out for the project.
Chapter 5

Results

This chapter will show some examples of the project in use, and assess the project’s success in meeting its design goals and its usefulness to the wider Cloud Haskell ecosystem.

5.1 Examples

In getting examples working, a couple of configuration steps are required:

- The user needs an Amazon Web Services account.[14]
- AWS access keys need to be generated.[2]
- The user needs to generate a Linux image with libssh2-1 installed.
- A patched (as described in 4.3.2) aws-sdk project needs to be added as a source to the sandbox distributed-process-aws is installed into.
- A configuration file needs to be created.(As described in 4.2.2)

5.1.1 Demos

The distributed-process-azure project has a number of demo programs that demonstrate the library working. As mentioned in Section 4.3.1, it was a goal of this project to get these programs to build on the new distributed-process-aws library. This was successful with minor alterations for configuration API changes.

They (along with those described below) can be installed along with the library by running cabal install -f build-demos.

These demos require that the user set manually create a virtual instance using the AWS EC2 Management Console named “CHDemo1” and give it a “service” tag with the value “CHDemoService”. This is a legacy of the Azure libraries only working with existing, manually created, VirtualMachines.
$ cloud-haskell-aws-fib ~/aws.config ubuntu CHDemoService 15
610

$ cloud-haskell-aws-echo ~/aws.config ubuntu CHDemo1 5000
# Echoing this back?
Echo: Echoing this back?

5.1.2 New Functionality

Altered versions of the above Echo and Fib\(^1\) programs are available too. These applications demonstrate the basic usage of the new run-time primitives in action. They do not take a named CloudService or VirtualMachine as a command-line argument. Instead they create a new CloudService at run-time and use that service to run the application on. Their output is identical to that outline above.

$ cloud-haskell-aws-scale-fib ~/aws.config ubuntu 15
610

$ cloud-haskell-aws-scale-echo ~/aws.config ubuntu 5000
# Echoing this back?
Echo: Echoing this back?

These examples showcase distributed-process-aws in action, running Cloud Haskell Processes on EC2. Manual usage of aws-service-api is also possible. What follows is an abridged account of using aws-service-api in the Haskell Read-Eval-Print-Loop.

*Network.AWS.ServiceManagement> cfg <- loadSetupFromFile "aws.config"
*Network.AWS.ServiceManagement> cs <- cloudServices cfg
*Network.AWS.ServiceManagement> cs
[]
*Network.AWS.ServiceManagement> cs' <- createService cfg "DemoService" 2
*Network.AWS.ServiceManagement> cs'
CloudService {cloudServiceName = "DemoService", cloudServiceVMs = []}
*Network.AWS.ServiceManagement> length $ cloudServiceVMs cs'
2
*Network.AWS.ServiceManagement> cs'' <- scaleUpService cfg cs' 1
*Network.AWS.ServiceManagement> length $ cloudServiceVMs cs''
3
*Network.AWS.ServiceManagement> cs''' <- scaleDownService cfg cs' 2
*Network.AWS.ServiceManagement> length $ cloudServiceVMs cs'''
1

\(^1\)Called cloud-haskell-aws-scale-echo and cloud-haskell-aws-scale-fib respectively.
The state of the AWS EC2 Management Console after this looks as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Instance ID</th>
<th>Instance Type</th>
<th>Availability Zone</th>
<th>Instance State</th>
</tr>
</thead>
<tbody>
<tr>
<td>DemoService-1e3d...</td>
<td>i-55c3a3b2</td>
<td>t2.micro</td>
<td>eu-west-1c</td>
<td>running</td>
</tr>
<tr>
<td>DemoService-9d0d...</td>
<td>i-bcc6e65b</td>
<td>t2.micro</td>
<td>eu-west-1c</td>
<td>terminated</td>
</tr>
<tr>
<td>DemoService-db4b...</td>
<td>i-f5c6e612</td>
<td>t2.micro</td>
<td>eu-west-1c</td>
<td>terminated</td>
</tr>
</tbody>
</table>
5.2 Assessment

At the beginning of the project, we stated that our goals were to build:

- An Amazon Web Services (AWS) Service Management library that allows for the creation, scaling, and management of Cloud Services and Virtual Machines on AWS Elastic Compute Cloud (EC2) in Cloud Haskell applications.

- A Cloud Haskell wrapper of this service management library that exposes its functionality to Cloud Haskell applications.

These have been completed in aws-service-api and distributed-process-aws. However, to properly assess how well the project achieved these goals, we will now compare the project with the design goals that we set out in Chapter 3.

Ease of Installation The libraries can be installed with one documented fix to the aws-sdk dependency as detailed above. Alternatively, the user can install the latest version of aws-sdk on GitHub. Beyond that temporary fix, the libraries build and run on the latest version of Haskell without issue.

Ease of Configuration As detailed in Chapter 4: Configuration, a user can easily configure the libraries via a configuration file which they pass to \texttt{awsSetup :: FilePath -> IO AWSSetup}. The level of manual intervention is low and, importantly, it is front-loaded. Once an appropriate image has been generated the user never needs to log into the EC2 Console again.

Scalability Not only have we provided functionality for adding and removing VirtualMachines, we have also provided higher-level wrappers that deal directly with scaling and creation from a CloudService. This allows users of our library to create, alter, and write Cloud Haskell programs with CloudServices of sizes where dealing with individual VirtualMachines would be too cumbersome.

Usability This project does the above with the minimal amount of API breakage between them and the Azure libraries. This will allows users of the Azure library to re-use their knowledge when picking up the new AWS libraries.

In summary, by-and-large the design goals of the project have been met. The Cloud Haskell ecosystem now has an AWS integration for the first time. Also, this integration allows for the run-time adjustment of the cloud cluster, another first for Cloud Haskell.
Chapter 6

Conclusion

In conclusion, I believe that this project was a success. For the first time, Cloud Haskell and Amazon Web Services have been integrated in `distributed-process-aws` and `aws-service-api`, which have both been released (and have 1.0+ version numbers at time of writing). These libraries will allow new Cloud Haskell users to get up and running on new hardware in a way that was not previously possible.

As we assessed earlier, the design goals of the project have been met. More qualitatively, I feel like this project has advanced the state-of-the-art in Cloud Haskell backends. The new scaling primitives in this service provider will allow for Cloud Haskell applications to take proper advantage of the cloud for the first time. Having spent a significant portion of this project fighting with the Haskell ecosystem in order to build the libraries on a modern Haskell compiler with modern versions of their dependencies, I hope that the current ease of installation will attract new members of the Haskell community to the Cloud Haskell ecosystem and push it forward.

Though the goals I set out in the beginning were met, there are some avenues for future work that I hope that I or the community can pursue. First is an exploration of how we could move `instanceTypes` from configuration to the library level. This would allow the end-user to create `cloudServices` out of heterogenous `instanceTypes`, rather than being tied to a value in a configuration file. Second, and this is more of a general research topic, I would like to explore how we could use Haskell’s expressiveness to abstract the boilerplate of writing Cloud Haskell applications. Some work is needed in both Cloud Haskell (to allow multiple transport protocols in one application) and the Haskell run-time (to allow the serialisation of anonymous and partially-applied functions) to do this properly, but it would be interesting to see what sort of Domain Specific Language could be built on top of Cloud Haskell once the implementations allow for it.

Having studied it in great depth over the past few months, I genuinely believe that the Cloud Haskell paradigm is the future of distributed con-
currency in Haskell and that Haskell and Cloud Haskell are the tools for programming on public clouds like Amazon Web Services. However, they need a greater level of community engagement to get their implementations to a production grade, and to push for supporting work inside the Haskell compiler and run-time. I hope my project can go some way to increasing that engagement and to showing what Haskell can do for programming in the large.
References


