UiO : Department of Informatics
University of Oslo

A Study In Monads

Leif Harald Karlsen
Master's Thesis Autumn 2013

Preface

This thesis is a study in monads in functional programming. As a functional language I will use Haskell, and will therefore assume some basic knowledge of functional programming and Haskell's syntax. In short, Haskell is a lazy, strongly and statically typed, curried, purely functional language with pattern matching, abstract data types, type polymorphism, and first class functions. A thorough introduction to Haskell can be found in "Real World Haskell" by Sullivan, Goerzen and Stewart[7].

Furthermore, I will also assume some familiarity with basic theoretical computer science, such as first order logic, lambda calculus with and without types, and basic mathematical and computer scientific concepts and expressions like bijective functions, homeomorphisms, structural induction, etc.

Throughout the thesis I will use a dagger, †, to denote my theoretical contributions. If a dagger is placed next to a chapter, it means that the chapter contains new theoretical material, the same holds for sections. All definitions, lemmas, theorems, etc. marked with a dagger are my own and not taken from anywhere else. Furthermore, a λ next to a section or a chapter marks that the chapter contains applicable code written by me. All such code can be found in full, written in pure ASCII in the appendix starting on page 119.

The thesis is divided into four parts. The first part, "Foundation And Definition", goes through all the definitions and results needed for the rest of the thesis. The next part, "Application", shows real applications of monads. The third part, "Deeper Theory", is mainly concerned with monads studied in a more formal way. The last part, "Conclusion And Appendix", is just that, the conclusion and appendix. A more detailed outline of the chapters is given below.

Chapter 1 In the first chapter we start by looking at the history, invention, and first use of the monad, together with some alternate constructions that solve some of the same problems. We then briefly visit one of the original motivations behind monads, namely the problem of IO in functional languages.

Chapter 2 The second chapter is mainly concerned with introducing the monad as it is used in a functional language, such as Haskell. There are two definitions of the monad, and both will be presented and shown to be equivalent. We continue by looking at a special notation used when writing monadic code, along with common auxiliary functions used throughout the rest of the thesis. The chapter continues with a discussion on the monadic

solution to IO and the consequences this has for IO in Haskell, and ends with common perspectives and views on monads. Examples of different monads are presented when the necessary material concerning each monad has been discussed. After reading this chapter the reader should know what monads are and have some intuition of where monads can be used, as well as be familiar with the most common monads.

Chapter 3 In this chapter we will look at how monads' functionality can be combined through what is called monad transformers. A discussion on how transformers are used and how they behave follows, before a few examples are presented. After this, we show how we can use the extended functionality of a monad in the transformed monads through the concept of lifting. We finish this chapter with a discussion on general monad morphisms. The reader should now have an understanding of how predefined monads can be combined to form new, more complex monads.

Chapter 4 Up until now we will have focused on predefined monads. In this chapter we will create a new monad, the Pair monad, prove its desired properties and view some applications. The Pair monad can represent computations in almost any collection. We will also look at the corresponding transformer and discuss the implementation in detail.

Chapter 5 In this chapter an extended example is shown, where we construct a small interpreter for lambda calculus with a graph reduction algorithm similar to that of Haskell. We will use the Pair monad defined in the previous chapter to represent lambda terms. The reader should be familiar with all of the monads used in this implementation.

Chapter 6 This chapter combines much of the material that has been presented so far in one large example. A framework for cryptographic algorithms are constructed from some of the presented monads, and some new. We will throughout this chapter see how monads can be combined and used to solve a more realistic and more complex problem than those previously presented.

Chapter 7 One common application of monads is implementation of domain specific languages. In this chapter we will investigate such an application, where we use monads as building blocks to create a Datalog-like language embedded in Haskell. We will look at how different monads can be combined to create different languages.

Chapter 8 In this chapter we will look at monads' origins, namely category theory and how monads are defined there. All categorical constructs needed will be presented and no prior knowledge of category theory is necessary.

Chapter 9 This chapter is more theoretical and presents a modal logic derived from typed lambda calculus with monads, through the Curry-Howard correspondence. We will use this corresponding logic to build up some intuition around the nature of monads, and use its Kripke semantic to construct a semantic for our lambda calculus. The semantic will be used to help us reason about the properties of different monads. This chapter is of a more investigative nature, but will hopefully provide the reader with more intuition on monads.

Chapter 10 In the last chapter we will look at other constructs related

to monads, namely the monad's dual, comonads, along with the three other constructs, arrows, applicative functors, and continuations. In turn, we will look at how these concepts are related to monads, and discuss which problems they solve.

Acknowledgements

First of all, I want to thank my supervisor Herman Ruge Jervell who, despite illness, continuously gave me great supervision with helpful comments and clear insight throughout this thesis. I also want to thank him for allowing me to investigate freely according to my interests and making this thesis my own. He was also the one who initially sparked my interest for theoretical computer science, as the lecturer of my first course in logic, as well as developing my curiosity even further through later courses and discussions.

A special thanks goes to my dearest Helene for supporting me and showing interest in my work throughout this degree, even in my excited elaborations on monads. My time as a master student wouldn't be the same without her by my side. I also want to thank her for a thorough proof reading and spell checking of this thesis, in addition to general advice on language.

I also want to thank my wonderful mother, father, and brother for their encouragement and support in my years of studies. I am grateful for their caring and interest in my work, despite it being quite far from what they are familiar with.

A great thanks also goes to my ingenious friends for interesting discussions and engaging conversations.

Contents

Part I

Foundation And Definition

Chapter 1

The Monad's History and Motivation

1.1 The invention of monads

It was originally Eugenio Moggi who, in 1991, introduced the monad as a way of simulating different types of computations in lambda calculus. In his article "Notions of computations and monads" [20] he used the notion of the monad from category theory to define a categorical semantic for lambda calculus with different types of computational effects. Effects such as indeterminism and side-effects cannot be represented by lambda calculi alone, so he extended the categorical semantics with monads to describe such features. We will see the categorical definition of the monad in Chapter 8.

However, Philip Wadler was the first one to use monads as a programming construct and utilise them to structure computations within programs. This was presented in his article "Monads for functional programming"[31]. He was at the time involved in the creation of Haskell, and needed a way of handling IO and other side effects. Before monads were used, Haskell applied two models to handle these features, one based on lazy lists and one based on continuations[14]. We will see more of continuations in chapter 10.

Since then monads have been shown to be a neat and expressive construct and used in a great amount of applications, which we will see throughout this thesis.

Even though one perhaps mostly hear about monads as a construct in Haskell or category theory, it can be, and is, defined in other languages as well, for instance other functional languages like F#[23] and Scala[21], but also imperative languages like C++[7]. The language with the most applications and most explicit use is perhaps still Haskell[7], and we will see that Haskell's syntax and type system is very much suited for monadic programming.

1.2 Motivation

Before we define the monad in the next chapter, we will give a small motivational example of one application of the monad, namely IO.

The functional programming paradigm has many interesting aspects which is appealing to most programmers: Features such as referential transparency and pattern matching makes it much easier to debug code and make secure and robust programs, but also allows functions to be memoized; higher order functions enables programmers to work and think on a greater level of abstraction, making code shorter and easier generalised; the close connection, through the Curry-Howard Correspondence, to logic makes typed functional languages ideal for making proof verifiers and theorem provers.

There are however some drawbacks to the listed features. Sadly (or perhaps luckily) the world is imperative, so one needs to compromise to implement them. One cannot for instance have only referential transparent functions, since a referential transparent function cannot have side effects. A program without any side effects is rather useless, as we can't gather any output from it. We must therefore allow side effects for some functions, like *print* and other IO-functions. But wouldn't that compromise everything? How can we know whether a function, say `f :: Int → Int`, is with or without side effects if we don't have access to the source code? And if we do gather some input, for instance asking for a character from the user, isn't that going to pollute the entire code with this destructive value? The function above could very well be defined as `f x = getInt + x`, where `getInt :: Int` gets an integer from the user.

We can use the type system to make a container for destructive values, to seal them off from the rest of the code. If we have a value with type α, we can let its destructive type be `IO` α. Now we have as type for the function defined above `f :: Int → IO Int`, and it is now obvious that this function isn't pure. Bot, say we want to apply the function `f` twice to a value `a`, how would we do this? After one application we have `(f a) :: IO Int` and all is well, but we cannot apply `f` once more, since `f` takes arguments of type `Int`, not `IO Int`. It is, of course, not a solution to add an escape function `escapeIO :: (IO a) → a`, as this would render the container pointless. Instead of letting the value out, we could rather put the function inside the container. A function with type `IO Int → (Int → IO Int) → IO Int` would do the trick.

This is exactly what the monad provides: A container with the ability to chain computations of that type together. Even greater, monadic IO is in fact pure. We will later see how this is possible. Monads are however not limited to just handling IO and destructive effects in a pure way, it is a general way of structuring computations in a separate environment.

One should also note that Haskell doesn't *need* monads, it is just a convenient way of handling IO and structuring features of a program. Furthermore, it is not a language feature, but rather a product of the ability to create abstract data types and type classes.

Chapter 2

Meet the Monad

In this chapter we will introduce the monad in a functional setting, show some properties of the construct, and look at some examples of common monads and their applications. We will start with the definition, and will build up intuition through examples along the way.

2.1 Defining the monad

As stated above, we start off quite naturally by defining the monad as it is defined in Haskell.

Definition 2.1.1. *The monad in functional programming is defined as a tuple* (`m`, `return`, $\gg=$) *through a type class as*

$$\begin{aligned}&\textbf{class}\ \texttt{Monad}\ \texttt{m}\ \textbf{where}\\ &\quad \texttt{return} :: \alpha \to \texttt{m}\,\alpha\\ &\quad (\gg=) \quad :: \texttt{m}\,\alpha \to (\alpha \to \texttt{m}\,\beta) \to \texttt{m}\,\beta\end{aligned}$$

where the following laws must hold

$$\begin{aligned}&\texttt{i)} \quad \texttt{return a} \gg= \texttt{f} \;= \texttt{f a}\\ &\texttt{ii)} \quad \texttt{ma} \gg= \texttt{return} \;= \texttt{ma}\\ &\texttt{iii)}\ \texttt{ma} \gg= \texttt{f} \gg= \texttt{g} = \texttt{m} \gg= \lambda\,\texttt{a} \to (\texttt{f a} \gg= \texttt{g})\end{aligned}$$

Monads are thought of as a structure defining a type of computation, where an element in the monad, `ma` $::\ \texttt{m}\,\alpha$, is a *computation* having a result (or results) of type α; `return a` evaluates to the trivial computation immediately returning `a`; ($\gg=$) binds computations together, such that `ma` $\gg=$ `f` evaluates `ma` and applies the function `f` to the result (or results) of this computation, which again results in a new computation. The meaning behind the words *computation* and *bind* will become clearer after some examples.

It is easier to remember the laws of the monad if we rewrite them with monadic function composition, called Kleisli Composition, which is defined as

$$(\odot) :: (\beta \to \texttt{m}\,\gamma) \to (\alpha \to \texttt{m}\,\beta) \to \alpha \to \texttt{m}\,\gamma$$

```
f ⊙ g = λa → f a ≫= g
```

Now the laws can be restated as

```
return ⊙ f  = f
f ⊙ return  = f
(f ⊙ g) ⊙ h = f ⊙ (g ⊙ h)
```

That is, $(\odot)$ is associative and has `return` as identity, or alternatively put, (`return`, $\odot$) is a monoid.

We said that `return` should construct the trivial computation, and this is captured in the two first laws. Furthermore, $(\gg=)$ is chaining computation together, and the third law just states that this chaining of computations behaves nicely.

Example 2.1.2. *A simple yet useful monad is the* `Maybe` *monad. This is a monad for working with computations with the possibility of failure. For this we first need to define the monadic type.*

```
data Maybe α = Just α | Nothing
```

So a computation of type `Maybe` α *can either be successful and be on the form* `Just a`*, having result* `a`*, or have failed and be* `Nothing`*. The* `Nothing` *value is similar to the* `Null` *pointer in Java or the* `void` *value in Python. The monad instance for the* `Maybe` *type is given by*

```
instance Monad Maybe where
  return        = Just
  Nothing ≫= _  = Nothing
  Just a ≫= f   = f a
```

`return` *just wraps the argument in* `Just`*. The value of a failed computation can't be used in a new computation, since it does not contain a value. Hence binding a failed computation to a new computation results in a failed computation. If we have a successful computation, we can use its value in a new computation.*

We can now use this monad to execute previously unsafe operations, safely.

```
safeDiv    :: Float → Float → Maybe Float
safeDiv n 0 = Nothing
safeDiv n m = return $ n/m


safeDivFive :: Float → Maybe Float
safeDivFive = safeDiv 5
safeDivFive 3 ≫= safeDivFive = Just (5/(5/3))
safeDivFive 0 ≫= safeDivFive = Nothing
```

Notice how we don't need any if-tests while chaining these computations. In Java or other pointer based languages one often encounters code like

```
String f(String str) {

  if(str == null){
      return null;
  }
  //Rest of method...
}
```

but by using the `Maybe` *monad, we have already specified how our computations should treat a failed computation.*

If a computation might fail, it could be interesting to be able to throw an exception to inform later functions what went wrong. This might also be achieved through a monad, shown in the following example.

Example 2.1.3. *The exception monad is defined as*

```
data Exceptional ϵ α = Exception ϵ | Success α

instance Monad (Exceptional ϵ) where
  return            = Success
  Exception b >>= _ = Exception b
  Success a >>= f   = f a
```

Chaining computations can now be done safely, and if some subcomputation fails with an exception, the entire computation fails with that exception. Exceptions can be thrown and caught with

```
throw  :: ϵ → Exceptional ϵ α
throw e = Exception e


catch :: Exceptional ϵ α → (ϵ → Exceptional ϵ α) → Exceptional ϵ α
catch (Exception e) h = h e
catch (Success a) _   = Success a
```

Now we can define

```
safeDiv    :: Float → Float → Exceptional String Float
safeDiv n 0 = throw "Divided by 0"
safeDiv n m = return $ n/m


safeDivFive :: Float → Exceptional String Float
safeDivFive = safeDiv 5
safeDivFive 3 >>= safeDivFive = Success (5/(5/3))
safeDivFive 0 >>= safeDivFive = Exception "Divided by 0"
```

As we saw in the previous examples, monads enables us to state *how* our computations should behave. We will later see that monads are much more expressive that just handling null-values and exceptions, but can in fact handle state, input and output, backtracking, and much more. We will now look at an alternate but equivalent definition, which is sometimes more convenient to use.

2.2 Alternate definition

In the previous section we defined the functional monad as it is defined in Haskell. There is however another definition which is more convenient in some cases, specially when working with monads which are containers. We will see an example of this shortly. This definition depends on a structural mapping, known as a functor.

Definition 2.2.1. *A functor is defined as*

```
class Functor f where
  fmap :: (α → β) → f α → f β
```

with the following requirements

```
i)  fmap id     = id
ii) fmap (f . g) = (fmap f) . (fmap g)
```

for `id` $:: \alpha \to \alpha$*, the identity function over all types.*

The functor, which in category theory is a mapping from one category to another, applies functions uniformly over a data type in the functional setting. We will look closer at the categorical functor in Chapter 8.

Example 2.2.2. *The* `Functor` *instance for the* `Maybe` *type can be defined as*

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

For lists, it can be defined as

```
instance Functor [ ] where
  fmap f [ ]     = [ ]
  fmap f (a : as) = (f a) : (fmap f as)
```

There is already a function `map` $:: (\alpha \to \beta) \to [\alpha] \to [\beta]$ *in the Haskell libraries, so we could just set* `fmap = map`*.*

Extending a functor to a monad can be done by defining the two functions `return` and `join`.

Definition 2.2.3. *A monad might be defined through the join function as follows,*

```
class Functor m ⇒ Monad m where
  return :: α → m α
  join   :: m (m α) → m α
```

where the following must hold

```
i)   join . (fmap join)   = join . join
ii)  join . (fmap return) = join . return = id
iii) join . (fmap (fmap f)) = (fmap f) . join
iv)  (fmap f) . return    = return . f
```

The `join` function is thought of as flattening a computation of a computation to a single computation. The equivalence of the two definitions is not difficult to show, but first we need a lemma. In the next proofs we write "$\overset{n}{=}$" if the equality holds due to the nth law in some assumed definition.

Lemma 2.2.4. *The definition of a monad using* $(\gg=)$ *implicitly defines a functor.*

Proof. Set

$$\texttt{fmap f a} = \texttt{a} \gg= \texttt{return . f}$$

In this proof we assume the laws of the monad in the first definition and will deduce the functor laws. The first law is easily seen by

$$\begin{aligned}(\texttt{fmap id})\,\texttt{ma} &= \texttt{ma} \gg= \texttt{return . id} \\ &= \texttt{ma} \gg= \texttt{return} \\ &\overset{2}{=} \texttt{ma} \\ &= \texttt{id ma}\end{aligned}$$

The equality in the second law of the functor is obtained by

$$\begin{aligned}&(\texttt{fmap f}) \,.\, (\texttt{fmap g})\,\$\,\texttt{ma} \\ &= \texttt{fmap f}\,(\texttt{ma} \gg= \texttt{return . g}) \\ &= (\texttt{ma} \gg= \texttt{return . g}) \gg= \texttt{return . f} \\ &\overset{3}{=} \texttt{ma} \gg= \lambda\,\texttt{x} \to (\texttt{return}\,(\texttt{g x}) \gg= \texttt{return . f}) \\ &\overset{1}{=} \texttt{ma} \gg= \lambda\,\texttt{x} \to \texttt{return . f . g}\,\$\,\texttt{x} \\ &= \texttt{ma} \gg= \texttt{return . f . g} \\ &= \texttt{fmap}\,(\texttt{f . g})\,\texttt{ma}\end{aligned}$$

□

Proposition 2.2.5. *The two definitions of the monad are equivalent.*

Proof. The lemma gave us a way of defining `fmap`, and we can define `join` from $(\gg=)$ by

$$\texttt{join a} = \texttt{a} \gg= \texttt{id}$$

for $\texttt{a} :: \texttt{m}\,(\texttt{m}\,\alpha)$. That is, we just pierce `a` and does nothing else with it. The other way is a bit harder, but by following the type signatures we get

$$\texttt{a} \gg= \texttt{f} = \texttt{join}\,\$\,\texttt{fmap f a}$$

What remains is to show that the laws are equivalent. Assume the laws of the definition using $(\gg=)$. Then we have the following,

$$\begin{aligned}\texttt{join}\,\$\,\texttt{fmap join a} &= \texttt{join}\,\$\,\texttt{a} \gg= \texttt{return . join} \\ &= \texttt{a} \gg= \texttt{return} \gg= \texttt{id} \gg= \texttt{id} \\ &\overset{2}{=} \texttt{a} \gg= \texttt{id} \gg= \texttt{id} \\ &= \texttt{join}\,\$\,\texttt{join a}\end{aligned}$$

For the second join law, we have for the left hand side

$$\begin{aligned}\texttt{join}\,\$\,\texttt{fmap return a} &= \texttt{join}\,\$\,\texttt{a} \gg\!= \texttt{return . return} \\ &= \texttt{a} \gg\!= \texttt{return . return} \gg\!= \texttt{id} \\ &\overset{2}{=} \texttt{return a} \gg\!= \texttt{id} \\ &\overset{1}{=} \texttt{id a}\end{aligned}$$

and for the other side

$$\begin{aligned}\texttt{join . return a} &= \texttt{return a} \gg\!= \texttt{id} \\ &\overset{1}{=} \texttt{id a}\end{aligned}$$

The third law can be seen to hold by the following steps,

$$\begin{aligned}&\texttt{join}\,\$\,\texttt{fmap}\,(\texttt{fmap f})\,\texttt{a} \\ &= \texttt{join}\,\$\,\texttt{a} \gg\!= \texttt{return}\,.\,(\texttt{fmap f}) \\ &= \texttt{a} \gg\!= \texttt{return}\,.\,(\texttt{fmap f}) \gg\!= \texttt{id} \\ &\overset{3}{=} \texttt{a} \gg\!= \lambda\,\texttt{x} \to \texttt{return}\,(\texttt{x} \gg\!= \texttt{return . f}) \gg\!= \texttt{id} \\ &\overset{1}{=} \texttt{a} \gg\!= \lambda\,\texttt{x} \to \texttt{id}\,(\texttt{x} \gg\!= \texttt{return . f}) \\ &= \texttt{a} \gg\!= \lambda\,\texttt{x} \to (\texttt{x} \gg\!= \texttt{return . f}) \\ &= \texttt{a} \gg\!= \lambda\,\texttt{x} \to (\texttt{id x} \gg\!= \texttt{return . f}) \\ &\overset{3}{=} \texttt{a} \gg\!= \texttt{id} \gg\!= \texttt{return . f} \\ &= \texttt{fmap f}\,(\texttt{join a})\end{aligned}$$

For the last law, observe

$$\begin{aligned}\texttt{fmap f}\,(\texttt{return a}) &= \texttt{return a} \gg\!= \texttt{return . f} \\ &\overset{1}{=} \texttt{return}\,(\texttt{f a})\end{aligned}$$

Now for the other way around, assume the laws using `join`, `return`, and `fmap`. Here we write "$\overset{\texttt{n}'}{=}$" for the nth `fmap` law. We now have for the first law

$$\begin{aligned}\texttt{return a} \gg\!= \texttt{f} &= \texttt{join}\,\$\,\texttt{fmap f}\,(\texttt{return a}) \\ &\overset{4}{=} \texttt{join}\,\$\,\texttt{return}\,(\texttt{f a}) \\ &\overset{2}{=} \texttt{f a}\end{aligned}$$

For the second law, we have

$$\begin{aligned}\texttt{a} \gg\!= \texttt{return} &= \texttt{join}\,\$\,\texttt{fmap return a} \\ &\overset{2}{=} \texttt{id a} \\ &= \texttt{a}\end{aligned}$$

The third law is a bit more work. For the left hand side, we derive

$$\begin{aligned}\texttt{a} \gg\!= \texttt{f} \gg\!= \texttt{g} &= \texttt{join}\,(\texttt{fmap f a}) \gg\!= \texttt{g} \\ &= \texttt{join}\,\$\,\texttt{fmap g}\,(\texttt{join}\,\$\,\texttt{fmap f a}) \\ &= \texttt{join}\,\$\,(\texttt{fmap g})\,.\,\texttt{join}\,.\,(\texttt{fmap f})\,\texttt{a} \\ &\overset{3}{=} \texttt{join}\,\$\,\texttt{join}\,.\,(\texttt{fmap}\,(\texttt{fmap g}))\,.\,(\texttt{fmap f})\,\texttt{a}\end{aligned}$$

$$\overset{2'}{=} \texttt{join} \,\$\, \texttt{join} \,.\, (\texttt{fmap}\, ((\texttt{fmap}\, \texttt{g}) \,.\, \texttt{f}))\, \texttt{a}$$

whereas for the right hand side, we obtain

$$\begin{aligned}
&\texttt{a} \gg\!= \lambda\, \texttt{x} \to (\texttt{f}\, \texttt{x} \gg\!= \texttt{g}) \\
&= \texttt{join} \,\$\, \texttt{fmap}\, (\lambda\, \texttt{x} \to (\texttt{f}\, \texttt{x}) \gg\!= \texttt{g})\, \texttt{a} \\
&= \texttt{join} \,\$\, \texttt{fmap}\, (\lambda\, \texttt{x} \to \texttt{join}\, (\texttt{fmap}\, \texttt{g}\, (\texttt{f}\, \texttt{x})))\, \texttt{a} \\
&= \texttt{join} \,\$\, \texttt{fmap}\, (\texttt{join} \,.\, (\texttt{fmap}\, \texttt{g}) \,.\, \texttt{f})\, \texttt{a} \\
&\overset{2'}{=} \texttt{join} \,\$\, (\texttt{fmap}\, \texttt{join}) \,.\, (\texttt{fmap}\, ((\texttt{fmap}\, \texttt{g}) \,.\, \texttt{f}))\, \texttt{a} \\
&\overset{1}{=} \texttt{join} \,\$\, \texttt{join} \,.\, (\texttt{fmap}\, ((\texttt{fmap}\, \texttt{g}) \,.\, \texttt{f}))\, \texttt{a}
\end{aligned}$$

and we are done. □

This gives us two ways to look at monads in a functional setting. Either a monad can be viewed as a way of chaining computations together, or a way of both lifting functions to the computational level and also flattening computations of computations to just computations. We have proved the equivalence of the two definitions of a monad, and can therefore use whichever definition we prefer. Let's look at an example of the two definitions.

Example 2.2.6. *The* `List` *monad is a very common monad, and has the following definition:*

```
instance Monad [] where
  return a = [a]
  ma >>= f = concat $ map f ma
```

Here `map` *was the function we saw above. The function*

```
concat         :: [[α]] → [α]
concat []      = []
concat (a : as) = a ++ (concat as)
```

is also a regular Haskell functions from the library. `concat` *takes a list of lists* `L` *and returns a list of all the elements of the lists inside* `L`*, whereas* `map` *takes a function and a list, and applies the function to each element in the list. The alert reader may have observed that what we really have done is defined the definition using* (≫=) *through the join definition, since for lists* `join = concat` *and* `fmap = map`*. One could also define it directly through the* (≫=) *operator as*

```
instance Monad [] where
  return a       = [a]
  [] >>= f       = []
  (a : as) >>= f = (f a) ++ (as >>= f)
```

The `List` *monad is often seen as a way of combining indeterministic computation, that is, computations that can return multiple values. Assume you play a dice game where you can bet an amount* `A` *of money, and you win*

`A` *times* (`f r`)*, where* `f` *is some function and* `r` *is the roll of a die. We then have a function*

```
game  :: int → [int]
game a =  [a ∗ (f 1), a ∗ (f 2), . . . , a ∗ (f 6)]
```

computing the possible winnings of a single play. You are of course interested to see what you can expect in the long run of this game. The following functions give you the answer:

```
play       :: [int] → int → [int]
play bets n = if n == 0 then
                bets
              else
                play (bets ≫= game) (n − 1)

earnings      :: int → int → float
earnings bet n = (mean $ play (return bet) n) − bet
```

where `n` *is the number of games to play,* `bet` *is the starting bet, and* `mean` :: [`int`] → `float` *calculates the mean of a list. Then* (`earnings bet n`) *will give you the expected earnings if you start betting* `bet`*, and bet everything you win for* `n` *games.*

The `Monad` class has some extra utility functions defined. Monads can, as stated earlier, simulate destructive effects, so in some cases we are not interested in the result of a computation, but rather its effect. Take e.g. the result of applying `print` :: `Show` $\alpha \Rightarrow \alpha \rightarrow$ `IO` (), which is a function in the `IO` monad. For such circumstances we have the following function,

```
(≫)       :: m α → m β → m β
ma ≫ mb = ma ≫= λ_ → mb.
```

That is, execute computation `ma`, discard the result, and then execute `mb`. Any monad also has a predefined `fail` :: `String` → `m` α, as `fail s = error s`. One should however, override this to fit the logic of the instantiating type. The full definition of the Monad class in Haskell is as follows,

```
class Monad m where
  return   :: α → m α
  (≫=)     :: m α → (α → m β) → m β
  (≫)      :: m α → m β → m β
  fail     :: String → m α
  ma ≫ mb = ma ≫= λ_ → mb
  fail     = error
```

2.3 Do-notation

Since many of the applications of monads involve simulating imperative features, it is often convenient to write that code as one would in an imperative language. Because of this there is a special way of writing monadic code, using what in Haskell is called a `do`-block. In a `do`-block one writes commands just as in an imperative language, line by line. However, all such code can be (and is by the compiler) rewritten to ordinary monadic functions using lambda abstractions, ($\gg=$), and ($\gg$). That is, it is only syntactic sugar. A `do`-block starts with the keyword `do` and then a block of lines, each line a command. A command is either a call to a function, or an assignment of a value to a variable. The assignment is either using the familiar let-expression, or a monadic assignment written with a left arrow. Here is an example of a `do`-block:

```
doExample = do
              update
              b ← getSomeVal
              let c = doSomething b
              return $ c + 1
```

The best way of understanding this type of code is to see how it translates to known code. We will use $\rightsquigarrow$ to denote the translation from `do`-notation to regular monadic code. Furthermore,

```
do
  line1
  line2
  ...
  linen
```

is equivalent to

do {$\texttt{line}_1$; $\texttt{line}_2$; ...; $\texttt{line}_n$}.

The following is taken from [7], and is a desugaring of `do`-blocks in the same way a Haskell compiler would do it. A single command translates to itself, so

do `com` $\rightsquigarrow$ `com`

A series of commands in a `do`-block is translated recursively such that

do {`com`; `rest`} $\rightsquigarrow$ `com` $\gg$ **do** {`rest`}

A let statement in a `do`-block behaves the same, albeit we do not need the *in* keyword.

do {**let** `decls`; `rest`} $\rightsquigarrow$ **let** `decls` **in do** {`rest`}

The monadic assignment is trickier to translate. We have that

$$\textbf{do}\,\{\texttt{a} \leftarrow \texttt{ma}; \texttt{rest}\} \rightsquigarrow \texttt{ma} \gg\!= \lambda\,\texttt{a} \rightarrow \textbf{do}\,\{\texttt{rest}\}$$

So whenever we write `a` $\leftarrow$ `exp`, the monadic value `exp` is computed, such that `a` is assigned the result of the computation. So for instance, if we had a line `a` $\leftarrow$ `Just` 1, then `a` would get the value 1. However, for `a` $\leftarrow [1, 2, 3]$ in the `List` monad, then `a` would first be assigned 1, then 2, and lastly 3, and the `do`-block is executed once for each value.

The small example above is translated to

```
doExample = update ≫
                getSomeVal ≫= λb →
                   let
                     c = doSomething b
                   in
                   return $ c + 1
```

Let's look at an example using this new notation coupled with a rather useful monad.

Example 2.3.1. *Manipulating state is alpha and omega in imperative programming, and some algorithms are more intuitively expressed through manipulation of state. However, we should not tolerate destructive effects in our pure code. One solution could be to pass a state value along to all our functions, but that is rather tiresome. The* `State` *monad comes to our rescue, simulating destructive effects in a functionally pure way.*

```
newtype State σ α = State { runState :: σ → (σ, α)}

instance Monad (State σ) where
  return a = State $ λs → (a, s)
  as ≫= f = State $ λk → let
                           (a, s) = runState as k
                         in
                         f a s
```

So something of type `State` $\sigma\,\alpha$ *is a function taking a state* `s` *and returns a tuple consisting of a state and a value;* `return a` *is just the trivial computation immediately returning its unaltered argument state value and* `a`*;* (`sa` $\gg\!=$ `f`) *unwraps* `sa` *with the current state* `k`*, applies* `f` *to the unwrapped value, and then feeds the new state* `s` *to the result.*

Notice that the state value is actually passed along every monadic function call, but is now done automatically. As with the `Maybe` *monad, we again had a feature we wanted our computations to (uniformly) have, and again the defined monad takes care of it in the background.*

Now, let's return to your gambling situation from the previous example. Perhaps betting everything you own isn't the best tactic. Perhaps you would like to just bet a fraction of what you own. Let's see how the state monad can help you keep track of how much you have, and how much you should bet. For our example, let's say you would like to bet $\frac{1}{3}$ *of what you have. We will need two very helpful functions defined in the* `State` *module in the Haskell libraries. The helpful functions are as follows:*

```
modify  :: (s → s) → State s ()
modify f = State $ λ s → ((), f s)

get :: State s s
get = State $ λ s → (s, s)
```

So `modify` *takes a function and uses it to modify the state value, whereas* `get` *sets the result of the computation to be equal to the state value. We are now ready to define functions for our gambling simulation.*

```
won  :: Int → State Int Int
won n = do
          modify (+ n)
          money ← get
          let nextBet = floor . fromIntegral $ money/3.0
          return nextBet

playGames          :: Int → Int → (Int, Int)
playGames start n = if n == 0 then
                      return start
                    else
                      playGames start (n − 1) >>= won . play
```

Here `play :: Int → Int` *is the winnings from one play,* `startBet` *is the first bet, and* `n` *is the number of games to play. In the state's tuple* (`a`, `s`)*, the state* `s` *is how much money you currently have and the value* `a` *is how much you want to bet the next round. The* `won` *function can be translated to*

```
won n =
  modify (+ n) >>
     get >>= λ money →
         let
           nextBet = floor . fromIntegral $ money/3.0
         in
         return nextBet
```

which further reduces to

```
won n = State $ λ k → ((floor . fromIntegral $ (k + n)/3.0), (k + n))
```

by β*-reduction.*

The monad laws can also be expressed in `do`-notation, as

```
do                =  do
  block               block
  x ← return a        rest[a/x]
  rest

do                =  do
  block               block
  a ← ma              ma
  return a

do                =  do
  block               block
  a ← do              com_1
         com_1        ⋮
         ⋮            a ← com_n
         com_n        rest
  rest
```

The laws are rather natural in this context, almost taken for granted one might think. It would probably be quite confusing to write code in `do`-notation if these qualities didn't hold. This is an important note, monad instances not satisfying the monad laws are actually unintuitive to work with.

We will now look at some of the functionality written around monads, both useful extensions and practical functions.

2.4 Auxiliary functions

In addition to the functions defined in the monad class, there are several type class extensions equipping monads with more functionality. One rather useful extension is the `MonadPlus` class.

Definition 2.4.1. *The* `MonadPlus` *class is defined as*

```
class Monad m ⇒ MonadPlus m where
  mzero :: m α
  mplus :: m α → m α → m α
```

Where we require (a subset of)

```
mzero `mplus` ma          = ma
ma `mplus` mzero          = ma
ma `mplus` (mb `mplus` mc) = (ma `mplus` mb) `mplus` mc
mzero >>= f               = mzero
ma >> mzero               = mzero
(ma `mplus` mb) >>= f     = (ma >>= f) `mplus` (mb >>= f)
```

where `` `mplus` `` *is just infixed* `mplus`*.*

One should note that the laws are not fully agreed upon, but the once stated above are perhaps the most canonical[2]. Different choices of laws gives different interpretations of both `mzero` and `mplus`. One interpretation allows us to make a monoid (the three topmost laws) over computations with the binary operator `mplus` and an accompanying identity `mzero`. The idea is to let `mzero` denote a failed computation, and have `mplus` combine the result of its two arguments. A common interpretation of `mplus` is that it represents a choice over its two argument computations. For the `Maybe` monad, `mzero` = `Nothing` and `mplus` returns the first successful computation, if any. For the `List` monad, `mzero` = [ ] and `mplus` = ++.

There is currently a reform proposal[2] which suggest to split the `MonadPlus` class into different classes, each satisfying one of the most common subsets of the laws above.

There are also several other extensions of the `Monad` class, and we will see some of them later in the thesis. We will now look at some useful monadic functions which we will apply throughout the thesis.

We saw how a functor `F` defines functionality for lifting any function $\texttt{f} :: \alpha \to \beta$ up to its type, such that $(\texttt{fmap f}) :: \texttt{F}\,\alpha \to \texttt{F}\,\beta$. We also saw how every monad uniquely defines a functor. There is an implementation of this lifting defined for every monad as

```
liftM    :: Monad m ⇒ (α → β) → m α → m β
liftM f ma = do
               a ← ma
               return $ f a
```

The definition above is equivalent (just written in `do`-notation) to the one for `fmap`, but now with the assumption that the data type is a monad. Even though every monad implicitly defines a functor, there need not be an explicit implementation of `fmap`. In such cases, we can however always use `liftM`. Applying `liftM` is called promoting a function to a monad. This is very useful, but we are not limited to only promoting functions of one argument. The expressiveness of monads lets us define this for functions of an arbitrary number of arguments. For two arguments, we have

```
liftM2 :: Monad m ⇒
          (α1 → α2 → β) → m α1 → m α2 → m β
liftM2 f ma1 ma2 = do
                     a1 ← ma1
                     a2 ← ma2
                     return $ f a1 a2
```

So for any natural number `n` we could define

```
liftMn :: Monad m ⇒
          (α1 → α2 → ... → αn → β) →
          m α1 → m α2 → ... → m αn → m β
```

```
liftMn f ma1 ma2 ... man = do
                             a1 ← ma1
                             a2 ← ma2
                             ⋮
                             an ← man
                             return $ f a1 a2 ... an
```

In the Haskell library there are definition up to `liftM5`, and one probably rarely needs more.

So far we have used the application function (`$`) quite a lot, albeit mainly to avoid parenthesis. There is a monadic version of this, defined as

```
ap :: Monad m ⇒ m (α → β) → m α → m β
ap = liftM2 id
```

This function lets us apply a function inside the monad to values also inside the monad. Say for instance we would have a computation in the `State` monad, where the state value was a function with type $\alpha \to \beta$. Then

```
get :: State (α → β) (α → β)
```

Normally, to apply this function, one would have to unwrap both the function and the argument from the state computation, apply it, and then wrap the result into the computation again, that is

```
do
  f ← get
  a ← ma
  return $ f a
```

Now we can just write

```
ap get ma
```

Another neat feature is that

```
(return f) `ap` ma1 `ap` ma2 `ap` ... `ap` man
```

is equivalent to

```
liftMn f ma1 ma2 ... man
```

so defining, for example, `liftM7` can now be done in one line, if one ever should need it.

There are some neat functions for controlling the flow of computations inside monads and we will now look at a couple of such functions. One often needs to filter lists, and to do this in the `List` monad, one could write

```
do
  x ← ma
  if p x then
    return x
  else
    [ ]
```

This is however tedious if you want to have several filters. This is where the neat function `guard :: MonadPlus m ⇒ Bool → m ()` is handy. It has the simple implementation

```
guard True  = return ()
guard False = mzero
```

All it does, is cut off a computation if its argument is `False`. So the above filter could rather be written

```
do
  x ← ma
  guard (p x)
  return x
```

There are two other useful functions for conditional execution of monadic expression, namely `when :: Monad m ⇒ Bool → m () → m()` and `unless` with the same type. In the expression `when b ma`, if `b` is `True` then `ma` is executed, but will return `return ()` if not. `unless` does just the opposite.

The last function we will discuss is

```
sequence           :: Monad m ⇒ [m α] → m [α]
sequence [ ]       = return [ ]
sequence (ma : mas) = do
                        a ← ma
                        as ← sequence mas
                        return (a : as)
```

and is defined in the library of the `List` monad and is useful in conjunction with any monad `m`. It takes as argument a list of computations in `m`, and executes each one in order. Every result is gathered in a list and wrapped in one large computation inside `m`. Notice how every result now is dependent upon the same computation. We will see applications of this function later in our extended examples.

There is a myriad of other useful functions, and most of them can be looked up in the Haskell library in the package `Control.Monad`.

2.5 The IO monad

We started this thesis by explaining a problem in purely functional programs, namely input and output. In this section we investigate how one particular monad can solve this problem, such the language remains pure. This particular monad is (not surprisingly) the `IO` monad. The following is taken from [17], and a more thorough discussion can be read there.

The `IO` monad is in many ways similar to the `State` monad since it actually has a state it passes along with its computations. It differs in the way that the states type is not something we can chose. The state in the `IO` monad is a representation of the real world, with a type not accessible to programmers.

There are two ways of implementing the `IO` monad in a pure language: Either the definition of the `IO` monad is not in language, but in a lower level language (like C) such that the monadic functions are primitives; or it could be defined as a normal monad like the following,

```
newtype IO α = IO { runIO :: RealWorld → (α, RealWorld)}

instance Monad IO where
  return a = IO $ λw → (a, w)
  ma >>= f = IO $ λw → let
                         (a, w') = runIO ma w
                       in
                       f a w'
```

This is how it is done in GHC[17].

There are two vital features of this monad. The first is that every time we print something, ask for a character from the user, or use any other IO-function, the internal representation of the real world is updated. The updated representation should never have been used before, since the current state of the world is never equal to a previous state.

The other vital feature of the `IO` monad is that it is inescapable, hence there is nothing of type `RealWorld` available to a programmer. That means that there is no function of type `f` :: (`IO` α) $\rightarrow$ β for a β not in the `IO` monad. To see the importance of this, let's assume there was a function like the one above. Assume also that we have a function `getInput` :: `IO` α, taking some input of type α from an external source (user, file, etc.). We would now have a function `g` = (`f getInput`) where `g` :: β, so assume we observe `g` = `b` for some value `b`. Then, by referential transparency, we should be able to substitute `g` by `b` everywhere in the code. But `g` is taking input from an external source, so we might not always get the same value. If we, on the other hand, had `f` :: `IO` α $\rightarrow$ `IO` β, we would have `g` :: `IO` β. Now `g`'s return value would depend on the current state of the world, which is given to `g` as an argument. As stated above the representation of the world is updated every time we interact with it, so we will never give `g` exactly the same arguments and is therefore not substituted.

These two features preserves referential transparency, and thereby provides one solution to the issue of combining IO with referential transparency in purely functional languages.

2.6 Perspectives on monads

As we have seen throughout this chapter, the monad can be applied to many different types of problems and we will see more in the chapters to come. Even though the monads are useful constructs and there are dozens of tutorials on the subject, they are somewhat notorious for being difficult to understand. However, many are debating this view, for instance "Real World Haskell", stating that monads really are quite simple[7], and that monads appear quite naturally from the functional paradigm.

Until now we have described monads as a construct simulating different kinds of computations. However, there are many perspectives on how one could interpret monads, so here we will give some examples of such interpretations.

Some of the monads we have seen, for instance `List` and `Maybe`, are naturally interpreted as containers. They represents data structures which contain values. With this interpretation, `return` creates a singleton container; `fmap` takes a function and a container and applies the function to every element in the container;$(\gg\!=)$ takes two arguments, the first is a container, and the second is a function which transforms a value to a container, and applies the function over each element in the container before flattening the resulting container of containers down to just a container.

Apart from data structures, the IO monad can also be seen as a container. Here, `c` :: `IO Char` is a character inside the `IO` container. As explained in the previous section, this container is inescapable.

Monads have, as we saw earlier, a special notation, namely the `do`-notation. This gives rise to yet another interpretation, monads as programmable semicolon. In a `do`-block we saw that at the end of every line there really is a $(\gg\!=)$. So $(\gg\!=)$ is used in quite a similar fashion as a semicolon, as they both define the borders between two commands. If we change the definition of $(\gg\!=)$ we change how a command should be interpreted, thereby making $(\gg\!=)$ a *programmable* semicolon.

Monads can also be viewed as a context, such that `ma` :: `m` α is a value depending a context specified by the monad. This is certainly the case for the State and IO monads. If we have a value (`State` (λ `s` $\rightarrow$ (`a` + `s`, `s`))) :: `State` σ α, we can clearly see that the result or value of the computation depends on a context, namely the state value `s`.

These perspectives are of course only to help our intuition, as the definition of a monad is equivalent no matter our perspective on that monad.

This completes our general discussion on monads. In the next section, we will see how we can combine monadic features to create new monads, as well as create monad morphisms.

Chapter 3

Combining Monads

We have until now seen how we can create a monad for a single type of computation, like indeterminism or statefulness. In most cases you will need more complex computations, and in such situations we can use monad transformers. They combine features and functionality of different monads into one single monad.

We will also look at monad morphisms, their properties, and examples of such morphisms.

3.1 Monad transformers

A monad transformer is a function over monads in the sense that its kind is $* \to * \to *$, where the first argument should be a monad, and the result of applying a transformer to a monad should also be a monad. Hence, a monad transformer can be viewed as a morphism from monads to monads, where the resulting monad is extended with one specific feature defined by the transformer. Monad transformers are defined in a similar way as a regular monad, but where we abstract over all argument monads.

Definition 3.1.1. *A monad transformer is a type* $\mathtt{t} :: * \to * \to *$ *such that for any given monad* $\mathtt{m}$, $(\mathtt{t}\,\mathtt{m})$ *is a monad. That means we should have definitions for*

$$\begin{aligned} &\mathtt{return} :: \mathtt{Monad}\,\mathtt{m} \Rightarrow \alpha \to \mathtt{t}\,\mathtt{m}\,\alpha \\ &(\gg\!=) \quad :: \mathtt{Monad}\,\mathtt{m} \Rightarrow \mathtt{t}\,\mathtt{m}\,\alpha \to (\alpha \to \mathtt{t}\,\mathtt{m}\,\beta) \to \mathtt{t}\,\mathtt{m}\,\beta \end{aligned}$$

satisfying the monad laws.

Let's look at the implementation of the transformer for the `Maybe` monad.

Example 3.1.2. *The* `Maybe` *monad transformer* `MaybeT` *is defined in Haskell in the following way:*

$$\mathbf{newtype}\ \mathtt{MaybeT}\,\mathtt{m}\,\alpha = \mathtt{MaybeT}\,\{\,\mathtt{runMaybeT} :: \mathtt{m}\,(\mathtt{Maybe}\,\alpha)\,\}$$

$$\mathbf{instance}\ \mathtt{Monad}\,\mathtt{m} \Rightarrow \mathtt{Monad}\,(\mathtt{MaybeT}\,\mathtt{m})\ \mathbf{where}$$

```
return  = MaybeT . return . Just
x >>= f = MaybeT $ do
                     v ← runMaybeT x
                     case v of
                        Nothing → return Nothing
                        Just y  → runMaybeT (f y)
```

Notice that the `do`*-block (in the definition of (*$\gg=$*)) is in the argument monad* `m`*, and that all we really do is pierce the two layers of computation and apply* `f` *to that value. The first line in the do block retrieves the* `Maybe` *value from the argument monad, and the case retrieves the result inside the* `Maybe` *monad (if any). The function* `f` *is then applied, and the result is wrapped up in the* `MaybeT` *constructor.*

Checking the laws should be rather straight forward for the first two, and the third can be proven by induction.

Observe how the `Maybe` monad is the innermost monad in the example above. This is so for any monad transformer, its argument monad is wrapped around the transformer monad. This is a crucial point, due to the importance of the order of the monads in a monad stack. Take e.g. `MaybeT (State` σ`)`. This monad is quite different from `(StateT` σ `Maybe)`. The first is a monad handling state where the values in the computation may be `Nothing`, such that the interior of a computation may look like either λ `s` $\rightarrow$ `(Just a, s)` or λ `s` $\rightarrow$ `(Nothing, s)`. The latter is a monad where we may have a value, and that value is a function over states. The interior of such a computation looks like either λ `s` $\rightarrow$ `Just (a, s)` or λ `s` $\rightarrow$ `Nothing`. Notice how this is a *combination of the functionality* of the two monads, and is not the same as the trivial combination of monads, like `State` σ `(Maybe a)` or `Maybe (State` σ `a)`. With such values the (>>=)-function can only work on one monad at the time.

Another important point is that there really is no limit as to how many monads you can combine. Feeding a monad transformer a new monad transformer enables you to add yet another monad. Say we would like `IO` on top of the two other monads in the first example above, we could just type our elements with type `MaybeT (StateT` σ `IO)`. This is extremely useful, as it makes adding or removing functionality quite easy. We will see how this is done in the extended examples.

Not all monads can be made monad transformers though. That does not mean it is impossible to implement the monad class, but rather that the resulting monads are not proper monads in the sense that they do not satisfy the monad laws. The transformer for the `List` monad is one such example.

Example 3.1.3. *The* `List` *monad has the following transformer.*

```
newtype ListT m α = ListT { runListT :: m [α] }

instance Monad m ⇒ Monad (ListT m) where
```

```
return     = ListT . return . return
mla >>= f = ListT $ do
                      la  <- runListT mla
                      lla <- sequence $ liftM (runListT . f) la
                      return $ concat lla
```

Notice how this really isn't that different from the `MaybeT` *transformer. After unwrapping* `mla` *to* `la` $::$ $[\alpha]$*, we map* (`runListT . f`) *over it. The result has type* $[(\texttt{m}\,[\alpha])]$*, so after applying* `sequence` *we get something of type* $\texttt{m}\,[[\alpha]]$*, which then is pierced and concatenated before injected into* `m` *again.*

(`ListT m`) is only a monad if the argument monad `m` is commutative, that is, monads where

```
do
  a <- exp1
  b <- exp2
  rest
```

is equivalent to

```
do
  b <- exp2
  a <- exp1
  rest
```

A proof of this can be seen by Jones and Duponcheel in "Composing Monads"[16].

In the Haskell libraries, if a monad has a proper monad transformer one often just defines the transformer. Then one can define the base monad as the transformer applied to the `Identity` monad. The `Identity` monad has the trivial definition:

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
  return a = Identity a
  ma >>= f = f (runIdentity ma)
```

The `Identity` monad contains no functionality, that is, `return` is just the identity function and ($\gg=$) is the same as regular function application. Applying a monad transformer to the `Identity` monad therefore results in a monad with only the transformers functionality. So for example, the `Maybe` monad can be defined through `MaybeT` as `MaybeT Identity`, such that

```
newtype Maybe α = MaybeT Identity α
```

Now we already have the monad instance for `Maybe`, since it was defined for `MaybeT m` for any monad `m`.

3.2 Lifting

While monad transformers enable us to combine monadic functionality through `return` and (≫=), a monad often has much more functionality attached to it. Say we have constructed the monad

```
type NewMonad σ = MaybeT (State σ)
```

how can we use functions associated with the `State` monad, such as `get :: State σ σ`? If we used this in a `do`-block in the `NewMonad` monad, we would get a compiler error, since `get` doesn't have type `MaybeT (State σ) σ`. The answer is through *lifting* (not to be confused with the function lifting we discussed in Chapter 2, albeit related). Including the definition of how a monad transformer `t` should work as a monad, one also have to define how a value is lifted from an underlying monad up to a value in the transformer monad.

Definition 3.2.1. *A monad transformer defines how it* lifts *values up from an underlying monad through*

```
class MonadTrans t where
  lift :: Monad m ⇒ m α → t m α
```

such that

```
lift . return    = return
lift (ma ≫= f) = (lift ma) ≫= (lift . f)
```

The first `return` and the first (≫=) in the first and second law respectively, are from the `m` monad, while the seconds are from the composite monad `t m`.

Example 3.2.2. *Lifting in the* `MaybeT` *transformer is defined as*

```
instance MonadTrans MaybeT where
  lift = MaybeT . (liftM Just)
```

It is easy to verify that the laws hold.

We can now use `get` in our `NewMonad` as (`lift get`), which has the correct type `MaybeT (State σ) σ`.

3.3 Monad morphisms

A monad morphism is a morphism from one monad to another.

Definition 3.3.1. *For any two monads* `m` *and* `n`*, a monad morphism,* `η :: ∀ α . m α → n α`*, from* `m` *to* `n` *is a property preserving map from computations in* `m` *to computations in* `n` *such that*

```
η (return a) = return a
η (ma ≫= f) = η ma ≫= (η . f)
```

Monad transformers are monad morphisms, where `lift` plays the role of η in the above definition. There are however other monad morphisms not traditionally expressed as a monad transformer, and they do not necessarily only add functionality, but might remove or completely change the functionality of the argument monad. For instance, assume one has a monad transformer `t` and a monad `m`. Then we have a monad morphism $\texttt{lift} :: \texttt{m}\,\alpha \to \texttt{t}\,\texttt{m}\,\alpha$, but one could also, in many cases, write a dual monad morphism $\texttt{down} :: \texttt{t}\,\texttt{m}\,\alpha \to \texttt{m}\,\alpha$, removing functionality. A function perhaps not normally thought of as a monad morphism is the function $\texttt{head} :: [\alpha] \to \alpha$ extracting the first element of a list. Observe that this really is a monad morphism from the `List` monad to the `Identity` monad, and that both of the laws above are satisfied. This function is only partial though, since it fails when applied to the empty list. If we however write

$$\begin{aligned}
&\texttt{head}' && :: [\alpha] \to \texttt{Maybe}\,\alpha \\
&\texttt{head}'\,(\texttt{x} : \texttt{xs}) && = \texttt{Just}\,\texttt{x} \\
&\texttt{head}'\,[\,] && = \texttt{Nothing}
\end{aligned}$$

we have made a monad morphism from the List monad into the Maybe monad.

As we can see, there are quite a lot of functions not normally thought of as monad morphisms, but knowing that they are might help reasoning and characterisation of programs. For instance, it is a lot cheaper to calculate $\texttt{head}'\,\texttt{xs} \gg\!= \texttt{head}' \,.\, \texttt{f}$ than $\texttt{head}'\,(\texttt{xs} \gg\!= \texttt{f})$, since the first limits the number of applications of `f`.

This concludes our discussion on monad transformers and monad morphisms. Throughout the chapters 5, 6, and 7 we will apply both constructs to solve different kinds of problems.

Chapter 4

Introducing a New Monad: The Pair Monad$^{\dagger\lambda}$

Throughout this chapter, we will construct a new monad, prove its correctness, look at some interpretations of computations in that monad, and lastly define its transformer. This monad is new and to my knowledge does not exist in the libraries.

4.1 Definition$^{\dagger\lambda}$

From Lisp we know how useful pairs can be as a data structure, and we will try to achieve some of that expressiveness in the realm of monads by constructing the `Pair` monad. One criteria for the `Pair` Monad is that it should be equivalent to the `List` Monad if we restrict our constructions to lists. That is, $(\texttt{Cons}\ \texttt{a}_1\ (\texttt{Cons}\ \texttt{a}_2\ \ldots\ (\texttt{Cons}\ \texttt{a}_\texttt{n}\ \texttt{Nil})\ \ldots)) \gg\!= \texttt{f}$ and $[\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_\texttt{n}] \gg\!= \texttt{f}$ should be equivalent for all `f` and $\{\texttt{a}_\texttt{i}\}_{\texttt{i}\leq\texttt{n}}$. In addition to the `Nil` and `Cons` constructors, we also need a way of combining two pairs to one.

Definition† 4.1.1. *The* `Pair` *type is defined as*

$$\textbf{Data}\ \texttt{Pair}\ \alpha\ =\ \textbf{Nil} \mid \textbf{Cons}\ \alpha\ (\texttt{Pair}\ \alpha) \mid \textbf{Comp}\ (\texttt{Pair}\ \alpha)\ (\texttt{Pair}\ \alpha)$$
$$\textbf{deriving}\ (\texttt{Eq}, \texttt{Show})$$

`Cons`[1] behaves as (:) in regular lists. `Comp`[2] constructs a new pair of two pairs, and thereby constructs a bifurcation. `Nil` is just the empty pair. In addition to construct lists, we can with these constructors also create arbitrary trees, lists of trees, trees of lists, etc. This freedom of form in combination with its similarity to lists is quite handy, and we will see some applications of this later.

`Pair` behaves like a functor in much the same way as List does.

[1] Short for "CONStruct".

[2] Short for either "COMbine Pairs" in regular applications, or "COMPute" when representing lambda terms, as we will see soon.

Definition† 4.1.2. `Pair` *is a functor, with the following functor instance.*

```
instance Functor Pair where
  fmap f Nil          = Nil
  fmap f (Cons a p)   = Cons (f a) (fmap f p)
  fmap f (Comp p1 p2) = Comp (fmap f p1) (fmap f p2)
```

With the functor instance defined we are one step closer to a monad. However, we need two more functions before the monad can be defined. First we need the functionality for appending and concatenation of pairs.

Definition† 4.1.3. *Appending two pairs is done with*

```
appendPair                  :: Pair α → Pair α → Pair α
appendPair Nil p            = p
appendPair (Cons a p1) p2   = Cons a (appendPair p1 p2)
appendPair (Comp p1 p2) p3  = Comp (appendPair p1 p3) p2
```

and concatenation of a pair of pairs down to just a pair, is done with

```
concatPair                  :: Pair (Pair α) → Pair α
concatPair Nil              = Nil
concatPair (Cons p1 p2)     = appendPair p1 (concatPair p2)
concatPair (Comp p1 p2)     = Comp (concatPair p1) (concatPair p2)
```

Notice that we in fact can set `concatPair = join`, so now we have all we need to define the monad instance for `Pair`.

Definition† 4.1.4. *The* `Pair` *monad is defined as*

```
instance Monad Pair where
  return a = Cons a Nil
  p >>= f  = concatPair $ fmap f p
```

We can easily see that if we omit the `Comp` constructor, all of the definitions above are equivalent to `List`'s, so the `Pair` type really is a true extension of `List`. Also, there are no functions taking a term only consisting of `Nil` and `Cons` to a term containing `Comp`. This means that whenever we want to use lists, we could just as well use pairs to allow for a possible extension to trees or other similar data structures.

After proving some basic properties of the monad, we will see how we can use this bifurcation in some interesting applications.

Lemma† 4.1.5. `Pair` *is a proper functor, that is*

```
fmap id    =  id
fmap (f . g) =  (fmap f) . (fmap g)
```

Proof. Easy induction on the construction of pairs. □

Lemma† 4.1.6. *We have the following properties of the functions above:*

i) (`fmap f`) *distributes over* `appendPair`,

ii) `appendPair` *is associative,*

iii) `concatPair` *distributes over* `appendPair`.

Proof. All proofs done by induction over the construction of pairs. □

Theorem† 4.1.7. `Pair` *is a proper monad, that is*

```
return a >>= f  =  f a
ma >>= return   =  ma
ma >>= f >>= g  =  ma >>= λx → (f x >>= g)
```

for all `a` $::$ α, `ma` $::$ `Pair` α, `f` $::$ $\alpha \to$ `m` β, *and* `g` $::$ $\beta \to$ `m` γ.

Proof. The first two laws are trivial, and seen to hold by a short induction proof. The third is a bit more involved. If `ma = Nil` both sides reduce to `Nil`. Assume `ma = Comp p1 p2`. First observe that

```
(Comp p1 p2) >>= f = concatPair $ fmap f (Comp p1 p2)
                   = Comp (concatPair $ fmap f p1)
                          (concatPair $ fmap f p2)
                   = Comp (p1 >>= f)
                          (p2 >>= f)
```

Repeating the steps above with the function `g`, we get the left hand side to be

```
(Comp p1 p2) >>= f >>= g = Comp (p1 >>= f >>= g)
                                (p2 >>= f >>= g)
```

Doing the same for the right hand side, we obtain

```
(Comp p1 p2) >>= λx → (f x >>= g)
 = Comp (p1 >>= λx → (f x >>= g))
        (p2 >>= λx → (f x >>= g))
```

which is equal to the left hand side by to the induction hypothesis. Now assume `ma = Cons v p`. We then have the following by the lemma above,

```
(Cons v p) >>= f >>= g
 = concatPair . (fmap g) . concatPair $ Cons (f v) (fmap f p)
 = concatPair . (fmap g) $ appendPair (f v) (concatPair $ fmap f p)
 = appendPair (concatPair . (fmap g) $ f v)
              (concatPair . (fmap g) . concatPair . (fmap f) $ p)
 = appendPair (f v >>= g)
              (p >>= f >>= g)
```

and for the other side we have

$$
\begin{aligned}
&(\texttt{Cons}\ \texttt{v}\ \texttt{p}) \gg\!= \lambda\,\texttt{x} \to (\texttt{f}\ \texttt{x} \gg\!= \texttt{g}) \\
&= \texttt{concatPair}\ \$\ \texttt{Cons}\ (\texttt{f}\ \texttt{v} \gg\!= \texttt{g})\ (\texttt{fmap}\ (\lambda\,\texttt{x} \to \texttt{f}\ \texttt{x} \gg\!= \texttt{g})\ \texttt{p}) \\
&= \texttt{appendPair}\ (\texttt{f}\ \texttt{v} \gg\!= \texttt{g}) \\
&\qquad\qquad (\texttt{p} \gg\!= \lambda\,\texttt{x} \to (\texttt{f}\ \texttt{x} \gg\!= \texttt{g}))
\end{aligned}
$$

which also are equal by the induction hypothesis. □

Notice that the only requirements on `appendPair` for `Pair` to satisfy the monad laws, are those given in lemma 4.1.6 and $\texttt{appendPair}\ \texttt{Nil} = \texttt{id}$. This means that we could also define the third clause of `appendPair` as

$$\texttt{appendPair}\ (\texttt{Comp}\ \texttt{p}_1\ \texttt{p}_2)\ \texttt{p}_3 = \texttt{Comp}\ \texttt{p}_1\ (\texttt{appendPair}\ \texttt{p}_2\ \texttt{p}_3)$$

The reason for choosing the original over this one, is just to enable an application we will see in the next chapter.

While using this monad, we will see that it is often the case that we want to substitute one particular value with a pair. In such cases we can use the following function.

```
subs      :: Eq α ⇒ α → Pair α → α → Pair α
subs a p x = if a == x then
               p
             else
               return x
```

If we have the following pair

$$
\begin{aligned}
\texttt{p} = \texttt{Cons}\ 1\ (\texttt{Cons}\ 2\ (\texttt{Comp}\ &(\texttt{Cons}\ 3\ \texttt{Nil}) \\
&(\texttt{Cons}\ 4\ \texttt{Nil})))
\end{aligned}
$$

we can substitute 3 with $\texttt{Comp}\ (\texttt{Cons}\ 5\ \texttt{Nil})\ (\texttt{Cons}\ 6\ \texttt{Nil})$ with

$$
\begin{aligned}
&\texttt{p} \gg\!= \texttt{subs}\ 3\ (\texttt{Comp}\ (\texttt{Cons}\ 5\ \texttt{Nil})\ (\texttt{Cons}\ 6\ \texttt{Nil})) \\
&= \texttt{Cons}\ 1\ (\texttt{Cons}\ 2\ (\texttt{Comp}\ (\texttt{Comp}\ &(\texttt{Cons}\ 5\ \texttt{Nil}) \\
&&(\texttt{Cons}\ 6\ \texttt{Nil})) \\
&\qquad (\texttt{Cons}\ 4\ \texttt{Nil})))
\end{aligned}
$$

Now that we have seen the definition and proof of correctness of the `Pair` monad, we will look at applications and interpretations of this new monad.

4.2 Usage and interpretation$^{\dagger\lambda}$

As we saw in the previous section, the `Pair` monad is a generalisation of the `List` monad, the monad of indeterministic computations. So what type of computations does the `Pair` monad model? As a generalisation of the `List` monad, the `Pair` monad can of course also model indeterministic computations, but how should we then interpret the new constructor `Comp`? Throughout this section we will look at some different interpretations of the constructors, and for each interpretation see what $(\gg\!=)$ represents.

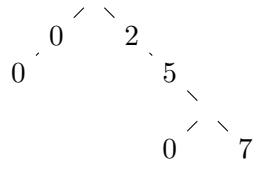
We know that for lists, a common interpretation of (:) is disjunction, that is, a list represents a disjunction of possible results from a computation. For

example, $[1, 2, 3]$ represents a computation with either result 1, or 2, or 3. $[\,]$ represents a failed computation, a computation with zero possible results. First we will look at two interpretations of pairs where we keep these interpretations of `Cons` and `Nil`, and let `Comp` represent either "or" or "and".

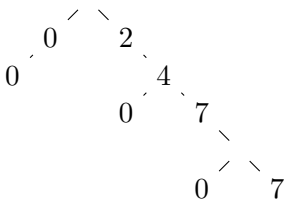
If `Comp` also represents "or", then the only difference between the `Pair` monad and the `List` monad is that `Pair` enables some structure on the result. Instead of having all possible values in a linear list, we can separate values via bifurcations. For instance in the dice example from the previous chapter, we could let each result be on the form $\texttt{Comp}\,\texttt{p}_1\,\texttt{p}_2$, where we put all 0s in $\texttt{p}_1$ and all greater numbers in $\texttt{p}_2$. Instead of having a linear result on the form

$$[0, 0, 2, 5, 0, 7]$$

we can construct the bifurcations above, and get



Notice that betting 0 in the dice game always gives 0 money back, so we will never get a non-zero value in a left subtree. Also, if one substitutes a non-zero value with a result $\texttt{Comp}\,\texttt{p}_1\,\texttt{p}_2$, the definition of `appendPair` ensures that the rest of the tree is appended to $\texttt{p}_2$. This preserves the structure also after substitution. Assume, if we bet 5 we can win either 0, 4, or 7. Making only this substitution in the above tree, we get



We could make this interpretation explicit, and by that I mean to transform a computation in the `Pair` monad to a computation in the `List` monad. We have a firm grasp of the meaning or semantic of a computation in the `List` monad, as described above. A translation from pairs to lists would then give rise to an interpretation of computations in the `Pair` monad.

Definition† 4.2.1. *A structured indeterministic computation in the* `Pair` *monad can be translated to a regular indeterministic computation with*

$$\begin{aligned}
&\texttt{pairAsStruct} && :: \texttt{Pair}\,\alpha \to [\alpha] \\
&\texttt{pairAsStruct}\,\texttt{Nil} && = [\,] \\
&\texttt{pairAsStruct}\,(\texttt{Cons}\,\texttt{a}\,\texttt{p}) && = (\texttt{a} : (\texttt{pairAsStruct}\,\texttt{p})) \\
&\texttt{pairAsStruct}\,(\texttt{Comp}\,\texttt{p}_1\,\texttt{p}_2) && = (\texttt{pairAsStruct}\,\texttt{p}_1) \mathbin{+\!\!+} (\texttt{pairAsStruct}\,\texttt{p}_2)
\end{aligned}$$

In fact, `pairAsStruct` is almost a monad morphism from pairs to lists. We have that `pairAsStruct` $(\texttt{p} \gg\!= \texttt{f})$ will have the same elements as `pairAsStruct p` $\gg\!=$ `pairAsStruct . f`, but not necessarily in the same order. If we use the alternate definition of `appendPair` as described in the end of previous section, `pairAsStruct` would be a proper monad morphism.

Before finishing of this interpretation of pairs as computations with structure, one should note that pairs have an infinite number of different failed computations. While lists only have the empty list, pairs have `Nil`, and any combination of only `Comp` and `Nil`. We can enumerate a subset of them,

$$\begin{aligned}
&\texttt{zero} \;::\; \texttt{Int} \to \texttt{Pair}\,\alpha \\
&\texttt{zero}\,0 = \texttt{Nil} \\
&\texttt{zero}\,\texttt{n} = \texttt{Comp}\,(\texttt{zero}\,(\texttt{n}-1))\,\texttt{Nil}
\end{aligned}$$

and use them to represent different types of failures. Note that $(\texttt{zero}\,\texttt{n}) \gg\!= \texttt{f} = \texttt{zero}\,\texttt{n}$ for any `n` and `f`, since there are no values contained in `zero n`. Say we have a computation in the `List` monad, `la` $\gg\!=$ `f`. If `f` applied to one of the values in `la` results in $[\,]$, this will simply disappear. Observe that if one particular value substitutes to `zero n` for $\texttt{n} > 0$ in the `Pair` monad, the failure will be a part of the resulting computation, as $\texttt{zero}\,(\texttt{n}-1)$. For example,

$$\begin{aligned}
&(\texttt{Cons}\,1\,(\texttt{Cons}\,2\,(\texttt{Cons}\,3\,\texttt{Nil}))) \gg\!= \texttt{subs}\,2\,(\texttt{zero}\,\texttt{n}) \\
&= \texttt{Cons}\,1\,(\texttt{Comp}\,(\texttt{zero}\,(\texttt{n}-1))\,(\texttt{Cons}\,3\,\texttt{Nil}))
\end{aligned}$$

So in addition to preserve some structure on our computations, we can also represent and differentiate between different failed computations.

Another possible interpretation of `Comp` is as "and". In other words, to let it represent a choice of two values, one choice from both its arguments. This interpretation makes `Comp` a Cartesian product over possible values. Although intuitive, we will see shortly that this interpretation is a bit problematic with respect to substitutions.

With such an interpretation, we cannot translate pairs with this interpretation to single level lists. However, a translation to list of lists, $[[\alpha]]$, can be done with the function defined below. Here, (:) for the top level list is interpreted as regular disjunction, while for the lists within this list we interpret (:) as conjunction.

Definition† 4.2.2. *A computation in the* `Pair` *monad can be interpreted as a choice over the composition of single values and Cartesian products over values with*

$$\begin{aligned}
&\texttt{pairAsCart} &&:: \texttt{Pair}\,\alpha \to [[\alpha]] \\
&\texttt{pairAsCart}\,\texttt{Nil} &&= [\,] \\
&\texttt{pairAsCart}\,(\texttt{Cons}\,\texttt{a}\,\texttt{p}) &&= ([\texttt{a}] : (\texttt{pairAsCart}\,\texttt{p}) \\
&\texttt{pairAsCart}\,(\texttt{Comp}\,\texttt{p}\,\texttt{p}') &&= \texttt{ap}\,(\texttt{fmap}\,(+\!\!+)\,(\texttt{pairAsCart}\,\texttt{p})) \\
&&&\qquad (\texttt{pairAsCart}\,\texttt{p}')
\end{aligned}$$

The last line might need an explanation to help our intuition. First we map $(+\!\!+)$ over the result from `pairAsCart p`, which creates a list of functions. This list has type $[[\alpha] \to [\alpha]]$. Each function in that list is then applied to every value in the result of `pairAsCart p'`. This generates a list of every possible combination of elements from each branch. For instance, if

$$\begin{aligned}\texttt{p} = (&\texttt{Cons}\ 1\ (\texttt{Comp}\ (\texttt{Cons}\ 2\ (\texttt{Cons}\ 3\ \texttt{Nil}))\\ &\qquad\qquad (\texttt{Cons}\ 4\ (\texttt{Cons}\ 5\ \texttt{Nil}))))\end{aligned}$$

then

$$\texttt{pairAsCart p} = [[1],\ [2,4],\ [2,5],\ [3,4],\ [3,5]]$$

If we let $\texttt{f} = \texttt{subs}\ 1\ (\texttt{Comp}\ (\texttt{Cons}\ 1\ \texttt{Nil})\ (\texttt{Cons}\ 0\ \texttt{Nil}))$, then

$$\texttt{pairAsCart p} \gg\!= \texttt{f} = [[0,1],\ [0,2,4],\ [0,2,5],\ [0,3,4],\ [0,3,5]].$$

The interpretation of the result of the substitution depends heavily on where in the tree the substituted value is located, but the function making the substitutions is oblivious to the placements of its argument. This makes this interpretation quite unintuitive to work with.

Above we interpreted `Cons` as "or", as with regular lists, and `Comp` as "and". We might do the reverse, interpret `Cons` as "and" and `Comp` as "or". Then every choice would represent a path from the root to a leaf node in the tree. This can be made explicit and translated to a list of lists with the same semantic as above, with the following definition.

Definition† 4.2.3. *A computation in the* `Pair` *monad can be interpreted as a choice of path from root to leaf with the function*

$$\begin{aligned}&\texttt{pairAsPathCh} &&:: \texttt{Pair}\ \alpha \to [[\alpha]]\\ &\texttt{pairAsPathCh}\ \texttt{Nil} &&= [[\,]]\\ &\texttt{pairAsPathCh}\ (\texttt{Cons a p}) &&= \texttt{fmap}\ (\texttt{a} :)\ (\texttt{pairAsPathCh p})\\ &\texttt{pairAsPathCh}\ (\texttt{Comp p p'}) &&= (\texttt{pairAsPathCh p}) +\!\!+ (\texttt{pairAsPathCh p'})\end{aligned}$$

If we let `p` and `f` be the same as above, we have that

$$\texttt{pairAsPathCh p} = [[1,2,3],\ [1,4,5]]$$

which is a list of all possible paths from root to leaves. Furthermore,

$$\texttt{pairAsPathCh \$ p} \gg\!= \texttt{f} = [[0],\ [1,2,3],\ [1,4,5]]$$

As we can see, substitutions extend and create new paths.

We have now seen interpretations of pairs as extensions of indeterministic computations. There is however another quite different interpretation of computations in `Pair`. In fact, Pairs naturally represents lambda terms and $(\gg\!=)$ can simulate β-reduction. Here `Cons a p` can represent a lambda abstraction of `a` over `p`, `Comp p1 p2` can represent the application of `p1` to `p2`, and a single variable can be represented as `Cons x Nil`. Then `p` $\gg\!=$ `f`

represents a substitution of variables `x` with lambda terms (`f x`) in `p`. Regular β-reduction is then modelled by `subs`, such that

$$\texttt{Comp}\,(\texttt{Cons}\,\texttt{x}\,\texttt{p})\,\texttt{p}' \overset{\beta}{\rightsquigarrow} \texttt{p}' \gg\!= \texttt{subs}\,\texttt{x}\,\texttt{p}$$

An interpretation satisfying the description above is given below.

Definition† 4.2.4. *A computation in the* `Pair` *monad can be interpreted as a lambda term with the following functions*

```
beta    :: Eq α ⇒ Pair α → Pair α
beta ps = case ps of
            Comp (Cons a p) p'  → if p == Nil then
                                    ps
                                  else
                                    p >>= subs a p'
            Comp (Comp p1 p2) p → Comp (beta $ Comp p1 p2) p
            _                   → ps

reduce    :: Eq α ⇒ Pair α → Pair α
reduce ps = case findFix beta ps of
              Nil        → Nil
              Cons a p   → Cons a (reduce p)
              Comp p1 p2 → Comp p1 (reduce p2)

findFix    :: Eq α ⇒ (α → α) → α → α
findFix f a = fixer a (f a)
  where fixer a1 a2 = if a1 == a2 then
                        a1
                      else
                        fixer a2 (f a2)
```

In this definition, `beta` reduces a lambda term one step, `reduce` reduces a lambda term to normal form, and `findFix` finds the fix point of a function over one particular element.

Going one step further, we could use elements with type `Pair (Pair` α`)` to represent lambda terms with pattern matching. With this, we can for instance represent `f (c x) = h x` as the pair

```
f = Cons (Comp (Cons c Nil)
               (Cons x Nil))
         (Comp (Cons (Cons h Nil) Nil)
               (Cons (Cons x Nil) Nil))
```

Every atomic value has to be wrapped twice in `Cons` to get the correct type, and all abstractions are now over pairs instead of variables. To apply such pattern matching functions, we could translate them in the same manner as any interpreter would have, namely back to regular lambda expressions. I have written such a translation, which can be found in the appendix on

page 121. The function is called `reducePatterns` and is a bit technical. We will not go through the details here, but the interested reader can look it up.

Applying `reducePatterns` to `f`, we would get the regular lambda function

```
reducePatterns f
 = Cons z
     (Comp (Cons f (Comp (Cons x p)
                         (Comp (Comp Nil Nil)
                               (Cons z Nil))))
           (Comp Nil (Cons z Nil)))
```

where

```
p = (Comp (Cons f Nil)
          (Cons x Nil))
```

We let `Nil` represents a meta function extracting $\mathtt{p}_1$ from $\mathtt{Comp}\,\mathtt{p}_1\,\mathtt{p}_2$, and $\mathtt{Comp}\,\mathtt{Nil}\,\mathtt{Nil}$ represents a meta function extrac ting $\mathtt{p}_2$. These functions are primitives, and need to be implemented in `beta`, so a small change in this function is required. Note that the reason `p` was placed at the right spot in the pair is due to the original definition of `appendPair`. If we were to use the alternate definition, `p` would be placed at the `Nil` after `z`.

We will see an application of pairs as lambda terms in the next chapter, where we create an interpreter for lambda calculus with graph sharing. There we will not use the functions defined above, but rather a more complex reduction strategy including an optimisation and logging of the interpretation process.

One should note that there already exist a `Tree` monad, which can simulate lists and similar data structures. It has the definition

```
data Tree α = Nil | Leaf α | Branch (Tree α) (Tree α)

instance Monad Tree where
  return              = Leaf
  Nil >>= f           = Nil
  (Leaf a) >>= f      = f a
  (Branch t1 t2) >>= f = Branch (t1 >>= f) (t2 >>= f)
```

However, this data type is not really an extension of lists, and cannot replace lists in the same way. To simulate a single level list one would have to use all constructors. Furthermore, there is no easy separation of a single element appended to a list and the joining of two trees to one. This would, for instance, make simulating lambda calculus difficult and far from natural. It also doesn't have the same interpretations as the `Pair` monad. So as we can see, the `Tree` monad and the `Pair` monad solves different problems.

In the next section we are going to see how we can define a monad transformer for pairs.

4.3 PairT$^{\dagger\lambda}$

Composition of monads is in many cases useful, and the `Pair` monad is no exception. In this section we will go through the definition of the monad transformer for `Pair`.

Definition† 4.3.1. *The definition of the transformer type,* `PairT`*, is the rather straight forward*

```
newtype PairT m α = PairT { runPairT :: m (Pair α) }
```

So an element with type `PairT` is just an element of type `m (Pair α)` wrapped in `PairT`, and unwrapped through the function `runPairT`.

Recall how the `sequence` function did much of the work in the definition of (≫=) in `ListT`. `PairT` is not that different from that of ListT, as its core is `sequenceP`, the `Pair` version of the list function `sequence`.

Definition† 4.3.2. *The sequencer-function for* `Pair` *is defined as*

```
sequenceP :: Monad m ⇒ Pair (m a) → m (Pair a)
sequenceP Nil             = return Nil
sequenceP (Cons ma mps)   = do
                              a ← ma
                              ps ← sequenceP mps
                              return $ Cons a ps
sequenceP (Comp mps mps′) = do
                              ps ← sequenceP mps
                              ps′ ← sequenceP mps′
                              return $ Comp ps ps′
```

The list version, `sequence`, took a list of computations, executed them in turn, and gathered the results of these computations in a list inside one large computation. Exactly the same is done in `sequenceP`, the only difference being that our data structure is pairs rather than lists.

We now have what we need to implement the `Monad` instance for the transformer.

Definition† 4.3.3. *The* `Pair` *monad transformer has the following definition,*

```
instance Monad m ⇒ Monad (PairT m) where
  return    = PairT . return . return
  mpa ≫= f = PairT $ do
                       pa ← runPairT mpa
                       ppa ← sequenceP $ liftM (runPairT . f) pa
                       return $ concatPair ppa
```

The `return` function is rather easy to grasp, first apply `Pair`'s `return` function, then apply `m`'s, and finally wrap the result in `PairT`.

The implementation of $(\gg\!=)$ is a bit more involved, and we will go through it step by step. In general, `PairT` is applied to a `do`-block in the monad `m`. The first thing done in the `do`-block is unwrapping `mpa` and binding the result of the unwrapped computation (of type $\texttt{m}\,(\texttt{Pair}\,\alpha)$) to `pa`. Hence $\texttt{pa} :: \texttt{Pair}\,\alpha$. Then the function $(\texttt{runPairT}\,.\,\texttt{f})$ is applied to every element inside `pa`. Recall that $\texttt{f} :: \alpha \to \texttt{PairT}\,\texttt{m}\,\beta$, so $(\texttt{runPairT}\,.\,\texttt{f}) :: \alpha \to \texttt{m}\,(\texttt{Pair}\,\beta)$. Hence, $\texttt{liftM}\,(\texttt{runPairT}\,.\,\texttt{f})\,\texttt{pa}$ results in a value of type $\texttt{Pair}\,(\texttt{m}\,(\texttt{Pair}\,\beta))$. To this value, `sequenceP` is applied, gathering all the results from the applications of $(\texttt{runPairT}\,.\,\texttt{f})$ in one large computation with type $\texttt{m}\,(\texttt{Pair}\,(\texttt{Pair}\,\beta))$. This value is now pierced, binding the result to $\texttt{ppa} :: \texttt{Pair}\,(\texttt{Pair}\,\beta)$. This value is concatenated down to a value of type $\texttt{Pair}\,\beta$ before being promoted to `m` by the `return` function.

Let's look at an example to build up some intuition.

Example 4.3.4. *We will look at a small computation in* `PairT [ ] Int`*. So let*

$$\texttt{mpa} = \texttt{PairT}\,[(\texttt{Cons}\,1\,\texttt{Nil}),\,(\texttt{Comp}\,(\texttt{Cons}\,2\,\texttt{Nil})\,(\texttt{Cons}\,4\,\texttt{Nil}))]$$

and

$$\begin{aligned}\texttt{f a} = \;&\textbf{if}\;\texttt{a} > 2\;\textbf{then}\\ &\quad\texttt{PairT}\,\$\,[(\texttt{Cons}\,\texttt{a}\,\texttt{Nil}),\,(\texttt{Cons}\,(\texttt{a}+1)\,\texttt{Nil})]\\ &\textbf{else}\\ &\quad\texttt{return a}\end{aligned}$$

We will go through $\texttt{mpa} \gg\!= \texttt{f}$ *step by step. Firstly, the* `PairT` *constructor is removed from* `mpa`*. Then* `pa` *is assigned* `Cons 1 Nil`*. Remember that the* `do`*-block is in the* `List` *monad, so* `pa` *will be assigned each element of the list in turn.*

After the assignment, we map $\texttt{runPairT}\,.\,\texttt{f}$ *over* `pa`*, resulting in* $\texttt{Cons}\,[(\texttt{Cons}\,1\,\texttt{Nil})]\,\texttt{Nil}$*, since* 1 *is less than* 2*. Now* `sequenceP` *is applied, which results in* $[\texttt{Cons}\,(\texttt{Cons}\,1\,\texttt{Nil})\,\texttt{Nil}]$*. Each result of this list is then assigned in turn to* `ppa`*, which currently only is* $\texttt{Cons}\,(\texttt{Cons}\,1\,\texttt{Nil})\,\texttt{Nil}$*. This pair is then concatenated down to* $\texttt{Cons}\,1\,\texttt{Nil}$*, before wrapped in a list, so the result is* $[\texttt{Cons}\,1\,\texttt{Nil}]$*.*

Now, we go back to the second line, and assign the next element in the first list to `pa`*, so now* $\texttt{pa} = \texttt{Comp}\,(\texttt{Cons}\,2\,\texttt{Nil})\,(\texttt{Cons}\,4\,\texttt{Nil})$*. The result of applying* $\texttt{runPairT}\,.\,\texttt{f}$ *to every element in* `pa` *is*

$$\begin{aligned}&\texttt{Comp}\,(\texttt{Cons}\,[\texttt{Cons}\,2\,\texttt{Nil}]\,\texttt{Nil})\\ &\qquad(\texttt{Cons}\,[(\texttt{Cons}\,4\,\texttt{Nil}),\,(\texttt{Cons}\,5\,\texttt{Nil})]\,\texttt{Nil})\end{aligned}$$

Then `sequenceP` *is applied to get*

$$\begin{aligned}&[\texttt{Comp}\,(\texttt{Cons}\,(\texttt{Cons}\,2\,\texttt{Nil})\,\texttt{Nil})\,(\texttt{Cons}\,(\texttt{Cons}\,4\,\texttt{Nil})\,\texttt{Nil}),\\ &\;\texttt{Comp}\,(\texttt{Cons}\,(\texttt{Cons}\,2\,\texttt{Nil})\,\texttt{Nil})\,(\texttt{Cons}\,(\texttt{Cons}\,5\,\texttt{Nil})\,\texttt{Nil})]\end{aligned}$$

Now, both of the two elements in that list is assigned to `ppa`*, concatenated, and wrapped in a list again, in turn. The two results of this are*

```
[Comp (Cons 2 Nil) (Cons 4 Nil)]

[Comp (Cons 2 Nil) (Cons 5 Nil)]
```

The three resulting lists are appended, and we get the following result,

```
PairT [Cons 1 Nil,
       Comp (Cons 2 Nil) (Cons 4 Nil),
       Comp (Cons 2 Nil) (Cons 5 Nil)]
```

and we are done.

We are almost done building the `PairT` transformer, but we still need to define how values are lifted into the `PairT` monad. All we really need to do is apply `Pair`'s `return` function to the value of the computation being lifted, and then wrap the result in `PairT`.

Definition† 4.3.5. `PairT`*'s* `lift` *function is defined as*

```
instance MonadTransformer PairT where
  lift = PairT . (liftM return)
```

Theorem† 4.3.6. `lift` *for* `PairT` *is a proper monad morphism, that is*

```
lift . return = return

lift (ma >>= f) = lift ma >>= lift . f
```

Proof. The first equality is trivial. The second is proven by a long sequence of rewritings, reducing both sides to

```
PairT $ concatPair (liftM ((liftM return) . f) ma
```

The left hand side is easily rewritten to the term above. The right hand side however, is more work. We will not go through all the steps, but by using the monad laws and the fact that `sequenceP . return = liftM return`, the proof is rather straight forward. □

Now we have implemented everything we need to combine the functionality of the `Pair` monad with other monads. However, there is still one thing one should think of every time one implements an instance of the monad class: Does our implementation satisfy the monad laws? That is, is `PairT m` really a proper monad for *any* monad `m`? As it turns out, it is not. This should come as no surprise, since I have already mentioned that `ListT m` is only a proper monad for commutative `m`, and we know that `Pair` is just a generalisation of `List`.

Theorem† 4.3.7. `PairT m` *is not a monad for all monad* `m`.

Proof. It is the third law that is not satisfied. For example, if we define

```
ma :: PairT (State Int) Int
ma = PairT $ return (Cons 1 (Cons 2 Nil))

update  :: Int → PairT (State Int) Int
update a = do
             s ← lift get
             lift $ put (s + a)
             return s

runBoth   :: PairT (State Int) Int → (Pair Int, Int)
runBoth ma = runState (runPairT ma) 0
```

we have that

```
runBoth $ ma >>= update >>= update
 = ((Cons 3 (Cons 3 Nil)), 4)

runBoth $ ma >>= λ a → (update a >>= update)
 = ((Cons 1 (Cons 3 Nil)), 4)
```

which are not equal. □

Theorem 4.3.8. `PairT m` *is a proper monad for commutative* `m`*.*

Proof. The proof can be obtained by generalising the proof of Mark P. Jones and Luc Duponcheel in "Composing Monads"[16] of the same property of `ListT`. They prove that for `ListT m` to be a proper monad, `sequence` needs to satisfy four laws. In their article `sequence` is named `swap`. They continue to prove these laws for commutative `m` with induction. We only need to extend their inductions proofs with the `Comp`-constructor, so a generalised proof must prove these properties for `sequenceP`.

We will not go through the proof here, as the proof is quite straight forward using the methods and results described in the article, but demands a great deal of rewriting and technical manipulation of large expressions.

Throughout the article they make heavy usage of comprehensions which is closely related to `do`-notation. An explanation of such syntax can be seen in Chapter 7. □

This concludes our investigation into the `Pair` monad, and we will now look at applications of this and many other monads.

Part II

Application

Chapter 5

Extended Example: Graph Reduction$^{\lambda}$

Haskell is a purely functional programming language, and as we discussed briefly in the introduction, one feature of functional languages is referential transparency. (That is, the resulting value of a given expression can replace the same expression anywhere in the code.) This feature lets us optimise our code through what is known as graph reduction. In this chapter we will, after an introduction to graph reduction, implement an interpreter for lambda calculus with graph reduction, using monads. All lines of code are my own, but the idea and concepts are gathered from Simon Peyton Jones' article on implementing functional languages[24].

5.1 The problem

In a normal β-reduction one reduces applications by substituting a term with a variable in the applied function's body. However, there are two main reduction strategies, innermost and outermost reduction. Outermost substitutes unreduced arguments, while innermost reduces its arguments before substitution. For instance, say $(\mathtt{f}\ 5) = 3$, then the outermost reduction of $(\lambda\,\mathtt{x}\,.\,(*\,\mathtt{x}\,\mathtt{x}))\,(\mathtt{f}\ 5)$ is

$$\begin{aligned}(\lambda\,\mathtt{x}\,.\,(*\,\mathtt{x}\,\mathtt{x}))\,(\mathtt{f}\ 5) &\overset{\beta}{\rightsquigarrow} (*\,(\mathtt{f}\ 5)\,(\mathtt{f}\ 5)))\\ &\overset{\beta}{\rightsquigarrow} (*\,3\,(\mathtt{f}\ 5))\\ &\overset{\beta}{\rightsquigarrow} (*\,3\,3)\\ &\overset{\beta}{\rightsquigarrow} 9\end{aligned}$$

while the innermost is

$$\begin{aligned}(\lambda\,\mathtt{x}\,.\,(*\,\mathtt{x}\,\mathtt{x}))\,(\mathtt{f}\ 5) &\overset{\beta}{\rightsquigarrow} (\lambda\,\mathtt{x}\,.\,(*\,\mathtt{x}\,\mathtt{x}))\,3\\ &\overset{\beta}{\rightsquigarrow} (*\,3\,3)\\ &\overset{\beta}{\rightsquigarrow} 9\end{aligned}$$

As we can see, the innermost has, in this case, the fewest reductions, since it reduces $(\mathtt{f}\ 5)$ before it sends it to a function which uses the expression

two different places. This is not always the case, take for instance the term `fst` (`f` 5, `f`5) where the outermost is

$$\begin{aligned}\texttt{fst}\,(\texttt{f}\ 5,\ \texttt{f}\ 5) &\overset{\beta}{\rightsquigarrow} (\texttt{f}\ 5)\\ &\overset{\beta}{\rightsquigarrow} 3\end{aligned}$$

The innermost is the longer

$$\begin{aligned}\texttt{fst}\,(\texttt{f}\ 5,\ \texttt{f}\ 5) &\overset{\beta}{\rightsquigarrow} \texttt{fst}\,(\texttt{f}\ 5,\ 3)\\ &\overset{\beta}{\rightsquigarrow} \texttt{fst}\,(3,\ 3)\\ &\overset{\beta}{\rightsquigarrow} 3\end{aligned}$$

Haskell uses outermost reduction to enable lazy evaluation[24] such that terms are only reduced when the result is needed (enabling more expressiveness, like streams etc.). But as we saw with outermost reduction, the term (`f` 5) might be calculated several times, although we really only have to reduce it once due to referential transparency. This is where graph reduction comes into play. The idea is to substitute pointers to the argument rather than the argument itself. So

$$(\lambda\,\texttt{x}\,.\,(*\,\texttt{x}\,\texttt{x}))\,(\texttt{f}\ 5) \overset{\beta}{\rightsquigarrow} (*\,\square_{(\texttt{f}\ 5)}\,\square_{(\texttt{f}\ 5)})$$

where $\square_{(\texttt{f}\ 5)}$ is a pointer to a cell with the expression (`f` 5) stored. If we need the result (or value) of $\square_{(\texttt{f}\ 5)}$, we reduce the expression which it points to, updates the cell with the reduced value, and substitutes in that value. The next time we need the result of the same expression, we can just retrieve it from the cell pointed to. A term stored in a cell can also contain pointers, so a term can be represented by a graph. Now every argument is reduced at most once while still having outermost reduction. This is called *sharing* and the graph that results from this sharing will be called the *Shared Term Graph* (STG). The process of reducing such a graph is called *graph reduction*.

In the next section we will see how we will represent lambda terms, pointers, and other functionality needed for an implementation of an interpreter of lambda calculus with graph reduction and sharing.

5.2 Building blocks and representation$^{\lambda}$

We start by presenting a convenient way of representing the lambda terms, namely through the `Pair` monad defined in the previous chapter. So the term in the previous section has the following representation

```
Comp (Cons "x"
           (Comp (Comp (Cons " * " Nil)
                       (Cons "x" Nil))
                 (Cons "x" Nil)))
     (Comp (Cons "f" Nil)
           (Cons "5" Nil))
```

As we saw above, we must have a way of representing pointers. However, there are no pointers in Haskell, so we will use a `Map` from `Pair`s to `Term`s to represent the edges of the STG. For this, we define the following type,

```
import qualified Data.Map as Map

type MyMap α = Map.Map (Pair α) (Term α)
```

where

```
data Term α = Red (Pair α) | UnRed (Pair α)
```

Our pointers are going from pairs to terms, which either consists of reduced or unreduced pairs. The `Map` data structure demands that its keys are orderable, that is, implementing the `Ord` type class. In the code in the appendix, on page 119, there is a proposal for ordering pairs, demanding that the values inside the pair are orderable. We could also rather just assume `Show` α and use ordering on strings, but this would lead to a lot of unnecessary comparison of characters, since the constructors would be a part of the strings.

The map is going to keep track of all the pointers in the graph, so most of the interpreter's functions will need access to it. We will therefore use the `State` monad with the map as the state value. We are going to implement an interpreter, and as we know, logging of the interpretation is quite useful. For such logging, we are going to use the `Writer` monad, which is a monad for sharing values that are relevant for later computations. This monad is a special case of the `State` monad, with transformer type

```
newtype WriterT μ m α = WriterT { runWriterT :: m (α, μ) }
```

where the monadic instance for the transformer is defined as

```
instance (Monoid μ, Monad m) ⇒ Monad (WriterT μ m) where
  return a = WriterT $ return (a, mempty)
  m >>= k = WriterT $ do
                        (a, w) ← runWriter m
                        (b, w′) ← runWriter (k a)
                        return $ (b, w `mappend` w′)
```

All it does is update and pass the `w`-value alongside the computations. Notice that while the `State` monad is a function over states, the `Writer` monad is not. Here we always start a computation with `mempty`.

The `WriterT` monad implements the `MonadWriter` class, which defines the function

```
tell :: MonadWriter μ m ⇒ m ()
```

`WriterT` implements this with

```
tell w = WriterT $ return ((), w)
```

The monad we are going to work with is therefore the following,

```
type STG α = WriterT String (State (MyMap α))
```

such that logging is taken care of by the `Writer` monad. For inserting a pointer to a pair with key `k` and value `a`, we define

```
insert   :: Ord α ⇒ Pair α → Term α → STG α ()
insert k a = lift $ modify (Map.insert k a)
```

We will also use the following lifted `Map` functions, and their implementation can be seen in the appendix on page 123.

```
member   :: Ord α ⇒ Pair α → STG Bool
retrieve :: Ord α ⇒ Pair α → STG α (Term α)
delete   :: Ord α ⇒ Pair α → STG α ()
```

Here `member` checks if the argument is a member of the `Map`, `retrieve` extracts a value, and `delete` deletes a value from the `Map`. We now have the building blocks needed for the implementation of our sharing lambda calculus.

5.3 Implementation of a small interpreter$^{\lambda}$

We implemented a small interpreter with pairs representing lambda terms in the previous chapter. Now we will add much more functionality, so we are not going to use that implementation here, but rather define a new, similar, but more complex, interpreter.

We will implement the interpreter top down with three main functions controlling the reduction process. One function will control the reduction on the top level in the same way `reduce` did in our simple implementation in previous chapter. So this function is ensuring that the resulting term is on normal form. Let's call the this function `reduceSTG`.

A second function will do a single, topmost β-reduction, and will call itself recursively in case of reductions of sub-terms are necessary to perform a single topmost reduction. This is necessary whenever we have a term on the form ((`f e`) `e'`), where we have to reduce (`f e`) to see which variable `e'` should be substituted for. In the above expression, the reduction of applying (`f e`) to `e'` is topmost, so (`f e`) should be reduced first. This function is named `reduceTop`.

The last main function, `evaluator`, is handling the pointers in the map, and reduces the terms when necessary using `reduceSTG`.

Let's start by defining the first of these functions.

```
reduceSTG  :: (Ord α, Show α) ⇒ Pair α → STG α (Pair α)
reduceSTG p = do
```

```
b ← member p
p′ ← if b then evaluator p else untilFix reduceTop p
case p′ of
  Nil        → return Nil
  Cons a ps  → do
                 ps′ ← reduceSTG ps
                 return $ Cons a ps′
  Comp p1 p2 → do
                 p′1 ← reduceSTG p1
                 p′2 ← reduceSTG p2
                 return $ Comp p′1 p′2
```

To reduce a term, we first check whether the term is shared earlier, and if so we apply `evaluator` to `p` to obtain the shared term. The term returned from `evaluator` should already be reduced. We will also log when reducing and using reduced terms. Below is the definition of this function.

```
evaluator :: (Ord α, Show α) ⇒ Pair α → STG α (Pair α)
evaluator p = do
  trm ← retrieve p
  case trm of
    Red p′    → do
                  tell $ "Used reduced value < "
                         ++ (show p′) ++ " > \n"
                  return p′
    UnRed p′ → do
                  delete p
                  p″ ← reduceSTG p′
                  insert p (Red p″)
                  tell $ "Reduced < " ++ (show p′)
                         ++ " > to < " ++ (show p″)
                         ++ " > .\n"
                  return p″
```

So `evaluator` just retrieves the term from the map, returns it if it already is reduced, or reduces it if not. Note that if `p` is unreduced, we have to delete `p` from the map before reducing it to prevent going in an infinite loop.

If it's not shared when given to `reduceSTG`, we need to iterate `reduceTop` over `p`, such that it is fully reduced on top level. This is done by finding the fix point of `reduceTop` over `p`, which the following function handles.

```
untilFix    :: (Monad m, Ord α) ⇒ (α → m α) → α → m α
untilFix f a = fixer a (f a)
  where fixer b mb = do
                       b′ ← mb
                       if b == b′ then
                         return b
                       else
                         fixer b′ (f b′)
```

Finally, we have the function doing the actual reduction.

```
reduceTop    :: (Ord α, Show α) ⇒ Pair α → STG α (Pair α)
reduceTop ps = case ps of
  Comp (Cons a p) p' → if p == Nil then
                         return ps
                       else
                         do
                           share p'
                           return $ p >>= subs a p'
  Comp (Comp p1 p2) p → do
                          p' ← reduceTop (Comp p1 p2)
                          return $ Comp p' p
  _                   → return ps
```

Notice that we don't reduce terms on the form `Comp (Cons f Nil) p`, since they correspond to `(f p)` where `f` is atomic. Also observe the laziness of our reductions, we only reduce a term if it is strictly necessary. So we never reduce an argument, and only reduce an applied function such that we can see which variable the function is abstracted over. However, `reduceSTG` does make sure that the term is on normal form on termination by reducing the sub-terms when we cannot reduce the top level of the term any further.

We still haven't defined `share`, which just stores the value in the map, if not already there. We log the values entering the map for convenience.

```
share   :: (Ord α, Show α) ⇒ Pair α → STG α ()
share p = do
            b ← member p
            if b then
              tell $ "Used shared value < " ++ (show p) ++ " > .\n"
            else
              do
                insert p (UnRed p)
                tell $ "Shared < " ++ (show p) ++ " > .\n"
```

Now we have everything we need to reduce a lambda term to normal form using shared term graphs. Our result is in the `STG α` monad, so to unwrap the reduced term outside the monad, we could define

```
unwrapSTG     :: (Ord α, Show α) ⇒ STG α (Pair α) → (Pair α, String)
unwrapSTG stg = let
                  wtr       = runWriterT stg
                  ((p, s), _) = runState wtr Map.empty
                in
                (p, s)
```

Furthermore, we could define

```
reduce :: (Ord α, Show α) ⇒ Pair α → (Pair α, String)
reduce = unwrapSTG . reduceSTG
```

such that a user never have to know of, or use, the `STG` monad at all. Let's test our implementation and apply it to the term

$((\lambda\,\mathtt{f}\ \rightarrow\ (\mathtt{f}\,(\mathtt{f}\,\mathtt{x})))\,((\lambda\,\mathtt{g}\ \rightarrow\ \mathtt{g})\,\mathtt{h}))$, which should reduce to $(\mathtt{h}\,(\mathtt{h}\,\mathtt{x}))$. So if we set

```
p = Comp (Cons 'f'
                 (Comp (Cons 'f' Nil)
                       (Comp (Cons 'f' Nil)
                             (Cons 'x' Nil))))
           (Comp (Cons 'g' (Cons 'g' Nil))
                 (Cons 'h' Nil))

(p', s) = reduce p
```

then

```
p' = Comp (Cons 'h' Nil)
          (Comp (Cons 'h' Nil)
                (Cons 'x' Nil)))

s = "Shared < Comp (Cons 'g' (Cons 'g' Nil)) (Cons 'h' Nil) > .
    Shared < Cons 'h' Nil > .
    Reduced < Comp (Cons 'g' (Cons 'g' Nil)) (Cons 'h' Nil) >
            to < Cons 'h' Nil > .
    Used reduced value < Cons 'h' Nil > .
    Used shared value < Cons 'h' Nil > .
    Used reduced value < Cons 'h' Nil > ."
```

Notice that even though $((\lambda\,\mathtt{g}\ \rightarrow\ \mathtt{g})\,\mathtt{h})$ is used twice in the expression, it is only reduced once, thanks to our sharing.

In this chapter we saw usage of the `Writer` and `State` monad in conjunction, and the `Pair` monad on its own. In the next chapter we will introduce more monads, and combine them in a more complex and realistic manner.

Chapter 6

Extended Example: Cryptographic Algorithms$^{\lambda}$

In this chapter we will show how monads can be combined to create an environment for writing cryptographic algorithms. We will start by combining the needed functionality associated with such programming. Functions and types will be introduced one at the time and explained. For the entire code, see the appendix starting on page 125. The ideas and algorithms concerning cryptography are gathered from Stinson's "Cryptography Theory and Practice"[27], but the implementations are my own.

After the base environment is defined, we will extend it with more functionality. The point of this is to show how easy one can extend monadic code.

6.1 Combining the needed functionality$^{\lambda}$

Perhaps the most crucial functionality needed in the implementation of cryptographic algorithms are access to random numbers. Cryptographic algorithms mix cryptographic keys with plain text in a reversible way, to hide its meaning. The cryptographic keys are mostly randomly generated numbers, sometimes with specific properties.

In Haskell there is a module called `System.Random` that enables us to retrieve random values. In this module there is a function `getStdGen :: IO StdGen` which returns a random number generator in the `IO` monad. Why is it in the `IO` monad, one might ask. When generating random numbers, one often use the previously generated value and runs it through a complicated algorithm, similar to a hash function. In order to start generating numbers however, we need a pseudo random number to start with. This number is gathered from outside the program, such that the program doesn't use the same values for every run. Since it communicates with the environment outside the program, the result should be in the `IO` monad, that is, it is dependent on the state of the World.

We can start using a generator retrieved from a call to `getStdGen` to generate random numbers by calling `next :: StdGen → (Int, StdGen)`.

Now we have a way of getting random values, but how should we enable access to the generator from functions needing it? We could define all functions to take an extra argument, namely the generator, and then explicitly pass it along with every function call. This is a solution, albeit cumbersome and untidy.

We will not settle for cumbersome and untidy, but instead seek a more elegant solution. There is in fact a predefined monad that perfectly suits our purpose, namely the `Random` monad from the module `Control.Monad.Random`. The base monad has type `Rand` $\gamma\,\alpha$, where γ is the random generator. This monad passes the generator along, much in the same way the state value was passed along in the `State` monad. Inside the monad we have access to, among others, the function `getRandom` :: (`Random` α, `MonadRandom` `m`) $\Rightarrow$ `m` α, since `Rand` implements `MonadRandom`. We can execute a computation in the `Random` monad with either of

```
evalRand :: RandomGen γ ⇒ Rand γ α → γ → α
```

or

```
runRand :: RandomGen γ ⇒ Rand γ α → γ → (α, γ)
```

depending on whether we want to keep the generator or not.

We will define a type for our encryption which we will extend as we define more and more functionality. For now, we have

```
type Encrypted α = Rand StdGen α
```

Example 6.1.1. *A simple implementation of One Time Pad encryption:*

```
enc :: Int → Encrypted Int
enc p = do
          k' ← getRandom
          let k = mod k' 2
          return $ xor p k

encryptAll :: [Int] → [Int] → Encrypted [Int]
encryptAll [ ] cs = return cs
encryptAll (p : ps) cs = do
                           c ← enc p
                           encryptAll ps (c : cs)

oneTimePad :: [Int] → IO [Int]
oneTimePad ps = do
                  let randcs = encryptAll ps [ ]
                  evalRand randcs getStdGen
```

Encryption isn't very interesting if we cannot reverse what we have done. Using only the Random monad we can encrypt, but since we don't keep the keys, we cannot decrypt the cipher text. One solution could be to make `encryptAll` take another list as argument to keep the keys. However, again there is a monad that suits our needs, the `Writer` monad seen in the previous chapter. We could now extend the monad we work in, as

$$\textbf{type}\ \texttt{Encrypted}\ \kappa\ \alpha\ =\ \texttt{RandT}\ \texttt{StdGen}\ (\texttt{Writer}\ [\kappa])\ \alpha$$

where κ is the type of our keys. To share our keys in the above example, all we have to do is insert (`lift $ tell [k]`) in `enc`, a call to `runWriter` in `oneTimePad`, and use `evalRandT` instead of `evalRand`. The rest of the code can go untouched.

Decryption is in many cases equivalent to encryption, for instance with the one time pad. There, decryption is done by xor'ing the keys with the cipher text. How would we do this in the example above? Obviously we cannot use the `enc` function. To be able to use the same logic for both encryption and decryption, we will move the key management outside the `Encryption` monad. We will then introduce a new monad, which will supply the computations with keys fed from the outside of the monad. This new monad, called conveniently `Supply`, keeps a (possibly infinite) list of values. We will use the transformer of this monad, and its definition is

$$\textbf{newtype}\ \texttt{SupplyT}\ \kappa\ \texttt{m}\ \alpha\ =\ \texttt{SupplyT}\ (\texttt{StateT}\ [\kappa]\ \texttt{m}\ \alpha)$$

As we can see, `SupplyT` is really just a new name for `StateT`, but with some new functionality. The supplied values are given to the computation when the computation is run through

$$\texttt{evalSupplyT}\ ::\ \texttt{Monad}\ \texttt{m}\ \Rightarrow\ \texttt{SupplyT}\ \kappa\ \texttt{m}\ \alpha\ \rightarrow\ [\kappa]\ \rightarrow\ \texttt{m}\ \alpha$$

`SupplyT` implements the `MonadSupply` class, which contains the following useful function

$$\texttt{supplies}\ ::\ \texttt{MonadSupply}\ \sigma\ \texttt{m}\ \Rightarrow\ \texttt{Int}\ \rightarrow\ \texttt{m}\ [\sigma]$$

such that `supplies n` returns a list of elements from the stream with length `n`. Substituting `RandT` with this monad, our new monad now looks like

$$\textbf{type}\ \texttt{Encrypted}\ \kappa\ \alpha\ =\ \texttt{SupplyT}\ \kappa\ (\texttt{Writer}\ [\kappa])\ \alpha$$

All we now need is a list of random values to feed our computations. This can be done with the following function.

```
generateSupply :: (Random κ, RandomGen γ) ⇒ Rand γ [κ]
generateSupply = liftM2 (:) getRandom generateSupply
```

All this code does is generate an infinite stream of random numbers, inside the `Rand` monad. If we would like an infinite stream of random bits as a list, this could be achieved by

```
bitSupply :: IO [Int]
bitSupply = do
              gen ← getStdGen
              let s' = generateSupply
                  s = evalRand s' gen
              return $ map (λ x → mod x 2) s
```

Normally in cryptography one does not generate all the key material randomly, since the keys must be transmitted in some way to the receiver of the cipher text. Usually one generates randomly one key of a fixed number of bits and then use that key to generate new keys[27]. This makes the `Supply` monad handy, since we can generate one key, and make a supply with derived keys.

Now we have everything we need to implement symmetric cryptographic algorithms.

Example 6.1.2. *Let's see how the One Time Pad is implemented using our new monad and functions. Encryption and decryption can both be done by the function*

```
enc :: Int → Encrypted Int Int
enc p = do
          (k : _) ← supplies 1
          lift $ tell [k]
          return $ xor p k


oneTimePad :: [Int] → [Int] → ([Int], [Int])
oneTimePad ps ks = let
                     sup = encryptAll ps
                     wrt = runSupplyT sup ks
                   in
                   runWriter wrtComp
```

`encryptAll :: [Int] → [Int] → Encrypted Int Int` *is equivalent, but with an updated type signature. In the code below, we first generate a supply of random bits* `randBits`*, encrypt a list of bits* `bs`*, and then decrypts them again.*

```
encDec :: [Int] → IO ()
encDec bs = do
              randBits ← bitSupply
              let (cs, ks) = oneTimePad ps randBits
              print cs
              let (ps, _) = oneTimePad cs ks
              print ps
```

6.2 Making a framework$^{\lambda}$

In this section we will make a general framework for cryptographic algorithms, such that a user only have to implement the key functionality

of an algorithm: How keys are mixed with the plain text to form the cipher text. For instance, the only thing a user should need to implement in the examples above, is the `enc` function. So we need to define general versions of the other functions. Let's start with the `encryptAll` function, with type

```
encryptAll :: (α → Encrypted κ α) → [α] → [α] → Encrypted κ [α]
```

where the first argument is the `enc` function. This could be implemented as

```
encryptAll enc [ ] cs = return cs
encryptAll enc (p : ps) cs = do
                               c ← enc p
                               encryptAll enc ps (c : cs)
```

What we are doing here is actually applying a function uniformly over a list, so we could instead map `enc` over `ps`. But (`map enc ps`) :: [`Encrypted` $\kappa\,\alpha$], and we want something of type `Encrypted` $\kappa\,[\alpha]$. The trick is to use the function

```
sequence :: Monad m ⇒ [m α] → m [α]
```

from the `List` monad, which we discussed earlier. We then get the neat implementation below.

```
encryptAll :: (α → Encrypted κ α) → [α] → Encrypted κ [α]
encryptAll enc ps = sequence $ map enc ps
```

Now `enc` is mapped over the elements of `ps` resulting in a list of computations. When `sequence` is applied, it creates one big computation returning a list of elements. These elements are the result of each computation in the first list. It is important to understand that after `sequence` is applied, they are all part of the same computation. That means that they update the same writer variable and use the same supply.

All we need now is a method for gathering the result of the resulting computation.

```
evalEncrypted :: Encrypted κ [α] → [κ] → ([α], [κ])
evalEncrypted encr ks = let
                          res = evalSupplyT encr ks
                        in
                        runWriter res
```

From these function we can define a general encryption method.

```
encrypt :: (α → Encrypted κ α) → [α] → [κ] → ([α], [κ])
encrypt enc ps ks = evalEncrypted (encryptAll enc ps) ks
```

In the example in the section before, to share one key `k` we had to write `lift $ tell [k]`. To simplify this we wrap such code in small functions.

```
shareKey  :: κ → Encrypted κ ()
shareKey k = lift $ tell [k]

shareKeys  :: [κ] → Encrypted κ ()
shareKeys ks = lift $ tell ks
```

We will finish this section with a couple of examples of our framework. In the next section we will see how easily one can extend this monadic code.

Example 6.2.1. *The implementation of One Time Pad is simply*

```
enc :: Int → Encrypted Int Int
enc p = do
          (k : _) ← supplies 1
          shareKey k
          return $ xor p k


oneTimePad :: [Int] → [Int] → ([Int], [Int])
oneTimePad = encrypt enc
```

Usage is as before, for instance as used in `encDec`.

Example 6.2.2. *Let's look at a more realistic example in terms of complexity. A popular family of cryptographic algorithms are the SPNs (Substitution Permutation Networks), which is a type of block cipher. Here encryption is done by a series of* rounds *defined by a round function. One round in such an encryption scheme consists of mixing in keys, a substitution, and a permutation. Typically, the number of rounds are between 12 to 16, depending on the algorithm and desired security. In the last round one drops the permutation. This is done such that one can use the same logic to decrypt the cipher text, just reverse the order of the applications of the keys. Chapter 3 in [27] handles SPNs in detail.*

Since SPNs are block ciphers we will work on lists of bits, so `enc :: [Int] → Encrypted Int [Int]`. *In our example we will assume implementations of the permutations,* `perm :: [Int] → [Int]`, *and the substitutions,* `subs :: [Int] → [Int]`. *We will also assume that key mixing is done by xor. We can then implement the round function as*

```
rndFunc :: [Int] → Int → Encrypted Int [Int]
rndFunc p n = do
                k ← supplies (length p)
                shareKeys k
                let c' = zipWith xor p k
                    c  = subs c'
                if n > 1 then
                  rndFunc (perm c) (n − 1)
                else
                  return c

enc :: [Int] → Encrypted Int [Int]
enc p = round p 16
```

if we want 16 rounds. We can now both decrypt and encrypt using `enc`*, but we need to reverse the keys received from encryption before using them to decrypt.*

```
spn :: [Int] → [Int] → ([Int], [Int])
spn = encrypt enc
```

6.3 Introducing state$^{\lambda}$

In many cases in an algorithm one is interested in the previous encrypted text. For instance in SPNs CBC mode (Cipher Block Chaining mode) where one xors the previously encrypted block with the current plain text before it is encrypted. So if $\mathtt{c_i}$ is the encrypted $\mathtt{p_i}$, then $\mathtt{c_{i+1}} = \mathtt{enc}\ (\mathtt{xor}\ \mathtt{p_{i+1}}\ \mathtt{c_i})$. The first plain text, $\mathtt{p_0}$, is xored with a default value, often called `IV`. However, currently we don't have access to $\mathtt{c_i}$ when applying `enc` to $\mathtt{p_{i+1}}$. In this section we will extend our framework to enable such encryption. For this we will use the `State` monad to store previously encrypted blocks. Our new and extended monad now has the type

```
type Encrypted κ α = StateT [κ] (SupplyT κ (Writer [κ])) α
```

All we have to change in the code of the framework is adding another `lift` in the `shareKey` and `shareKeys` functions, and inserting one more line in the `let`-block in `evalEncrypted`, as such

```
evalEncrypted :: Encrypted κ [α] → [κ] → ([α], [κ])
evalEncrypted encr ks = let
                          res' = evalStateT encr [IV]
                          res  = evalSupplyT res' ks
                        in
                        runWriter res
```

The rest can go unchanged, including the implementations of One Time Pad and the SPN.

The purpose of the state was to store values needed for later encryption. For manipulating this state, we will define some helpful functions.

```
push  :: α → Encrypted κ ()
push a = modify (a :)


peek :: Encrypted κ κ
peek = do
         s ← get
         return $ head s


pop :: Encrypted κ κ
pop = do
        k ← peek
        modify tail
        return k
```

We can now implement an SPN with CBC mode.

```
enc :: [Int] → Encrypted Int [Int]
enc p = do
          c′ ← pop
          c ← rndFunc (xor p c′) 14
          push c
          return c
```

where `rndFunc` is as before.

This is how easily monadic code can be extended with new features: Extend the type and do some small changes in the base code. Amazingly, our code is still backwards compatible with already implemented applications, like `oneTimePad` and `spn`. This is a feature that any programmer would appreciate, and makes monadic code rather easy to maintain and extend.

Chapter 7

Building Blocks for Query Languages†

In this chapter we will investigate how monads and monad transformers can be used to reason about and construct queries for relational databases. We will generalise the work of Grust's query language with monad comprehensions based on a catamorphic language. We will base our work on his article "Monad Comprehension: A Versatile Representation for Queries"[12]. Furthermore will we combine this result with the work of Hinze's Prological backtracking monad from "Prological Features in a Functional Setting: Axioms and Implementations"[13].

7.1 Catamorphisms

In this section we will investigate the concept of a catamorphism, which will be used in the next sections as a basis for a monadic, comprehension based query language. Catamorphisms are homomorphisms over initial algebras, that is, a property preserving translation from one inductively generated algebra to another. For instance, the catamorphism from the initial algebra for lists, written ([ ], (:)), to the initial algebra of XOR, (`false`, ($\oplus$)), is the unique homomorphism `h`, such that

$$\begin{aligned} \texttt{h}\,[\,] &= \texttt{false} \\ \texttt{h}\,(\texttt{x} : \texttt{xs}) &= \oplus\ \texttt{x}\,(\texttt{h}\,\texttt{xs}) \end{aligned}$$

In Haskell there is a special function, `foldr`[1], for generating homomorphisms from lists to other initial algebras,

$$\begin{aligned} &\texttt{foldr} && :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ &\texttt{foldr} \otimes \texttt{z}\,[\,] && = \texttt{z} \\ &\texttt{foldr} \otimes \texttt{z}\,(\texttt{x} : \texttt{xs}) && = \otimes\ \texttt{x}\,(\texttt{foldr} \otimes \texttt{z}\,\texttt{xs}) \end{aligned}$$

So for any $(\otimes) :: \alpha \to \beta \to \beta$ and $\texttt{z} :: \beta$ having the same properties as (:) and [ ] respectively, `foldr` ($\otimes$) `z` is a homomorphism. There is, however, also a generalisation of this function in the libraries, `Data.Foldable`, with type

[1] The 'r' at the states which way the fold is executed, from left to right. There is also a `foldl` variant.

```
foldr :: Foldable t ⇒ (α → β → β) → β → t α → β
```

which generalises application to all foldable structures that implement the type class `Data.Foldable`. This is the function we will use to generate catamorphisms. Catamorphisms are highly expressive and can express several key features of functional programming. A small list of such are given below.

```
maximum :: Foldable t ⇒ t Int → Int
maximum = foldr max 0

or      :: Foldable t ⇒ t Bool → Bool
or      = foldr (||) false

(++)    :: [α] → [α] → [α]
(++) a b = foldr (:) a b

sum     :: Foldable t ⇒ t Int → Int
sum     = foldr (+) 0

toList  :: Foldable t ⇒ t α → [α]
toList  = foldr (:) [ ]
```

We can define general data structure mappings with

```
gmap      :: Foldable t ⇒
             γ → (β → γ → γ) → (α → β) → t α → γ
gmap z c f = foldr (λ x xs → c (f x) xs) z
```

We can also use `foldr` to define a fix point generator with

```
fix  :: (α → α) → α
fix f = foldr (λ _ → f) undefined (repeat undefined)
```

showing how `foldr` can express general iteration. Catamorphisms are in fact expressive enough to construct almost all queries in common query languages for databases[12].

One should note that the `foldr` function is not capable of expressing all catamorphisms, and is therefore somewhat weaker in expressiveness. For example, catamorphisms can compute the height of a tree, while `foldr` cannot[32]. This is due to the fact that `foldr` only iterates a structure from left to right (`foldl` from right to left), while there is no such restriction in the definition of a catamorphism. For our purpose however, the power of `foldr` will do, and a generalisation of our work here to all catamorphisms could be a topic for future work.

Now we will look at some nice features of catamorphisms, making them well fit as a foundation for query languages. The first property is an optimisation called deforestation[11]. We will in the following theorem let (`cata z c`) be a catamorphism where the resulting data structure has

constructor `c` and zero element `z`. For instance the general mapping `gmap z c f` has its constructors factored out, so if we rather fix one function `f`, and write

$$\texttt{mapf z c} = \texttt{gmap z c f}$$

we can easily alter the resulting data structure while preserving the element-wise mapping `f`. With this in mind, we are ready to present the following neat optimisation.

Theorem 7.1.1. *For any catamorphism* $(\texttt{foldr}\,(\otimes)\,\texttt{n})$ *and any catamorphism* $(\texttt{cata z c})$ *we have*

$$(\texttt{foldr}\,(\otimes)\,\texttt{n}) \,.\, (\texttt{cata z c}) = \texttt{cata n} \otimes$$

Proof. By induction on the structure of the argument. Let the argument structure have zero element $\texttt{z}'$ and constructor $\texttt{c}'$. For the base case we have

$$\begin{aligned}(\texttt{foldr} \otimes \texttt{n}) \,.\, (\texttt{cata z c})\,\$\,\texttt{z}' &= \texttt{foldr} \otimes \texttt{n z} \\ &= \texttt{n} \\ &= \texttt{cata n} \otimes \texttt{z}'\end{aligned}$$

The induction step is easily shown with

$$\begin{aligned}(\texttt{foldr} \otimes \texttt{n}) \,.\, (\texttt{cata z c})\,\$\,\texttt{c}'\,\texttt{x xs} &= \texttt{foldr} \otimes \texttt{n}\,\$\,\texttt{c}\,\texttt{x}'\,(\texttt{cata z c})\,\texttt{xs} \\ &= \otimes\ \texttt{x}'\,((\texttt{foldr} \otimes \texttt{n}) \,.\, (\texttt{cata z c})\,\$\,\texttt{xs}) \\ &= \otimes\ \texttt{x}'\,(\texttt{cata n} \otimes \texttt{xs}) \\ &= \texttt{cata n} \otimes (\texttt{c}'\,\texttt{x xs})\end{aligned}$$

where $\texttt{x}'$ is the result of $(\texttt{cata z c})$'s effect on `x`. □

What this theorem really express is that we can omit intermediary data structures. This is indeed a neat optimisation, since the left-hand side of the equation in the theorem iterates the data structure twice, while the right-hand only does this once. Let's look at an example from Gill, Launchbury, and Jones' article[11] of such an optimisation.

Example 7.1.2. *Assume we have the two catamorphisms*

```
and :: Foldable t ⇒ t Bool → Bool
and = foldr (&&) true

mapPred :: (α → Bool) → [α] → [Bool]
mapPred p = map p
```

We can now make finite universal quantification over lists with the function

```
forall  :: (α → Bool) → [α] → Bool
forall p = and . (mapPred p)
```

determining if all elements of the argument list satisfy `p`*. However, if we rather factor out and abstract over the constructors, that is*

```
mapPred :: Foldable t ⇒
           (α → Bool) → β → (Bool → β → β) → t α → β
mapPred p z c = gmap z c p
```

we can now make finite universal quantification over a general data structure with zero `z` *and constructor* `c`*, with the catamorphism*

```
forall  :: Foldable t ⇒ (α → Bool) → t α → Bool
forall p = and . (mapPred p z c)
```

Furthermore, from the theorem above we know that we can simplify that to

```
forall p = mapPred p true (&&)
```

In the end of the next section, we will look at other optimisations. We have now defined the foundation for our language, but catamorphisms by themselves are uneasy to work with and does not give an intuitive framework for writing queries. In the next section, we will introduce an abstraction layer on top of the catamorphisms, known as monad comprehensions, providing a syntax closer to modern query languages.

7.2 Queries as monad comprehensions†

We have seen how `do`-notation makes monadic code easier to write and read. For the `List` monad, there is however yet another layer of syntactic sugar. This new notation is used for writing *list comprehensions*, a way of defining lists based on other lists, predicates, and functions, with a set-like structure. The syntax can be written in BNF as

```
mc := [e | qs] | ma
qs := q | qs, qs
q  := v ← mc | boolExp
```

where `e` is any lambda expression (including new comprehensions) with possible free variables bound in `qs`, `ma` is any list, and `boolExp` is a boolean expression. So `[x * x | x ← [1..5], odd x]` results in the list $[1, 9, 25]$. This syntax is syntactically quite similar to relational calculus, but is in fact more expressive[12]. Such syntax is easily translated to `do`-notation. Assume `T` is the translation morphism from comprehension syntax to `do`-notation, then

```
T [e | q0, q1, ..., qn] = do { Tq q0; Tq q1; ...; Tq qn; return (T e)}
T exp                   = exp

Tq (v ← mc)             = v ← (T mc)
Tq boolExp              = guard boolExp
```

for any expression `exp` that is not a comprehension. If we apply this function to the example above, we get

```
T [x * x | x ← [1..5], odd x] = do
                                  x ← [1..5]
                                  guard (odd x)
                                  return $ x * x
```

We could also translate the syntax directly to regular monadic functions as

```
T [e | q0, q1, ..., qn] = Tq q0 (Tq q1 (... Tq qn (return T(e)) ...))
T exp                   = exp

Tq boolExp k            = guard boolExp >> k
Tq (v ← mc) k           = T mc >>= λ v → k
```

The comprehension syntax is special for lists in Haskell. The only dependency is has is through the use of `guard` which depends on `MonadPlus`, hence it could be defined for any monad implementing this. We will therefore abstract over all `MonadPlus` instances, and write $[\mathtt{e} \mid \mathtt{qs}] :: (\mathtt{m}\,\alpha)$ for a comprehension in monad `m` when needed[2].

Furthermore, we will also allow nested comprehensions of different monads, such as $[\mathtt{e} \mid \mathtt{v} \leftarrow [\mathtt{e}' \mid \mathtt{qs}] :: (\mathtt{m}'\,\alpha), \mathtt{p}\,\mathtt{v}] :: (\mathtt{m}\,\beta)$. To translate this, we need the catamorphisms defined in the previous section,

$$\mathtt{foldr} :: \mathtt{Foldable}\,\mathtt{t} \Rightarrow (\alpha \to \beta \to \beta) \to \beta \to \mathtt{t}\,\alpha \to \beta,$$

hence we also require our comprehension monads to be `Foldable`. We can now define a monad translator function from any foldable monad $\mathtt{m}'$ to a `MonadPlus` monad `m`.

Definition† 7.2.1. *A translation from a monad* $\mathtt{m}'$ *to a monad* `m` *is done with*

```
trans :: (Foldable m', MonadPlus m) ⇒ m' α → m α
trans = foldr mplus' mzero
    where mplus' x xs = mplus (return x) xs
```

We can then ornament `T` *with the name of the base monad of the comprehension it translates, as* $\mathtt{T}_\mathtt{m}$*, and add the following line to the above definition.*

$$\mathtt{T}_\mathtt{m}\,[\mathtt{e} \mid \mathtt{q}_0, \mathtt{q}_1, ..., \mathtt{q}_n] :: (\mathtt{m}'\,\alpha) = (\mathtt{trans}\,\$\,\mathtt{T}_{\mathtt{m}'}\,[\mathtt{e} \mid \mathtt{q}_0, \mathtt{q}_1, ..., \mathtt{q}_n]) :: (\mathtt{m}\,\alpha)$$

We will use these comprehensions as a base language on top of our catamorphisms for expressing queries. Let's look at how regular SQL queries can be expressed in this comprehension language. Regular SQL queries on the form

```
select e
  from r
 where p
```

[2]This notation is just to help the reader determine which monad we are working in. In an implementation this could be handled by the type inference algorithm.

can be translated to the comprehension [e | v ← r, p]. A regular SQL semi-join,

```
select e
  from r , s
 where p
```

can be represented as a comprehension as [e | v_1 ← r, v_2 ← s, p]. We are not limited to just SQL as our top level query language, but could also use other dialects or completely different languages. For instance, Grust shows how comprehensions can serve as a backend for the path describing language Xpath[12].

To complete the translation from SQL syntax, or some other language, all the way to catamorphisms, we need our monadic functions `fmap` and `join` to be expressible as catamorphisms. To achieve this we have to demand that our monads satisfy some properties. Such monads will be called *natural* monads, and the properties are given in the following definition. From now on we will write the infix function (`|`) for `mplus` where it is appropriate.

Definition† 7.2.2. *A monad* `m` *is called* natural *if it has* `Foldable` *and* `MonadPlus` *instances satisfying*

i) `fmap f mzero = mzero`

ii) `join mzero = mzero`

iii) `fmap f (a | b) = (fmap f a) | (fmap f b)`

iv) `join (a | b) = (join a) | (join b)`

v) `ma | mzero = ma`

vi) `(a | b) | c = a | (b | c)`

vii) `foldr ⊗ z (return a) = ⊗ a z`

viii) `foldr ⊗ z (a | b) = foldr ⊗ (foldr ⊗ z b) a`

Although these assumptions on our monadic computations seem limiting, they are in fact rather natural. There are several monads satisfying the criteria of naturality. For instance, it is quite easy to check that the monads `List`, `Bag` and `Set` from Grust's article are natural. In fact, natural monads are closed under a type of composition.

Theorem† 7.2.3. *If* `t` *is the monad transformer for some monad* `m` *such that*

```
newtype t n α = T { runT :: n (m α)}
```

for some constructor `T`*, and* `n` *is a natural monad, then* `t n` *is natural, with a* `Foldable` *instance given by*

```
foldr :: (α → β → β) → β → t n α → β
foldr f z nma = foldr f z $ (runT nma) >>= trans
```

and a `MonadPlus` *instance equal to* `n`*'s.*

Proof. Since the `MonadPlus` instance is equal to `n`'s and `n` is natural, `t n` will satisfy the naturality axioms concerning `mplus` and `mzero`. Observe that we need to add and remove the constructor `T` where appropriate, such that `mplus a b = T $ mplus (runT a) (runT b)`, and `mzero = T mzero`.

What remains is proving the last two axioms, those involving `foldr`. For the first, observe that by naturality of `n`,

```
foldr f z (return a)
 = foldr f z $ runT (T $ return (return a)) >>= trans
 = foldr f z $ return (return a) >>= trans
 = foldr f z $ foldr mplus mzero (return a)
 = foldr f z $ return a
 = f a z
```

For the last axiom, we have that

```
foldr f z (a ⁞ b)
 = foldr f z $ runT (a ⁞ b) >>= trans
 = foldr f z $ (runT a >>= trans) ⁞ (runT b >>= trans)
 = foldr f (foldr f z $ runT b >>= trans) (runT a >>= trans)
 = foldr f (foldr f z b) (runT a >>= trans)
 = foldr f (foldr f z b) a
```

where we got from the second to the third line by distributivity of `join` and `fmap`, and from the third to the fourth by axiom viii), both by naturality of `n`. The last two lines are concluded by definition of `foldr` for `t n`. □

Notice that all we demand from `m` in the above theorem is that it should be foldable, so it doesn't need to be natural. Hence, any monad transformer `t` applied to a natural monad `n` such that the unwrapped type is equal to `n (m α)` is natural. In the next section, we will prove the naturality of yet another set of monads.

Grust worked explicitly on the constructors of the monads, such as (:) and [ ] for lists. We will abstract these away and rather use `return` and `mplus` as a way of constructing our computations. This lets us generalise our work also to monads which lacks such constructors.

Definition† 7.2.4. *If a computation is equal to either* `mzero`*,* `return a`*, or* `ma ⁞ ma′` *for some* `ma :: m α`*,* `ma′ :: m α`*, we will say that the computation is on* natural form*. Furthermore, we will call the restriction of the set of all computations of a monad down to just computations on natural form for the* natural restriction*.*

Note that having computations on natural form isn't necessarily a desired property, but rather a constraint to make our future results provable. If one is able to generalise the results in rest of this chapter to computations on some other form, one should. However, all computations in `List`, `Bag`, and `Set` are on natural form, and we will mainly work with monads where this is the case.

Before we can define the monadic functions through catamorphisms, we need the following lemma.

Lemma† 7.2.5. *For any natural monad* `m`*, any* `ma :: m α` *on natural form, and any* `mb :: m β`*, we have*

$$\texttt{foldr}\,(\shortmid)\,\texttt{mb}\,\texttt{ma} = (\texttt{foldr}\,(\shortmid)\,\texttt{mzero}\,\texttt{ma}) \shortmid \texttt{mb}$$

Proof. We will let `z = mzero` in the following proof. The proof is by induction on `ma`. For both `ma = z` or `ma = return a` the equality is trivial. So assume $\texttt{ma} = \texttt{ma}_1 \shortmid \texttt{ma}_2$ and that the result holds for $\texttt{ma}_1$ and $\texttt{ma}_2$. Then we have

$$\begin{aligned}
\texttt{foldr}\,(\shortmid)\,\texttt{mb}\,(\texttt{ma}_1 \shortmid \texttt{ma}_2) &= \texttt{foldr}\,(\shortmid)\,(\texttt{foldr}\,(\shortmid)\,\texttt{mb}\,\texttt{ma}_2)\,\texttt{ma}_1 \\
&= (\texttt{foldr}\,(\shortmid)\,\texttt{z}\,\texttt{ma}_1) \shortmid (\texttt{foldr}\,(\shortmid)\,\texttt{mb}\,\texttt{ma}_2) \\
&= (\texttt{foldr}\,(\shortmid)\,\texttt{z}\,\texttt{ma}_1) \shortmid ((\texttt{foldr}\,(\shortmid)\,\texttt{z}\,\texttt{ma}_2) \shortmid \texttt{mb}) \\
&= ((\texttt{foldr}\,(\shortmid)\,\texttt{z}\,\texttt{ma}_1) \shortmid (\texttt{foldr}\,(\shortmid)\,\texttt{z}\,\texttt{ma}_2)) \shortmid \texttt{mb} \\
&= (\texttt{foldr}\,(\shortmid)\,(\texttt{foldr}\,(\shortmid)\,\texttt{z}\,\texttt{ma}_2)\,\texttt{ma}_1) \shortmid \texttt{mb} \\
&= (\texttt{foldr}\,(\shortmid)\,\texttt{z}\,(\texttt{ma}_1 \shortmid \texttt{ma}_2)) \shortmid \texttt{mb}
\end{aligned}$$

□

We are now able to state the `Monad` instance from the `Foldable` and `MonadPlus` instances.

Theorem† 7.2.6. *For the natural restriction of any natural monad* `m`*, we have*

i) $\texttt{join} = \texttt{foldr}\,(\shortmid)\,\texttt{mzero}$

ii) $\texttt{fmap} = \texttt{gmap}\,\texttt{mzero}\,(\shortmid')$

where $\texttt{a} \shortmid' \texttt{b} = (\texttt{return}\,\texttt{a}) \shortmid \texttt{b}$.

Proof. The proof is again by induction on the construction of the monads computations `ma`. Given either `ma = mzero`, `ma = return a` the result is trivial for both, so assume the equalities holds for some $\texttt{ma}_1$ and $\texttt{ma}_2$ and that $\texttt{ma} = \texttt{ma}_1 \shortmid \texttt{ma}_2$. It should be straight forward to see that

$$\begin{aligned}
\texttt{join}\,(\texttt{ma}_1 \shortmid \texttt{ma}_2) &= (\texttt{join}\,\texttt{ma}_1) \shortmid (\texttt{join}\,\texttt{ma}_2) \\
&= (\texttt{foldr}\,(\shortmid)\,\texttt{mzero}\,\texttt{ma}_1) \shortmid (\texttt{foldr}\,(\shortmid)\,\texttt{mzero}\,\texttt{ma}_2) \\
&= \texttt{foldr}\,(\shortmid)\,(\texttt{foldr}\,(\shortmid)\,\texttt{mzero}\,\texttt{ma}_2)\,\texttt{ma}_1 \\
&= \texttt{foldr}\,(\shortmid)\,\texttt{mzero}\,(\texttt{ma}_1 \shortmid \texttt{ma}_2)
\end{aligned}$$

by the induction hypothesis, the lemma above, and naturality of `m`, respectively. The second equality remains. Recall that

$$\texttt{gmap}\,\texttt{z}\,\texttt{c}\,\texttt{f} = \texttt{foldr}\,(\lambda\,\texttt{x}\,\texttt{xs} \to \texttt{c}\,(\texttt{f}\,\texttt{x})\,\texttt{xs})\,\texttt{z}.$$

Let's write

$$\texttt{f}' = \lambda\,\texttt{x}\,\texttt{xs} \to ((\texttt{f}\,\texttt{x}) \shortmid' \texttt{xs})$$

such that

$$\texttt{gmap mzero } (\mid') \texttt{ f } = \texttt{ foldr f' mzero}$$

to save ink. Then

$$\begin{aligned}
\texttt{fmap f } (\texttt{a} \mid \texttt{b}) &= (\texttt{fmap f a}) \mid (\texttt{fmap f b}) \\
&= (\texttt{foldr f' mzero a}) \mid (\texttt{foldr f' mzero b}) \\
&= \texttt{foldr f' } (\texttt{foldr f' mzero b}) \texttt{ a} \\
&= \texttt{foldr f' mzero } (\texttt{a} \mid \texttt{b}) \\
&= \texttt{gmap z } (\mid') \texttt{ f}
\end{aligned}$$

again by the induction hypothesis, the lemma above, and naturality of `m`, respectively.

One can easily see from the distributivity and identity properties of `join` and `fmap f` that the result of applying either of them to a computation in natural form results in a computation in natural form. □

We now see that our comprehensions can be translated to a language only constructed by catamorphisms, and is therefore subject to the nice features we saw in the previous section.

This completes the translation, and we now have the desired hierarchy: SQL syntax or some other suitable language on the top as an interface for a user; Monad comprehensions in the middle, similar to relational calculus; Catamorphisms at the bottom, enabling mathematical reasoning and optimisation of queries.

We will show one more of the optimisations from Grust's article, and show that it holds for all natural monads. The optimisation is concerned with a type conversion function used in some SQL dialects, known as `element`. This function unwraps a singleton of some collection, and throws an error either if there are zero or more than one element. Whenever we apply this function to a query, there is a chance that we do a lot of unnecessary computations. Take the following scenario,

```
element(select f x y
          from xs as x, ys as y)
```

If `xs` and `ys` contains more than one element each, this query is calculating `f x y` more than once, which is a waste since an error should be called right away. So an optimiser should rewrite the above query to

```
f (element xs) (element ys)
```

Now an error would be called immediately if either `xs` or `ys` contains more than one element. We will see that this translation is valid in our functional language. Grust uses the following definition of `element`,

$$\begin{aligned}
&\texttt{element} = \texttt{snd} \,.\, (\texttt{foldr} \otimes \texttt{z}) \\
&\quad \textbf{where } \texttt{z} \qquad\quad = (\texttt{true}, \bot) \\
&\qquad\quad \texttt{x} \otimes (\texttt{c}, \texttt{e}) = \textbf{if } \texttt{c} \textbf{ then} \\
&\qquad\qquad\qquad\quad (\texttt{false}, \texttt{x}) \\
&\qquad\qquad\qquad\ \textbf{else} \\
&\qquad\qquad\qquad\quad \bot
\end{aligned}$$

where evaluating $\bot$ raises an exception.

Lemma† 7.2.7. *Assume* `m` *is a natural monad where* `mplus` *preserves the quantity of elements. Then* `element` *is a morphism from the natural restriction of* `m`*, to the Identity monad, such that*

$$\texttt{element} \,.\, \texttt{return} = \texttt{id}$$

$$\texttt{join} \,.\, \texttt{element} \quad = \texttt{element} \,.\, \texttt{element}$$

and in addition, if for any `xs`*,* $\texttt{size}\,\texttt{xs} \leq \texttt{size}\,(\texttt{fmap}\,\texttt{f}\,\texttt{xs})$*, then*

$$\texttt{element} \,.\, (\texttt{fmap}\,\texttt{f}) = \texttt{f} \,.\, \texttt{element}$$

where $\texttt{size} = \texttt{foldr}\,(\lambda\,\texttt{x} \to (+1))\,0$*.*

Proof. It is quite easy to see that $\texttt{element} \,.\, \texttt{return} = \texttt{id}$ by axiom vii) of naturality, which proves the first result. For the second equality, we get $\bot$ for both sides if the argument is either `mzero` or $(\texttt{a} \mid \texttt{b})$ for non-zero `a` and `b`. If the argument is $\texttt{return}\,\texttt{a}$, we get

$$\begin{aligned}\texttt{element}\,\$\,\texttt{join}\,(\texttt{return}\,\texttt{a}) &= \texttt{element}\,\texttt{a} \\ &= \texttt{element}\,\$\,\texttt{element}\,(\texttt{return}\,\texttt{a})\end{aligned}$$

by the first equality.

In the last law we again get an error if the argument to either of the sides is empty or a monadic sum, since $\texttt{fmap}\,\texttt{f}$ never decreases the number of elements in its argument, and `mplus` preserves the quantity of elements. If the argument is on the form $\texttt{return}\,\texttt{a}$, we have

$$\begin{aligned}\texttt{element}\,\$\,\texttt{fmap}\,\texttt{f}\,(\texttt{return}\,\texttt{a}) &= \texttt{element}\,\$\,\texttt{return}\,(\texttt{f}\,\texttt{a}) \\ &= \texttt{f}\,\texttt{a} \\ &= \texttt{f} \,.\, \texttt{element}\,\$\,\texttt{return}\,\texttt{a}\end{aligned}$$

by the last monad law for the `join`-definition and $\texttt{element} \,.\, \texttt{return} = \texttt{id}$ respectively. □

Notice that the requirement on `f` is necessary in the lemma, and that this implies that `element` isn't a proper monad morphism. This can be seen by the counter example below.

$$\texttt{f} = \lambda\,\texttt{x} \to \textbf{if}\ \texttt{x} == 1\ \textbf{then}\ \texttt{return}\,\texttt{x}\ \textbf{else}\ [\,]$$

$$\texttt{element}\,([1,2] \gg\!= \texttt{f}) = 1$$

$$\texttt{element}\,[1,2] \gg\!= \texttt{element} \,.\, \texttt{f} = \bot$$

We also need the requirement on `mplus`, since for monads like `Set`, where $\texttt{mplus}\,\texttt{a}\,\texttt{a} = \texttt{a}$, we for instance can get

$$\texttt{element}\,\$\,\texttt{fmap}\,(\lambda\,\texttt{x} \to 1)\,\{1,2,3\} = 1$$

$$(\lambda\,\texttt{x} \to 1)\,\$\,\texttt{element}\,\{1,2,3\} = \bot$$

Using the lemma above, we are automatically able to generalise Grust's result to all natural monads satisfying the lemma.

Theorem 7.2.8. *For any natural monad* `m` *where* `mplus` *preserves quantity, the example query from above,*

```
element (select f x y
                from xs as x, ys as y)
```

is equivalent to

```
f (element xs) (element ys).
```

for `xs` *and* `ys` *on natural form.*

Proof. We have that the first of the two expressions translates to

```
element [f x y | x ← xs, y ← ys].
```

Continuing the translation down to monadic code, and using lemma 7.2.7, we get

```
element [f x y | x ← xs, y ← ys]
 = element (xs >>= λx → ys >>= λy → return $ f x y)
 = element . join $ fmap (λx → ys >>= λy → return $ f x y) xs
 = element $ ys >>= λy → return $ f (element xs) y
 = element . join $ fmap (λy → return $ f (element xs) y) ys
 = element . element $ fmap (λy → return $ f (element xs) y) ys
 = element . return $ f (element xs) (element ys)
 = f (element xs) (element ys)
```

Observe that we were able to use the lemma in the lines 3 and 6, since neither of the functions

```
fmap (λx → ys >>= λy → return $ f x y)

fmap (λy → return $ f (element xs) y)
```

decreases the size of its argument. □

This was just one small example of optimising rewritings, several more can be done. For example, by induction we can show that the above result holds for functions `f'` with any arity, not just for binary functions. There are several other similar optimisations, and it would be interesting to see which is satisfied by our natural monads. This could be a topic for future research.

This concludes our discussion on optimisations. For more the interested reader can consult Grust's article.

7.3 Backtracking†

The language we have constructed is well suited for expressing query languages like SQL. In this section we will see that we can extend our language, without cost, with backtracking computations similar to that of Datalog. We will use the work of Hinze[13] and his backtracking monad.

The idea is to create a backtracking core for which monads can implement. Hinze defined this core as

```
class Monad m ⇒ Backtr m where
  fail :: m α
  (|)  :: m α → m α → m α
  once :: m α → m (Maybe α)
  sols :: m α → m [α]
```

Since `fail` represents the same as `mzero` and `(|)` is equivalent to our usage above, we will rather use the following definition:

```
class MonadPlus m ⇒ Backtr m where
  once :: m α → m (Maybe α)
  sols :: m α → m [α]
```

Here `mzero` represents a failed computation as usual, `(|)` represents the actual backtracking, that is $\texttt{m}_1$ `|` $\texttt{m}_2$ should be interpreted "try $\texttt{m}_1$ first, if that fails try $\texttt{m}_2$". The `once`-function extracts one result if there is one, while `sols` gathers all successful computations' results in a list. Hinze assumes that (`mzero`, `(|)`) is a proper monoid, and apart from defining a `Foldable` instance for his backtracking computations, he demands all other laws of naturality.

With this core we can write facts and rules with resemblance to a Datalog program. For instance

```
child         :: Backtr m ⇒ String → m String
child "peter" = (return "sandra") | (return "abe") | (return "carl")
child "carl"  = (return "olivia") | (return "alex")
child "alex"  = (return "stuart") | (return "sue")
child _       = mzero
```

represents facts, while

```
grandChild :: Backtr m ⇒ String → m String
grandChild = child ⊙ child
```

introduces a new relation based on a "rule". For example

```
grandChild "carl"
 = child ⊙ child $ "carl"
 = child "carl" ≫= child
 = (return "olivia") | (return "alex") ≫= child
 = (return "olivia" ≫= child) | (return "alex" ≫= child)
 = mzero | (return "alex" ≫= child)
 = return "alex" ≫= child
 = (return "stuart") | (return "sue")
```

Here we can clearly see the backtracking, where the computation `child "olivia"` failed.

We could go further and define

```
descendant :: Backtr m ⇒ String → m String
descendant = child ⊖ (descendant ◎ child)
```

where $(\ominus)$ is the union of two relations defined as

```
(⊖)    :: Backtr m ⇒ (α → m β) → (α → m β) → (α → m β)
f ⊖ g = λa → f a ∣ g a
```

If we read `r ⊖ q` as "r or q" and `r ◎ q` as "r of a q", then the above reads as "a descendant is a child or a descendant of a child", while "a grandChild is a child of a child". This syntax can remind one of the syntax of description logic, although far from equivalent.

General transitive closure of relations can be defined as

```
transitiveClosure  :: Backtr m ⇒ (α → m α) → α → m α
transitiveClosure r = r ⊖ (transitiveClosure r ◎ r)
```

Hinze discusses the transitive closure in more detail and defines an improved version of the one above.

The backtracking core gives us some elementary backtracking features seen in languages such as Prolog and Datalog. We don't have logical variables, nor are our rules declarative since our definitions are sequential and the order of the rules does matter. However, we still have all the positive features of Haskell combined with backtracking, which for instance results in higher order relations.

Hinze further defines a backtracking monad transformer adding backtracking features to any monad. The definition of this transformer is as follows.

```
newtype BacktrT m α = ∀β . (α → m β → m β) → m β → m β

instance Monad (BacktrT m) where
  return a = λk → k a
  m >>= f = λk → m (λa → f a k)
```

A computation in `BacktrT m` is a function taking two arguments. The first is a function which is applied to every value contained in the computation in a backtracking manner. The result is a function with type $\mathtt{m}\,\alpha \to \mathtt{m}\,\alpha$. The argument to this function will be returned if the backtracking computation fails.

With this in mind, it shouldn't be too hard to grasp the meaning of `m >>= f`. Here, the values in `m` will be bound to `a`, for which `f` will be applied. Then we abstract over the argument function and feed this to the result of each `f a`.

Furthermore, we can augment the resulting monads with the backtracking core implementing the following instances:

```
instance Monad m ⇒ MonadPlus (BacktrT m) where
  mzero = λk → id
  m ∣ n = λk → (m k) . (n k)
```

```
instance Monad m ⇒ Backtr (BacktrT m) where
  once m = λ k f → m first nothing ≫= λ x → k x f
  sols m = λ k f → m cons nil ≫= λ x → k x f
```

where

```
nothing  :: m (Maybe α)
nothing  = return Nothing
first    :: Monad m ⇒ α → m (Maybe α) → m (Maybe α)
first a _ = return (Just a)

nil      :: m [α]
nil      = return [ ]
cons     :: Monad m ⇒ α → m [α] → m [α]
cons a mx = do
              x ← mx
              return (a : x)
```

Example 7.3.1. *Let's look at the example above, with the definition of the relation* `child`*. We can update its type to* `child :: String → BacktrT [ ] String`*, and for instance write*

```
(child "carl") (:) [ ] = ((return "olivia") ∣ (return "alex")) (:) [ ]
                       = ((λ k → k "olivia") ∣ (λ k → "alex")) (:) [ ]
                       = (λ k → (k "olivia") . (k "alex")) (:) [ ]
                       = ["olivia", "alex"]
```

We could also bind a computation to `child`*, as*

```
(child "carl") ≫= child
= λ k → (child "carl") (λ a → f a k)
= λ k → ((return "olivia") ∣ (return "alex")) (λ a → child a k)
= λ k → (child "olivia" k) . (child "alex" k)
= λ k → (mzero k) . (((return "stuart") ∣ (return "sue")) k)
= λ k → id . ((k "stuart") . (k "sue"))
= λ k → (k "stuart") . (k "sue")
```

If `m` *is the computation above, then* `once m = return (Just "stuart")`*, while* `sols m = return ["stuart", "sue"]`.

Notice how we don't need to assume that `m` is a monad in the monadic instance definition of `BacktrT m`. That means that `BacktrT m` is a monad for any data type `m :: ∗ → ∗`, not just monads. Also, the only requirement on `m` in the definition of the backtracking core, `Backtr`, is that is is a monad, so `m` doesn't need a `Backtr` instance.

There are some other neat features of this monad not present in most other backtracking monads. One notable feature is that both (≫=) and (∣) execute in constant time. If we want to retrieve all solutions of the computation in some data structure, we still need to visit every value.

However, fetching `n` elements, for some fixed integer `n`, still runs in constant time.

We are now ready to bridge our catamorphism based query language with backtracking. It turns out that the framework defined in the previous sections is expressive enough to include the above-defined backtracking feature. First of all, note that if we restrict ourselves to only the functions defined above, all computations are on normal form. This can be proven by a short induction proof. Furthermore, we have already stated that almost all the laws of naturality are satisfied, apart from the definition of a `Foldable` instance with the desired properties. To define the `Foldable` instance for `BacktrT m`, we need to add a dependency on the argument monad `m`. It needs to be escapable, that is, it needs to implement the `Run` class.

```
class Monad m ⇒ Run m where
  run :: m α → α
```

Here we require `run . return = id`. The requirement of implementing `Run` is not really a catch if we intend to work with natural monads due to the following result.

Lemma† 7.3.2. *All natural monads are escapeable, and can therefore implement* `Run`*.*

Proof. Since all computations in any natural monads can be transformed into a list through

```
toList :: Foldable t ⇒ t α → [α]
toList = foldr (:) [ ]
```

we can let `run = head . toList`. □

Since `head` is a partial function, `run` will also be. `run` will fail if its argument is `mzero`. We will shortly see that for our application, this is no problem. Most other monads, like `State` or `Pair`, can also implement `Run` easily. For `State` we can just set `run ma = fst $ runState ma s` for some appropriate state value `s`. Notice that this state value might be different for each `State σ`, and `State σ` is only escapeable if σ is inhabited. For `Pair` we could just extract the left-most value, if the pair contains a value. We can now define the `Foldable` instance for `BacktrT`.

Definition† 7.3.3. *The* `Foldable` *instance for the Backtracking monad is defined as*

```
instance Run m ⇒ Foldable (BacktrT m) where
  foldr f z ma = run $ ma (liftB f) (return z)

liftB :: Monad m ⇒ (α → β → β) → α → m β → m β
liftB f a mb = liftM (f a) mb
```

Folding a function `f` over a backtracking computation is done by lifting `f` and feeding this lifted function into the computation. The lifted function is then applied to every result of the computation. We now have something of type $\mathtt{m}\,\beta \to \mathtt{m}\,\beta$, which is a function taking as argument the desired output in case the computation fails. This is then fed the monadically wrapped up `z`, such that a failed computation results in `return z`. The desired result is now computed, but is inside the monad `m`, hence we apply the unwrapping function `run`.

Notice that if the argument to `foldr f z` is `mzero`, `run` will be applied to `return z` and never `z`, so even though `run` is partial, `foldr` is not.

We will prove the correctness of this definition, but first we need a lemma.

Lemma† 7.3.4. *For any runnable monad* `m`*, any function* $\mathtt{f} :: \alpha \to \beta \to \beta$*, any* $\mathtt{z} :: \beta$*, and any* $\mathtt{ma} :: \mathtt{BacktrT}\,\mathtt{m}\,\alpha$*, there exists a* $\mathtt{b} :: \beta$ *such that*

$$\mathtt{ma}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z}) = \mathtt{return}\,\mathtt{b}$$

Proof. Seen easily by induction on `ma`. For either `ma` $=$ `mzero` or `ma` $=$ `(return a)`, the result is trivial. Assume `ma` $=$ `a | b`. Then

$$\begin{aligned}(\mathtt{a} \mid \mathtt{b})\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z}) &= (\lambda\,\mathtt{k} \to (\mathtt{a}\,\mathtt{k})\,.\,(\mathtt{b}\,\mathtt{k}))\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z}) \\ &= \mathtt{a}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{b}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z})) \\ &= \mathtt{a}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{b}') \\ &= \mathtt{return}\,\mathtt{a}'\end{aligned}$$

where the two last equalities follow by the induction hypothesis. □

Theorem† 7.3.5. *For any runnable monad* `m`*,* `BacktrT m` *is a natural monad.*

Proof. There are two laws concerning `foldr`. The first is

$$\mathtt{foldr}\,\mathtt{f}\,\mathtt{z}\,(\mathtt{return}\,\mathtt{a}) = \mathtt{f}\,\mathtt{a}\,\mathtt{z}.$$

We can easily see that

$$\begin{aligned}\mathtt{foldr}\,\mathtt{f}\,\mathtt{z}\,(\mathtt{return}\,\mathtt{a}) &= \mathtt{run}\,\$\,(\lambda\,\mathtt{k} \to \mathtt{k}\,\mathtt{a})\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z}) \\ &= \mathtt{run}\,\$\,(\mathtt{liftB}\,\mathtt{f})\,\mathtt{a}\,(\mathtt{return}\,\mathtt{z}) \\ &= \mathtt{run}\,\$\,\mathtt{return}\,(\mathtt{f}\,\mathtt{a}\,\mathtt{z}) \\ &= \mathtt{f}\,\mathtt{a}\,\mathtt{z}\end{aligned}$$

The second law is

$$\mathtt{foldr}\,\mathtt{f}\,\mathtt{z}\,(\mathtt{a} \mid \mathtt{b}) = \mathtt{foldr}\,\mathtt{f}\,(\mathtt{foldr}\,\mathtt{f}\,\mathtt{z}\,\mathtt{b})\,\mathtt{a}$$

By the above lemma and the fact that `run . return` $=$ `id`, we have

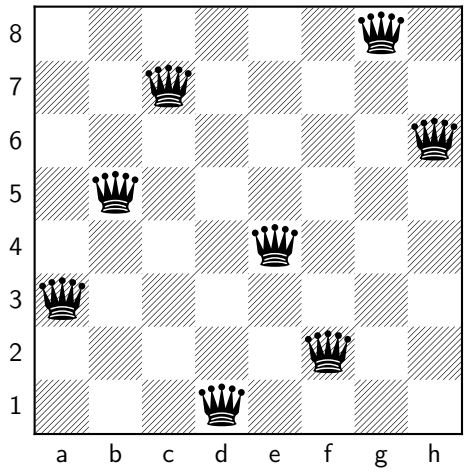
$$\begin{aligned}\mathtt{foldr}\,\mathtt{f}\,\mathtt{z}\,(\mathtt{a} \mid \mathtt{b}) &= \mathtt{run}\,\$\,(\mathtt{a} \mid \mathtt{b})\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z}) \\ &= \mathtt{run}\,\$\,(\lambda\,\mathtt{k} \to (\mathtt{a}\,\mathtt{k})\,.\,(\mathtt{b}\,\mathtt{k}))\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z}) \\ &= \mathtt{run}\,\$\,\mathtt{a}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{b}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{z})) \\ &= \mathtt{run}\,\$\,\mathtt{a}\,(\mathtt{liftB}\,\mathtt{f})\,(\mathtt{return}\,\mathtt{b}')\end{aligned}$$

$$
\begin{aligned}
&= \texttt{foldr f b' a} \\
&= \texttt{foldr f (run \$ return b') a} \\
&= \texttt{foldr f (run \$ b (liftB f) (return z)) a} \\
&= \texttt{foldr f (foldr f z b) a}
\end{aligned}
$$

□

The only dependency the argument to `BacktrT` has, is that is should be a runnable monad. This means that we can construct queries where the result is in any such monad, also unnatural monads, like `State`. Let's end this section with an example showing an application of our backtracking query language.

Example 7.3.6. *Let's look at an example of a combination of backtracking and our monadic query language. We will use Hinze's implementation of the* `n`*-queens problem[13], and show how we can write this in comprehension syntax. The* `n`*-queens problem is the problem of placing* `n` *queens on an* $\texttt{n} \times \texttt{n}$ *chess board, such that no two queens can attack each other. Notice that every solution can be written as a list of integers, the index is the column while the value is the row. For example*



□

can be written as $[3, 5, 7, 1, 4, 2, 8, 6]$. *Notice further that every solution is a permutation of this list, and that for every placement we only have to check whether queens can attack each other on the diagonal. We will represent diagonals as two lists, one for up diagonals (going up from left to right), and one for down diagonals, both with numbers from* -7 *to* 7. *A query solving this problem is given below.*

$$
\begin{aligned}
&\texttt{place} :: \texttt{Backtr m} \Rightarrow \texttt{Int} \to [\texttt{Int}] \to [\texttt{Int}] \to [\texttt{Int}] \to \texttt{m}\,[\texttt{Int}] \\
&\texttt{place i rs d}_1\,\texttt{d}_2 = [(\texttt{q} : \texttt{qs}) \,|\, (\texttt{q}, \texttt{rs}') \leftarrow \texttt{select rs}, \\
&\qquad\qquad\qquad\qquad \texttt{q} - \texttt{i} \notin \texttt{d}_1, \\
&\qquad\qquad\qquad\qquad \texttt{q} + \texttt{i} \notin \texttt{d}_2, \\
&\qquad\qquad\qquad\qquad \texttt{qs} \leftarrow \texttt{place}\,(\texttt{i} - 1)\,\texttt{rs}'\,(\texttt{q} - \texttt{i} : \texttt{d}_1)\,(\texttt{q} + \texttt{i} : \texttt{d}_2)]
\end{aligned}
$$

where `select` *is defined as*

$$
\texttt{select} :: \texttt{Backtr m} \Rightarrow [\alpha] \to \texttt{m}\,(\alpha, [\alpha])
$$

```
select [ ]     = mzero
select (a : x) = return (a, x) ∣
                 [(b, a : x′) | (b, x′) ← select x]
```

So `place` *tries to place out queens, backtracks if it fails in either of the guards, and returns all possible solutions. The first argument to* `place` *is the number of queens not yet placed on the board; the second is the list of possible placements; the third and fourth are the currently attacked up and down diagonals respectively. The* `select` *function creates a monadic sum of selections of elements, where one selection is a tuple* (`a`, `xs`) *where* `a` *is the selected item and* `xs` *is the list of the unselected items.*

We can now define

```
queens :: Backtr m ⇒ Int → m [Int]
queens n = place n [1..n] [ ] [ ]
```

which solves the problem. We could then, for instance, execute the computation in `BacktrT [ ]` *and retrieve the solutions of the 8th-queens problem as a list with*

```
solution :: BacktrT [ ] [Int]
solution = queens 8

solutionList :: [[Int]]
solutionList = solution (:) [ ]
```

We could easily change the backtracking monad used by changing the type of `solution`*, also to unnatural monads. Say we wanted to put all solutions in the state value of a computation in the* `State` *monad, this can be done with*

```
addToState     :: α → State [α] () → State [α] ()
addToState a sa = sa ≫ modify (a :)


solution :: BacktrT (State [[Int]]) [Int]
solution = queens 8

solutionState :: State [[Int]] ()
solutionState = solution addToState (return ())
```

Another neat feature of our language is that we can use all functions already defined for any natural monad. Say for instance `select` *was a predefined function into the* `List` *monad,* `select` :: $[\alpha] \to [(\alpha, [\alpha])]$*. Then we could still just use it as it is used above. Our translator function* $\mathtt{T}_{\mathtt{m}}$ *defined in the previous section would then just inject* `trans` *before* `select rs` *in the query above. This would translate* `select rs` *to the definition of* `select` *we have defined above.*

These features show how easily we can integrate our query language with other monads and the rest of Haskell.

Part III

Deeper Theory

Chapter 8

Category Theory

Since the monad originally is a categorical construct, it would seem only natural to devote a chapter to the category theoretic foundation for monads. This will also familiarise mathematicians and computer scientists from the more theoretical disciplines.

8.1 Categories

Category theory originated as the theory of structure preserving transformations and is now used as a general mathematical language [4]. Its main construct is the category.

Definition 8.1.1. *A category consists of a collection of objects and a collection of arrows satisfying the following:*

i) For every arrow f, there are objects $A = dom(f)$ and $B = cod(f)$ called the domain and codomain of f respectively, and we write $f : A \to B$.

ii) For every couple of arrows $f : A \to B$ and $g : B \to C$ where $cod(f) = dom(g)$, there exists a composite arrow, written $g \circ f : A \to C$.

iii) Arrow composition is associative, that is, $(f \circ g) \circ h = f \circ (g \circ h)$ for any arrows f, g, and h.

iv) For every object A there is an identity arrow $1_A : A \to A$.

v) For any arrow $f : A \to B$ there are identity arrows 1_A and 1_B such that $f \circ 1_A = 1_B \circ f = f$.

Example 8.1.2. *One of the perhaps most well known example of a category is the category of sets, where objects are sets and arrows are functions over sets. So $f : A \to B$ is a function from the set A to the set B, arrow composition is regular function composition, hence $(g \circ f)(x) = g(f(x))$.*

Example 8.1.3. *We have studied monads in functional programming, so how does categories come into functional programming? In typed functional languages, such as Haskell, the type system is a category. The types are the objects and functions are arrows. To verify this, note the following:*

i) Every function has an argument type, which is its domain, and a return type, which is its codomain.

ii) In Haskell we have a composition function

$$(.) :: (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$
$$\texttt{g . f} = \lambda \texttt{x} \to \texttt{g (f x)}$$

playing the role of $\circ$*. It is trivial to verify that* $(\texttt{f . g})\texttt{ . h} = \texttt{f . (g . h)}$*, so it is clearly associative.*

iii) We have a polymorphic identity function

$$\texttt{id} :: \forall\, \alpha \,.\, \alpha \to \alpha$$
$$\texttt{id} = \lambda \texttt{x} \to \texttt{x}$$

which can be made the identity function for any type by explicitly typing it. We can e.g. write $\texttt{id} :: \gamma \to \gamma$ *for the identity function of the* γ *type.*

iv) For any function $\texttt{f} :: \alpha \to \beta$ *we have* $\texttt{f . id} = \texttt{id . f} = \texttt{f}$*.*

It is important to note that even though both the notation and my examples suggest that arrows are functions, they need not be. There are several examples of arrows which aren't functions. Take e.g. preorders (sets with a transitive and reflexive relation). If we let the elements of the set be objects and let there be an arrow between related elements, we get a category. Deductions are also arrows, if we let logical sentences be objects. Then every deduction be an arrow from its assumption to its conclusion. For these, and several other examples, see Awodey's book[4].

8.2 Functors and natural transformations

Categories are all well and good, but currently we have no way of relating different categories, and we can't do anything with our arrows except for combining them. For this we need the concept of a functor, structure preserving morphisms of categories.

Definition 8.2.1. *A functor* $F : \mathcal{C} \to \mathcal{C}'$ *is a mapping from a category* $\mathcal{C}$ *to a category* $\mathcal{C}'$ *such that for any objects* A *and* B*, and any arrows* $f : A \to B$ *and* $g : B \to C$ *from* $\mathcal{C}$*,*

$$F(f) : F(A) \to F(B)$$
$$F(1_A) = 1_{F(A)}$$
$$F(g \circ f) = F(g) \circ F(f).$$

Functors can be used as translations between equivalent categories, embedding a smaller category into a larger one, or even collapsing a large category into a smaller one. If we create endo-functors, functors from a category back into itself, we can express morphisms from one arrow to

another. We have seen functors as types with kind $* \to *$, with functor mappings defined through the `fmap` function in the `Functor` type class.

We will now climb higher on the mountain of abstraction. While functors represents morphisms of arrows, category theory was originally made for studying morphisms of functors [19].

Definition 8.2.2. *A natural transformation $\eta : F \to G$ from a functor $F : \mathcal{C} \to \mathcal{C}'$ to a functor $G : \mathcal{C} \to \mathcal{C}'$ is a family of arrows defining $\eta_X : F(X) \to G(X)$ for each object X in $\mathcal{C}'$, such that*

$$\eta_B \circ F(f) = G(f) \circ \eta_A$$

for any arrow $f : A \to B$ in $\mathcal{C}$. For any functors $T : \mathcal{C}'' \to \mathcal{C}$ and $K : \mathcal{C}' \to \mathcal{C}''$, we write $\eta_T : FT \to GT$ for the family $\eta_{T(X)} : F(T(X)) \to G(T(X))$, and $K\eta : KF \to KG$ for the family $K(\eta_X) : K(F(X)) \to K(G(X))$.

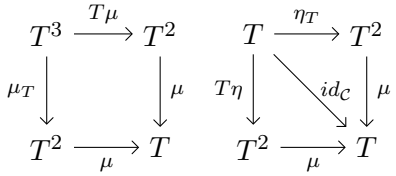
We have seen many natural transformations in the previous chapters. Natural transformations are morphisms from one functor to another, hence any function `f :: F α → G α` is a natural transformation, a mapping from the functor `F` to the functor `G`. For example, all monad morphisms and many of the catamorphisms we saw in the previous chapter are natural transformations.

This concludes our discussion on general categorical constructs. For more examples of categories, functors, and natural transformations and a more thorough discussion, see [4]. We will now turn towards our main goal, the definition of the monad.

8.3 Categorical monads

The monad is traditionally defined through the use of one endo-functor and two natural transformations, which is the categorical version to our definition using `join`.

Definition 8.3.1. *Given a category $\mathcal{C}$, a categorical monad is a triple $\langle T, \mu, \eta \rangle$, where $T : \mathcal{C} \to \mathcal{C}$ is an endo-functor, $\mu : T^2 \to T$ and $\eta : id_{\mathcal{C}} \to T$ are two natural transformations such that $\mu \circ T\mu = \mu \circ \mu_T$ and $\mu \circ T\eta = \mu \circ \eta_T = id_{\mathcal{C}}$, where $id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ is the identity functor on $\mathcal{C}$. As commuting diagrams, we have*



□

From this one can define the following

Definition 8.3.2. *A Kleisli triple is given by $\langle T, \eta, \_^* \rangle$, where T and η is as given above, and for a given $f : A \to T(B)$ we have $f^* : T(A) \to T(B)$, such that ${\eta_A}^* = id_{T(A)}$, $f^* \circ \eta_A = f$ for every $f : A \to T(B)$, and $g^* \circ f^* = (g^* \circ f)^*$ for every $f : A \to T(B)$ and $g : B \to T(C)$.* □

This definition corresponds to our regular definition, where the Kleisli Star $\_^*$ is equivalent to $(\gg\!=)$, only with the order of the arguments switched. One can prove that there is an isomorphic correspondence between monads and Kleisli triples, by observing that $\mu_A = {id_{T(A)}}^*$, and for a given $f : A \to T(B)$ we have $f^* = \mu_B \circ T(f)$. This is quite similar to the proof restricted to the two type theoretic definitions we did in Chapter 2.

If we introduce the categorical Kleisli composition ($\circledcirc$), that is for $g : A \to T(B)$ and $g : B \to T(C)$,

$$f \circledcirc g = (f^* \circ g)^*$$

the laws above are expressed as

i) $f \circledcirc \eta = f$

ii) $\eta \circledcirc f = f$

iii) $(f \circledcirc g) \circledcirc h = f \circledcirc (g \circledcirc h)$

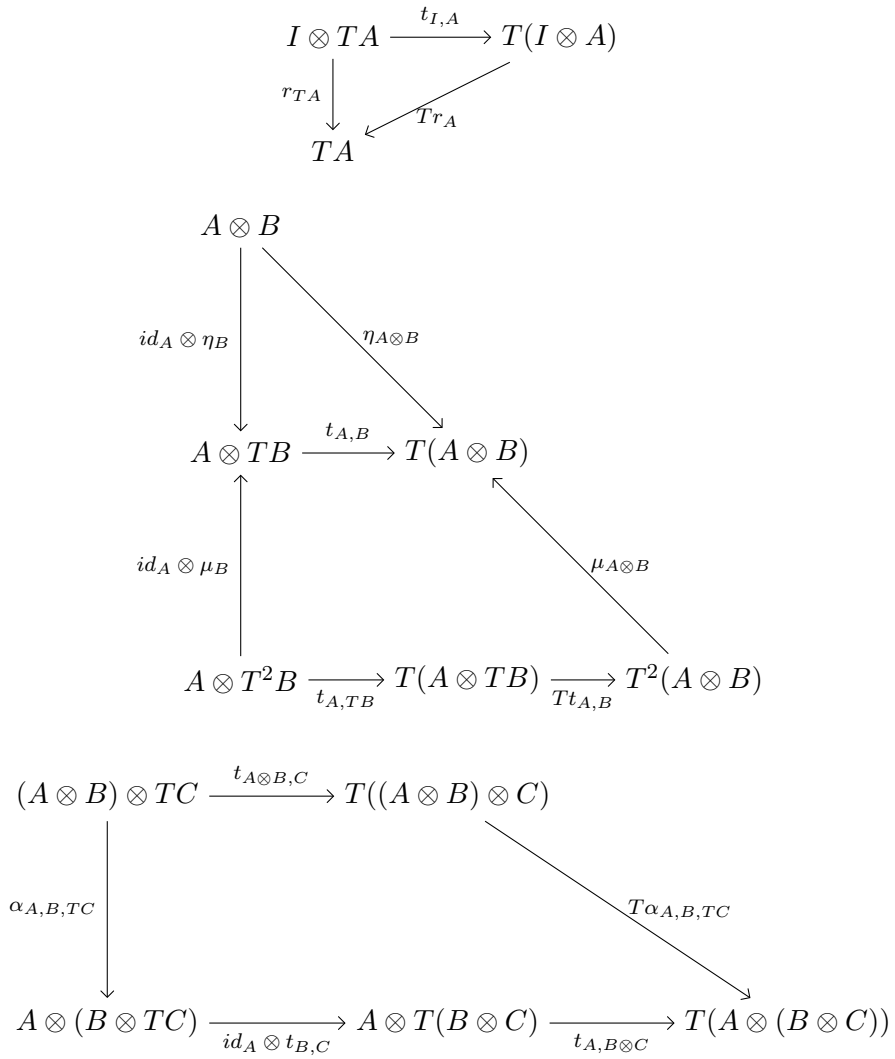
Observe that these laws are the properties of a category. Hence, a monad is a category over arrows of the form $f : A \to T(B)$, for any objects A, B, and Kleisli composition as arrow composition.

Monads are quite important in category theory, specially in the study of adjunctions. An adjunction is a relationship between certain functors, and if two functors satisfy such a relationship they are called adjoint functors. These functors are very important in mathematics, as Awodey puts it: "Indeed, I will make the admittedly provocative claim that adjointness is a concept of fundamental logic and mathematical importance that is not captured elsewhere in mathematics."[4] In his book, he shows how adjunctions can represent both universal quantification in logic and image operations of continuous functions in topology, stating that these two seemingly unrelated concepts actually are the same thing in category theory.

Monads are interesting in the study of adjunctions, since every adjunction gives rise to both a monad, and every monad gives rise to an adjunction. Also, if an endo-functor T is part of a monad, then T is the composition of two adjoint functors. The proof for these statements as well as a deep discussion on the relationship between monads and adjunctions can be seen in Awodey's book[4].

Before we finish this this chapter, we will look at a particular difference in the concept of the categorical monad and type theoretic monad. In fact, what we in functional programming call a monad really corresponds to what in category theory is called a *strong monad*[20].

Definition 8.3.3. *A strong monad is a monad $\langle T, \mu, \eta \rangle$ over a monoidal category $\langle \mathcal{C}, \otimes, I \rangle$ coupled with a natural transformation $t_{A,B} : A \otimes TB \to T(A \otimes B)$, such that the following diagrams commute*



where $r_A : (I \otimes A) \to A$ *and* $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$ *are natural isomorphisms.* □

A strong monad is just a monad behaving nicely under a Cartesian product. However, for any functional monad `m` and Cartesian product over types $(,)$, we can define the function

```
strength :: Monad m ⇒ (α, m β) → m (α, β)
strength (a, mb) = mb ≫= λ b → return (a, b)
```

This function satisfies the properties of the natural transformation $t_{A,B} : A \otimes TB \to T(A \otimes B)$ required for a strong monad[22]. So we never have to consider strength of a monad in Haskell, they are all strong monads.

Chapter 9

Monads and the Curry-Howard Correspondence†

In this chapter we will investigate how monads appear through the Curry-Howard Correspondence. The correspondence states that there is an isomorphism between the lambda expressions in typed lambda calculus and the proofs in constructive logic. The type of the arguments corresponds to the assumptions in the proof; the return value of the expression corresponds to the conclusion of the proof; the morphism of the arguments to the return value in the function (or the function body) corresponds to the actual proof. So a type is inhabited if and only if the corresponding formula is a theorem. The correspondence maps different type theories to different logics. For instance, simply typed lambda calculus corresponds to constructive propositional logic.

We will in this chapter see how simply typed lambda calculus extended with monads, called Computational Lambda Calculus, is isomorphic to a subsystem of constructive S4 modal logic, known as CL or Lax logic. Some familiarity with modal logic is necessary to appreciate this chapter. For an introduction to modal logic, see for instance Blackburn, Benthem, and Wolter's book "Handbook of modal logic"[6].

9.1 New notation

We start by introducing some new notation for the monad, a more appropriate representation for use in a proof calculus.

Definition 9.1.1. *A monad consists of a type T coupled with the two expressions $val(x)$ and $(let\ x \Leftarrow e\ in\ f)$, such that if $x : \varphi$ then $val(x) : T\varphi$, and if $f : T\psi$ with $x : \varphi$ free and $e : T\varphi$ then $(let\ x \Leftarrow e\ in\ f) : T\psi$. The*

expressions satisfies the following three laws:

$$\begin{aligned} \mathit{let}\ x \Leftarrow (\mathit{val}(e))\ \mathit{in}\ f &= f[e/x] \\ \mathit{let}\ x \Leftarrow e\ \mathit{in}\ (\mathit{val}(x)) &= e \\ \mathit{let}\ y \Leftarrow (\mathit{let}\ x \Leftarrow e\ \mathit{in}\ f)\ \mathit{in}\ g &= \mathit{let}\ x \Leftarrow e\ \mathit{in}\ (\mathit{let}\ y \Leftarrow f\ \mathit{in}\ g), \end{aligned}$$

It is easy to see that $\mathit{val}(x)$ represents ($\mathtt{return\,x}$), whereas ($\mathit{let}\ x \Leftarrow e\ \mathit{in}\ f$) represents $\mathtt{e} \gg\!= \lambda\,\mathtt{x} \to \mathtt{f}$ (or as do notation, $\mathbf{do}\,\{\mathtt{x} \leftarrow \mathtt{e};\ \mathtt{f}\}$), and the three laws are equivalent to the functional definition.

We will, in definitions and proofs, let

$$\begin{array}{c} D \\ \Gamma \vdash \varphi \end{array}$$

denote the proof, D, which concludes with $\Gamma \vdash \varphi$, so $\Gamma \vdash \varphi$ is *included* in D. On the other hand,

$$\frac{D}{\Gamma \vdash \varphi}$$

denotes the proof D *extended* with $\Gamma \vdash \varphi$, that is, $\Gamma \vdash \varphi$ is not a part of D.

Much of the discussion in this chapter in based on the work of Benton, Bierman, and de Paiva in [5] and Pfenning and Davies in [26].

9.2 There and back again†

We start by defining the calculus, before deriving some fundamental properties of the system. Computational lambda calculus has the following rules of derivation

- $\dfrac{}{\Gamma, e : \varphi \vdash e : \varphi}$
- $\dfrac{}{\Gamma \vdash * : 1}$
- $\dfrac{\Gamma, x : \varphi \vdash f : \psi}{\Gamma \vdash \lambda x : \varphi . f : \varphi \to \psi}$
- $\dfrac{\Gamma \vdash f : \varphi \to \psi \qquad \Gamma \vdash e : \varphi}{\Gamma \vdash (f\ e) : \psi}$
- $\dfrac{\Gamma \vdash e : 0}{\Gamma \vdash \nabla_{\varphi}(e) : \varphi}$
- $\dfrac{\Gamma \vdash e : \varphi}{\Gamma \vdash \mathit{val}(e) : T\varphi}$
- $\dfrac{\Gamma \vdash e : T\varphi \qquad \Gamma, x : \varphi \vdash f : T\psi}{\Gamma \vdash (\text{let } x \Leftarrow e \text{ in } f) : T\psi}$

In addition to the regular axioms of simply typed lambda calculus, we have the axioms concerning the monadic computations mentioned above. We will let Λ be the set of lambda terms, $\mathcal{V}_{\Lambda}$ be the set of term variables, $\mathcal{T}$ be the set of types, and $\mathcal{V}$ be the set of type variables.

Benton, Bierman, and de Paiva proves *substitution*, *subject reduction*, *strong normalisation*, and *confluence* properties of the calculus. We will assume these properties and focus more on the interplay between the CL-logic and the lambda calculus. They also defined conjunction and disjunction in their calculus. This is just an easy extension and the interested reader can look it up in the above mentioned paper.

Example 9.2.1. *Let's see how the list monad would appear in our system. Assume $T\varphi$ is the type of a list of elements of type φ, and that we have a constant $nil : T\varphi$ and a constructor $cons : \varphi \to T\varphi \to T\varphi$. We will also need a way of joining two lists to one, $append : T\varphi \to T\varphi \to T\varphi$. With*

$$\begin{aligned} append(nil, e) &= e \\ append(cons(e, e'), e'') &= cons(e, append(e', e'')) \end{aligned}$$

we can introduce the following monadic β-reductions

$$\begin{aligned} val(e) &= cons(e, nil) \\ let\ x \Leftarrow nil\ in\ f &= nil \\ let\ x \Leftarrow cons(e, e')\ in\ f &= append(f[e/x], let\ x \Leftarrow e'\ in\ f) \end{aligned}$$

It is easy to verify that this definition of the List monad is equivalent to Haskell's.

To study the link between our type theory and logic, we will go through the Curry-Howard Correspondence and define a forgetful map $\mathcal{F}$ from expressions in the type theory, to statements in propositional CL-logic.

Definition 9.2.2. *$\mathcal{F}$ is a forgetful map, such that for any term e and any type φ we have that $\mathcal{F}(e : \varphi) = \mathcal{P}(\varphi)$ where $\mathcal{P}$ is defined recursively as*

$$\begin{aligned} \mathcal{P}(\alpha) &= \alpha,\ \alpha \in \mathcal{V} \\ \mathcal{P}(1) &= \top \\ \mathcal{P}(0) &= \bot \\ \mathcal{P}(\varphi \to \psi) &= \mathcal{P}(\varphi) \supset \mathcal{P}(\psi) \\ \mathcal{P}(T\varphi) &= \bigcirc\mathcal{P}(\varphi) \end{aligned}$$

Clearly, $\mathcal{P}$ is a bijection. The map is extended to sets of expressions, sequents and proofs in the natural way. We will also use $\mathcal{P}^{-1}$ for the obvious inverse of $\mathcal{P}$. Furthermore, we write φ' for the translated propositional statement derived from the type φ by $\mathcal{F}$ and $\mathcal{P}$, as a more compact notation when needed. We will extend this notation to sets of statements, as well as worlds.

We will let $\mathcal{P}_{\mathcal{T}} = \{\mathcal{P}(\varphi) | \varphi \in \mathcal{T}\}$.

The derivation rules resulting from applying the map to the rules from computational lambda calculus are the normal natural deduction rules for intiutionistic propositional logic, but with the following two new rules

- $\dfrac{\Gamma \vdash_{CL} \varphi}{\Gamma \vdash_{CL} \bigcirc\varphi}$

- $\dfrac{\Gamma \vdash_{CL} \bigcirc\varphi \qquad \Gamma, \varphi \vdash_{CL} \bigcirc\psi}{\Gamma \vdash_{CL} \bigcirc\psi}$

These obviously correspond to the monadic rules. Hence the monadic type T in type theory corresponds to the modal operator $\bigcirc$. We will later discuss how this operator can be interpreted in a more classical modal logic. For now, the reader can think of it as a type of possibility.

Now we have a way of going from computational lambda calculus to CL-logic. Getting back is a bit more tricky. Since we forgot something on our way to CL-logic, we now need to reconstruct what was forgotten. This can be done by the following construction map:

Definition† 9.2.3. *$C : \mathcal{D} \times \mathcal{E} \to \Lambda$ is a construction map which constructs a lambda term from the derivation of a statement in CL-logic. Here $\mathcal{D}$ is the set of CL-derivations, $\mathcal{E}$ is the set of environments on the form $\{(x_1 : \varphi_1), (x_2 : \varphi_2), \ldots, (x_n : \varphi_n)\}$ such that $\varphi_i \in \mathcal{T}$ and $x_i \in \mathcal{V}_\Lambda$ are all distinct. $\mathcal{C}$ is defined recursively on proofs as follows:*

$$\mathcal{C}\left(\frac{}{\Gamma \vdash_{CL} \top}, \sigma\right) = *$$

$$\mathcal{C}\left(\frac{}{\Gamma, \varphi \vdash_{CL} \varphi}, \sigma\right) = x \quad (s.t\ (x : \varphi) \in \sigma)$$

$$\mathcal{C}\left(\frac{\overset{D}{\Gamma, \varphi \vdash_{CL} \psi}}{\Gamma \vdash_{CL} \varphi \supset \psi}, \sigma\right) = \lambda x : \mathcal{P}^{-1}(\varphi).\mathcal{C}\left(D, \sigma \cup \{x : \mathcal{P}^{-1}(\varphi)\}\right)$$

$$\mathcal{C}\left(\frac{\overset{D}{\Gamma \vdash_{CL} \varphi \supset \psi} \qquad \overset{D'}{\Gamma \vdash_{CL} \varphi}}{\Gamma \vdash_{CL} \psi}, \sigma\right) = \left(\mathcal{C}\left(D, \sigma\right)\ \mathcal{C}\left(D', \sigma\right)\right)$$

$$\mathcal{C}\left(\frac{\overset{D}{\Gamma \vdash_{CL} \bot}}{\Gamma \vdash_{CL} \varphi}, \sigma\right) = \nabla_\varphi\left(\mathcal{C}\left(D, \sigma\right)\right)$$

$$\mathcal{C}\left(\frac{\overset{D}{\Gamma \vdash_{CL} \varphi}}{\Gamma \vdash_{CL} \bigcirc\varphi}, \sigma\right) = val\left(\mathcal{C}\left(D, \sigma\right)\right)$$

$$\mathcal{C}\left(\frac{\overset{D}{\Gamma \vdash_{CL} \bigcirc\varphi} \qquad \overset{D'}{\Gamma, \varphi \vdash_{CL} \bigcirc\psi}}{\Gamma \vdash_{CL} \bigcirc\psi}, \sigma\right) = let\ x \Leftarrow \mathcal{C}\left(D, \sigma\right)\ in\ \mathcal{C}\left(D', \sigma \cup \{(x : \varphi)\}\right)$$

where x is fresh in the third and the last line.

Notice that $\mathcal{C}$ really is indeterministic, since there might be more than one element satisfying the clause $(x : \varphi) \in \sigma$ in the second line of the definition. For instance, $\mathcal{C}$ applied to the proof

$$\dfrac{\dfrac{\varphi, \varphi \vdash \varphi}{\varphi \vdash \varphi \supset \varphi}}{\vdash \varphi \supset \varphi \supset \varphi}$$

can give either of $\lambda x : \varphi \lambda y : \varphi.x$ or $\lambda x : \varphi \lambda y : \varphi.y$. For our purposes, we will only be interested whether for a given proof and a given term, $\mathcal{C}$ applied to the proof *can* return the term, so this really is not a problem.

Example 9.2.4. *We will see how we can convert a derivation of $\varphi \supset (\psi \supset \bigcirc\varphi)$ to a lambda term. We have*

$$\dfrac{\dfrac{\dfrac{\varphi, \psi \vdash_{CL} \varphi}{\varphi, \psi \vdash_{CL} \bigcirc\varphi}}{\varphi \vdash_{CL} \psi \supset \bigcirc\varphi}}{\vdash_{CL} \varphi \supset (\psi \supset \bigcirc\varphi)}$$

Feeding this through $\mathcal{C}$ we get

$$\begin{aligned}
\mathcal{C}\left(\dfrac{\dfrac{\dfrac{\varphi, \psi \vdash_{CL} \varphi}{\varphi, \psi \vdash_{CL} \bigcirc\varphi}}{\varphi \vdash_{CL} \psi \supset \bigcirc\varphi}}{\vdash_{CL} \varphi \supset (\psi \supset \bigcirc\varphi)}, \{\}\right) &= \lambda x : \varphi.\mathcal{C}\left(\dfrac{\dfrac{\varphi, \psi \vdash_{CL} \varphi}{\varphi, \psi \vdash_{CL} \bigcirc\varphi}}{\varphi \vdash_{CL} \psi \supset \bigcirc\varphi}, \{(x : \varphi)\}\right) \\
&= \lambda x : \varphi.\lambda y : \psi.\mathcal{C}\left(\dfrac{\varphi, \vdash_{CL} \varphi}{\varphi, \psi \vdash_{CL} \bigcirc\varphi}, \{(x : \varphi), (y : \psi)\}\right) \\
&= \lambda x : \varphi.\lambda y : \psi.val\left(\mathcal{C}\left(\dfrac{}{\varphi, \psi \vdash_{CL} \varphi}, \{(x : \varphi), (y : \psi)\}\right)\right) \\
&= \lambda x : \varphi.\lambda y : \psi.val(x)
\end{aligned}$$

As expected $\lambda x : \varphi.\lambda y : \psi.val(x) \ : \ \varphi \to (\psi \to T\varphi)$.

The definition of $\mathcal{F}$ and $\mathcal{C}$ enables us to move back and forth between the two systems. This is of course only interesting if what we manage to do in one of the systems holds in the other.

Theorem† 9.2.5. *Given a lambda term e, a type φ, and an environment Γ, we have*

$$\Gamma \vdash_\lambda e : \varphi \Leftrightarrow \exists D \left(\left(\begin{array}{c} D \\ \Gamma' \vdash_{CL} \varphi' \end{array}\right) \wedge \mathcal{C}(D, \Gamma) = e\right)$$

Proof. ($\Rightarrow$) is trivial by the definitions of $\mathcal{F}$ and $\mathcal{P}$. For ($\Leftarrow$), observe that if we derive a function, $\mathcal{C}'$, mapping $\mathcal{C}$ over each depth of the proof, we can construct a proof in computational lambda calculus from a proof in CL-logic. $\mathcal{C}'$ is defined inductively, with rules on the form

$$\mathcal{C}'\left(\frac{D}{\Gamma \vdash_{CL} \varphi}, \sigma\right) = \frac{\mathcal{C}'(D, \sigma')}{\sigma \vdash_\lambda \mathcal{C}(\hat{D}, \sigma) : \mathcal{P}^{-1}(\varphi)}$$

where $\hat{D}$ is just short for $\dfrac{D}{\Gamma \vdash_{CL} \varphi}$, σ' is either just σ or σ with an element $(x : \psi)$ added if needed. If we have a bifurcation we can just map $\mathcal{C}'$ over each of them, adjusting σ as needed.

With this we can from a proof D and a specified lambda term e, construct a proof of $\Gamma \vdash e : \varphi$. □

Example 9.2.6. *We can now see what functions in the computational lambda calculus different tautologies in CL-logic gives. It is easy to see that*

$$\dfrac{\dfrac{\dfrac{\dfrac{\varphi \supset \psi \vdash_{CL} \varphi \supset \psi \qquad \varphi \vdash_{CL} \varphi}{\varphi \supset \psi, \varphi \vdash_{CL} \psi}}{\varphi \supset \psi, \varphi \vdash_{CL} \bigcirc\psi} \qquad \bigcirc\varphi \vdash_{CL} \bigcirc\varphi}{\varphi \supset \psi, \bigcirc\varphi \vdash_{CL} \bigcirc\psi}}{\varphi \supset \psi \vdash_{CL} \bigcirc\varphi \supset \bigcirc\psi}}{\vdash_{CL} (\varphi \supset \psi) \supset (\bigcirc\varphi \supset \bigcirc\psi)}$$

Furthermore, if we name the proof above D, we have

$$\mathcal{C}(D, \{\}) = \lambda f.\lambda e.(let\ x \Leftarrow e\ in\ val(fx)) : (\varphi \to \psi) \to (T\varphi \to T\psi)$$

which is a functor derived from a monad.

The above example gives us some insight into how we can interpret $\bigcirc$ in classical intuitionistic modal logic, since here $\nvdash (\varphi \supset \psi) \supset (\Diamond\varphi \supset \Diamond\psi)$. Hence it cannot be equivalent to just $\Diamond$. However, assuming $\Rightarrow$ is implication in intuitionistic modal logic, we do have $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Diamond\varphi \Rightarrow \Diamond\psi)$. If $\mathcal{I}$ is a translation from CL-logic to intuitionistic modal logic, Pfenning and Davies showed in [26] that we need both

$$\mathcal{I}(\bigcirc\varphi) = \Diamond\Box\mathcal{I}(\varphi)$$
$$\mathcal{I}(\varphi \supset \psi) = \Box\mathcal{I}(\varphi) \Rightarrow \mathcal{I}(\psi).$$

It is clear what this would mean in modal logic. Is is not so clear as to how we can think of $\Box\varphi$ in our computational lambda calculus. For the sake of our intuition, we can think of $\Box\varphi$ as a stable value, a value that is persistent and will never cease to exist. Of course, in our computational lambda calculus, all values are stable, so this is of no particular interest to us. As Pfenning and Davies pointed out, it could however be used to model computations effected by destruction of values due to memory deallocation during a computation.

So a monadic value corresponds to a proof of a possibly necessarily true statement. Therefore, a new and fruitful way of thinking of a monadic value $ma : T\varphi$ is that it possibly necessarily evaluates to something of type φ. This is quite obvious if we limit ourselves to monads like Maybe or Exception, but how can this possibility be interpreted in monads in general? In every monad we work on some type of *computation*. The key difference between a value and a computation are the intrinsic *possibilities*. A value is constant while a computation might fail, not terminate, depend on other values etc. In the non-monadic function application (`f a`), there is only one possible value which `f` will be applied to, namely `a`, which we can obtain by just inspecting it. However, this is not the case for the monadic `ma` $\gg\!=$ `f'`. If we were to inspect `ma` we still couldn't say with certainty which value would be given to `f'`.

In the next section we will look at a possible semantic for the computational lambda calculus, and hopefully this will further our intuition on the nature of monads.

9.3 Computational Kripke models†

We will now define a possible semantic via Kripke models for CL-logic, and then see how we can use this to derive a semantic for our computational lambda calculus. For this we use the CL-Kripke model from Benton, Bierman and de Paiva:

Definition 9.3.1. *A Kripke model for CL-logic is a tuple* $(W, V, \leq, R, \vDash_{CL})$*, such that* W *is a non-empty set of worlds;* $V : \mathcal{V} \to \wp(W)$ *is a mapping from propositional variables to subsets of worlds;* R *and* $\leq$ *are partial preorders on* W*;* $\vDash_{CL}$ *is a relation between worlds and formulae such that for all* $w \in W$ *we have*

- $w \vDash_{CL} p \Leftrightarrow w \in V(p),\ p \in \mathcal{V}$
- $w \vDash_{CL} \top$
- $w \vDash_{CL} \bot \Leftrightarrow \forall p \in \mathcal{V}. w \vDash_{CL} p$
- $w \vDash_{CL} \varphi \supset \psi \Leftrightarrow \forall v \geq w (v \vDash_{CL} \varphi \Rightarrow v \vDash_{CL} \psi)$
- $w \vDash_{CL} \bigcirc\varphi \Leftrightarrow \forall v \geq w \exists u (vRu \wedge u \vDash_{CL} \varphi)$.

Both R *and* $\leq$ *are required to be hereditary, that is for every* $w, v \in W$*, and* $\varphi \in \mathcal{P}_{\mathcal{T}}$ *we have*

- *if* $w \vDash_{CL} \varphi$ *and* $w \leq v$ *then* $v \vDash_{CL} \varphi$*,*
- *if* $w \vDash_{CL} \varphi$ *and* wRv *then* $v \vDash_{CL} \varphi$.

With this definition it is quite easy to see that CL-logic is a sublogic of constructive S4, but where we are restricted to only one modality, the composition of possibility and necessity. We have soundness and completeness from Benton, Bierman, and de Paiva.

Theorem 9.3.2. $\vdash_{CL} \varphi \Leftrightarrow \forall w. w \vDash_{CL} \varphi$.

Proof. By standard Henkin constructions [5]. □

Definition 9.3.3. *We will make use of the following abbreviations:*

- $w \vDash_{CL} \Gamma$ *denotes* $\forall \varphi \in \Gamma. w \vDash_{CL} \varphi$*,*
- $\Gamma \vDash_{CL} \varphi$ *denotes* $\forall w (w \vDash_{CL} \Gamma \Rightarrow w \vDash_{CL} \varphi)$.

Lemma† 9.3.4. $\Gamma, \psi \vDash_{CL} \varphi \Leftrightarrow \Gamma \vDash_{CL} \psi \supset \varphi$

Proof. ($\Rightarrow$): Assume $\forall w. w \vDash_{CL} \Gamma, \psi \Rightarrow w \vDash_{CL} \varphi$ and fix a world w. Also assume $w \vDash_{CL} \Gamma$. By the hereditary property of $\leq$, we know that for any world $v \geq w$ we have $v \vDash_{CL} \Gamma$, so if $v \vDash \psi$ then $v \vDash_{CL} \Gamma, \psi$. By our assumption we then have $v \vDash_{CL} \varphi$, and since $v \geq w$ was arbitrary, we can conclude $w \vDash_{CL} \psi \supset \varphi$.

($\Leftarrow$): Assume $\forall w. (w \vDash_{CL} \Gamma \Rightarrow w \vDash_{CL} \psi \supset \varphi)$ and fix a world w. Assume also that $w \vDash \Gamma, \psi$. Since $w \vDash_{CL} \Gamma$ we have $w \vDash_{CL} \psi \supset \varphi$, and by reflexivity of $\leq$ we then have $w \vDash \varphi$. □

Corollary† 9.3.5. $\Gamma \vdash_{CL} \varphi \Leftrightarrow \Gamma \vDash_{CL} \varphi$.

Proof. Since every deduction is finite, we can assume that $\Gamma = \{\psi_1, \psi_2, \ldots, \psi_n\}$. Let $\varphi^* \equiv \psi_1 \supset \psi_2 \supset \cdots \supset \psi_n \supset \varphi$. By the Deduction theorem we have

$$\Gamma \vdash_{CL} \varphi \Leftrightarrow \vdash_{CL} \varphi^*$$

With induction and the above lemma, we get

$$\forall w.(w \vDash_{CL} \Gamma \Rightarrow w \vDash_{CL} \varphi) \Leftrightarrow \forall w.w \vDash_{CL} \varphi^*$$

Conclude

$$\begin{aligned}\Gamma \vdash_{CL} \varphi &\Leftrightarrow \vdash_{CL} \varphi^* \\ &\Leftrightarrow \forall w.w \vDash_{CL} \varphi^* \\ &\Leftrightarrow \forall w.(w \vDash_{CL} \Gamma \Rightarrow w \vDash_{CL} \varphi) \\ &\Leftrightarrow \Gamma \vDash_{CL} \varphi.\end{aligned}$$

□

Before we can deduce a semantic for computational lambda calculus, we need to state how proofs and worlds are related. We will therefore first define an extension on the semantic above, where each proposition is ornamented with a proof of that proposition.

Definition† 9.3.6. *An ornamented Kripke model for CL-logic is a 6-tuple $(W, V, \leq, R, \vDash_{CL}, \Vdash)$, such that $(W, V, \leq, R, \vDash_{CL})$ is a regular Kripke model for CL logic, and $\Vdash$ is a relation between worlds and terms of the form $D : \varphi$, where D is a proof and φ is a regular formula, such that*

- $w \Vdash \left(\dfrac{}{\Gamma, \varphi \vdash \varphi}\right) : \varphi \Leftrightarrow w \vDash_{CL} \varphi$
- $w \Vdash \left(\dfrac{}{\Gamma \vdash \top}\right) : \top$
- $w \Vdash \left(\begin{array}{c} D \\ \Gamma \vdash \bot \end{array}\right) : \bot \Leftrightarrow \ \forall \varphi \in \mathcal{P}_{\mathcal{T}}.w \Vdash \left(\dfrac{D}{\Gamma \vdash \varphi}\right) : \varphi$
- $w \Vdash \left(\dfrac{\begin{array}{c} D \\ \Gamma, \varphi \vdash \psi \end{array}}{\Gamma \vdash \varphi \supset \psi}\right) : \varphi \supset \psi \Leftrightarrow \forall v \geq w\,(v \vDash_{CL} \varphi \Rightarrow v \Vdash D : \psi)$
- $w \Vdash \left(\dfrac{\begin{array}{cc} D & D' \\ \Gamma \vdash \psi \supset \varphi & \Gamma \vdash \psi \end{array}}{\Gamma \vdash \varphi}\right) : \varphi \Leftrightarrow w \Vdash D : \psi \supset \varphi \wedge w \Vdash D' : \psi$
- $w \Vdash \left(\dfrac{\begin{array}{c} D \\ \Gamma \vdash \varphi \end{array}}{\Gamma \vdash \bigcirc\varphi}\right) : \bigcirc\varphi \Leftrightarrow \forall v \geq w \exists u\,(vRu \wedge u \Vdash D : \varphi)$
- $w \Vdash \left(\dfrac{\begin{array}{cc} D & D' \\ \Gamma \vdash \bigcirc\varphi & \Gamma, \varphi \vdash \bigcirc\psi \end{array}}{\Gamma \vdash \bigcirc\psi}\right) : \bigcirc\psi \Leftrightarrow$
  $w \Vdash D : \bigcirc\varphi \wedge \forall v \geq w(v \vDash_{CL} \varphi \Rightarrow v \Vdash D' : \bigcirc\psi)$

and in addition

- $w \Vdash D : \varphi \wedge (w \leq v \vee wRv) \Rightarrow v \Vdash D : \varphi$
- $w \Vdash D : \varphi \Rightarrow D$ *is a proper proof, that is, every leaf of* D *is on either of the forms,* $\Gamma', \psi \vdash \psi$ *or* $\Gamma' \vdash \top$*, and every extension of a subproof matches a calculus rule.*

Definition† 9.3.7. $\Gamma \Vdash D : \varphi$ *denotes* $\forall w(w \vDash_{CL} \Gamma \Rightarrow w \Vdash D : \varphi)$.

Now we have related worlds and proofs in a clear way. Our goal is to collapse the above semantic to a semantic for typesetting of lambda terms. However, first we need to prove some properties of the ornamented semantic.

Lemma† 9.3.8. *If* $\left(\begin{array}{c} D \\ \Gamma \vdash \varphi \end{array} \right)$ *a proper proof, then* $\Gamma \Vdash D : \varphi$.

Proof. The proof is by induction on the depth of D. For D equal to any of

$$\left(\frac{}{\Gamma, \varphi \vdash \varphi}\right), \left(\frac{}{\Gamma \vdash \top}\right), \left(\frac{\begin{array}{c} D \\ \Gamma \vdash \bot \end{array}}{\Gamma \vdash \varphi}\right), \left(\frac{\begin{array}{cc} D & D' \\ \Gamma \vdash \psi \supset \varphi & \Gamma \vdash \psi \end{array}}{\Gamma \vdash \varphi}\right)$$

the result follows directly from the definition of $\Vdash$, so we will show the proof of the last three extensions.

- If D is on the form

$$\frac{\begin{array}{c} D \\ \Gamma, \varphi \vdash \psi \end{array}}{\Gamma \vdash \varphi \supset \psi}$$

  and is a proper proof, then we also have that

$$\begin{array}{c} D \\ \Gamma, \varphi \vdash \psi \end{array}$$

  is a proper proof. Then from the induction hypothesis, it follows that

$$\forall w \left(w \vDash \Gamma, \varphi \Rightarrow w \Vdash \left(\begin{array}{c} D \\ \Gamma, \varphi \vdash \psi \end{array} \right) : \psi \right).$$

  This is, by the heredity of $\leq$, equivalent to

$$\forall w \left(w \vDash \Gamma \Rightarrow \forall v \geq w \left(v \vDash \varphi \Rightarrow v \Vdash \left(\begin{array}{c} D \\ \Gamma, \varphi \vdash \psi \end{array} \right) : \psi \right) \right).$$

  which by definition is equivalent to

$$\forall w \left(w \vDash \Gamma \Rightarrow w \Vdash \left(\frac{\begin{array}{c} D \\ \Gamma \vdash \varphi \supset \psi \end{array}}{} \right) : \varphi \supset \psi \right).$$

- Or, if D is on the form

$$\frac{\begin{array}{c} D \\ \Gamma \vdash \varphi \end{array}}{\Gamma \vdash \bigcirc\varphi}$$

then by the induction hypothesis we get $\forall w\,(w \vDash \Gamma \Rightarrow w \Vdash D : \varphi)$. It follows from the hereditary property of $\leq$ and the reflexivity of R that $\forall w\,(w \vDash \Gamma \Rightarrow \forall v \geq w \exists u\,(vRu \wedge v \Vdash D : \varphi))$, which is equivalent to the desired

$$\forall w \left(w \vDash \Gamma \Rightarrow w \Vdash \left(\frac{D}{\Gamma \vdash \bigcirc\varphi} \right) : \bigcirc\varphi \right).$$

- Finally, if D is on the form

$$\frac{\begin{array}{cc} D & D' \\ \Gamma \vdash \bigcirc\varphi & \Gamma, \varphi \vdash \bigcirc\psi \end{array}}{\Gamma \vdash \bigcirc\psi},$$

then by the induction hypothesis we have both $\forall w\,(w \vDash \Gamma \Rightarrow w \Vdash D : \bigcirc\varphi)$ and $\forall w\,(w \vDash \Gamma, \varphi \Rightarrow w \Vdash D' : \bigcirc\psi)$. The last of the two implies, by the hereditary property of $\leq$, that

$$\forall w \left(w \vDash \Gamma \Rightarrow \forall v \geq w \left(v \vDash \varphi \Rightarrow v \Vdash D' : \bigcirc\psi \right) \right).$$

Hence,

$$\forall w \left(w \vDash \Gamma \Rightarrow \left(w \Vdash D : \bigcirc\varphi \wedge \forall v \geq w \left(v \vDash \varphi \Rightarrow v \Vdash D' : \bigcirc\psi \right) \right) \right),$$

which is equivalent to

$$\forall w \left(w \vDash \Gamma \Rightarrow w \Vdash \left(\frac{\begin{array}{cc} D & D' \end{array}}{\Gamma \vdash \bigcirc\psi} \right) : \bigcirc\psi \right).$$

□

The above lemma just states that if a world satisfies all assumptions of a proof, then the world also "satisfies" the proof.

Corollary† 9.3.9. *The natural deduction calculus for CL-logic is sound and complete with respect to the ornamented Kripke semantics, that is, for any proof D,*

$$\left(\begin{array}{c} D \\ \Gamma \vdash \varphi \end{array} \right) \Leftrightarrow \Gamma \Vdash D : \varphi$$

Proof. ($\Rightarrow$) follows from lemma 9.3.8. All that remains is to prove ($\Leftarrow$). We will prove the contrapositive, so assume

$$\neg \left(\begin{array}{c} D \\ \Gamma \vdash \varphi \end{array} \right)$$

in order to prove that there exists a model such that

$$\exists w \in \mathcal{W}(w \vDash \Gamma \wedge \neg w \Vdash D : \varphi).$$

Since all proofs satisfied by a world is well formed, we have that for all models $\forall w \in \mathcal{W}(\neg w \Vdash D : \varphi)$, so all that remains is to prove that there exists a model such that $\exists w.w \vDash_{CL} \Gamma$. Let Γ' be the largest consistent subset of Γ. Then, by completeness of CL-logic, there exists a model such that $\forall w \in \mathcal{W}.w \vDash_{CL} \Gamma'$. Either $\Gamma = \Gamma'$ and we are done, or $\Gamma \backslash \Gamma' = \{\varphi_i \supset \bot\}_{i \in I}$ for some $\varphi_i \in \Gamma'$. If the latter is the case, pick one world w in that model, and extend the model with one top world $w' \geq w$, where $w' \vDash_{CL} \bot$. We now have $\forall \varphi \in \Gamma'.w' \vDash_{CL} \varphi \supset \bot$, hence $w' \vDash_{CL} \Gamma$. □

Definition† 9.3.10. *A Kripke model for computational lambda calculus is a 5-tuple* $(W, V, \leq, R, \vDash_\lambda)$*, defined through an ornamented Kripke model* $(W, V, \leq, R, \vDash_{CL}, \Vdash)$ *as*

$$\Gamma \vDash_\lambda e : \varphi \Leftrightarrow \exists D \left(\mathcal{F}(\Gamma) \Vdash D : \mathcal{P}(\varphi) \wedge \mathcal{C}(D, \Gamma) = e\right)$$

In addition, we will only allow variables to inhabit one type, that is, $w \vDash_\lambda x : \varphi \wedge w \vDash_\lambda x : \psi \Rightarrow \varphi = \psi$.

Theorem† 9.3.11. $\Gamma \vdash_\lambda e : \varphi \Leftrightarrow \Gamma \vDash_\lambda e : \varphi$

Proof. We have that

$$\begin{aligned}
\Gamma \vDash_\lambda e : \varphi \quad &\Leftrightarrow \quad \exists D \left(\mathcal{F}(\Gamma) \Vdash D : \mathcal{P}(\varphi) \wedge \mathcal{C}(D, \Gamma) = e\right) \\
&\Leftrightarrow \quad \exists D \left(\left(\begin{array}{c} D \\ \mathcal{F}(\Gamma) \vdash_{CL} \mathcal{P}(\varphi) \end{array} \right) \wedge \mathcal{C}(D, \Gamma) = e \right) \\
&\Leftrightarrow \quad \Gamma \vdash_\lambda e : \varphi
\end{aligned}$$

□

Using the definition of $\Vdash$ we can derive the rules for $\vDash_\lambda$. Doing this we get the following set of rules:

- $w \vDash_\lambda x : \varphi \Leftrightarrow w \vDash_{CL} \mathcal{P}(\varphi)$
- $w \vDash_\lambda * : 1$
- $w \vDash_\lambda e : 0 \Leftrightarrow \forall \varphi \in \mathcal{T}.w \vDash_\lambda \nabla_\varphi(e) : \varphi$
- $w \vDash_\lambda (\lambda x : \varphi.e) : \varphi \to \psi \Leftrightarrow \forall v \geq w(v \vDash_\lambda x : \varphi \Rightarrow v \vDash_\lambda e : \psi)$
- $w \vDash_\lambda (f\ e) : \varphi \Leftrightarrow \exists \psi \in \mathcal{T}\,(w \vDash_\lambda f : \psi \to \varphi \wedge w \vDash_\lambda e : \psi)$
- $w \vDash_\lambda val(e) : T\varphi \Leftrightarrow \forall v \geq w \exists u(vRu \text{ and } u \vDash_\lambda e : \varphi)$
- $w \vDash_\lambda (let\ x \Leftarrow\ e\ in\ f) : T\varphi \Leftrightarrow$
  $\exists \psi \in \mathcal{T}\,(w \vDash_\lambda e : T\psi \wedge \forall v \geq w(v \vDash_\lambda x : \psi \Rightarrow w \vDash_\lambda f : T\varphi))$

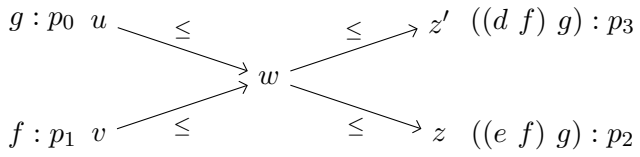
We finished the last section by using the correspondence to modal logic to build up intuition. We will now see how our semantic can help us further in this regard. Firstly, we discuss the worlds. From the fact that $w \vDash_\lambda x : \varphi \Leftrightarrow w \vDash_{CL} \varphi$, it is intuitive to think of the worlds as environments where different worlds satisfy different types. Every type inhabited in a world, is at least inhabited by variables, and from these one can build more complex terms to inhabit other types.

So worlds are environments, but what is the meaning of the relations $\leq$ and R? We know that if $w \leq w'$ we have $w \vDash_\lambda e : \varphi \Rightarrow w' \vDash_\lambda e : \varphi$ but not necessarily the other way around. So w' is an extension of w in the sense of types. These extensions allow for new, more complex types to be created in w'.

Example 9.3.12. *Take the term*

$$(\lambda x : p_0 \to p_1 \to \varphi.\ (x\ f)\ g) : (p_0 \to p_1 \to \varphi) \to \varphi$$

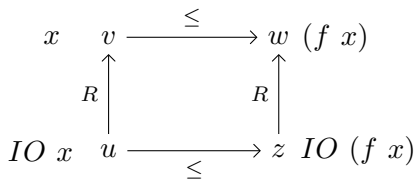
A model for this could be $(W, V, \leq, R, \vDash)$ *where* $W = \{u, v, w, z, z'\}$, $V(p_0) = \{u, w, z, z'\}$, $V(p_1) = \{v, w, z, z'\}$, $V(p_2) = \{z\}$, *and* $V(p_3) = \{z'\}$. *Then* $u \leq w$, $v \leq w$, $w \leq z$, *and* $w \leq z'$. *If* $z \vDash_\lambda e : p_0 \to p_1 \to p_2$ *and* $z' \vDash_\lambda d : p_0 \to p_1 \to p_3$, *then as a graph*



From the example we can observe that $(W, \leq)$ can be seen as a dependency graph, relating worlds with values that can directly be used to form new expressions. This can be used to model several things. For instance scope of variables where each set $\{w' | w' \geq w\}$ is the scope of elements defined in w, or to model dependencies of different definitions where a world is a module or perhaps a class. Such modelling could perhaps be the topic of future research.

The R relation is a bit different. It does not define dependencies in the same way. Say $w \vDash_\lambda e : T\varphi$, all we now know is that there should exist a world u R-related to w where there is an element e' inhabiting φ. We cannot, for certain, use this element in a non-monadic expression in w nor any $\leq$-related world, since e' need not be defined in such a world. It might however, be used in a world z if $u \leq z$. This represents two important features of monads: There is a possibility attached to the result value; the computation is performed in a separate environment. Let's look at some examples of monads interpreted with this semantic.

Example 9.3.13. *We first consider the IO monad, where computations are done in a separate environment which values cannot escape. Assume we have a value* x *received from a user and a function* f *we can apply to that value. A model satisfying this scenario can look like*



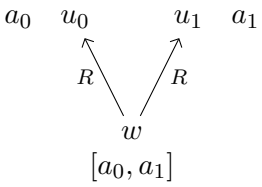
Here the worlds v and w are "inside" the IO monad, and the two worlds u and z are outside. A property of the IO monad is that there never should be, for any model involving the IO monad, a world inside the IO monad w' such that for a world outside the IO monad u', $w' \leq u'$. One can think of the worlds inside the IO monad as inhabiting the RealWorld type while all other worlds does not, hence they cannot be $\leq$-related.

We could do the same type of modelling with any `State` *related monad. If we let the type of the state value only be inhabited in certain worlds, we can then easily separate worlds manipulating or otherwise depending on the state value, and those who do not. Say we had a program P using the* `State` *monad and our state had type σ. Then, if in any model for P we had that for some type γ,*

$$\forall w \in \mathcal{W}(\exists s \in \Lambda . w \vDash_\lambda s : \sigma \Leftrightarrow \exists e \in \Lambda . w \vDash_\lambda e : \gamma)$$

we know that γ depends on σ.

Example 9.3.14. *Let's look at the* `List` *monad and what the R-relation represents here. Assume that for some worlds u_0, u_1, $u_0 \vDash_\lambda a_0 : \alpha$ and $u_1 \vDash_\lambda a_1 : \alpha$, but $u_0 \nvDash_\lambda a_1 : \alpha$ and $u_1 \nvDash_\lambda a_0 : \alpha$. This can be achieved by letting a_0 contain a variable with a type not inhabited in u_1 and likewise for a_1 and u_0. If $w \vDash_\lambda append : [\alpha] \to [\alpha] \to [\alpha]$, and wRu_0 and wRu_1, we have that $w \vDash_\lambda [a_0, a_1]$. Now the R-relation represents a choice of values in the indeterministic computation $[a_0, a_1]$.*



As we can see, it is possible for the results of an indeterministic computation to live in different worlds. Making a choice of one result value implies a choice of one of the possible worlds.

As we can see, the R-relations between the worlds represents the properties of a monad. This might be a useful tool for studying the differences and similarities of different monads.

We could extend our modal logic with multiple modalities, $\{\bigcirc_i\}$, where every T_i would represent a monad. We could then study relations and mappings between monads, which would make a framework for studying monad transformers and other monad morphisms.

This concludes our discussion on monads and the Curry-Howard Correspondence, and we will now explore other constructs related to the monad.

Chapter 10

Other Related Constructs

We know that in Haskell there are a myriad of type classes, each with different properties and applications. In this chapter we will look at some of those type classes, namely those which are closely related to monads.

10.1 Applicative functors

We saw in the beginning of this thesis that every monad can express a functor, that is, by defining a monad one indirectly also defines a functor. In this section we will see that a monad indirectly defines an even more powerful concept, the *applicative* functor. This type class extends the functionality of the `Functor` type class. The `Applicative` type class has the following definition.

```
class Functor f ⇒ Applicative f where
  pure  :: α → f α
  (<*>) :: f (α → β) → f α → f β
```

such that

```
pure id <*> a                = a
pure f <*> pure a            = pure (f a)
a <*> pure b                 = pure ($ b) <*> a
fmap f a                     = pure f <*> a
pure (.) <*> a <*> b <*> c = a <*> (b <*> c)
```

An applicative functor can lift functions into and apply functions inside its type. For example,

```
instance Applicative [ ] where
  pure a          = [a]
  [ ] <*> _       = [ ]
  (f : fs) <*> xs = (fmap f xs) ++ (fs <*> xs)
```

is the `Applicative` instance for lists. How is this related to monads? Well, since `pure` is just supposed to inject a value into the type, it is equivalent to the monadic `return`. Furthermore, we have that $(<*>) = \texttt{ap}$, where

```
ap       :: Monad m ⇒ m (α → β) → m α → m β
ap mf ma = do
             f ← mf
             a ← ma
             return $ f a
```

which is the application of a function from inside the monad, that we defined earlier in this thesis. It is easy to verify that the laws above are satisfied with these definitions of `pure` and (<∗>). So all monads are in fact applicative functors[7].

Another interesting question is then whether all applicative functors are monads. If we write

```
(<∗>′) :: f α → f (α → β) → f β
(<∗>′) = flip (<∗>)
```

we can see that the types of (<∗>′) and (⩾=) are not so different. However, if we try to apply a function of the form `f` :: $\alpha \rightarrow$ `f` β to an element in the applicative functor, we get something of type `f f` β, and not something of type `f` β. Applicative functors lacks the ability to flatten such a type, that is, we cannot from the three functions

```
fmap  :: (α → β) → f α → f β
pure  :: α → f α
(<∗>) :: f (α → β) → f α → f β
```

create a function `join` :: `f f` $\alpha \rightarrow$ `f` α. This is what makes a monad more powerful than an applicative functor, it can flatten layers of computations down to a single layer.

For examples and a deeper discussion on applicative functors, see Yorgey's "Typeclassopedia"[32].

10.2 Arrows

In this subsection we will look at another construct which the monad implicitly defines, namely the arrow. The arrow was first introduced by John Hughes in [15] as a construct solving many of the same problems as the monad. Its core type class is defined as

```
class Arrow a where
  arr  :: (β → γ) → a β γ
  (⋙) :: a β γ → a γ δ → a β δ
```

So while the monadic type only takes one argument, an arrow takes two. An element `a` $\beta\ \gamma$ is thought of as a computation taking input of type β and returns a result of type γ. The first function in the above definition, `arr`, then creates an arrow from a function, while the second function, (⋙), is composition of arrows.

An arrow implementation should satisfy the following laws:

i) $(\texttt{f} \ggg \texttt{g}) \ggg \texttt{h} = \texttt{f} \ggg (\texttt{g} \ggg \texttt{h})$

ii) $\texttt{arr}\,(\texttt{f} \ggg \texttt{g}) = \texttt{arr}\,\texttt{f} \ggg \texttt{arr}\,\texttt{g}$

iii) $\texttt{arr}\,\texttt{id} \ggg \texttt{f} = \texttt{f} = \texttt{f} \ggg \texttt{arr}\,\texttt{id}$

That is, $(\ggg)$ is a monoid with `arr id` as identity, and `arr` distributes over $(\ggg)$.

Notice the similarity of this definition to that of the monad: Where `return` makes a computation from a value, `arr` makes an input dependent computation from a function; and where $(\gg\!=)$ chains computations, $(\ggg)$ chains input dependent computations.

In fact, for every monad `m` we can make an arrow by just making the input explicit. That is, Hughes shows that we can define

$$\textbf{newtype}\ \texttt{Kleisli}\,\texttt{m}\,\alpha\,\beta = \texttt{K}\,(\alpha \to \texttt{m}\,\beta)$$

$$\begin{aligned}&\textbf{instance}\ \texttt{Monad}\,\texttt{m} \Rightarrow \texttt{Arrow}\,(\texttt{Kleisli}\,\texttt{m})\ \textbf{where}\\ &\quad \texttt{arr}\,\texttt{f} \quad\ = \texttt{K}\,(\lambda\texttt{b} \to \texttt{return}\,\$\,\texttt{f}\,\texttt{b})\\ &\quad \texttt{K}\,\texttt{f} \ggg \texttt{K}\,\texttt{g} = \texttt{K}\,(\lambda\texttt{b} \to \texttt{f}\,\texttt{b} \gg\!= \texttt{g})\end{aligned}$$

to make any monad an arrow. So simply put $\texttt{arr}\,\texttt{f} = \texttt{return}\,.\,\texttt{f}$ and $(\ggg) = (\circledcirc)$.

An arrow instance defines functionality for making and composing arrows, but currently we have no way of combining output from two arrows. For instance, as Hughes explains in his article, we cannot make a simple `add` function analogous to the monadic

$$\begin{aligned}&\texttt{add} \quad\ :: \texttt{Monad}\,\texttt{m} \Rightarrow \texttt{m}\,\texttt{Int} \to \texttt{m}\,\texttt{Int} \to \texttt{m}\,\texttt{Int}\\ &\texttt{add}\,\texttt{mx}\,\texttt{my} = \texttt{mx} \gg\!= \lambda\texttt{x} \to \texttt{my} \gg\!= \to \lambda\texttt{y} \to \texttt{return}\,(\texttt{x} + \texttt{y})\end{aligned}$$

To enable this, Hughes extended his class to include the function

$$\texttt{first} :: \texttt{a}\,\beta\,\gamma \to \texttt{a}\,(\beta,\,\delta)\,(\gamma,\,\delta)$$

converting a computation from β to γ to a computation over a tuple applying the argument arrow to the first element in the tuple. The other element in the tuple is untouched. This might seem like a small extension, but it allows us to express much more, for instance function lifting, which solves our problem above. Defining lifting from the three functions `arr`, $(\ggg)$, and `first` is quite involved and several intermediate functions are needed. These intermediate functions are rather useful so we will write out their definitions.

After defining `first`, it would be natural to have a `second`, and we can define this as

$$\begin{aligned}&\texttt{second} \quad :: \texttt{Arrow}\,\texttt{a} \Rightarrow \texttt{a}\,\beta\,\gamma \to \texttt{a}\,(\delta,\beta)\,(\delta,\,\gamma)\\ &\texttt{second}\,\texttt{f} = \texttt{arr}\,\texttt{swap} \ggg \texttt{first}\,\texttt{f} \ggg \texttt{arr}\,\texttt{swap}\\ &\qquad\qquad \textbf{where}\ \texttt{swap}\,(\texttt{x},\texttt{y}) = (\texttt{y},\texttt{x})\end{aligned}$$

We can then define

```
(***)    :: Arrow a ⇒ a β γ → a δ ϵ → a (β, δ) (γ, ϵ)
f *** g = first f >>> second g
```

which applies `f` to the first element and `g` to the second element of the input pair. The last function needed to define lifting is the ability to combine the result of two arrows into a pair.

```
(&&&)  :: Arrow a ⇒ a β γ → a β δ → a β (γ, δ)
f &&& g = arr (λ b → (b, b)) >>> (f *** g)
```

Now we can define lifting of a binary function over two arrows by first combining the results of the two arrows in a tuple, and then apply the arrow obtained from the operator to the two elements of the tuple. So

```
liftA2 :: Arrow a ⇒ (γ → δ → ϵ) → a β γ → a β δ → a β ϵ
liftA2 op f g = (f &&& g) >>> arr (λ (b, c) → op b c)
```

Finally we can define

```
add :: Arrow a ⇒ a β Int → a β Int → a β Int
add = liftA2 (+)
```

Let's write that out in terms of our three core functions to see what really happens:

```
add f g =arr (λ b → (b, b)) >>>
         first f >>>
         arr swap >>>
         first g >>>
         arr swap >>>
         arr (λ (b, c) → b + c)
```

To sum up, first duplicate the input to a tuple, apply `f` to the first element, swap places, apply `g` to the (currently) first element, swap place again, then apply (+) to the two results. As we can see, arrows are well suited for manipulating one part of the computation at the time, and combining the parts again.

What does arrows lack that monads have, that is, which function or type class does an arrow need to implement to become as expressive as a monad? Hughes shows in his article that the following type class is all it takes:

```
class Arrow a ⇒ ArrowApply a where
  app :: a (a β γ, β) γ
```

That is, we need to have an arrow which takes as input an arrow and an input value to that arrow, applies the arrow to the value, and returns the result. He then shows that for any `ArrowApply a` we have

```
newtype ArrowApply a ⇒ ArrowMonad a b = M { unM :: (a Void b)}

instance Monad (ArrowMonad a) where
```

```
return a  = M $ arr (λz → a)
M m >>= f = M (m >>> arr (λx → (unM $ f x, ⊥)) >>> app)
```

where $\bot$ :: `Void`, which eliminates the dependency on the input type. Notice that `return a` is a computation that ignores its input, and so is the case for all elements in any `ArrowMonad a`. This is forced by the fact that $\bot$ cannot be used for anything, and is also the only element inhabiting `Void`. In the definition of ($\gg=$) we can then safely apply the resulting arrow from (`f x`) on $\bot$ to unwrap the value inside, again resulting in an input-independent computation.

This concludes our discussion on arrows. Non-monadic examples of arrows are quite involved, but can be looked up in Hughes article cited above.

10.3 Comonads

In category theory it is often the case that a structure has a co-structure. A co-structure is a structure where all the mapping are reversed, so for every mapping in the original structure `F : A → B`, one would have a mapping `F' : B → A`. The monad has such a structure, namely the comonad. The `Comonad` is defined in Haskell through the following type class,

```
class Functor w ⇒ Comonad w where
  extract   :: w α → α
  duplicate :: w α → w w α
```

such that

```
extract . duplicate      = id
fmap extract . duplicate = id
duplicate . duplicate    = fmap duplicate . duplicate
```

where `extract` and `duplicate` are the dual to `return` and `join` respectively. Alternatively, we could define it with

```
class Comonad w where
  extract :: w α → α
  extend  :: (w α → β) → w α → w β
```

where `extend` is the (flipped and prefixed) dual to ($\gg=$). This definition should satisfy

```
extend extract    = id
extract . extend f = f
extend f . extend g = extend (f . extend g)
```

In the Haskell libraries there is a function which is the direct dual to ($\gg=$),

```
(=>>)  :: Comonad w ⇒ w α → (w α → β) → w β
```

```
ma ⋙ f = extend f ma
```

Now the laws look a bit more familiar, albeit a small difference in the last.

```
ma ⋙ extract   = ma
extract ma ⋙ f = f ma
ma ⋙ f ⋙ g     = ma ⋙ λm → f (m ⋙ g)
```

One can go from the first definition to the next by setting

```
extend f = (fmap f) . duplicate
```

and from the second to the first by

```
fmap f    = extend (f . extract)
duplicate = extend id
```

Comonads are discussed in detail by Uustalu and Vene in [28]. They think of `wa :: w α` as a value depending on a context, namely `w`.

Some examples of comonads are given below.

Example 10.3.1. *The list monad does not have a proper comonad, since then* `extract` *would only be a partial function, as it has no return value if applied to the empty list. However, we could define the data type for non-empty lists as*

```
data NEList α = [α] | (α : (NEList α))
```

Now we can define

```
instance Comonad [ ] where
  extract           = head
  duplicate [a]     = [[a]]
  duplicate (a : as) = (a : as) : (duplicate as)
```

So `duplicate` *gathers its argument and all tails of that argument in a list, such that* `duplicate` $[1,2,3,4] = [[1,2,3,4],[2,3,4],[3,4],[4]]$. *Furthermore, we have* $[1,2,3,4] \ggg$ `sum` $= [10,\ 9,\ 7,\ 4]$.

The comonad above might be useful for simulating computations on the effect of ordered removal of elements from a list. However, the example below is perhaps closer to a canonical usage of comonads. Here we clearer see the context dependent computations as described by Uustalu and Vene. This example is inspired by Dan Piponi's article[1] on comonads and cellular automata.

Example 10.3.2. *In this example we will look at a comonad for pointers in a stream. A pointer is here modelled by one list containing all elements to the left (or before) the element pointed to, the actual element pointed to, and a list of all elements to the right of (or after) the element pointed to.*

```
data Pointer α = P [α] α [α]
```

Now we can move the pointer left and right with

```
moveLeft                 :: Pointer α → Pointer α
moveLeft (P (y : ys) a xs) = P ys y (a : xs)
moveLeft _               = error "Cannot move any more left."

moveRight                :: Pointer α → Pointer α
moveRight (P ys a (x : xs)) = P (a : ys) x xs
moveRight _              = error "Cannot move any more right."
```

We will assume that the streams are infinite, so there really is no need for the wild card in the two definitions. Before we can define the comonad instance, we need the following two help functions.

```
pointAllLeft   :: Pointer α → [Pointer α]
pointAllLeft p  = tail $ iterate moveLeft p

pointAllRight  :: Pointer α → [Pointer α]
pointAllRight p = tail $ iterate moveRight p
```

These two functions just make a list of all possible left or right pointers from the given pointer. So, for instance,

```
pointAllLeft (P [2, 1..] 3 [4, 5..]) = [(P [1..] 2 [3, 4, 5..]), (P [..] 1 [2, 3, 4, 5..])..]
```

Now we can define our comonad instance.

```
instance Comonad where
  extract (P ys a xs) = a
  duplicate p         = P (pointAllLeft p)
                          p
                          (pointAllRight p)
```

The `extract` *function just returns the value pointed to, while* `duplicate` *makes a pointer to a pointer in a stream of pointers. If we write* `wa ⋙ f`*, then* `f` *is applied to* `wa` *and all possible pointers achievable by moving left or right from* `wa`*.*

Notice that what we have made is quite similar to a Turing machine. If we set

```
read :: Pointer α → α
read = extract
```

and

```
write              :: α → Pointer α → Pointer α
write b (P ys a xs) = P ys b xs
```

we can view a pointer as a state with the head currently over the element pointed to. We can then move the head left and right on the tape, and also read and write to the tape. With this we can write rules as for a regular

Turing machine. However, (⋙) allows us to apply a rule uniformly to every state achievable by only movement. This is quite useful for simulating cellular automatons, as Dan Piponi writes on his blog[1]. A cellular automata determines the value in a cell on rules based on its neighbouring cell's values, and every cell's value is updated every step. This is where the uniformly mapping of rules is useful. A popular example of a cellular automata is Conway's Game of Life[10].

Let's make a small example of such an automata which models growth of some sort of a life form. We will let every cell have one of two values, either `True` *or* `False` *depending on whether that cell is inhabited or not, respectively. We will have the following rules of transitions from one state to the next:*

- *If a cell is inhabited and has two inhabited neighbouring cells, it dies as a result of overpopulation.*
- *If a cell is uninhabited and has at least one neighbouring inhabited cell, the cell becomes inhabited as a result of growth.*

Notice that for any pointer (`P (y : ys) a (x : xs)`)*, the above rules can be reduced to the following:*

```
rule                       :: Pointer Bool → Bool
rule (P (y : ys) a (x : xs)) = if a then
                                 not (x && y)
                               else
                                 x || y
```

Let's also define some help functions, transforming a pointer into a finite list with the element pointed to as the mid element.

```
toFiniteList              :: Pointer Bool → Int → [Char]
toFiniteList (P ys a xs) n = (reverse $ take n ys) ++ [a] ++ (take n xs)
```

Furthermore, we will display `True` *as* `'x'` *and* `False` *as* `'_'` *for visual purposes. Now we have*

```
p :: Pointer Bool
p = P (True : (repeat False)) False (True : (repeat False))

toFiniteList p 10 = "________x_x________"

toFiniteList (p ⋙ rule) 10 = "_______xxxxx_______"
```

If we print each iteration of `rule` *over* `p`*, that is* `p ⋙ rule ⋙ rule ⋙ ...`*, and use white space instead of underscore, we get the nice fractal seen in figure 10.1. If we change the values on the start tape, we get a completely different pattern.*

The above example clearly shows how comonads can be viewed as a context. Comonads also have transformers which combines contexts in the same way monad transformers combine effects. A discussion of comonad transformers along with more examples can be seen in the article by Uustalu and Vene[28].

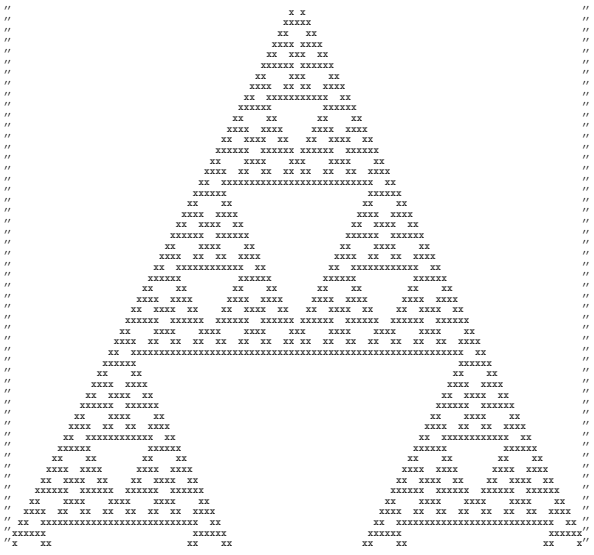


Figure 10.1: Resulting pattern from iterating rule over p.

10.4 Continuations

A continuation is something that contains a representation of the future computation. Programming which explicitly handles continuations is called *continuation passing style* (CPS) and such programs are able to control and manipulate this future computation. Continuations can be represented in Haskell as programming with elements of type $(\alpha \to \rho) \to \rho$, such that the argument to this function represents the future computation. Within such functions one has a name for the future computation, and can continue this any where in the body of the function. This allows for great control of the computation's flow, and enables features such as escape of the computation, backtracking, and much more. Let's set

$$\textbf{newtype}\ \texttt{Cont}\ \rho\ \alpha\ =\ \texttt{Cont}\ \{\ \texttt{runCont}\ ::\ (\alpha\ \to\ \rho)\ \to\ \rho\}$$

The first immediate connection between continuations and monads, is that for every continuation with result type ρ, there is a monad:

```
instance Monad (Cont ρ) where
  return a = Cont $ λ k → k a
  ma >>= f = Cont $ λ k → runCont ma (λ a → runCont (f a) k)
```

The `Cont` monad combines computations in CPS, `return` constructs the trivial continuation which applies the future computation to a value, and `ma >>= f` abstracts over the future computation, applies `f` to the results of `ma`, unwraps this result and feeds it the future computation `k`. We will now study continuations in more detail before we examine this connection any further.

A very powerful concept in conjunction with continuations is something called "call-with-current-continuation", often represented as a function with name `callCC`. This concept, as a function, originated in the Scheme language[3], but was invented as an escape operator in 1965 by Peter Landin[18]. It takes a function as argument and applies this to the current continuation. Whenever this current continuation is called, the computation jumps right to it, and continues from there. Hence, `callCC` can, amongst other things, simulate forward goto-statements.

In Haskell, `callCC` can be defined as a regular function:

```
callCC :: ((α → Cont ρ β) → Cont ρ α) → Cont ρ α
callCC f = Cont $ λk → runCont (f (λa → λ_ → k a)) k
```

It allows us to bring the future continuation `k` into scope explicitly. The difference between `Cont` and `callCC` is best expressed by looking at the differences in their types.

```
Cont   :: ((α → ρ)          → ρ)          → Cont ρ α
callCC :: ((α → Cont ρ β) → Cont ρ α) → Cont ρ α
```

In `callCC` we have the ability to manipulate, sequence, and construct entire continuations from values given to the argument function, while `Cont` is only able to manipulate the immediate future computation. For instance with `callCC` we can construct nested `goto`-statements, as

```
contEx :: Int → Cont ρ Int
contEx n =
  callCC (λlabel0 → do
     a1 ← callCC (λlabel1 → do
        a2 ← do
           when (n ≥ 10) $ label0 n
           when (n ≥ 5) $ label1 n
           return n
        return $ a2 + 5)
     return $ a1 + 5)
```

Here, every $\texttt{label}_\texttt{N}$ is a continuation representing a label we can jump to. If we call a label, we jump up to the line it was defined and continue from there. In the above function, our goal is to return something greater or equal to 10 for any positive argument integer `n`. If `n` is less than 5, we continue the computation as usual, add 5 twice and returns that result. If `n` is greater or equal to 5, but less than 10, we only have to add 5 once, so we jump to $\texttt{label}_1$ and skip the first addition. We now add 5 and returns. If `n` already is greater or equal to 10, we can just escape the entire computation, and jump right back to start.

As we can see from the example above, continuations give great control over computations flow and can also simulate different types of computations. As mentioned in the first chapter for example, there is a model of IO using continuations. We have already seen that continuations can be represented by monads, but can every monad be represented by

continuations, that is, are monads and continuations equally expressive? It turns out that the answer is negative. There are several papers on this subject, so let's look at some of the differences between monads and continuations.

Simon Peyton Jones and Phillip Wadler shows in "Imperative functional programming"[25] that monadic IO is strictly more powerful than IO based on continuations. Furthermore, Wadler shows in "The essence of functional programming"[29] that continuations always provides an escape possibility through `callCC`, while monads can choose whether to include such a feature or not.

However, Wadler compares monads to an extension of continuations with new functions in "Monads and composable continuations"[30]. Here he extends the continuation language with the two functions `shift` and `reset`, which was invented by Danvy and Filinsky[8]. These functions increases the control over the computation flow even further, and the resulting continuation-based language is too general to be embraced as a monad.

Filinsky proves in "Representing monads"[9] that if a language can represent composable continuation and a single state value, that language can express any monadic computation, such as indeterminism, exception handling, etc. This clearly states the expressive powers of continuations, and begs the question whether the continuation monad in conjunction with the state monad can express any other monad. We will not dig any further into this, but it might be an interesting topic for future research.

From the discussion above, we can conclude that continuations are able to simulate many computational features and are closely related to monads, but the two constructs are not equivalent. The interested reader can consult the papers mentioned above for a deeper study on the relationship between the two concepts.

This concludes our discussion on constructs similar (or otherwise related) to monads, and we will now proceed to summarise and conclude this thesis.

Part IV

Conclusion And Appendix

Chapter 11

Conclusion

Throughout this thesis we have studied different aspects of monads, and in some cases developed some new theory. In this section I will summarise my experience working with monads, followed by a section about my contributions and proposals for future work.

11.1 My views on monadic programming

In this section we will summarise my experiences and views on monads and monadic programming.

We have seen that monads are not a tool for solving one particular set of problems, but rather a general framework for expressing solutions in a small but expressive language. Monadic programming has the ability to define how computations should behave in a uniform way. This frees the programmer from writing a lot of previously needed code, such as if-tests for checking void values or explicit exception handling. Predefining how computations behave also makes the code cleaner and easier to read. To check the correctness of an implementation, much of the work can be put into just checking the correctness of the monad.

In many of our applications, we saw how monads allows for great abstractions, simulating different computations and effects. Monadic code can easily be extended with new functionality using monad transformers and lifting. This makes core functionality easy to maintain and develop without even thinking about compatibility issues with previously written code.

Using `do`-notation and effectful monads, one can write imperative-like code without the need for destructive functions. This enables an imperative style for writing algorithms more intuitively expressed in an imperative manner.

On the other hand, I think heavy usage of `do`-notation in some cases might lead to unexpected results, due to the great level of abstraction from the original monadic functions. There is also the danger of assuming some imperative features of commands in a `do`-block, since the appearance is so similar to imperative code. The same caution is raised in "Real World Haskell"[7], where the authors suggest one should go "sugar-free" until one becomes an intermediate user of monads. Explicitly writing ($\gg=$)

reminds the programmer what the code really means, but also enables the programmer to manipulate monadic code to a greater extent.

We have seen many positive features of monadic programming. However, the monad is perhaps a more difficult concept to grasp than other programming constructs, such as objects in object oriented languages or actors in the actor model. After one builds up some intuition on the usage of monads, I think that continued use should come without strain and appear quite natural. Apart from demanding a bit more time to learn, I see no reason why monadic programming shouldn't become a widely used programming paradigm.

11.2 My contributions and suggested future work

Apart from serving as yet another tutorial on monads, this thesis explores some new theory and applications. In this section, we will summarise my contributions along with proposals for future work.

The first of my contributions in this thesis was to define a new monad, the `Pair` monad. A monad, which is a direct generalisation of the `List` monad, we saw was well suited for representing lambda calculi with different properties. It might be interesting to look at other uses of this monad, for instance with one of the interpretations presented in that chapter. Another possibility for future work on this topic could be to investigate how its transformer can be used in conjunction with other monads.

The applications I wrote in chapter 5 and 6 were mainly for illustrating monadic programming. However, the monads used are quite convenient for each purpose. For writing an interpreter, using the combination of `Writer` and `State` is quite natural. The same goes for cryptographic libraries and `Supply`, `Writer` and `State`. If one wants to write such applications, these monads and the functionality I defined around them might be a good start.

In chapter 7, I generalised Grust's work on monadic comprehension languages for queries to all natural monads. The definition and results were general enough to contain Hinze's backtracking transformer applied to any monad. This made the resulting query language quite easy to integrate, as we can let the result of a query be wrapped in any monad. As a proposal for future work, it might be interesting to generalise all the optimisations from Grust's article to any natural monad. One could also try to generalise further, and see if one could remove some of the axioms of naturality, for instance the criteria of natural form, without losing the abilities needed for forming a basis for query languages. It would also be nice to see whether one is able to derive my generalisations for all catamorphisms, not just folds.

The last of my contributions in this thesis was concerning the Curry-Howard Correspondence. Here I made the correspondence explicit with a forgetful mapping and a construction mapping. After defining how worlds and proofs were related, I derived a Kripke semantic for the computational lambda calculus. I then used the maps to prove completeness of the calculus with respect to this new semantic. We saw that the `R`-relation relating worlds inside a monad with outside worlds, described the features of each monad.

It would be very interesting to see if we can get even more information from this `R`-relation of the semantic for different monads. As stated in that same chapter, it would also be quite interesting to see how multiple modalities could represent different monads and monad transformers.

This concludes both the discussion of my contributions and future work, and this thesis.

Chapter 12

Appendix

12.1 Code

12.1.1 PairMonad

The PairMonad module

```
module PairMonad where

import Control.Monad
import Control.Monad.Trans

data Pair a = Nil | Cons a (Pair a) | Comp (Pair a) (Pair a)
     deriving (Eq, Show)

appendPair :: Pair a -> Pair a -> Pair a
appendPair Nil x = x
appendPair (Cons a b) c = Cons a (appendPair b c)
appendPair (Comp a b) c = Comp (appendPair a c) b

concatPair :: Pair (Pair a) -> Pair a
concatPair Nil = Nil
concatPair (Cons p1 p2) = appendPair p1 (concatPair p2)
concatPair (Comp p1 p2) = Comp (concatPair p1) (concatPair p2)

instance Functor Pair where
    fmap f Nil = Nil
    fmap f (Cons a p) = Cons (f a) (fmap f p)
    fmap f (Comp p1 p2) = Comp (fmap f p1) (fmap f p2)

instance Monad Pair where
    return a = Cons a Nil
    p >>= f = concatPair (fmap f p)

instance MonadPlus Pair where
    mzero = Nil
    mplus = appendPair

subs :: Eq a => a -> Pair a -> a -> Pair a
subs x p a = if a == x
             then p
             else return a

zero :: Int -> Pair a
zero 0 = Nil
zero n = Comp (zero (n-1)) Nil

newtype PairT m a = PairT { runPairT :: m (Pair a) }
```

```
sequenceP :: (Monad m) => Pair (m a) -> m (Pair a)
sequenceP Nil = return Nil
sequenceP (Cons ma mps) = do
                            a <- ma
                            ps <- sequenceP mps
                            return (Cons a ps)
sequenceP (Comp mps mps') = do
                              ps <- sequenceP mps
                              ps' <- sequenceP mps'
                              return (Comp ps ps')

instance Monad m => Monad (PairT m) where
    return = PairT . return . return
    mpa >>= f = PairT $ do
                          pa <- runPairT mpa
                          ppa <- sequenceP $ liftM (runPairT . f) pa
                          return $ concatPair ppa

instance MonadTrans PairT where
  lift ma = PairT $ do a <- ma
                       return (Cons a Nil)

instance Ord a => Ord (Pair a) where
  compare p p' = case p of
    Nil        -> case p' of
      Nil -> EQ
      _   -> LT
    Cons a q   -> case p' of
      Nil        -> GT
      Cons a' q' -> let c = compare a a'
                    in
                     if (c == EQ) then compare q q'
                     else c
      _          -> LT
    Comp p1 p2 -> case p' of
      Comp p1' p2' -> let c = compare p1 p1'
                      in
                       if c == EQ then compare p2 p2'
                       else c
      _            -> GT
```

```
module PairInterp where

import PairMonad
import Control.Monad

----------Pair as a structured indeterministic computation--------
pairAsStruct :: Pair a -> [a]
pairAsStruct Nil = []
pairAsStruct (Cons a p) = (a:(pairAsStruct p))
pairAsStruct (Comp p1 p2) = (pairAsStruct p1) ++ (pairAsStruct p2)

----------Pair as a Cartesian product-----------------------------
pairAsCart :: Pair a -> [[a]]
pairAsCart Nil = []
pairAsCart (Cons a p) = ([a]:(pairAsCart p))
pairAsCart (Comp p p') = ap (fmap (++) (pairAsCart p)) (pairAsCart p')

----------Pair as a choice of path from root to leaf--------------
pairAsPathCh :: Pair a -> [[a]]
pairAsPathCh Nil = [[]]
pairAsPathCh (Cons a p) = fmap (a:) (pairAsPathCh p)
pairAsPathCh (Comp p p') = (pairAsPathCh p) ++ (pairAsPathCh p')

----------Pair as lambda expressions------------------------------
beta :: Eq a => Pair a -> Pair a
beta ps = case ps of
  (Comp (Cons a p) p')  -> if p == Nil
                           then ps
                           else p >>= subs a p'
  (Comp (Comp p1 p2) p) -> Comp (beta $ Comp p1 p2) p
  _                     -> ps

reduce :: Eq a => Pair a -> Pair a
reduce ps = case findFix beta ps of
  Nil        -> Nil
  Cons a p   -> Cons a (reduce p)
  Comp p1 p2 -> Comp p1 (reduce p2)

findFix :: Eq a => (a -> a) -> a -> a
findFix f a = fixer a (f a)
  where fixer a1 a2 = if a1 == a2
                      then a1
                      else fixer a2 (f a2)

----------Pattern matching lambda terms---------------------------

--Same as beta, but with two primitive functions:
-- i)  (Comp Nil (Comp p1 p2)) -> p1
-- ii) (Comp (Comp Nil Nil) (Comp p1 p2)) -> p2
betaWP :: Eq a => Pair a -> Pair a
betaWP ps = case ps of
  Comp Nil (Comp p1 _)            -> p1
  Comp (Comp Nil Nil) (Comp _ p2) -> p2
  Comp (Cons a p) p'              -> if p == Nil
                                     then ps
                                     else p >>= subs a p'
  Comp (Comp p1 p2) p             -> Comp (betaWP $ Comp p1 p2) p
  _                               -> ps

reduceWP :: Eq a => Pair a -> Pair a
reduceWP ps = let ps' = findFix betaWP ps
              in
               case ps' of
                 Nil        -> Nil
                 Cons a p   -> Cons a (reduceWP p)
```

```
                  Comp p1 p2 -> Comp p1 (reduceWP p2)


evPattern :: Pair String -> Pair String
evPattern p@(Comp (Cons f Nil) (Cons x Nil)) =
  Cons (show p) (Comp (Cons f (Comp (Cons x Nil)
                                    (Comp (Comp Nil Nil)
                                     (Cons (show p) Nil))))
                  (Comp Nil (Cons (show p) Nil)))

evPattern p@(Comp f (Cons xs Nil)) =
  Cons (show p) (Comp (Cons xs (Comp (evPattern f)
                                    (Comp Nil (Cons (show p) Nil))))
                  (Comp (Comp Nil Nil)
                   (Cons (show p) Nil)))

evPattern p = p

reducePatterns :: Pair (Pair String) -> Pair String
reducePatterns p = p >>= evPattern
```

12.1.2 Graph reduction

Graph reduction

```
module GraphReduction where

import PairMonad
import Control.Monad.Trans.Writer
import Control.Monad.Trans.State
import qualified Data.Map as Map
import Control.Monad.Trans

data Term a = Red (Pair a) | UnRed (Pair a)
type MyMap a = Map.Map (Pair a) (Term a)
type STG a = WriterT String (State (MyMap a))

untilFix :: (Monad m, Ord a) => (a -> m a) -> a -> m a
untilFix f a = fixer a (f a)
  where fixer b mb = do
          b' <- mb
          if b == b'
            then return b
            else fixer b' (f b')

insert :: Ord a => Pair a -> Term a -> STG a ()
insert k a = lift $ modify (Map.insert k a)

member :: Ord a => Pair a -> STG a Bool
member k = do s <- lift get
              return $ Map.member k s

retrieve :: Ord a => Pair a -> STG a (Term a)
retrieve k = do s <- lift get
                return $ s Map.! k

delete :: Ord a => Pair a -> STG a ()
delete p = lift $ modify (Map.delete p)

share :: (Ord a, Show a) => Pair a -> STG a ()
share p = do
  b <- member p
  if b
    then tell $ "Used shared value <" ++ (show p) ++ ">.\n"
    else do insert p (UnRed p)
            tell $ "Shared <" ++ (show p) ++ ">.\n"

evaluator :: (Ord a, Show a) => Pair a -> STG a (Pair a)
evaluator p = do
  trm <- retrieve p
  case trm of
    Red p'   -> do tell $ "Used reduced value <"
                     ++ (show p') ++ ">.\n"
                   return p'
    UnRed p' -> do delete p'
                   p'' <- reduceSTG p'
                   insert p (Red p'')
                   tell $ "Reduced <" ++ (show p')
                     ++ "> to <" ++ (show p'') ++ ">.\n"
                   return p''

reduceTop :: (Ord a, Show a) => Pair a -> STG a (Pair a)
reduceTop ps = case ps of
  Comp (Cons a p) p' -> if p == Nil
                        then return ps
                        else do share p'
                                return $ p >>= subs a p'
  Comp (Comp p1 p2) p -> do p' <- reduceTop (Comp p1 p2)
```

```
                              return $ Comp p' p
  _                        -> return ps

reduceSTG :: (Ord a, Show a) => Pair a -> STG a (Pair a)
reduceSTG p = do
  b <- member p
  p' <- if b then evaluator p else untilFix reduceTop p
  case p' of
    Nil         -> return p'
    Cons a ps   -> do ps' <- reduceSTG ps
                      return $ Cons a ps'
    Comp p1 p2 -> do p1' <- reduceSTG p1
                     p2' <- reduceSTG p2
                     return $ Comp p1' p2'

unwrapSTG :: (Ord a, Show a) => STG a (Pair a) -> (Pair a, String)
unwrapSTG stg = let wtr        = runWriterT stg
                    ((p,s),_) = runState wtr Map.empty
                in
                 (p,s)

reduce :: (Ord a, Show a) => Pair a -> (Pair a, String)
reduce = unwrapSTG . reduceSTG
```

12.1.3 Cryptographic framework

Cryptographic framework without state

```
module CryptoNoState where

import Control.Monad.Trans.State
import Control.Monad.Random
import System.Random
import Control.Monad.Trans
import Data.Bits
import Control.Monad
import Control.Monad.Supply
import Control.Monad.Writer

type Encrypted k a = SupplyT k (Writer [k]) a

shareKey :: k -> Encrypted k ()
shareKey k = lift $ tell [k]

shareKeys :: [k] -> Encrypted k ()
shareKeys ks = lift $ tell ks

encryptAll :: (a -> Encrypted k a) -> [a] -> Encrypted k [a]
encryptAll enc ps = sequence (map enc ps)

evalEncrypted :: (Encrypted k [a]) -> [k] -> ([a],[k])
evalEncrypted encr ks = let res = evalSupplyT encr ks
                        in
                         runWriter res

generateSupply :: (Random k, RandomGen g) => Rand g [k]
generateSupply = liftM2 (:) getRandom generateSupply

bitSupply :: IO [Int]
bitSupply = do gen <- getStdGen
               let s' = generateSupply
                   s = evalRand s' gen
               return $ map (\x -> mod x 2) s
```

Cryptographic framework with state

```
module Crypto where

import Control.Monad.Trans.State
import Control.Monad.Random
import System.Random
import Control.Monad.Trans
import Data.Bits
import Control.Monad
import Control.Monad.Supply
import Control.Monad.Writer

type Encrypted k a = StateT [k] (SupplyT k (Writer [k])) a

push :: k -> Encrypted k ()
push k = modify (k:)

peek :: Encrypted k k
peek = do s <- get
          return $ head s

pop :: Encrypted k k
pop = do k <- peek
         modify tail
         return k

shareKey :: k -> Encrypted k ()
shareKey k = lift $ lift $ tell [k]

shareKeys :: [k] -> Encrypted k ()
shareKeys ks = lift $ lift $ tell ks

encryptAll :: (a -> Encrypted k a) -> [a] -> Encrypted k [a]
encryptAll enc ps = sequence (map enc ps)

evalEncrypted :: (Encrypted k [a]) -> [k] -> ([a],[k])
evalEncrypted encr ks = let rsltSt = evalStateT encr []
                            rsltSu = evalSupplyT rsltSt ks
                        in runWriter rsltSu

generateSupply :: (Random k, RandomGen g) => Rand g [k]
generateSupply = liftM2 (:) getRandom generateSupply

bitSupply :: IO [Int]
bitSupply = do gen <- getStdGen
               let s' = generateSupply
                   s = evalRand s' gen
               return $ map (\x -> mod x 2) s
```

Example of usage of the cryptographic framework

```
import Crypto
import Data.Bits
import System.Random
import Control.Monad.Supply
import Control.Monad
import Control.Monad.Trans
import Control.Monad.Writer
import Control.Monad.Trans.State

--One time pad
encOTP :: Int -> Encrypted Int Int
encOTP a = do (k:_) <- lift $ supplies 1
              shareKey k
              return $ xor a k

oneTimePad :: [Int] -> [Int] -> ([Int], [Int])
oneTimePad ps ks = encrypt enc ps ks

-------------------------------------------------------------------

--Dummy functions
perm :: [a] -> [a]
perm = id
subs :: [a] -> [a]
subs = id

-------------------------------------------------------------------

--Round function for SPNs
rndFunc :: [Int] -> Int -> Encrypted Int [Int]
rndFunc p n = do k <- supplies (length p)
                 shareKeys k
                 let c'' = zipWith xor p k
                     c'  = subs c''
                     c   = if (n > 1) then perm c' else c'
                 rndFunc c (n-1)

--enc for SPNs
encSPN :: [Int] -> Encrypted Int [Int]
encSPN p = rndFunc p 16

spn :: [Int] -> [Int] -> ([Int], [Int])
spn ps ks = encrypt encSPN ps ks

-------------------------------------------------------------------

--CBC mode for SPNs
encCBC :: [Int] -> Encrypted Int [Int]
encCBC p = do c' <- pop
              c  <- rndFunc (xor p c') 14
              push c
              return c
```

Bibliography

[1] `http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html`, December 2006.

[2] `http://www.haskell.org/haskellwiki/Monadplus`, June 2013.

[3] Harold Abelson, RK Dybvig, CT Haynes, GJ Rozas, NI Adams IV, DP Friedman, E Kohlbecker, GL Steele Jr, DH Bartley, R Halstead, et al. Revised report on the algorithmic language scheme. *ACM SIGPLAN Lisp Pointers*, 4(3):1–55, 1991.

[4] Steve Awodey. *Category Theory (Oxford Logic Guides)*. Oxford University Press, USA, 2 edition, August 2010.

[5] P.N. Benton, G.M. Bierman, and V.C.V. de Paiva. Computational types from a logical perspective i. 1995.

[6] Patrick Blackburn, Johan Van Benthem, and Frank Wolter. *Handbook of Modal Logic*. Studies in logic and practical reasoning - ISSN 1570-2464 ; 3. Elsevier, 2007.

[7] John Goerzen Bryan O'Sullivan and Don Stewart. *Real World Haskell*. O'Reilly, 2009.

[8] Olivier Danvy and Andrzej Filinski. *A functional abstraction of typed contexts*. Datalogisk Institut, 1989.

[9] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM.

[10] Martin Gardner. Mathematical Games The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[11] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[12] Torsten Grust. Monad Comprehensions: A Versatile Representation for Queries.

[13] Ralf Hinze. Prological features in a functional setting axioms and implementations. In *In Third Fuji Int. Symp. on Functional and Logic Programming*, pages 98–122, 1998.

[14] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992.

[15] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67 – 111, 2000.

[16] Mark P. Jones and Luc Duponcheel. Composing monads. Technical report, 1993.

[17] Simon P. Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.

[18] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, February 1965.

[19] Saunders Mac Lane. *Categories for the working mathematician.* Springer-Verlag, New York :, 1971.

[20] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[21] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Acm Sigplan Notices*, volume 43, pages 423–438. ACM, 2008.

[22] D. Orchard and A. Mycroft. A notation for comonads. 2012. In Post-Proceedings of IFL'12.

[23] Tomas Petricek and Don Syme. Syntax matters: Writing abstract computations in f#. *Pre-proceedings of TFP (Trends in Functional Programming), St. Andrews, Scotland*, 2012.

[24] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[25] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[26] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. In *Mathematical Structures in Computer Science*, page 2001, 1999.

[27] Douglas Stinson. *Cryptography: Theory and Practice,Second Edition.* CRC/C&H, 2nd edition, 2002.

[28] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263 – 284, 2008. ¡ce:title¿Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008)¡/ce:title¿.

[29] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[30] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55, 1994.

[31] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[32] Brent Yorgey. Typeclassopedia. *The Monad.Reader*, (13):17–68, March 2009.