Imperial College London

BEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Linearity as an Effect

Author:
Hashan Punchihewa

Supervisor:
Dr. Nicolas Wu

Second Marker:
Dr. Anthony Field

15th June 2020

Submitted in partial fulfillment of the requirements for the BEng Computing of Imperial College London

Abstract

A novel encoding of linearity is presented in Haskell, using parameterised algebraic effects and parameterised algebraic effect handlers. The framework built allows the treatment of data as resources, such that the type of a computation encodes information about how the computation manipulates resources. This allows constraints to be statically enforced on resources, that are not captured by conventional type systems. The encoding is written using the type-level programming extensions of GHC, and a type-checker plugin is provided that aids the compiler in verifying the constraints.

The encoding is used to implement two case studies. The first is an implementation of session types to ensure that communication through channels is type-safe. The second is safe concurrency, by verifying that the use of mutable arrays won't cause data races. The expressiveness of our system is evaluated, considering alternative ways that our encoding could be implemented, as well as alternative encodings in the literature.

Acknowledgements

I would like to thank my supervisor Dr. Nicolas Wu for his support and guidance over the course of the project.

I would like to thank my personal tutor Dr. Ben Glocker for his support and encouragement throughout my degree.

I would like to thank my family for their support and belief in me throughout my life.

Contents

Chapter 1

Introduction

1.1 Motivation

Milner (1978) coined the slogan 'well typed programs do not go wrong'. The idea is that type systems provide static guarantees on the correctness of computer programs. While type systems found in programming languages today do prevent an important class of errors, certain errors will not be caught and cause programs to malfunction at runtime. Consider the program below, which successfully compiles in Haskell. It opens a file and attempts to write to the file. There are already two errors: the programmer opens the file with only the permission to read; and the programmer fails to close the file handle.

```
main :: IO ()
main = do
  handle ← openFile "thesis.tex" ReadMode
  hPutStr handle "Fin"
```

The fact that the type system is unable to catch these errors, reflects the fact that programming languages treat data as abstract entities, which can be used without limitation. While this view of data may be appropriate for certain data, data can also be used to represent resources such as file handles, communication channels and peripherals. These resources are subject to constraints on how they can be used. The idea of constraints fits data that might not be considered as a conventional resource, for example, sensitive user information. An organisation might want to have static guarantees that private information cannot be leaked outside of a system.

The view of data as a resource is called linearity. This reflects its origins in linear logic (Girard, 1987), where logical propositions are treated as resources that must be used once and exactly once. It has a counterpart in the linear λ-calculus, where arguments to functions must be used once and exactly once (Benton et al., 1993). The constraints that this project seeks to implement are broader than the linear λ-calculus, however. We seek to build a system, that can control not only how much a resource can be 'consumed', but that can also constrain the operations that can be applied to a resource in a given state.

The goal of this project is to provide an encoding of these constraints in Haskell, using the notion of computational effects. Common examples of computational effects are input/output, state, exceptions and nondeterminism. For instance, reading a file or writing to a variable are effects. A key difference between imperative languages and functional languages such as Haskell is the treatment of effects. A misconception is that Haskell forbids effects. This is not true; it is possible to interact with the file system and to have mutable state in Haskell. The difference is that in Haskell, effects are used through structures such as monads and algebras.

In fact in Haskell it is enough to look at the type of a function to understand what effects it performs. The outcome of this project is to do the same thing with linearity, i.e. for the state of resources in the computation to be encoded in the type of a computation. For instances, it should be possible to see whether there are any open file handles, and if so whether they can be read from or written to from the type. To combine the notions of computational effects and linearity, parameterised effects are used. Parameterised effects extend the notion of effects by parameterising computation by two states, one reflecting the state of resources before the computation and one reflecting the state of resources after the computation.

A major difference between this piece of work and others in the literature, is that other attempts at linearity focus on building new programming languages tailor made for linearity (Orchard et al., 2019; Matsakis and Klock, 2014). By contrast, Haskell is a well-established programming language. It is used actively in industry, and there is a rich ecosystem of libraries and tooling available. Haskell has an expressive type system, reflected in its treatment of computational effects mentioned earlier. In addition, the de facto implementation of the language, the Glasgow Haskell Compiler (GHC) supports advanced type-level computation, through a number of language extensions.

Therefore, there is a clear motivation in bringing linearity to Haskell. Lin-

earity allows for more powerful type systems, with stronger guarantees on correctness, leading to more robust programs. Parameterised effects allow us to unify computational effects with linearity. The type-level infrastructure that already exists in Haskell, provides the setting for an idiomatic encoding. Furthermore, the prevalence of Haskell among functional programmers attests to the utility of such an encoding.

1.2 Contributions

The objective of this project is to provide an encoding of linearity in the GHC implementation of Haskell. The goal is for an expressive encoding of linearity, that is natural for programmers to use and extend.

After we review the background on computational effects, linearity and type-level programming that underlies the work done during the project (Chapter 2), we make the following contributions:

- We build a framework consisting of type-level utilities which can be used to encode linearity (Chapter 3). This starts with the development of parameterised algebraic effects in Haskell. Then contexts are introduced to allow effects to interact with multiple resources. Scoped operations are introduced to allow more complex effects to be modelled. Finally, type inference is improved by introducing a GHC type-checker plugin.

- We demonstrate the utility of the framework through an encoding of session types (Chapter 4). We start with background knowledge on session types, and its relation to linear types. We then show how this can be encoded easily in our framework. We then use this encoding to evaluate the framework.

- We encode mutable arrays in our framework to provide another example of how linearity can be encoded in our framework (Chapter 5). We start with a discussion of mutable arrays in a multithreaded environment, and how this can be modelled in our framework. Then a basic encoding is provided, that prevents unsafe access to arrays by concurrent threads. The example of quicksort is considered to highlight shortcomings in the initial solution, and several enhancements are given that allow the

sorting algorithm to be encoded. Finally we evaluate the framework in the context of mutable arrays.

After this, we conclude with a summary of the work presented in this report (Chapter 6). We give an overall discussion of the strengths and weaknesses of our framework. We also review other examples of linearity in the literature, and how these encodings compare to our approach. We then end with some research directions that could form the basis of future work.

Chapter 2

Background

This chapter sets out the background to the implementation of linearity, that the rest of the report concentrates on. First a brief introduction to category theory is given through the lens of Haskell. The introduction works towards the mathematical structure of a monad, which is used to provide the semantics for computational effects, but also to structure effectful programs in functional languages.

After introducing category theory, an introduction to algebraic effects is presented. This is an alternative view of computational effects to monads, with roots in universal algebra. It is then shown how algebraic effects can be formulated in a category-theoretic light, and how this formulation can be used provide an encoding in Haskell.

After an overview of the two different approaches to computational effects is given, a generalisation of effects, called parameterised effects, is introduced. Parameterised effects give operations two type-level indices corresponding to the state of the effect before and after the operation. We first consider parameterised effects in the context of monads, before moving onto parameterised effects in the context of algebras.

Having discussed effects, we then turn our attention to linearity. We give a brief explanation of linearity in the context of the linear λ-calculus, before considering how linearity can be used to encode more generic constraints. We consider file handles and network sockets as examples.

The final part of this chapter looks at some of the key tools in the type-level armamentarium provided by Haskell, that will be used to build the encoding of linearity. We discuss the distinction between values, types and kinds that underlie the Haskell type system, as well as three extensions that

greatly enhance the power of the Haskell type system.

2.1 Category Theory

This section reviews the category-theoretic background to computational effects; a standard reference for the subject is presented by MacLane (1971). Category theory abstracts mathematical entities as objects and mathematical structure as relationships between entities, called morphisms.

First the technical definitions of categories, objects and morphisms are given. Then more structure is introduced in the form of functors and monads. This is accompanied with their corresponding implementations in Haskell. This leads to an explanation of how monads can be used to encode computational effects. Then the issue of composing effects is raised, with monad transformers being introduced as an attempt to solve this problem.

A category is a collection of objects and a collection of morphisms. Each morphism is assigned an object called the domain and another object called the codomain. A morphism $f : A \to B$ denotes a morphism f with domain A and codomain B. For any two morphisms $f : A \to B$ and $g : B \to C$, there is a composition of the two morphisms $f; g : A \to C$. Further, for every object A there is an identity morphism id_A, which acts as the unit in composition.

A simple example of a category is the category of sets. Here the objects are sets and the morphisms are functions. Composition of morphisms is function composition, and the identity morphism is the identity function. Another example of a category is the category of groups. Here the objects are groups and the morphisms are group homomorphisms. The composition of morphisms and identity morphism in this category is defined similarly to the category of sets.

A special object in a category is the initial object. An object is initial, if there exists exactly one morphism from it to every object. In the category of sets, the empty set is the initial object. In the category of groups, the group with only the identity element is the initial object.

A mapping between categories, is called a functor. A functor F, between categories C and D is denoted as $F : C \to D$. An object A in C is mapped to $F(A)$ in D, and a morphism f is mapped to a morphism $F(f)$ in D. A functor must preserve the structure of a category. In particular, it must preserve the domain and codomain of morphisms, the identity morphism

and the composition of morphisms. A functor $F : C \to C$ is called an endofunctor.

One example of a functor is the forgetful functor $Grp \to Set$. This forgets the group structure in Grp and maps every group to its underlying set. An example of an endofunctor is the powerset functor $Set \to Set$, that maps every set to its powerset. Another endofunctor for any category C is the identity functor, which maps every object and morphism to itself.

In Haskell, it is convenient to imagine one category called *Hask*, whose objects are types and whose morphisms are functions. The notion of an endofunctor is provided by the *Functor* typeclass.

```
type Functor :: (Type → Type) → Constraint
class Functor f where
  fmap f :: (a → b) → f a → f b
```

A functor in Haskell consists of a unary type constructor, which maps the objects (i.e. types) and a function *fmap* that maps the morphisms (i.e. functions). A functor in Haskell must also obey certain conditions, called the functor laws. These are given below, and correspond to coherence conditions already mentioned that preserve the structure of a functor in category theory. The first law states that *fmap* preserves the identity function, and the second law states that *fmap* respects the composition of functions.

```
fmap id       ≡ id
fmap (f ∘ g) ≡ fmap f ∘ fmap g
```

A natural transformation between two functors $F : C \to D$ and $G : C \to D$, denoted as $\alpha : F \to G$ is a family of morphisms in D, $\alpha_X : FX \to GX$ for each X in C. There are certain coherence conditions that a natural transformation is subject to. An example in Haskell of a natural transformation is given below. It maps from the *Maybe* functor to the *List* functor, by mapping *Nothing* to the empty list, and *Just x* to the singleton list consisting solely of x.

```
transform :: Maybe a → [a]
transform Nothing = []
transform (Just x) = [x]
```

A monad is an endofunctor $T : C \to C$, equipped with two natural transformations, called unit $\eta : \mathrm{id}_C \to T$ and multiplication $\mu : T^2 \to T$.

Again there are certain additional coherence conditions. In Haskell, a monad is defined by the following typeclass.

$$
\begin{aligned}
&\textbf{type } \mathit{Monad} :: (\mathit{Type} \to \mathit{Type}) \to \mathit{Constraint} \\
&\textbf{class } (\mathit{Functor}\ m) \Rightarrow \mathit{Monad}\ m \textbf{ where} \\
&\quad (\gg\!=) \ :: m\ a \to (a \to m\ b) \to m\ b \\
&\quad \mathit{return} :: a \to m\ a \\
&\quad \mathit{join} \quad :: m\ (m\ a) \to m\ a
\end{aligned}
$$

The *return* operation corresponds to the unit natural transformation. The *join* operation corresponds to the multiplication natural transformation. Note that $\gg\!=$ can be defined in terms of *join* and vice versa.

$$
\begin{aligned}
&\mathit{join}\ m \ = m \gg\!= \mathit{id} \\
&m \gg\!= f = \mathit{join}\ (\mathit{fmap}\ f\ m)
\end{aligned}
$$

An adjunction is a relationship between two functors. There is an adjunction between a functor $F : D \to C$ and $G : C \to D$, if there are two natural transformations: $\epsilon : FG \to \mathrm{id}_C$ and $\eta : \mathrm{id}_D \to GF$. F is called the left adjoint and G is called the right adjoint. The forgetful functor from groups to sets, mentioned earlier, has a left adjoint functor from sets to groups called the free functor. This functor constructs a free group for each set. The free group has a carrier set which is all the syntactic expressions that can be made from the group operations and the members of the original set, i.e. the free functor represents the construction of an abstract syntax tree. The group operations are defined as simply syntactic operations on the abstract syntax tree.

Another example of an adjunction in this style is between the category of monads for a category C and the category of endofunctors for a category C. The forgetful functor simply maps each monad to its underlying endofunctor. If this functor has a left adjoint, then it is the free functor. For each endofunctor F, the free functor produces, the free monad of F. For the category *Hask*, the free functor does exist i.e. a free monad can be constructed from every endofunctor. The free functor is given below. It takes an endofunctor f and constructs a monad *Free* f.

$$
\begin{aligned}
&\textbf{type } \mathit{Free} :: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
&\textbf{data } \mathit{Free}\ f\ a = \mathit{Pure}\ a \mid \mathit{Impure}\ (f\ (\mathit{Free}\ f)) \\
&\textbf{instance } \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (\mathit{Free}\ f) \textbf{ where}
\end{aligned}
$$

```
  fmap :: (a → b) → Free f a → Free f b
  fmap f (Pure a)    = Pure (f a)
  fmap f (Impure b) = Impure (fmap f b)
instance Functor f ⇒ Monad (Free f) where
  return :: a → Free f a
  return = Pure
  join :: Free f (Free f a) → Free f a
  join (Pure a)    = a
  join (Impure b) = Impure (fmap join b)
```

The term computational effect is used to describe a myriad of phenomena such as state, input/output and non-determinism. Moggi (1989) found that these effects could be given categorical semantics as strong monads on a Cartesian-closed category. Wadler (1993) introduced the idea of using monads to structure programs in functional languages. In Haskell, this is done with do-notation, which allows the monadic operations to be expressed conveniently. The *incr* function is given below as example involving the *State* effect. It takes an argument n which is the value to increment the state by.

```
incr :: Int → State Int
incr n = do
  val ← get
  put (val + n)
```

It is natural for programs to interact with multiple effects, and thus a way of composing computational effects is desired. However monads do not compose arbitrarily, which leads to the notion of a monad transformer, which can embed any monad in a particular monad. This is given by the following typeclass.

```
type MonadTrans :: (Type → Type) → Constraint
class MonadTrans t where
  lift :: (Monad m) ⇒ m a → t m a
```

The *lift* operation allows the embedded monad's operations to be 'lifted' into the new monad. An example is given below where the *Reader* and the *IO* monads are combined, with the *ReaderT* transformer, which corresponds to the *Reader* monad. Since *ReaderT* is the monad transformer being used, the *ask* operation is used as normal, whereas the *open* operation needs to

be embedded inside an invocation of *lift*, since *IO* is below *ReaderT* in the monad transformer stack.

```
readPrivateKey :: (ReaderT String IO) String
readPrivateKey = do
  path ← ask
  lift (open path)
```

The *readPrivateKey* function obtains a path from the *ReaderT* transformer, and reads the file at the path using the *IO* monad. The problem with the code above is that the function depends on a particular monad transformer stack. Ideally, one would wish to write code that works with any monad transformer stack provided certain effects are present. This can be done through the final tagless approach (Kiselyov, 2010), where typeclasses constrain the monad transformer stack to ensure certain effects are present.

```
readPrivateKey' :: (MonadReader Config m, MonadIO m) ⇒ m String
readPrivateKey' = do
  path ← ask
  liftIO (open path)
```

A version of *readPrivateKey* that uses this approach is given above. While this technique improves the generality of the code, this does not fix the fundamental shortcomings of monad transformers. In recent years, an alternative approach to structuring effectful programs has been developed. This approach gives a more concrete separation between the syntax and semantics of effects, and a cleaner approach to the composition of effects.

So far the vocabulary of category theory has been introduced to describe how monads can be used to structure programs in Haskell. The problem of composition of effects is then described. The canonical solution of the monad transformer is explained, as well as its shortcomings. This motivates the introduction of algebraic effects as an alternative way of structuring effectful programs.

2.2 Algebraic Effects

2.2.1 Theoretical Background

In this section, the theory of algebraic effects is reviewed. This starts with its roots in universal algebra (Burris and Sankappanavar, 1981). This is

built upon by showing how algebras can be used to encode computational effects (Plotkin and Power, 2004). The notion of algebraic effect handlers is presented, to provide the semantics of algebras in a compositional way (Plotkin and Pretnar, 2013). Universal algebra can be understood from the perspective of category theory (Hyland and Power, 2007), and monads are used to help encode algebraic effects in Haskell. Finally the techniques for implementing the composition of effects in Haskell are presented (Kiselyov et al., 2013).

As opposed to treating the monad arising from a computational effect as primitive, algebraic effects treat the operations that give rise to the effect as primitive. In particular, it seeks to capture these effects as an algebraic theory, in the same way that a monoid or a group can be captured as an algebraic theory.

In universal algebra, a signature $\langle \sigma, ar \rangle$ is a set of function symbols σ each with a non-negative arity, given by the function $ar : \sigma \to \mathbb{N}$. Given a set of variables X and a signature $\langle \sigma, ar \rangle$, a set of terms T can be built up inductively:

- Each variable $x \in X$ is a term.
- If $\mathsf{op} \in \sigma$ is an operation with $ar(\mathsf{op}) = n$ and $t_1, \ldots, t_n$ are terms, then $\mathsf{op}(t_1, \ldots, t_n)$ is a term.

An equation is made from two terms t_1 and t_2, and is denoted as $t_1 = t_2$. An algebraic theory is defined as a signature coupled with a set of equations. An example of an algebraic theory is the theory of monoids. The signature consists of the nullary function symbol e, the identity element, and the binary function symbol $\star$. The equations defining it are given as follows:

$$a \star (b \star c) = (a \star b) \star c$$
$$a \star e = a$$
$$e \star a = a$$

In the same way that the notion of a monoid can be captured algebraically, so can the computational effect of non-determinism, with a signature consisting of one binary operation or (i.e. non-deterministic choice).

$$\begin{aligned}\mathsf{or}(a, a) &= a \\ \mathsf{or}(a, b) &= \mathsf{or}(b, a) \\ \mathsf{or}(\mathsf{or}(a, b), c) &= \mathsf{or}(a, \mathsf{or}(b \vee c))\end{aligned}$$

An interpretation for a signature consists of a carrier set S and an interpretation for each n-ary function symbol f as a function $[\![f]\!] : S^n \to S$. Given an assignment of variables into the carrier set, terms can be evaluated. A model for a signature is an interpretation for theory, such that the equations of the theory hold for every possible variable assignment. An example of a model for the theory of monoids is the natural numbers with 0 as the identity element and + as the binary operation.

Note that each algebraic theory has a model called the free model. This is constructed by taking the set of terms (i.e. the syntax) and constructing the quotient set under the equations of the theory. Then the operations can be interpreted as functions that build terms. An example of a free model is the free group, mentioned earlier.

Now a generalisation of algebras will be presented. A signature is now defined as $\langle \sigma, par, ar \rangle$, where σ is still a set of operations symbols, par is an indexed family of sets for each operation symbol, and ar is an indexed family of sets for each operation symbol. Let us consider what has changed. Firstly, operations are parameterised by a set par_{op} for each op. Secondly, rather than taking a natural number as an arity, an operation takes a set ar_{op} as an arity. An operation op with the set P for a parameter, and A for arity is denoted as $\mathsf{op} : P \rightsquigarrow A$.

Let us see how these changes affect the set of terms. Unlike before, the set of terms consists of trees, rather than strings. The set of terms T for a set of variables X and a signature $\langle \sigma, par, ar \rangle$ is defined as follows:

- For each variable $x \in X$, then $\mathsf{return}\ x$ is a leaf.
- For each operation $\mathsf{op} : P \rightsquigarrow A$, where $p \in P$ and $f : A \to T$, then there is a tree with a root labelled with p. In addition there are a (possibly infinite) number of subtrees. For $a \in A$, there is a subtree $f(a)$.

The paths of the tree from the root to the leaves can be thought of as all the possible (terminating) computations. The function given to an operation

can be seen as a continuation. The notions of interpretations and equations can be extended to this new setting. Here is an example of the theory of state given below. The signature consists of two operations $\mathsf{get} : 1 \rightsquigarrow S$ and $\mathsf{put} : S \rightsquigarrow 1$, with 1 being a singleton set, consisting of only the unit element and S being the set of values that can be stored.

$$\begin{aligned}
\mathsf{get}((), \lambda s.\ \mathsf{get}((), \lambda t.\ \kappa\ s\ t)) &= \mathsf{get}((), \lambda s.\ \kappa\ s\ s) \\
\mathsf{get}((), \lambda s.\ \mathsf{put}((), \kappa)) &= \kappa\ () \\
\mathsf{put}(s, \lambda t.\ \mathsf{get}((), \kappa)) &= \mathsf{put}(s, \lambda t.\ \kappa s) \\
\mathsf{put}(s, \lambda t.\ \mathsf{put}(t, \kappa)) &= \mathsf{put}(t, \kappa)
\end{aligned}$$

2.2.2 Algebras in Haskell

We shall consider a category-theoretic formulation of algebras, called the F-algebra for an endofunctor $F : C \to C$. An F-algebra $\langle A, a \rangle$ is an object A in C and a morphism $a : FA \to A$. Intuitively, the set A represents the carrier set, the endofunctor F encodes the operations that form the algebra and the morphism a evaluates the operations. To formalise this in Haskell, a typeclass called *Algebra* is used.

```
type Algebra :: (Type → Type) → Type → Constraint
class (Functor f) ⇒ Algebra f a where
  eval :: f a → a
```

An instance declaration of *Algebra f a* states that the carrier set *a* along with the evaluation function *eval* forms an algebra for the endofunctor *f*. Let us consider a concrete example in Haskell. The endofunctor representing the algebra's operation is encoded below as *Mon*, with the *Functor* instance so that it is an endofunctor.

```
data Mon a = Sum a a | Unit
instance Functor Mon where
  fmap :: (a → b) → F a → F b
  fmap f (Sum x y) = Sum (f x) (f y)
  fmap f Unit      = Unit
```

There are two data constructors *Sum* and *Unit* corresponding to the monoid operations. An example of an algebra for this endofunctor would be

the carrier set *Int* with the evaluation function given below. This evaluation arrow assigns *Unit* the value 0, and assigns *Sum a b* the summation of a and b.

```
instance Algebra Mon Int where
  eval :: Mon Int → Int
  eval Unit       = 0
  eval (Sum a b) = a + b
```

The collection of algebras for an endofunctor F form the category of F-algebras. The initial algebra in this category is special, it represents the syntax formed from the endofunctor. It is analogous to the notion of free models and free groups, mentioned earlier. It is also related to the notion of free monads, which can be used to construct the initial algebra. The initial algebra for an endofunctor can be defined as the fixed point of the endofunctor.

```
type Fix :: (Type → Type) → Type
newtype Fix f = Fix (f (Fix f))
instance Functor f ⇒ Algebra f (Fix f) where
  eval :: f (Fix f) → Fix f
  eval = Fix
```

Let us consider what *Fix Mon* would look like. It would consist of *Unit*, *Sum Unit Unit*, *Sum Unit* (*Sum Unit Unit*), etc. This is a slight abuse of notation, since the *Fix* data constructor has been omitted for readability. The semantics can now be given through a morphism from the initial algebra to another algebra. This is called a catamorphism, and is given below by the *cata* function. This will recursively evaluate the tree, for a given algebra.

```
cata :: (Algebra f a) ⇒ Fix f → a
cata (Fix f) = eval (fmap cata f)
```

A problem is that the only base case is the *Unit* operation given by the endofunctor. It would be desirable to enrich our endofunctor with a collection of base values. For instance, one might want a theory of monoids with integers, or alternatively a theory of monoids with strings. This is done with the *Base* functor. For a type *Base f b a*, there are two data constructors: *Pure* lifts a value of type b into the data type, while *Impure* lifts a value of type *f a* into the data type.

```
data Base f b a = Pure b | Impure (f a)
instance Functor f ⇒ Functor (Base f) where
  fmap :: (a → b) → Base f a → Base f b
  fmap f (Pure x)   = Pure x
  fmap f (Impure x) = Impure (f x)
```

To obtain the syntax for the theory of monoids with integers, the composition of *Fix* and *Base* can be used.

```
type Syntax = Fix (Base Mon Int)
```

It turns out, however, that this composition has already been seen before. The composition of *Fix* and *Base* is simply *Free* i.e. the free monad.

```
data Free f a = Pure a | Impure (f (Free f))
type Syntax' = Free Mon Int
```

Now an updated version of *cata* can be provided, that works with the free monad. This takes an additional parameter *gen*, called the generator, that maps base values to the carrier set.

```
cata' :: (Algebra f a) ⇒ (b → a) → Free f b → a
cata' gen (Pure b)   = gen b
cata' gen (Impure a) = eval (fmap (cata' gen) a)
```

To define the semantics for *Free Mon Int*, the function *cata id* can be used, since there's already an *Algebra* instance for *Mon* and *Int*. This construction is powerful, and can be used as a basis for algebraic effects. Consider the state algebra, this can be given as:

```
data State s a = Get (s → a) | Put s a
```

So in order to insert an effect into an algebra, you would have to use the constructor for the effect, e.g. *Get* and then wrap it with *Impure*. To make this easy, smart constructors are used. Not only do they wrap the effect with *Impure*, they also put *return* in the continuation.

```
get :: Free (State s) s
get = Impure (Get return)
```

$put :: s \to Free\ (State\ s)\ ()$
$put\ v = Impure\ (Put\ v\ (return\ ()))$

An example of code using the state algebra is the *incr* function, which retrieves the state value, and then increments it by a given amount a. This highlights the natural way that code using algebraic effects can be written with do-notation, thanks to the underlying monad instance, as well as the use of smart constructors.

$incr :: Int \to Free\ (State\ Int)\ ()$
$incr\ a = \mathbf{do}$
  $s \leftarrow get$
  $put\ (s + a)$

It should be noted that the result of *incr* is a syntactic structure, with no semantics attached. In the same way as with monoids, a catamorphism can be used to provide the semantics by interpreting the computation tree. This highlights one of the main advantages of algebraic effects over monads, which is that there is a clean separation between syntax and semantics.

We can give the semantics for $Free\ (State\ s)\ a$ as functions of the type $s \to a$. This is done by the *cataState* function. It interprets base values a, as constant functions, that ignore the state value passed in. The algebra instance then shows how to interpret $State\ s\ (s \to a)$ into $s \to a$.

instance $Algebra\ (State\ s)\ (s \to a)$ **where**
  $eval\ (Get\ c) = \lambda s.\ c\ s\ s$
  $eval\ (Put\ s\ c) = \lambda\_.\ c\ s$
$cataState = cata'\ (\lambda x.\ \lambda\_.\ x)$

2.2.3 Composition of Effects

In the same way that monad transformers are used to combine different effects, it is desirable to combine effects in the algebraic effects paradigm. The change to allow this to happen on the level of syntax is the introduction of the coproduct between two endofunctors. This is done in the style introduced in Swierstra (2008).

type $(\oplus) :: (Type \to Type) \to (Type \to Type) \to (Type \to Type)$
data $(f \oplus g)\ a$ **where**

```
  Inl :: f a → (f ⊕ g) a
  Inr :: g a → (f ⊕ g) a
data Void
```

This allows us to combine effects using the type constructor $\oplus$. The endofunctor *Void* is used to terminate the coproduct. For instance, the effect *State s* alone would be encoded as *State s* $\oplus$ *Void*. If there were another effect *Exception*, this could be added to the chain of effects, as (*State s* $\oplus$ *Exception*) $\oplus$ *Void*.

Injecting operations into the coproduct would be done using the data constructors *Inl* and *Inr*. However, manually using these functions would be tedious and require us to know all the effects available and their ordering. This can be avoided by using injections. The constraint *Member l u* holds if the functor *l* appears in *u*. It provides the injection function *inj* that will take an effect and use *Inr* and *Inl* appropriately depending on the position of the effect within the coproduct.

```
type Member :: (Type → Type) → (Type → Type) → Constraint
class Member l u where
  inj :: l a → u a
instance Member f (f ⊕ g) where
  inj = Inl
instance (Member f h) ⇒ Member f (g ⊕ h) where
  inj = Inr ∘ inj
```

This means that the smart constructors for state can be written with *inj* as well as *Impure*.

```
get = Impure (inj (Get return))
put v = Impure (inj (Put v (return ())))
```

Now suppose there is another effect which allows us to abort the execution of the program. This would have a single constructor *Abort*, with no continuation. The function *incrAndFail* combining the use of *Exception* and *State* is given below. It increments the state by an amount *n*, but then aborts the computation.

```
data Exception a = Abort
abort = Impure (inj Abort)
```

$$
\begin{aligned}
&\mathit{incrAndFail}\ n = \mathbf{do} \\
&\quad \mathit{val} \leftarrow \mathit{get} \\
&\quad \mathit{put}\ (\mathit{val} + n) \\
&\quad \mathit{abort}
\end{aligned}
$$

The question is how to provide the semantics for the composition of these two effects in a way that is composable. This is done through algebraic effect handlers. Algebraic effect handlers usually interpret only a single effect. If an effect occurs, which they cannot handle, they leave it for another handler. They do this by embedding themselves in the continuation, so if an effect that is meant for them occurs later in the computation tree, they will still be able to handle it.

For instance a handler for the *State* effect would be a catamorphism that interprets the computation tree of the type $\mathit{Free}\ (\mathit{State}\ b \oplus g)\ a$ as a computation tree of type $\mathit{Free}\ g\ (a, s)$.

$$
\begin{aligned}
&\mathit{runState} :: s \to \mathit{Free}\ (\mathit{State}\ s \oplus g)\ a \to \mathit{Free}\ g\ (a, s) \\
&\mathit{runState}\ s\ \mathit{op} = \mathbf{case}\ \mathit{op}\ \mathbf{of} \\
&\quad \mathit{Pure}\ a \qquad\qquad\qquad\quad \to \mathit{Pure}\ (a, s) \\
&\quad \mathit{Impure}\ (\mathit{Inl}\ (\mathit{Get}\ c)) \quad \to \mathit{runState}\ s\ (c\ s) \\
&\quad \mathit{Impure}\ (\mathit{Inl}\ (\mathit{Put}\ s'\ c)) \to \mathit{runState}\ s'\ c \\
&\quad \mathit{Impure}\ (\mathit{Inr}\ a) \qquad\quad \to \mathit{Impure}\ (\mathit{fmap}\ (\mathit{runState}\ s)\ a)
\end{aligned}
$$

Similarly, the handler for the *Exception* effect would also be a catamorphism that traverses the computation tree, removing the *Exception* effect. The output type of the new computation would be $\mathit{Maybe}\ a$ reflecting the fact that the computation could fail.

$$
\begin{aligned}
&\mathit{runException} :: \mathit{Free}\ (\mathit{Exception} \oplus g)\ a \to \mathit{Free}\ g\ (\mathit{Maybe}\ a) \\
&\mathit{runException}\ \mathit{op} = \mathbf{case}\ \mathit{op}\ \mathbf{of} \\
&\quad \mathit{Pure}\ a \qquad\qquad\quad \to \mathit{Pure}\ (\mathit{Just}\ a) \\
&\quad \mathit{Impure}\ (\mathit{Inl}\ \mathit{Abort}) \to \mathit{Pure}\ \mathit{Nothing} \\
&\quad \mathit{Impure}\ (\mathit{Inr}\ a) \qquad \to \mathit{Impure}\ (\mathit{fmap}\ \mathit{runException}\ a)
\end{aligned}
$$

There also needs to be a handler for the placeholder effect *Void*. This is given by the *run* function which is always meant to run as the top-most handler and whose sole purpose is to extract the result of the computation tree.

$$run :: Free\ Void\ a \to a$$
$$run\ (Pure\ a) = a$$

There are two reasonable output types after evaluating the computation tree from *incrAndFail*, either $(Maybe\ a, s)$ or $Maybe\ (a, s)$, reflecting whether the state should be kept if there is an error. Luckily with algebraic effect handlers, both can be obtained, depending on the order in which the handlers are interleaved. In the first case, the result will be $(Nothing, 2)$, while in the second case the result will be *Nothing*.

$$(run \circ runState\ 1 \circ runException)\ incrAndFail \equiv (Nothing, 2)$$
$$(run \circ runException \circ runState\ 1)\ incrAndFail \equiv Nothing$$

This reflects a major advantage of algebraic effects. Algebraic effect handlers allow syntax and semantics to be composed with ease. Different semantics can be obtained by providing different handlers. In fact, the interactions between effects can be controlled by changing the order in which handlers interpret the computation tree. More sophisticated handlers can even be introduced that interpret effects in terms of other effects.

To highlight the practical advantages on the level of software engineering from this increase in compositionality, consider an effect for interacting with a database. One semantics could be given that reinterprets this effect in terms of the *IO* effect. Another semantics could be used for testing; this semantics could mock the database connection.

2.3 Parameterised Effects

So far two different formalisations of computational effects have been explored: monads and algebraic effects. This section explores a generalisation of an effect, called a parameterised effect, introduced by Atkey (2009a). This is not to be confused with the closely related concept of indexed monads, that have been introduced by McBride (2011).

It is shown how this generalisation can be applied to both monads and algebraic effects (Atkey, 2009b). The corresponding implementation of parameterised monads in Haskell is also given, however an implementation of parameterised algebraic effects is not given in Haskell here. As far as the author is aware, parameterised algebraic effects have not been implemented in functional programming languages.

First let us define a parameterised functor, with parameterising category S, as a functor $T : S^{\text{op}} \times S \times C \to C$. The objects of the parameterising category can be seen as predicates on states. The first parameter is called the precondition and the second parameter is the postcondition.

The object $T(P, Q, X)$ is used to represent an effectful computation that starts in the state that satisfies P and ends in the state that satisfies predicate Q. An arrow $w : Q \to R$ in the parameterising category, represents a weakening of the predicate Q. Hence there is an arrow $T(P, Q, X) \to T(P, R, X)$, that represents weakening the postcondition of the computation, given by $T(P, w, X)$. Similarly, an arrow $s : O \to P$ in C^{op} represents a strengthening of a predicate O. There is an arrow $T(O, Q, X) \to T(P, Q, X)$ that represents strengthening the precondition, given by $T(s, Q, X)$.

An encoding into Haskell is given with the *IFunctor* typeclass. Similar to the function *fmap*, there is a function *imap*. Intuitively, it means that if you can reach type b from type a, and you have a computation that arrives at a with precondition i and postcondition j, then you can obtain a computation that arrives at b with the same pre and postcondition.

$$\begin{aligned}&\textbf{type } \mathit{IFunctor} :: \forall p.\ (p \to p \to \mathit{Type} \to \mathit{Type}) \to \mathit{Constraint}\\&\textbf{class } \mathit{IFunctor}\ f \textbf{ where}\\&\quad \mathit{imap} :: (a \to b) \to f\ i\ j\ a \to f\ i\ j\ b\end{aligned}$$

A parameterised monad T, whose parameterising category is S, is a parameterised functor T, equipped with two natural transformations, $\eta_{S,A} : A \to T(S, S, A)$ called unit as well as $\mu_{S_1,S_2,S_3,A} : T(S_1, S_2, T(S_2, S_3, A)) \to T(S_1, S_3, A)$ called multiplication. The unit transformation allows you to lift values into computations, and the multiplication transformation allows you to sequence computations. Since a value doesn't actually perform computation, the indices for the unit transformation stay the same, reflecting that there is no change in state. The multiplication transformation requires that the postcondition of the first computation and the precondition of the second computation be the same.

An encoding into Haskell is given with the *IMonad* typeclass. As with the ordinary monad, the *return* function represents the unit transformation, and the *join* function represents the multiplication transformation.

$$\begin{aligned}&\textbf{type } \mathit{IMonad} :: \forall p.\ (p \to p \to \mathit{Type} \to \mathit{Type}) \to \mathit{Constraint}\\&\textbf{class } (\mathit{IFunctor}\ m) \Rightarrow \mathit{IMonad}\ m \textbf{ where}\\&\quad \mathit{return} :: a \to m\ i\ i\ a\end{aligned}$$

$$
\begin{aligned}
&\mathit{join} \quad :: m\ i\ j\ (m\ j\ k\ a) \to m\ i\ k\ a \\
&(\gg\!=) \ :: m\ i\ j\ a \to (a \to m\ j\ k\ b) \to m\ i\ k\ b
\end{aligned}
$$

In the same way that monads can be generalised to parameterised monads, so can algebras to parameterised algebras. Note that this leads to an unfortunate situation with terminology, since parameter is also used to refer to the argument given to an operation. Algebras are parameterised by a category C. A parameterised algebra consists of $\langle \Sigma, par, ar, pre, post \rangle$. Like before, Σ is a set of operation symbols, and par_{op} and ar_{op} are sets for the parameter and arity. In addition, there is pre_{op} and $post_{\mathsf{op}}$ which are objects in the category C. The set of terms is now defined as $T(V, p, q)$ where p is the precondition and q is the postcondition:

- For every morphism $a : p \to q$ in C, and variable $x \in V$, there is a term $\mathsf{return}\ (a, x)$ in $T(V, p, q)$.
- For every operation op with parameter A, arity B, precondition q and postcondition r, function $f : B \to T(V, r, s)$, $s : p \to q$ and argument $a \in A$, there is a term $\mathsf{op}(s, a, f) \in T(V, p, s)$.

To conclude this section a generalisation of effects has been presented, where types representing computations are indexed by predicates on the state before and after the computation runs. It has been shown how this can be used to give an encoding of parameterised monads in Haskell, while the underlying theory behind parameterised algebras has been discussed. This idea will prove valuable in the encoding of linearity that will be given.

2.4 Linearity

We now turn our focus to linearity, which at a high-level refers to a view of data as a resource. Its origins are in linear logic (Girard, 1987) where propositions must be used exactly once. In the same way that intuitionistic propositional logic is connected with the ordinary λ-calculus through the Curry-Howard isomorphism, intuitionistic linear logic is connected to a linear version of the λ-calculus (Benton et al., 1993). We start with an explanation of the typing rules of this calculus, before explaining how linearity can be used to consider a broader range of constraints, by indexing resources by type-level state.

In the linear λ-calculus, a function with type $A \multimap B$ can only use (or 'consume') its argument once. This means the function $\lambda x.x$ is well typed as $\tau \multimap \tau$ for any τ. However, the function $\lambda xy.x$ is not well typed, since it discards its argument y. Further, the function $\lambda xy.xyy$ is not well typed since the argument y is used twice.

$$\frac{}{\Gamma, x : \phi \vdash x : \phi}(ax) \qquad \frac{\Gamma_1 \vdash A : \phi \multimap \psi \qquad \Gamma_2 \vdash B : \phi}{\Gamma_1, \Gamma_2 \vdash AB : \psi}(app)$$

$$\frac{\Gamma, x : \phi \vdash A : \psi}{\Gamma \vdash \lambda x.A : \phi \multimap \psi}(abs)$$

The core typing rules for the linear λ-calculus are presented above. The key difference lies in the application rule, where Γ_1 and Γ_2 are disjoint. This ensures that resources are not used twice. Consider how linearity can be used to implement file handles, in a language with linearity. Once a file handle is opened it can only be used once. Operations such as *read* and *write*, which allow a file handle to be reused after the operation has completed will simply return a copy of the file handle. Meanwhile, the operation *close* will not return a copy of the file handle. A *duplicate* operation might return two copies of the file handle.

So far linearity allows the programmer to control whether an operation consumes a resource, preserves the resource or even produces more copies of the resource. However linearity can do much more, by constraining the operations that can be performed on a resource, depending on a resource's state. Consider data representing a network socket in a server. The network socket might be in a *Listening* state, waiting for a connection. One could imagine an operation *accept*, that blocks until a client attempts a connection, and takes the network socket to an *Open* state. Now that the network socket is in the *Open* state it should not be possible to use the *accept* operation again.

The way to ensure safety at a type level would be to parameterise the socket data type by a phantom type parameter, which expresses the state of the socket. The *accept* operation would only allow a socket in the *Listening* state as input, and produce a socket in the *Open* state as output. However, this only works if there is linearity. Otherwise, the original socket passed to *accept* could be reused, thwarting the attempts at type-safety.

Thus linearity allows for a far broader range of constraints than one might have anticipated *prima facie*. The fact that many real-world resources have

a notion of 'state', be they printers or communication channels reinforces the view that linearity is an appropriate way of treating resources in a computer system. Moreover, the fact that parameterised effects allow computation to be indexed by predicates on state, and that linearity allows operations to be constrained based on type-level state, provides a glimpse at how parameterised effects will be used to encode linearity.

2.5 Type Level Programming

In this section, type-level programming is introduced in Haskell. First the ordinary Haskell type system is explained, through the distinction between values, types and kinds. Then three extensions that greatly enhance the expressiveness of Haskell's type system are introduced. These will be taken advantage of the most in the succeeding chapters, in order to produce an encoding of linearity.

Examples of values are 1, `"hello"`, *True* and $\lambda x \to x * x$. These are data that exist at runtime. These values are grouped into types. For the values given previously, the types are *Int*, *String*, *Boolean* and $Int \to Int$ respectively. In particular, one can construct types from other types. Given a set of base types T e.g. *Int*, *String* and *Boolean*, the set of all types is defined as follows.

$$\tau \ := \ T \mid \tau \to \tau$$

Similarly types can be categorised as kinds, and kinds can be generated from other kinds. There is one base kind *Type*, whose inhabitants include all the standard Haskell types. In addition, algebraic data types allow us to define types that depend on types, albeit in a very restricted form. In the example below the type constructor *Maybe* is a type-level function from *Type* to *Type*.

type *Maybe* :: *Type* → *Type*
data *Maybe a* = *Just a* | *Nothing*

Notice that *Maybe Boolean* has kind *Type*, however *Maybe* has kind $Type \to Type$. The set of all kinds in Haskell is defined as follows.

$$\kappa \ := \ Type \mid \kappa \to \kappa$$

A significant extension to this is the data kinds language extension (Yorgey et al., 2012). This extension allows users to introduce new kinds, but without requiring any special syntax. It automatically promotes types defined as algebraic data types, so that the type will also function as a kind, and the values of the type will function as types themselves. For instance, the type *Bool* whose inhabitants are the data constructors *True* and *False*, has the promoted data type *Bool*, which is a kind. The promoted data type *Bool* is inhabited by the promoted data constructors *True* and *False*, which are types with no values as inhabitants.

Another significant part of type-level programming in Haskell is the type families language extension (Schrijvers et al., 2008), which allows more expressive functions on the type level. In the example below, the data type *Nat* encodes natural numbers. The nullary type constructor Z has the kind *Nat* and the type constructor S is a type-level function from *Nat* to *Nat*. The type family *Sub* encodes the subtraction operation on natural numbers.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} = Z \mid S\ \mathit{Nat} \\
&\textbf{type}\ \mathit{Sub} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
&\textbf{type}\ \mathit{family}\ \mathit{Sub}\ a\ b\ \textbf{where} \\
&\quad \mathit{Sub}\ a\ Z = a \\
&\quad \mathit{Sub}\ (S\ a)\ (S\ b) = \mathit{Sub}\ a\ b
\end{aligned}
$$

In the same way that terms in Haskell can be polymorphic in terms of types, type families can be polymorphic in terms of kinds using the polymorphic kinds extension. In the example below the type family *First* retrieves the first component of a type-level pair, and will work for any type-level pairs regardless of the kinds a and b.

$$
\begin{aligned}
&\textbf{type}\ \mathit{First} :: (a, b) \to a \\
&\textbf{type}\ \mathit{family}\ \mathit{First}\ a\ \textbf{where} \\
&\quad \mathit{First}\ (x, y) = x
\end{aligned}
$$

The encoding of linearity in Haskell will consist primarily of type-level machinery. This will involve a multitude of language extensions. The most significant have been reviewed here. Promoted data kinds allow the creation of new kinds and types, type families allow functions on the type-level and kind polymorphism allows type-level code that works with different kinds.

Chapter 3

Implementation

This chapter develops the encoding of linearity in Haskell. This starts with a discussion of the overall design of the encoding. Then an implementation of parameterised algebraic effects is developed. This is used to build very simple effects, with the key shortcoming being that they can only encapsulate a single resource. The section that follows looks at overcoming this. It develops an encoding for the storage effect, which is able to interact with many resources.

While the storage effect is contrived, its generality means that the core building blocks, that are needed to implement it can also be used as the foundation for a range of other effects that interact with resources. Moreover it provides a concrete setting to study the design space for these building blocks. The storage effect allows us to develop two other components: a type-checker plugin to improve type inference; and scoped operations. The chapter ends with some brief remarks on alternative design decisions. The outcome of this chapter is an ensemble of type-level utilities that will allow the development of more advanced case studies, in the succeeding chapters.

3.1 Design

This section shall give an outline of the encoding of linearity at a high-level. The hope is to give a concrete idea as to how effects can be used to encode linearity. As discussed in the background, linearity can be thought of as ensuring entities are used once. This allows more complex constraints to be encoded on resources using types indexed by states. The problem is that

in Haskell, there is no way to ensure that an entity isn't used more than once. This means that if a resource is given as input to an operation, which produces the same resource, indexed by a modified state, then there is no way to prevent the original reference to the resource, that will still be indexed by the original state, from being reused.

The solution presented by this project is based on the observation that this is only a problem at the type-level. The actual runtime value contained by the reference is correct, it is simply that the type given to the reference is indexed by the wrong state, meaning that type errors cannot be prevented. The solution is to store the type-level state elsewhere, and have the type of the resource indexed by a reference to the state. That way the operation that modifies the resource does not need to return a new version of the resource indexed by the modified state. Instead the location containing the state, that is referenced by the resource's type, is updated with the new state of the resource.

The question that follows is where to store the state. In the background, parameterised effects were introduced. Operations involving these effects had type-level indices representing predicates on the state of computation. In our encoding, these indices will be used to store the type-level state of resources involved in the computation. The precondition will be the state of resources before the operation, and the postcondition will be the state of resources after the operation. Operations that require resources to be in a particular type-level state, or that change the type-level state of resources, will be operations belonging to the parameterised algebra.

The simplest case is where there is only one resource, in which case there isn't any need for references at all, since there is only one resource for operations to be performed on. This is what the next section works towards, by building parameterised algebras in Haskell.

In order to interact with multiple effects, there needs to be a way of storing the states of multiple resources in the state of the parameterised algebra, and a way of indicating which type-level state, a resource refers to. This is done by having a list of states called a context. Then resources will be indexed by the position of the resource's state in that list. This will be developed using the storage effect as an example.

3.2 Parameterised Effects

In this section, we shall develop the infrastructure for parameterised algebraic effects, building upon the standard techniques for encoding algebraic effects discussed in preceding chapter. We first show how the freer monad construction builds upon the free monad construction introduced earlier to produce a more sophisticated representation of algebraic effects (Kiselyov and Ishii, 2015). We then show how this can be adapted to the context of parameterised algebraic effects. Finally we consider how to take the coproduct of different effects and how to inject individual effects into a coproduct. We show how this encoding can be used to model two examples: a parameterised version of the state effect; the file context effect.

3.2.1 Encoding Effects

In the background it was shown how an algebraic effect could be encoded as a functor, whose data constructors are the operations of the effect. This functor could be combined with the free monad to build up the syntax of the algebra.

$$\textbf{data}\ \mathit{Free}\ f\ a\ = \mathit{Pure}\ a \mid \mathit{Impure}\ (f\ (\mathit{Free}\ f))$$
$$\textbf{data}\ \mathit{State}\ s\ a = \mathit{Get}\ (s \to a) \mid \mathit{Put}\ s\ a$$

In both operations of $\mathit{State}\ s$, there is a continuation representing the rest of the computation. For the *Get* operation, the continuation is $s \to a$. Meanwhile for *Put* the continuation is a, but this could alternatively be written as $() \to a$ to be consistent. Since this continuation appears in almost all operations, it is common to factor it out. Instead of the type parameter of $\mathit{State}\ s$ being the type of the rest of the computation, it is the type of the argument passed to the continuation. In the background, this was called the arity of the operation. In order to constrain the type parameter of $\mathit{State}\ s$, a generalised abstract data type is used.

$$\begin{aligned}&\textbf{data}\ \mathit{State}\ s\ a\ \textbf{where}\\ &\quad \mathit{Get} :: \mathit{State}\ s\ s\\ &\quad \mathit{Put} :: s \to \mathit{State}\ s\ ()\end{aligned}$$

In this definition of *State*, the final type parameter of *Get* has to be the same as the type being stored. The final parameter of *Put* is the

unit type, since there is nothing useful to give to the continuation. In order to add the continuation, the *Coyoneda* constructor is used. The type *Coyoneda* (*State* s) a represents a computation of the state algebra, which results in the type a. The first argument to the *Coyoneda* data constructor is the continuation, and the second is the operation. Note that the type parameter of the operation, and the argument to the continuation have to match.

$$\textbf{data}\ \mathit{Coyoneda}\ f\ a = \forall b.\ \mathit{Coyoneda}\ (b \to a)\ (f\ b)$$

One property of *Coyoneda* f is that it is a functor for any data type f. The type *Free* (*Coyoneda* (*State* s)) a represents the syntax of computations involving the *State* algebra. In the background, it was shown that *Fix* and *Base* could be combined to produce *Free*. Similarly, *Free* and *Coyoneda* can be combined to produce *Freer*. It is a 'freer' construction, because while *Free* f requires that f be a functor, for *Free* f to be a functor and monad, *Freer* f is a functor and a monad regardless of what f is.

$$\begin{aligned}&\textbf{type}\ \mathit{Freer} :: (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})\\&\textbf{data}\ \mathit{Freer}\ f\ a\ \textbf{where}\\&\quad \mathit{Pure}\ \ \ :: a \to \mathit{Freer}\ a\\&\quad \mathit{Impure} :: (b \to \mathit{Freer}\ f\ a)\ (f\ b)\end{aligned}$$

The question that now arises is how to adapt this for parameterised algebras. In the current formulation, the type of an effect such as *State* s is $\mathit{Type} \to \mathit{Type}$, i.e. there is only one type parameter, which represents the type of the argument passed to the continuation. In the new formulation, the type of an effect will be $k \to k \to \mathit{Type} \to \mathit{Type}$, i.e. there are three type parameters. The first will be the precondition, the second will be the postcondition and the third will be the type of the argument passed to the continuation.

Notice that the first two type parameters belong to kind k, which can change depending on the effect. The kind k is the kind whose types are the possible values for the type-level state of the effect. Different effects will use different kinds to represent their type-level state. The kind k corresponds to the parameterising category of the effect. For convenience, a type synonym *Operation* k is introduced to denote the type of an effect with parameterising category k. It will become clearer later why this synonym is called *Operation* and not simply *Effect*.

```
type Operation k = k → k → Type → Type
```

We shall now present an example of a parameterised algebra in this style, the parameterised state effect. Unlike the usual state effect, which can only store data of a particular type, the type of the data stored in the parameterised version can change. The precondition is the type being stored before the operation, and the postcondition is the type being stored after the operation. This effect is parameterised by the category of all Haskell types, and consequently has type *Operation Type*.

```
type State :: Operation Type
data State p q a where
  Get :: State p p p
  Put :: q → State p q ()
```

Let us consider both operations in turn. The first operation *Get* does not change the state at all. Hence, the postcondition is the same as the precondition p. Furthermore, the type it gives to the continuation is the same as the type it stores i.e. p. The operation *Get* also does not require any arguments.

Meanwhile, the operation *Put* does take a single argument, the data to be stored in the state. The type of this data is also the type of the postcondition. The fact that the precondition p does not appear anywhere else in the type of the operation *Put*, reflects the fact that *Put* can happen after any operation. The type of the argument given to the continuation is the unit type, as it has nothing meaningful to pass to the continuation.

To provide a contrast to the parameterised state effect, another simple example of a parameterised algebraic effect is presented, the file context effect. Unlike the state effect, the parameterising category is *FileState*, which is a promoted data type with two members *Opened* and *Closed*. The file context effect is for encapsulating access to a file. The *Opened* state is used when there is a file handle is open. The *Closed* state is used when there is no file handle.

```
data FileState = Opened | Closed
type FileCtx :: Operation FileState
data FileCtx p q a where
  Open :: FilePath → FileCtx Closed Opened ()
```

$$
\begin{aligned}
&\mathit{Read} :: \mathit{FileCtx}\ \mathit{Opened}\ \mathit{Opened}\ \mathit{String} \\
&\mathit{Close} :: \mathit{FileCtx}\ \mathit{Opened}\ \mathit{Closed}\ () \\
\end{aligned}
$$

Now that there is a way of encoding effects, there needs to be a way of creating a computation tree from the effect. This is done with a parameterised version of the freer monad, called *IFreer*. The type parameter f is the data type encoding the effect, the type parameter p is the precondition, the type parameter q is the postcondition, and the type parameter a is the output of the computation.

The data constructor *Pure* allows a value to be raised to a computation, provided that the pre and postcondition are the same. The data constructor *Impure* allows two computations to be sequenced. The first computation is an operation, given by f. The second computation is given by the continuation. Note that the postcondition of the first computation must be the same as the precondition of the second computation.

$$
\begin{aligned}
&\textbf{type}\ \mathit{IFreer} :: \forall k.\ \mathit{Operation}\ k \to \mathit{Operation}\ k \\
&\textbf{data}\ \mathit{IFreer}\ f\ p\ q\ a\ \textbf{where} \\
&\quad \mathit{Pure}\ \ \ :: a \to \mathit{Free}\ f\ p\ p\ a \\
&\quad \mathit{Impure} :: f\ p\ q\ b \to (b \to \mathit{Free}\ f\ q\ r\ a) \to \mathit{Free}\ f\ p\ r\ a
\end{aligned}
$$

In the same way that the freer monad can be given a functor and monad instance, parameterised freer monads can be given an indexed functor and an indexed monad instance.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{IFunctor}\ (\mathit{IFreer}\ f)\ \textbf{where} \\
&\quad \mathit{imap} :: (a \to b) \to \mathit{IFreer}\ f\ p\ q\ a \to \mathit{IFreer}\ f\ p\ q\ b \\
&\quad \mathit{imap}\ f\ (\mathit{Pure}\ a)\ \ \ \ \ \ = \mathit{Pure}\ (f\ a) \\
&\quad \mathit{imap}\ f\ (\mathit{Impure}\ a\ c) = \mathit{Impure}\ a\ (\mathit{fmap}\ (\mathit{imap}\ f)\ c) \\
&\textbf{instance}\ \mathit{IMonad}\ (\mathit{IFreer}\ f)\ \textbf{where} \\
&\quad \mathit{return} :: a \to \mathit{IFreer}\ f\ p\ p\ a \\
&\quad \mathit{return} = \mathit{Pure} \\
&\quad (\gg\!=) :: \mathit{IFreer}\ f\ p\ q\ a \to (a \to \mathit{IFreer}\ f\ q\ r\ b) \\
&\quad (\mathit{Pure}\ a) \gg\!= f\ \ \ \ \ \ = f\ a \\
&\quad (\mathit{Impure}\ a\ c) \gg\!= f = \mathit{Impure}\ a\ (\mathit{imap}\ (\gg\!= f)\ c)
\end{aligned}
$$

In the case of $\mathit{Pure}\ a$, the *imap* function simply applies the function f to the value a. In the case of $\mathit{Impure}\ a\ c$, the *imap* function recurses on the computation tree that is the result of the continuation c. The *return*

function lifts a value to a computation, by calling *Pure*. The $\gg=$ function composes two computations, by recursing till it reaches the leaves of the first computation tree and inserting the second computation tree.

3.2.2 Composing Effects

The next step is to build a way of combining algebraic effects together, i.e. to take the coproduct of the effects. Earlier an approach to this for conventional algebraic effects was shown by having a binary coproduct operator. A more advanced approach is to have a coproduct data type indexed by a type-level list, e.g. $[State\ Int, Exception]$. This cannot be easily translated into a parameterised setting, however. The reason for this is that while conventional effects will have the same kind, parameterised effects will not have the same kind as they have different parameterising categories. For example, the type $[State, FileCtx]$ is malformed in Haskell.

A value-level heterogeneous list can be obtained by using higher-rank polymorphism. However, it is not possible to have higher-rank kind polymorphism in Haskell, so a kind of the form $[\forall x.\ Operation\ x]$ cannot be encoded. The solution to this is to use a heterogeneous list itself parameterised by a standard type-level list. The heterogeneous list contains the effects, and the type-level list contains the parameterising categories. This is done with the data type *HListEffect*, with the constructor *Nil* for an empty heterogeneous list and $\ominus$ to prepend an effect to the front of a heterogeneous list.

```
type HListEffect :: [Type] → Type
data HListEffect ks where
  Nil :: HListEffect []
  (⊖) :: Operation k → HListEffect ks → HListEffect (k : ks)
```

So the example would be encoded as $State \ominus FileCtx \ominus Nil$, which would have kind $HListEffect\ [Type, FileState]$. Henceforth a list of effects will be referred to as an *effect chain*. A solution is needed to encode the combined type-level state for an effect chain. This leads to a confusing situation with terminology. For clarity, the term *effect state* will be used to refer to the type-level state associated with an individual effect, e.g. *State* or *FileCtx*. Meanwhile, the term *combined state* will be used to refer to the combination of different effect states belonging to an effect chain.

The coproduct of an effect chain will be indexed by the combined state for the effect chain. This is formed by creating a heterogeneous list consisting of the individual effect states. The list is heterogeneous, since the effect states will have different kinds, depending on the parameterising category. This list will be given by the *HList* data type, with the constructor *Unit* for the combined state corresponding to an empty effect chain (i.e. *Nil*), and the constructor $\oplus$ for prepending an effect state to the front of a heterogeneous list.

$$
\begin{aligned}
&\textbf{type } \mathit{HList} :: [\mathit{Type}] \to \mathit{Type} \\
&\textbf{data } \mathit{HList}\ \mathit{ks} \textbf{ where} \\
&\quad \mathit{Unit} :: \mathit{HList}\ [\,] \\
&\quad (\oplus)\ \ :: k \to \mathit{HList}\ \mathit{ks} \to \mathit{HList}\ (k : \mathit{ks})
\end{aligned}
$$

Consider the effect chain $\mathit{State} \ominus \mathit{FileCtx} \ominus \mathit{Nil}$. A combined state for this could be $\mathit{String} \oplus \mathit{Closed} \oplus \mathit{Unit}$, with kind $\mathit{HList}\ [\mathit{Type}, \mathit{FileState}]$. One might consider whether it's possible to define *HListEffect* in terms of *HList*. Defining two separate structures makes the code simpler since a effect chain *HListEffect ks* will have an index of kind *HList ks*. The fact that the parameter is the same for each, makes writing code that works with these two structures simpler. This is seen in the definition of the coproduct.

$$
\begin{aligned}
&\textbf{type } \mathit{Coproduct} :: \\
&\quad \mathit{HListEffect}\ \mathit{ks} \to \mathit{HList}\ \mathit{ks} \to \mathit{HList}\ \mathit{ks} \to \mathit{Type} \to \mathit{Type} \\
&\textbf{data } \mathit{Coproduct}\ \mathit{as}\ \mathit{ps}\ \mathit{qs}\ x \textbf{ where} \\
&\quad \mathit{Inl}\ \ :: a\ p\ q\ x \\
&\qquad \to \mathit{Coproduct}\ (a \ominus \mathit{as})\ (p \oplus \mathit{ps})\ (q \oplus \mathit{ps})\ x \\
&\quad \mathit{Inr}\ \ :: \mathit{Coproduct}\ \mathit{as}\ \mathit{ps}\ \mathit{qs}\ x \\
&\qquad \to \mathit{Coproduct}\ (a \ominus \mathit{as})\ (p \oplus \mathit{ps})\ (p \oplus \mathit{qs})\ x
\end{aligned}
$$

As an example the effect *Closed* would be injected into the coproduct as *Inr* (*Inl Closed*). A possible pre and postcondition pair would be $\mathit{Int} \oplus \mathit{Opened} \oplus \mathit{Unit}$ and $\mathit{Int} \oplus \mathit{Closed} \oplus \mathit{Unit}$. However the pre and postconditions $\mathit{Int} \oplus \mathit{Opened} \oplus \mathit{Unit}$ and $\mathit{String} \oplus \mathit{Opened} \oplus \mathit{Unit}$ would not be valid. The reason is that the operation *Close* should only affect the effect state in the combined state corresponding to the state effect. All other effect states should stay the same. This is a property that is enforced by the definition of the coproduct.

The *Inl* constructors lifts an effect into the coproduct. The parameterising heterogeneous list contains the particular effect as the first item, and an

arbitrary number of other effects afterwards. The pre and postconditions are $p \oplus ps$ and $q \oplus ps$. The fact that ps is the same in both, reflects the fact that for the other effects, the effect states must stay the same.

The *Inr* constructor 'pads' the injected effect, by adding other effects in the parameterising list that come before it. The indices $p \oplus ps$ and $p \oplus qs$ reflect the fact that since the effect state for injected effect can be found in ps and qs, they could be different. But the effect now being added to the parameterising list, should have the same effect state for both the pre and postcondition.

An important thing to note is that there are multiple layers of indexing occurring. The *Coproduct* data type is indexed by the effect chain, i.e. a heterogeneous list of effects, which itself is indexed by a list of parameterising categories. The *Coproduct* data type is also indexed by two combined states, i.e. heterogeneous lists of effect states, which are themselves indexed by a list of parameterising categories.

3.2.3 Injecting Effects

The final piece of infrastructure to build parameterised effects is a way of injecting effects into a coproduct easily, without having to manually use the constructors *Inl* and *Inr*. We do this in two steps. First the effect chain is searched for the index of the particular effect within the list, as a type-level natural, with a type family *FindEffect*. Then the index and the effect chain are passed into a typeclass that defines a function *inj*. Using a type family to pre-compute the index, avoids problems with overlapping instances. If the effect cannot be found in the effect chain, then a type error occurs.

```
type FindEffect :: Operation k → HListEffect ks → Nat
type family FindEffect a as where
  FindEffect a (a ⊖ as) = Z
  FindEffect b (a ⊖ as) = (S (FindEffect b as))
  FindEffect c Nil
    = TypeError (Text "Effect " ⊙ ShowType c
      ⊙ Text " not found in signature")
```

The typeclass definition for injection is presented below. The type family *MemberAux i l u* states the effect l is the ith component of the effect chain u. Further, it provides a function *inj* to inject the effect l into *Coproduct u*.

If the effect has pre and postconditions p and q, then p and q should be at the ith component of ps and qs respectively. However, note this is not enforced by the type of *MemberAux*.

```
class MemberAux i l u | i u → l where
  inj :: l p q x → Coproduct u ps qs x
```

This is acceptable since *inj* will not be used directly. Instead it will be used through a function *inject*, that injects an effect into a coproduct and then a freer monad. This *inject* function will use two typeclasses *Contains* and *Evolve*, to ensure that p, q, ps and qs are related correctly. The constraint *Contains i p ps* means that effect state p is present in combined state ps at index i. The constraint *Evolve ps i q qs* means combined state qs is constructed by taking combined state ps and putting effect state q in index i. Attempting to put these constraints on the *inj* function is unnecessary, but also difficult as it complicates type-checking.

An alternative would be to include the pre and postcondition as parameters of the typeclass *MemberAux*. This would mean a typeclass *MemberAux* would have parameters *i l u p q ps qs*, where p and q are the effect state for the effect, and ps and qs are the combined state for the effect chain. However, this is undesirable as it means that you need a different *MemberAux* whenever the effect states change.

```
inject
    :: (i ~ FindEffect l u, MemberAux i l u, Contains i p ps
       , Evolve ps i q qs)
    ⇒ l p q x → u ps qs x
    → ITree u ps qs x
inject = Impure. inj @i return
```

It will be common to want to check that an effect is part of an effect chain, and find the effect state at a particular index within the combined state. This is unsurprising since smart constructors will be written in terms of *inject*, and so the *MemberAux* and *Contains* constraints will be propagated onto these constructors. To make this simpler a type synonym *Member* is introduced.

```
type Member i l u p ps
   = (i ~ FindEffect l u, MemberAux i l u, Contains i p ps)
```

To conclude this section, we have given an adaption of the typical infrastructure for algebraic effects, that accommodates parameterised effects. We showed two examples of parameterised effects, *State* and *FileCtx*. Both of these effects encapsulated a resource. *State* showed how the type of the arguments passed to the operation and that type of the argument that the operation passes to the continuation can depend on the pre and postcondition of the operation. *FileCtx* showed how certain operations could only occur with certain preconditions.

A key deficiency is that the examples so far only allowed a single instance of the resource to be encapsulated. *State* only allowed one item to be stored, and *FileCtx* only allowed one file to be open. This is not practicable for typical programming scenarios, where you will want effects to be able to interact with multiple resources.

3.3 Multiple Resources

In the previous section, we developed the infrastructure for encoding a parameterised extension of algebraic effects. In this section, we consider how to develop effects that can interact with multiple resources. We will do this by taking an effect, called the storage effect as an example. The effect will function similarly to parameterised state, with two key differences. First, an arbitrary number of things with different types can be stored. Second, each 'storage cell' will be associated with a count of how many 'instances' of the value exist. This section will first show how the operations of the storage effect can be encoded, before showing how they can be handled. Then examples of this effect are given.

3.3.1 Encoding Operations

The first thing to consider is the parameterising category. Since the effect needs to interact with multiple resources, the parameterising category needs to be a collection of some kind. We choose to use a type-level list, which will be referred to as a context. The context will be the effect state for the storage effect. Each element in the context will be a pair containing two pieces of information about a storage cell. The first will be the type it contains, the second will be a natural number referring to how many instances of it that you have. The effect state for the storage effect, *StorageState* is given below.

```
type StorageState = [(Type, Nat)]
data Nat          = Z | S Nat
```

Previously, a distinction has been made between *combined state* and *effect state*. The combined state is made up a series of effect states. In the case of the storage effect, the effect state is a context, consisting of pieces of information that correspond to each resource. For convenience, the term *resource state* will be introduced, to refer to the state corresponding to an individual resource within a context.

As an example, the effect state $[(Int, S\ (S\ Z)), (String, S\ Z)]$ represents a context with two resources. At index 0, the resource state represents an integer with two instances. At index 1, the resource state represents a string with a single instance.

There will be two operations that make up the effect: registration and use. Registering a value will construct a new 'storage cell' and provide a reference, that can be used to access that storage cell. You must give the value to be stored and the count of how many instances of the value you have. Given a context $[(Int, S\ Z)]$, registering an *Int* with a single instance, will give a new context $[(Int, S\ Z), (Int, S\ Z)]$.

The reference will have the type $ENat\ n$, where n is a natural number. *ENat* is a value-level type witness of a natural number. At the type-level it will be indexed by a *Nat*, a Peano representation of a natural number. However, at runtime the data type will contain a 32-bit integer. The *ENat* data type will also be used to encode how many instances of a resource there are.

A data constructor with the same name is used to create *ENat* values. It is unsafe in the sense that it falls on the user of the function, to make sure that the runtime value corresponds with the type-level index. While using a Peano representation of natural numbers at runtime would avoid this and be completely type-safe, it would also be highly inefficient. *ENat* values *zero*, *one* and *two* are created for convenience.

```
type ENat :: Nat → Type
data ENat n where
  ENat :: Int → ENat n
zero = ENat @Z 0
one  = ENat @(S (Z)) 1
two  = ENat @(S (S (Z))) 2
```

In order to help encode this operation, the *Append* and *Len* type families are introduced. These are characteristic of the type families that will be introduced to handle contexts, in that they are kind-polymorphic so they can work with any type-level list. This means that they can be used for other effects that follow this model, of having a context of resources represented by a type-level list. The *Append* type family is used to append a new resource state to the end of a context, while the *Len* type family provides the length of a context.

$$
\begin{aligned}
&\textbf{type } \mathit{Append} :: \forall m.\ [m] \to m \to [m] \\
&\textbf{type } \textit{family } \mathit{Append}\ \mathit{xs}\ \mathit{y} \textbf{ where} \\
&\quad \mathit{Append}\ [\,]\ \mathit{y} \quad\ = \mathit{y} : [\,] \\
&\quad \mathit{Append}\ (\mathit{x} : \mathit{xs})\ \mathit{y} = \mathit{x} : (\mathit{Append}\ \mathit{xs}\ \mathit{y}) \\
&\textbf{type } \mathit{Len} :: \forall m.\ [m] \to \mathit{Nat} \\
&\textbf{type } \textit{family } \mathit{Len}\ \mathit{xs} \textbf{ where} \\
&\quad \mathit{Len}\ [\,] \quad\ = \mathit{Z} \\
&\quad \mathit{Len}\ (\mathit{x} : \mathit{xs}) = \mathit{S}\ (\mathit{Len}\ \mathit{xs})
\end{aligned}
$$

Using a reference to a storage cell will give you the value it holds. You must give the reference and the number of instances that you want to use. It will then decrement the associated count by how many instances you want to use. For instance, if you have a context $[(\mathit{Int}, \mathit{S}\ \mathit{Z})]$ and you use one instance, this will leave you with $[(\mathit{Int}, \mathit{Z})]$. In order to encode this operation, the type families *Lookup* and *Replace* are introduced. The type family *Lookup* provides the resource state at a particular index in the context, while the type family *Replace* updates the resource state at a particular index in the context.

$$
\begin{aligned}
&\textbf{type } \mathit{Replace} :: \forall m.\ [m] \to \mathit{Nat} \to m \to [m] \\
&\textbf{type } \textit{family } \mathit{Replace}\ \mathit{xs}\ \mathit{i}\ \mathit{y} \textbf{ where} \\
&\quad \mathit{Replace}\ (\mathit{x} : \mathit{xs})\ \mathit{Z}\ \mathit{y} \quad\ = \mathit{y} : \mathit{xs} \\
&\quad \mathit{Replace}\ (\mathit{x} : \mathit{xs})\ (\mathit{S}\ \mathit{n})\ \mathit{y}\ = \mathit{x} : (\mathit{Replace}\ \mathit{xs}\ \mathit{n}\ \mathit{y}) \\
&\textbf{type } \mathit{Lookup} :: \forall m.\ [m] \to \mathit{Nat} \to m \\
&\textbf{type } \textit{family } \mathit{Lookup}\ \mathit{xs}\ \mathit{i} \textbf{ where} \\
&\quad \mathit{Lookup}\ [\,]\ \_ \quad\ = \mathit{TypeError}\ (\mathit{Text}\ \texttt{"Could not find index"}) \\
&\quad \mathit{Lookup}\ (\_ : \mathit{xs})\ (\mathit{S}\ \mathit{n}) = \mathit{Lookup}\ \mathit{xs}\ \mathit{n} \\
&\quad \mathit{Lookup}\ (\mathit{x} : \_)\ \mathit{Z} \quad\ = \mathit{x}
\end{aligned}
$$

The encoding for the storage effect is given below. The types for the operations are more sophisticated than previous effects, reflecting the increase

in complexity. The operation *Register* takes a parameter of type t, which is the value to be stored, and a *ENat*, indexed by the natural number c, which is the number of instances of the resource there are. The result of the *Register* operation is the length of the original context as an *ENat*, i.e. the index *Len p* which the inserted value can be found within the context. The *Register* operation appends the new resource state (t, c) to end of the context p.

The *Use* operation takes two *ENat* values, one indexed by c_2 which is the number of instances of the resource desired, and another indexed by n which is the index of the resource state in the context. The result of the *Use* operation is the type of the value stored in the storage cell. The *Use* operation finds the resource state in the context p at index n, and uses the *Sub* type family to ensure there are sufficient instances of the resource left.

```
type StorageOperation :: Operation StorageState
data StorageOperation p q a where
  Register
    :: ENat c
    → t
    → StorageOperation p (Append p (t, c)) (ENat (Len p))
  Use
    :: (Lookup p n ∼ (t, c1))
    ⇒ ENat c2
    → ENat n
    → StorageOperation p (Replace p n (t, Sub c1 c2)) k
```

We also give smart constructors *registerMany* and *useMany* to use the operations, which use the *inject* function from the previous section, to inject the operation into a coproduct for an effect chain. The types of two constructors naturally mirror the data constructors for their corresponding operations. The convenience functions *use* and *register* are also given to handle the case where you want to register or use one instance of a resource.

```
registerMany
  :: (Member StorageOperation u p ps i
     , Evolve ps i (Append p (t, c)) qs)
  ⇒ ENat c
  → t
  → ITree u ps qs (ENat (Len p))
```

```
registerMany c t = inject (Register c t) return
register = registerMany one

useMany
  :: (Member StorageOperation u p ps i
     , Lookup p n ~ (t, c1)
     , Evolve ps i (Replace p n (t, Sub c1 c2)) qs)
  => ENat c2
  -> ENat n
  -> ITree u ps qs t
useMany v n = inject (Use v n) return
use = useMany one
```

So far we have seen how the operations for the storage effect can be encoded. The notion of a context, made up of resource states, has been introduced. The type families *Append*, *Lookup*, *Len* and *Replace* have been introduced that work on contexts.

3.3.2 Handling Operations

In this section, we turn our attention to providing the semantics for the storage effect. The way this will be done is by storing a list of type $[Any]$, where *Any* is a GHC data type representing any Haskell value. This means that coercions have to be used when working with the list, but this is acceptable and safe. This is because only the code in the handlers performs the coercions. So this does not impact the type-safety of the code that will use the storage effect.

Note that this is not a particularly efficient implementation, since a Haskell list has $O(n)$ lookup and append costs, but this is not a concern at this point. Later examples of handlers will avoid the use of lists, to allow for more efficient implementations.

The handler consists of *runLinear* which wraps around a helper function *runLinearH*, that contains the main logic. The wrapper function *runLinear* provides two things. Firstly, it ensures the precondition for the storage effect is $[\,]$, i.e. that the computation tree starts from an empty context. Secondly, it ensures that the computation tree ends with a context that is entirely consumed through the use of *AllZero*.

```
runLinear :: (AllZero j)
  => IFreer (StorageOperation ⊖ as) ([] ⊕ ps) (j ⊕ qs) x
```

```
    → IFreer as ps qs x
runLinear = runLinearH [ ]
```

The constraint *AllZero* ensures that the context is fully consumed i.e. that the counts for each resource is zero. A big consideration is that good error messages are presented. This is why although it may seem natural to encode a constraint using a typeclass, the constraint will be implemented with a type family instead. In particular, it would be useful to know all the indices at which resources have not been fully consumed. An auxiliary type family will be used to find this information as a type-level list. Then a wrapper function will produce an error if the list is non-empty.

```
type VerifyAux :: StorageState → Nat → [Nat]
type family VerifyAux xs i where
  VerifyAux [ ] i          = [ ]
  VerifyAux ((_, Z) : xs) i = VerifyAux xs (S i)
  VerifyAux ((_, _) : xs) i = (i : VerifyAux xs (S i))
type VerifyWrapper :: [Nat] → ()
type family VerifyWrapper as where
  VerifyWrapper [ ] = ()
  VerifyWrapper ns = TypeError (Text "Indices " ⊙ VerifyStr ns)
type AllZero :: StorageState → Constraint
type AllZero a = VerifyWrapper (VerifyAux a Z) ∼ ()
```

The helper function *runLinearH* performs the main logic of the handler. In the case for *Use*, it ignores the type witness corresponding to the number of instances to store, since this information is only useful at the type-level. Similarly, the case for *Register* ignores the type witness corresponding to the number of instances to take, since again this is only useful at the type-level. The code for *Use* retrieves the integer within *ENat* and obtains the resource at that index in l.

The function *unsafeCoerce* is used to convert the resource from type *Any* to the type that is expected as the result of the operation. This is safe since the syntax of the storage effect will ensure that the user of the effect can only use the operations in a way that is type-safe. The *Register* operation adds a resource to the end of l and stores the length of the list, i.e. the index of the new resource, within an *ENat* which is the result of the operation.

```
runLinearH ::
  → [Any]
```

```
    → IFreer (StorageOperation ⊖ as) (p ⊕ ps) (q ⊕ qs) x
    → IFreer as ps qs x
runLinearH l op = case op of
  Pure a → return a
  Impure (Inl (Use _ (ENat i))) cnt
    → runLinearH l (cnt (unsafeCoerce (l !! i)))
  Impure (Inl (Register b _)) cnt
    → runLinearH (l ++ [unsafeCoerce b]) (cnt (ENat (length l)))
```

To conclude this section, it has been shown how a handler can be written for the storage effect. A key property of *runLinear* is the *AllZero* constraint which ensures that all resources have been consumed at the end of the context.

3.3.3 Examples

We shall now look at some code that type-checks under the framework. In this example i and k are pointers to the resources. i will have type *ENat Z* and k will have type *ENat* (*S Z*). The nice thing is that *register* can be used with data of different types. Furthermore, j and l are known by the type system to be of types *Int* and *String*. Consequently, Haskell is able to deduce that the result of the entire computation is $(Int, String)$. Note that this is without any type signatures.

```
example = run ∘ runLinear do
  i ← register 10
  j ← use i
  k ← register "Hello"
  l ← use k
  return (j, l)
```

The following example will fail to type-check, as the second *Use* operation will try to construct the type *Sub Z* (*S Z*) which will result in an error message, stating that there are no instances of the resource available.

```
example1 = run ∘ runLinear do
  i ← register 10
  j ← use i
```

```
  use i
  return j
```

This following example will also fail to type-check. This is due to the *AllZero* constraint on on the postcondition. It will state that there are resources that were not fully consumed at indices 0 and 1.

```
example2 = run ∘ runLinear do
  register 10
  register "Hello"
  return ()
```

Note that it is possible to directly index resource states within the context, without using identifiers. However, this would not allow you to write type-unsafe code. The following example fails to compile, as it attempts to look up an index not in the context. This leads to an error in *Lookup*.

```
example3 = run ∘ runLinear do
  use one
  return ()
```

So far we've seen how an effect which interacts with multiple resources can be encoded and how its handlers can be implemented. The ideas discussed in this section underlie more advanced encodings of linearity. A particularly important idea is the notion of a context of resources, that is manipulated by type families such as *Append* and *Replace*. In addition, *ENat* showed how the type-level reference to a resource state and the actual runtime value that refers to the resource can differ.

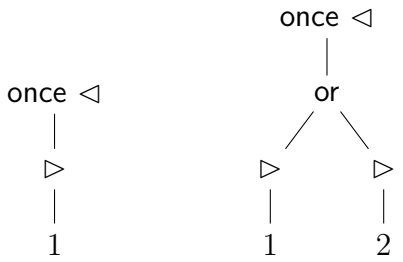
3.4 Scoped Operations

In this section, scoped operations are added to the framework of parameterised algebraic effects. First, prior work on scoped operations for algebraic effects is introduced (Wu et al., 2014; Piróg et al., 2018). Then an implementation for normal algebraic effects is considered. Then this is adapted to the setting of parameterised algebras, with careful consideration of how effect state changes when entering and exiting the inner scope. Finally, two examples of scoped operations are added to the storage effect.

3.4.1 Background

In imperative programming languages, scopes are used in conditionals and loops. In the context of algebraic effects, a scoped operation is an operation which changes the behaviour of operations that come after it. Scoped operations can be used to model multithreading and exception handling. The precise set of operations that the scoped operation affects is called the inner scope. Prior literature has shown that it is not enough to allow the handlers of algebraic effects to delimit the scope of an effect, as this breaks the separation between syntax and semantics, that is a core feature of algebraic effects. Instead the scope that a scoped operation affects should be delimited by syntax.

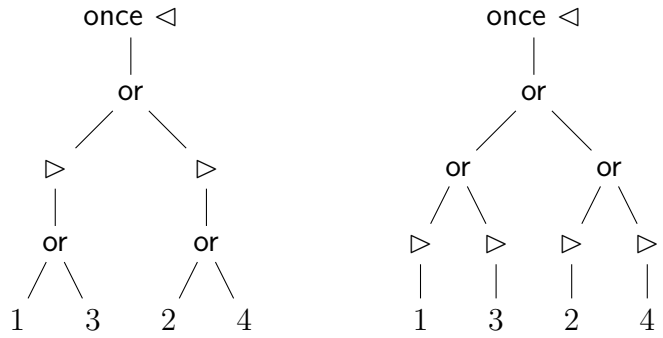
An example of a scoped operation will be given in the context of pruning non-deterministic computations. The operation $\mathsf{or}(a, b)$ was introduced in the background to model a computation that could be either a or b. The operation once is a scoped operation that will choose the first alternative. For instance, $\mathsf{once}(1)$ results in 1, since there is only one alternative to choose from. Whereas $\mathsf{once}(\mathsf{or}(1, 2))$ will result in 1, since this is the first alternative. These can be represented by the following trees.



In the trees above, $\triangleleft$ is used to indicate the start of a scope, and $\triangleright$ is used to indicate the end of a scope. Consider the inequality below between two computations using or and once. The right-hand side results in 1 out of a choice between four alternatives. The left-hand side results in $\mathsf{or}(1, 3)$. The reason is that 1 is chosen from $\mathsf{or}(1, 2)$, and then this is passed on to the continuation that produces $\mathsf{or}(1, 3)$. Note that if once were an algebraic operation, then it would have to commute with bind, and so the two sides would have to be equal. The fact that the two expressions are not equal, shows that this is not the case.

$$\mathsf{once}(\mathsf{or}(1, 2)) \gg\!= \lambda x.\ \mathsf{or}(x, x + 2) \not\equiv \mathsf{once}(\mathsf{or}(\mathsf{or}(1, 3), \mathsf{or}(2, 4)))$$

To understand the difference between the two expressions, it is illustrative to look at the computation trees for them.



The expression on the left-hand side is represented by the computation tree on the left. The ▷ occurs before the second or operation on any path from the root to a leaf. This reflects the fact that the scope created by the once operation ends before the second or operation. Therefore once does not affect that or operation. The computation tree on the right, represents the expression on the right-hand side of the inequality. The ▷ that ends the once operation occurs after the second or operation on any path from the root to a leaf. Therefore once does take into account that or operation.

So far the background behind scoped operations in the literature has been summarised. The example of pruning non-deterministic computations with once has been given to give a intuitive understanding of how scoped operations create a scope that can change the behaviour of other operations.

3.4.2 Development

One way of modelling scoped operations in normal algebraic effects would be to use a new type constructor that would represent the scoped operations. The type constructor would take no arguments, and its data constructors would be the possible scoped operations. Then *Freer* would take this type constructor as another argument g. There would also be a new data constructor *Scope* in freer, for scoped operations. This data constructor would have three arguments: the scoped operation, a computation tree representing the inner scope; and a computation tree representing the continuation.

```
data Freer f g a where
  Pure  :: a → Freer f g a
```

$$
\begin{aligned}
&\mathit{Impure} :: (b \to \mathit{Freer}\ f\ g\ a) \to f\ b \to \mathit{Freer}\ f\ g\ a \\
&\mathit{Scope}\ \ :: g \to \mathit{Freer}\ f\ g\ () \to \mathit{Freer}\ f\ g\ a \to \mathit{Freer}\ f\ g\ a
\end{aligned}
$$

Notice that the continuation does not take a parameter and that the output type of the computation tree representing the inner scope is the unit type. For some effects, it might be desirable for the inner scope to produce output of a particular type, or give the continuation a parameter of a particular type. It may be the case that these two should be related. They might even be the same, e.g. if the output of the inner scope should be passed to the continuation. In some cases, the type passed to the continuation might be dependent on, but not the same as, the output type of the inner scope, e.g. for a future.

To handle this the type constructor representing the inner scope can be given two type parameters representing the output type of the inner scope and the parameter to be passed to the continuation. The type constructor is then free to constrain these two type parameters as it wishes.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Freer}\ f\ g\ a\ \textbf{where} \\
&\quad \mathit{Pure}\ \ \ :: a \to \mathit{Freer}\ f\ g\ a \\
&\quad \mathit{Impure} :: (b \to \mathit{Freer}\ f\ g\ a) \to f\ b \to \mathit{Freer}\ f\ g\ a \\
&\quad \mathit{Scope}\ \ :: g\ x\ x' \to \mathit{Freer}\ f\ g\ x \to (x' \to \mathit{Freer}\ f\ g\ a) \to \mathit{Freer}\ f\ g\ a
\end{aligned}
$$

The question that arises is how to adapt this for parameterised effects. The data constructor *Scope* involves three different computation trees. There are two computation trees given as part of the input to the data constructor: the inner scope, and the continuation. There is one computation tree which is the output of the data constructor. This will be referred to as the overall tree. Each of these computation trees will have a pre and postcondition.

There are two questions that arise: how should these type parameters be related to each other; and should any of these be constrained by the scoped operation i.e. passed as input to g. To understand which choices to make, it is useful to look at the *Impure* data constructor in *IFreer*, where there is two computation trees: the continuation, and the overall tree.

Note that in the *Impure* data constructor, the postcondition of the continuation is the postcondition of the overall tree. This has type parameter r, and it is not passed as input to f, i.e. it is not constrained by the operation. This makes sense since a non-scoped operation shouldn't affect what happens after the operation is finished. This holds for scoped operations too.

```
data IFreer f g p q x where
  Pure   :: a → Free f p p a
  Impure :: f p q b → (b → f q r a) → Free f p r a
```

This leaves four type parameters to consider: the precondition of the overall tree, the pre and postcondition of the inner scope, and the precondition of the continuation. In other words: the effect state before entering the inner scope, the effect state after entering the inner scope, the effect state before exiting the inner scope, and the effect state after exiting the inner scope.

One approach to handle this simply would be to argue that neither entering nor exiting the inner scope should change the effect state. Hence the precondition for the overall tree and the inner scope should be the same, as well as the postcondition for the inner scope and the precondition to the continuation.

However, we choose not to assume this to be the case, and to err on the side of allowing scoped operations to be more expressive. It will turn out that allowing the effect state to change when entering and exiting the scope will allow very interesting examples to be modelled. This leads to the following data constructor *Scope* in *IFreer*.

```
data IFreer f g p q x where
  Scope :: g p p' q' q x x' → IFreer f g p' q' x
        → (x' → IFreer f g q r a) → IFreer f g p r a
```

As opposed to *Impure* which has three indices representing the effect state, *Scope* has five: p, p', q', q and r. The parameter r is the postcondition of the continuation and the overall tree, discussed earlier. The parameters p, p', q' and q represent the effect state before entering the scope, after entering the scope, before exiting the scope and after exiting the scope respectively. For convenience, these will be referred to as the outer precondition, inner precondition, inner postcondition and outer postcondition.

All four can be constrained by the scoped operation g, meaning g now takes six type parameters rather than two in *Freer*. The parameters p' and q' are the pre and postcondition for the inner scope. The parameters p and q are the type parameters for the precondition of the overall tree and the precondition of the continuation.

Since the type constructor g has six type parameters, four of which being effect states, the type constructor g has kind *Scope* k, where k represents the parameterising category. This type synonym is given below.

```
type Scope k = k → k → k → k → Type → Type → Type
```

Now effects consist of two parts: a constructor for normal (non-scoped) operations, with kind *Operation k*; and a constructor for scoped operations, with kind *Scope k*. It is convenient to be able to define a single type *Effect k* that combines these two.

```
data Effect a where
  MkEffect :: Operation a → Scope a → Effect a
```

Several changes need to be made to the existing code for defining coproducts, in order to accommodate this. First the data type for an effect chain *HListEffect* is defined in terms of *Effect*, rather than *Operation*. Second, a *CoproductScope* is written that is analogous to the *Coproduct* for operations. Third, these two are combined to write *CoproductEffect*. Fourth, a function *injS* is added to the *MemberAux* typeclass, for injecting scoped operations. Finally, a function *injectS* is written, that is analogous to *inject*.

To conclude this section, we've discussed the different effect states involved with scoped operations, and how this implementation of scoped operations allows the effect state to change when entering a scope (i.e. from the outer precondition to the inner precondition) and when exiting a scope (i.e. from the inner postcondition to the outer postcondition). The changes to *IFreer*, the data type that represents computation trees have also been explained.

3.4.3 Examples

We shall consider two simple scoped operations that might be used in the storage effect. The first is *local*, which creates a local scope, that allows resources from the outer scope to be used, but that does not allow resources in the inner scope to be accessible outside. This is analogous to how variables interact with scopes in typical imperative programming languages. The second is *isolate*, which creates a completely isolated scope, where variables outside cannot be accessed inside.

```
type StorageScope :: Scope StorageState
data StorageScope p p' q' q x x' where
  Local :: ((q, rest) ~ Split q' (Len p), AllZero rest)
        ⇒ StorageScope p p q' q x x
```

Observe that there is no change from the outer precondition to the inner precondition. Both effect states are p reflecting the fact that resources defined outside an inner scope, can be accessed within the scope. However, the inner postcondition q' is modified before becoming the outer postcondition q. This is because only the resources that were present before the scope began are kept.

The *Split* type family takes the context q' and splits it into two contexts q and *rest*, with q being the resources that were present beforehand and *rest* being the resources that were added by the scope. There is an additional constraint *AllZero* which ensures that the newly added resources in *rest* have been completely used.

Consider the example below of a use of the *local* operation. A resource i is constructed outside the scope, and another resource j is constructed inside the scope. Both resources are used inside the scope. The output of the inner scope is the output of the entire scoped operation, and consequently of *example*, since *local* is the final operation.

```
example = run ∘ runLinear do
  i ← register 10
  local do
    i' ← use i
    j  ← register 4
    j' ← use j
    return (i' + j')
```

Now consider the following example of code that would fail to compile. Here the inner scope only registers the resource j and the output of the inner scope is a resource j which is passed to the outer scope. However, attempting to obtain the value of j will fail, since it is no longer present.

```
example = run ∘ runLinear do
  i  ← register 10
  j  ← local (register 4)
  i' ← use i
  j' ← use j
  return (i' + j')
```

By contrast, the *isolate* operation is defined slightly differently. Since the resources registered outside are not accessible in the inner scope, a type

family *RemoveResources* is used to iterate through the resources and set the number of instances to zero. Moreover like *local*, the resources introduced in the inner scope are not accessible outside, so the outer postcondition and outer precondition are the same. They are both given by the type variable p. There is no need to *Split* the inner postcondition q', since all resources should be zero, including those introduced out of the scope, owing to the use of *RemoveResources*.

```
data StorageScope p p' q' q where
  Isolate :: (AllZero q')
          ⇒ StorageScope p (RemoveResources p) q' p x x
```

To conclude this section, we've seen examples of how the effect state can change when entering and exiting a scope, and how these constraints can be encoded as a data type.

3.4.4 Dual Scoped Operations

One might consider whether it might be desirable to have operations consisting of more than one inner scope. In the next chapter, an example will be given where an operation with two inner scopes is needed, which will be called a dual scoped operation. Dual scoped operations work in the same way as scoped operations.

There are two inner scope preconditions, two inner scope postconditions, and two inner scope output types. One is for the first inner scope, and another for the second inner scope. This together with the effect state before entering and after exiting a scope lead to a total of nine type parameters.

```
type DualScope a
  = a → a → a → a → a → a → Type → Type → Type
```

Adding dual scoped operations requires similar changes to the ones that adding scoped operations required. For instance, *Effect* is given an additional parameter, *IFreer* also needs an additional type parameter, the definition of coproducts is extended and injection functions are added.

```
type ITree a ps qs x
  = IFreer (GetOperation a) (GetScope a) (GetDualScope a) ps qs x
```

Since *IFreer* takes three type constructors now, a type synonym is defined called *ITree*. The *ITree* type synonym takes an effect, finds the data types representing the operations, the scoped operations and the dual scoped operations, and constructs the *IFreer* type.

3.5 Type Checker Plugin

In this section, the issue of type inference is considered. First we show that Haskell's partial type signatures feature allows type signatures to be inferred for functions, where handlers are not invoked. Then the complexity of the inferred type signature is discussed along with how it could be simplified. This motivates the introduction of a type-checker plugin. Then the various parts of the plugin are explained.

3.5.1 Motivation

There is one major shortcoming so far though. All the examples of the storage effect that have successfully compiled have been expressions that invoked the handlers *run* and *runLinear*. Consider the examples *example* and *example′*. They are almost identical, except *example* evaluates the computation tree, but *example′* does not. While *example* will compile, *example′* will fail to compile. The problem is due to ambiguous type variables. Firstly, the use of *run* and *runLinear* constrains the effect chain, to a concrete expression with no type variables, i.e. $\mathit{StorageOperation} \ominus \mathit{Nil}$. Secondly, the use of *runLinear* constrains the precondition to the concrete expression $[\,] \oplus \mathit{Unit}$. Without this information, there would be too many ambiguous type variables for Haskell to deduce a type.

$$
\begin{aligned}
&\mathit{example} = \mathit{run} \circ \mathit{runLinear}\ \mathbf{do}\\
&\quad i \leftarrow \mathit{register}\ 10\\
&\quad j \leftarrow \mathit{use}\ i\\
&\quad \mathit{return}\ j\\
&\mathit{example}' = \mathbf{do}\\
&\quad i \leftarrow \mathit{register}\ 10\\
&\quad j \leftarrow \mathit{use}\ i\\
&\quad \mathit{return}\ j
\end{aligned}
$$

In fact, the situation is more nuanced than that. The expression can, in fact, be made to type-check by specifying a type signature. The problem is that the type signature needed to make this expression type check is too long to write out manually. It is possible to get Haskell to derive this signature for itself, using the partial type signatures extension (Winant et al., 2014). This extension allows you to specify a type signature with gaps called 'holes' that are indicated with underscores. GHC will endeavour to find the type that is appropriate for the hole. In the extreme case, you can use it to find the entire signature.

$$example' :: (\_) \Rightarrow \_$$

A partial type signature is given above to deduce the entire signature of $example'$. There are two holes in the type signature, one for any constraints, and one for the actual type. From the partial type signature above, GHC derives the type signature given below. While this is an improvement from the user having to specify the type signature manually, the complexity of the type signature for such a simple example leaves much to be desired.

$$\begin{aligned}
&example' \\
&\quad :: (Num\ t, MemberAux\ (FindEffect\ LinEffect\ u)\ LinEffect\ u, \\
&\qquad Contains\ (FindEffect\ LinEffect\ u)\ a\ i, \\
&\qquad Contains\ (FindEffect\ LinEffect\ u)\ a_1\ j, \\
&\qquad Evolve\ i\ (FindEffect\ LinEffect\ u)\ (Append\ a\ (t, S\ Z))\ j, \\
&\qquad Evolve\ j\ (FindEffect\ LinEffect\ u) \\
&\qquad (Replace\ a_1\ (Len\ a)\ (b, Sub\ (S\ Z)\ k))\ k, \\
&\qquad Lookup\ a_1\ (Len\ a) \sim (b, k)) \\
&\quad \Rightarrow ITree\ u\ i\ k\ b
\end{aligned}$$

Let us examine the code and the deduced type signature to understand where the complexity comes from. The code registers a value of type t in the context with one copy, then removes this copy, and returns the value. The type variable i is the combined state at the beginning of the function, and the type variable a is the effect state for the storage effect. This is shown in the first *Contains* constraint. The type variable j refers to the combined state, after a new resource is stored in the context.

The difference between i and j is that a becomes $Append\ a\ (t, S\ Z)$. This is shown in the first *Evolve* constraint. The type variable a_1 refers

to the effect state in the combined state j corresponding to the storage effect. This is shown in the second *Contains* constraint. The value a_1 is equal to *Append* a $(t, S\ Z)$. However, Haskell is not able to deduce that a_1 is equal to *Append* a $(t, S\ Z)$. If it were, this would remove the second *Contains* constraint defining a_1, and give a context where a_1 is replaced with *Append* a $(t, S\ Z)$:

$$
\begin{aligned}
&\mathit{example}' \\
&\quad :: (\mathit{Num}\ t, \mathit{MemberAux}\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u)\ \mathit{LinEffect}\ u, \\
&\qquad \mathit{Contains}\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u)\ a\ i, \\
&\qquad \mathit{Evolve}\ i\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u)\ (\mathit{Append}\ a\ (t, S\ Z))\ j, \\
&\qquad \mathit{Evolve}\ j\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u) \\
&\qquad (\mathit{Replace}\ (\mathit{Append}\ a\ (t, S\ Z))\ (\mathit{Len}\ a)\ (b, \mathit{Sub}\ (S\ Z)\ k))\ k, \\
&\qquad \mathit{Lookup}\ (\mathit{Append}\ a\ (t, S\ Z))\ (\mathit{Len}\ a) \sim (b, k)) \\
&\quad \Rightarrow \mathit{ITree}\ u\ i\ k\ b
\end{aligned}
$$

The type variable k is the combined state that is formed from the combined state j, by replacing the effect state corresponding to the storage effect, which is *Append* a $(t, S\ Z)$ by *Append* a (t, Z). This is because one instance is used, so $S\ Z$ becomes Z. Haskell encodes this with the second *Evolve* constraint, but the expression that Haskell uses for the new effect state is more complicated than *Append* a (t, Z), even with a_1 solved. One reason for this is the *Lookup* constraint that defines b and k.

Haskell cannot deduce that (b, k) is equal to $(t, S\ Z)$. This is problematic since both b and k are found in *Replace*. Haskell does not intuitively understand that looking up the resource state in *Append* a $(t, S\ Z)$ at index *Len* a, does not require one to know the value of a at all, and is in fact $(t, S\ Z)$. If Haskell were able to deduce this, the type signature would look as follows.

$$
\begin{aligned}
&\mathit{example}' \\
&\quad :: (\mathit{Num}\ t, \mathit{MemberAux}\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u)\ \mathit{LinEffect}\ u, \\
&\qquad \mathit{Contains}\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u)\ a\ i, \\
&\qquad \mathit{Evolve}\ i\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u)\ (\mathit{Append}\ a\ (t, S\ Z))\ j, \\
&\qquad \mathit{Evolve}\ j\ (\mathit{FindEffect}\ \mathit{LinEffect}\ u) \\
&\qquad (\mathit{Replace}\ (\mathit{Append}\ a\ (t, S\ Z))\ (\mathit{Len}\ a)\ (t, Z))\ k) \\
&\quad \Rightarrow \mathit{ITree}\ u\ i\ k\ t
\end{aligned}
$$

Even with this, Haskell would not fully be able to simplify the second *Evolve* constraint, involving k. Similar to the problem with *Lookup*, Haskell

is not able to deduce that if you replace the resource state at index *Len a* by (t, Z) in the context *Append a* $(t, S\ Z)$ you do not need to know the value of a, to obtain the result *Append a* (t, Z). If this were done, then the type signature would be as follows.

$$
\begin{aligned}
&example' \\
&\quad :: (Num\ t, MemberAux\ (FindEffect\ LinEffect\ u)\ LinEffect\ u, \\
&\qquad Contains\ (FindEffect\ LinEffect\ u)\ a\ i, \\
&\qquad Evolve\ i\ (FindEffect\ LinEffect\ u)\ (Append\ a\ (t, S\ Z))\ j, \\
&\qquad Evolve\ j\ (FindEffect\ LinEffect\ u)\ (Append\ a\ (t, Z))\ k) \\
&\quad \Rightarrow ITree\ u\ i\ k\ t
\end{aligned}
$$

The type signature can be further improved still. The *Evolve* constraint involving k, can be defined in terms of i rather than j. The reason for this is that in going from i to j and j to k the same effect state is modified. The previous modifications can therefore be forgotten. This yields the following type signature.

$$
\begin{aligned}
&example' \\
&\quad :: (Num\ t, MemberAux\ (FindEffect\ LinEffect\ u)\ LinEffect\ u, \\
&\qquad Contains\ (FindEffect\ LinEffect\ u)\ a\ i, \\
&\qquad Evolve\ i\ (FindEffect\ LinEffect\ u)\ (Append\ a\ (t, S\ Z))\ j, \\
&\qquad Evolve\ i\ (FindEffect\ LinEffect\ u)\ (Append\ a\ (t, Z))\ k) \\
&\quad \Rightarrow ITree\ u\ i\ k\ t
\end{aligned}
$$

There is one more simplification that can still be performed. It should be noted that the variable j only appears in the first *Evolve* constraint. Therefore there is no need for j to even appear in the type signature. This *Evolve* constraint will limit j to exactly one possible value, and since there are no other constraints on j, this constraint cannot fail to hold. It can therefore be removed from the signature leading to the following.

$$
\begin{aligned}
&example' \\
&\quad :: (Num\ t, MemberAux\ (FindEffect\ LinEffect\ u)\ LinEffect\ u, \\
&\qquad Contains\ (FindEffect\ LinEffect\ u)\ a\ i, \\
&\qquad Evolve\ i\ (FindEffect\ LinEffect\ u)\ (Append\ a\ (t, Z))\ k) \\
&\quad \Rightarrow ITree\ u\ i\ k\ t
\end{aligned}
$$

Brief consideration will be given to why performing these simplifications is beneficial. For one, complicated type signatures make it harder for a programmer to understand the type of a function. Furthermore, complicated

type signatures can lead to error messages that are more difficult to understand. Indeed, they could mask the fact that there is an error at. There might be no way to satisfy the constraints of the function, but this can only be discovered by simplifying constraints to obtain a contradiction. In addition, the level of unneeded complexity of the deduced type signatures increases significantly as the code becomes more complicated. This is exacerbated by the fact that a function 'passes on' its constraints to the function that calls it.

In order to drastically simplify constraints, a type-checker plugin is introduced. GHC offers different interfaces to modify its internal behaviour (Pickering et al., 2019). A recent addition is the type-checker plugin interface, which allows type-checker plugins to modify the constraint solver. The type *TcPlugin* that defines a type-checker plugin is given below. Most of the work occurs in the *tcPlugin* function that is called by GHC. GHC provides the function with a list of given constraints, which are constraints that are known to hold and a list of wanted constraints, that GHC has yet to prove.

$$
\begin{aligned}
&\textbf{data}\ \mathit{TcPlugin} = \forall s.\ \mathit{TcPlugin}\\
&\quad \{\, \mathit{tcPluginInit}\ \ :: \mathit{TcPluginM}\ s\\
&\quad ,\ \mathit{tcPluginSolve} :: s \to \mathit{TcPluginSolver}\\
&\quad ,\ \mathit{tcPluginStop}\ \ :: s \to \mathit{TcPluginM}\ ()\\
&\quad \}
\end{aligned}
$$

To conclude this section, we have seen how partial type signatures are both a blessing, in that they mean the user does not manually have to write out a complex type signature, and a curse in that the feature generates overly complicated type signatures. We have also proposed a solution to this in the form of a type-checker plugin, that will be explored in the following sections.

3.5.2 Problem of Cycles

In this section, the problem of cycles created by *Evolve* constraints is introduced and it is shown how this complicates type-checking. A solution is presented in the form of *Ref* constraints.

In the previous section, there were two *Evolve* constraints. One from the combined state i to the combined state j, and another from the combined state j to the combined state k. These two *Evolve* constraints are simplified into a single *Evolve* constraint from i to k.

One can imagine *Evolve* constraints as describing a directed graph showing how the combined state changes during the course of the function. In the case above, there would be three nodes i, j and k, with edges from i to j and j to k. This graph has an easily observable start and end point. A problem arises if there exists a cycle in the graph, then it becomes more difficult to see which combined state the function starts in, and which combined state the function ends in.

It is, of course, possible to find this out by looking at the type parameters of *ITree*. The problem is that the type-checker plugin is not given this information by GHC. All that the type-checker plugin is given is a list of constraints. In some cases, there should be an unambiguous start and end point inferrable from the graph, but this is not the case if there is a cycle.

Without a clear start and end point, two problems arise. The first optimisation that is performed with *Evolve* consists of replacing the 'from' combined state in terms of the combined state, that the function begins with. For instance the *Evolve* constraint from j to k is rewritten to i to k. Without a clear initial state, this is not possible.

The second optimisation that is performed with *Evolve* is removing unneeded *Evolve* constraints. An *Evolve* constraint is removed if it begins in the initial state, but is not used to define the final state of the function. For instance, the *Evolve* constraint from i to j is no longer needed, and is removed. This leaves only the constraint from i to k. Without a clear start and end point this optimisation cannot be performed.

The way to get around this is to introduce a 'dummy' constraint called *Ref* that has no other purpose but to signal to the type-checker the start and end state of a function. A constraint *Ref ps qs* means that the function starts in combined state *ps* and ends in combined state *qs*. The constraint *Member* will be rewritten to include *Ref*.

```
type Member l u p ps i qs
  = (i ~ FindEffect l u, MemberAux i l u
    , Contains i p ps, Ref ps qs)
```

This information can then be given to the type-checker plugin by including a *Member* constraint in the partial type signature, as below. The *Ref* constraint will then be passed to the type-checker plugin as part of the list of given constraints.

example′

$$:: (Member\ l\ u\ p\ ps\ i\ qs, \_) \Rightarrow ITree\ u\ ps\ qs\ ()$$

Even with the addition of this information, the partial type signature is still straightforward to write. Moreover, the presence of the *Ref* constraint means that the type-checker plugin is able to do a significantly better job.

3.5.3 Design

In this section, a high-level view of the architecture of the type-checker plugin will be given. The data structures used internally by the plugin will be explained. Then the different stages that comprise the plugin will be discussed.

GHC gives the type-checker plugin a list of given constraints and a list of wanted constraints in its own internal representation, defined in the GHC source code. Constraints are encoded by the *Ct* data type, types are encoded by the *Type* data type, type constructors are encoded by the *TyCon* data type and variables are encoded by the *Var* data type.

In order to make it easier to perform optimisations on the constraints, the type-checker plugin converts GHC's internal representation into a custom tree representation. There are two advantages to this custom representation. Firstly, the representation makes it easier to use pattern matching to differentiate between different types of constraints as well as different type families. Secondly, the representation has special representations for certain patterns, e.g. consecutive appends of new resource states to the end of a context.

Three data types are used for this custom representation: *OpConstraint*, *OpType* and *OpFun*. The data type *OpConstraint* has constructors for different types of constraints: equality between two types, *Contains* constraints, *Evolve* constraints and *Ref* constraints. Constraints given by other type-classes, are ignored by our plugin. Giving each type of constraint its own data constructor is how the custom tree representation makes it easier to differentiate between constraints.

```
data OpConstraint
  = OpEquality OpType OpType Bool
  | OpContains Var Var OpType OpType OpType Bool
  | OpEvolve Var Var OpType OpType OpType OpType Bool
  | OpRef Var Var
```

The data type *OpType* encodes types. It plays an analogous role to *Type* in GHC's internal representation, and has similar constructors for variables and type applications. There is also a catch-all constructor in *OpType* for constructors that exist in *Type* but not *OpType*. There are also constructors in *OpType* for special patterns. For instance, the data constructor *OpMultiAppend* encodes consecutive appends of new resource states to the end of a context.

```
data OpType
  = OpApp OpFun [OpType]
  | OpVar Var
  | OpLift Type
  | OpMultiAppend OpType [OpType]
  | ...
```

The data type *OpFun*, used by *OpType*, encodes a type constructor. It performs an analogous role to GHC's internal representation for type constructors, *TyCon*. The use of *OpFun* is how the tree representation makes it easier to differentiate between different type families. There is also a catch-all constructor in *OpFun* to encode other type constructors, that simply wraps around *TyCon*.

```
data OpFun
  = OpReplace
  | OpLookup
  | OpLen
  | OpAppend
  | ...
```

Now a description will be given of the different stages that make up the type-checker plugin. The type-checker plugin begins by going through each of the wanted constraints given by GHC and converting them into the custom tree representation if this is possible. If this is not possible, the constraint is ignored.

The type-checker plugin also does the same thing with the given constraints. The given constraints are used for two purposes. Firstly, to populate a 'database' that maps type variables to associated information. Secondly, to identity if the list of given constraints contains a *Ref* constraint. The type-checker plugin expects at most one *Ref* constraint; if there are multiple, they

are simply ignored. If there isn't one, the type-checker plugin can still run, but not all optimisations can be performed.

After this, a fixed-point algorithm runs. The algorithm takes a list of constraints to be solved, and goes through each constraint in an attempt to simplify the constraint. It keeps on doing this until no change occurs in the set of constraints. The algorithm has four parts:

1. The first part consists of variable substitutions, that are performed using the database of information about variables. Variables in constraints are replaced if the database can be used to identify the value that they hold.

2. The second part consists of the simplification of constraints. For equalities, *Contains* constraints and *Evolve* constraints, there are distinct rules that are applied to simplify the constraints.

3. The third part consists of iterating through the constraints and breaking them up into smaller constraints if possible. This is possible if the constraints involve an injective type constructor. For instance, if $(a, b) \sim (x, y)$ this can be used to deduce $a \sim x$ and $b \sim y$.

4. The fourth part consists of iterating through constraints, and checking if there is any new information about variables that can be added to the database.

After the fixed-point algorithm is terminated, the plugin only keeps the constraints that have been simplified. For the constraints that have been modified, the original versions of the constraints that were given to the plugin are given back to the GHC as constraints that have been solved. Then the simplified versions of the constraints are given to GHC as constraints that need to be solved. Since constraints can be broken down into multiple constraints during the course of the type-checker plugin, some book-keeping is required to track how constraints change over time.

3.5.4 Simplifications

So far we have given the motivation behind the type-checker plugin and a high-level idea of how it works. This section will go through the three different types of constraints: equalities; *Contains* constraints; and *Evolve*

constraints. The simplifications that can be performed on the constraints during the fixed point algorithm are explored.

The first type of constraint that will be considered is a *Contains* constraint. If a *Contains* constraint specifies that the combined state ps contains effect state p at index i, and there is an *Evolve* constraint the specifies that the combined state ps is defined by taking some other combined state and replacing the effect state at index i with p_1, then it can be deduced that $p \sim p_1$. This is implemented by checking for an *Evolve* constraint that defines ps in the database. This optimisation results in the *OpContains* constraint being replaced by an *OpEquality* constraint.

The second type of constraint that will be considered is an *Evolve* constraint. There are two possible optimisations. If an *Evolve* constraint defines a combined state k in terms of a combined state j, and that combined state j is defined using another *Evolve* constraint from the initial state i, such that the same effect state is changed, then the *Evolve* constraint can define k in terms of the initial state i. This is done by looking for the *Evolve* constraint that defines j in the database.

In addition, if an *Evolve* constraint is used to define a combined state, that is not the terminal state of the function, in terms of the initial state, then this constraint can be removed. This is done by simplifying the constraint to a special 'trivial' constraint.

The final type of constraint to be considered is an equality. Simplification is done by traversing the expressions contained in the equality, identifying patterns that could be potentially optimised, performing checks to verify the optimisation would be valid, and finally producing the new expression. This is carried out in a bottom-up fashion, so that sub-expressions are optimised first, before an attempt is made to optimise the parent expression.

Simplifications broadly fit into two types. The goal of the first type is not to meaningfully change the actual Haskell type being represented, but to provide a 'neater' representation. Some of these simplifications are given below in the form of identities.

$$
\begin{aligned}
&\mathit{OpAppend}\ x\ y && \equiv \mathit{OpMultiAppend}\ x\ [y] \\
&\mathit{OpAppend}\ (\mathit{OpMultiAppend}\ x\ ys)\ y && \equiv \mathit{OpMultiAppend}\ x\ (ys +\!\!+ [y]) \\
&\mathit{OpLen}\ x && \equiv \mathit{OpShift}\ x\ 0 \\
&\mathit{OpShift}\ (\mathit{OpMultiAppend}\ x\ ys)\ n && \equiv \mathit{OpShift}\ x\ (n +\!\!+ \mathit{length}\ ys) \\
&\mathit{OpSucc}\ (\mathit{OpShift}\ x\ n) && \equiv \mathit{OpShift}\ x\ (n+1) \\
&\mathit{OpShift}\ (\mathit{OpReplace}\ xs\ i\ x) && \equiv \mathit{OpShift}\ xs\ n
\end{aligned}
$$

The first two of these rules deal with the constructor *OpMultiAppend* in *OpType*. These two rules use *OpMultiAppend* to compress multiple appends to a list into a single data type. A value *OpMultiAppend x xs* represents a list of values *xs*, that are being appended to a context *x*. The reason this is useful is that during a function, multiple new resources could be added to a context.

The next four of these rules deal with the constructor *OpShift* in *OpType*. These four rules use *OpShift* to represent a constant number added to the length of a list. A value *OpShift x n* represents the length of the list *x* added to the constant *n*. This is useful because as you add resources to a context, the natural number used to index these additional resources, will be defined as the length of a list, consisting of the original list and possibly some appended elements.

$$
\begin{aligned}
&\mathit{OpLookup}\ (\mathit{OpMultiAppend}\ a\ xs)\ (\mathit{OpShift}\ a\ i) \\
&\quad \equiv xs\ !!\ i \\
&\mathit{OpReplace}\ (\mathit{OpMultiAppend}\ a\ xs)\ (\mathit{OpShift}\ a\ i)\ x \\
&\quad \equiv \mathit{OpMultiAppend}\ a\ (\mathit{replace}\ xs\ i\ x)
\end{aligned}
$$

The goal of the other type of simplifications is to make a meaningful change to the Haskell type being represented. Two examples are given above. Both take advantage of *OpMultiAppend* and *OpShift*. The first rule allows you to find the resource state within a context, if the context is constructed with one or more appends to the end of a smaller context, and the index is defined as the length of the smaller context incremented by a constant, i.e. the index will be one of the items appended.

The second rule allows a new effect state to be found by replacing the resource state in the original effect state, with some other value. It relies on the original effect state being made from a smaller context with one or more appends at the end, and the index to be replaced being one of those appended items, i.e. the length of the smaller context incremented by some constant.

3.5.5 Optimising Map Expressions

In this section, the work on scoped operations and the type-checker plugin is combined, to allow for optimisations that work on a wider range of scoped operations.

Previously it was shown how scoped operations could control how the context of resources changes as the scope is entered and exited. In the case of entering a scope, this typically involves a function that iterates through each element of the context before the scope is entered, and performs the same function to each item. It would be convenient if this could be encoded using higher-order type-level functions, for instance a type-level *map* function. Not only would this be good for readability, but it would help the type-checker plugin. The type-checker plugin would be able to understand how the type-level *map* function would interact with other type-level functions such as *Append* and *Replace* and be able to optimise expressions accordingly.

Unfortunately Haskell doesn't allow higher-order type-level programming directly, as this requires unsaturated type families (Kiss et al., 2019), which is not currently supported by Haskell. However, it is possible to emulate higher-order functions based on a technique called defunctionalisation. This works by associating the function with a constructor in a data type. This constructor will be referred to as a symbol. For instance, suppose you wanted a function that incremented each natural number in a list by one.

$$\begin{aligned}&\textbf{data}\ \mathit{Fun}\ a\ \textbf{where}\\&\quad \mathit{IncrementByOne} :: \mathit{Fun}\ \mathit{Nat}\end{aligned}$$

The symbol for this function is *IncrementByOne*, which belongs to the data type $\mathit{Fun}\ a$. The data type doesn't matter, but the type parameter does. In this case, it is *Nat* and it states what type, the type-level function will operate on.

Then there is a type family called *Apply*, which allows the functions corresponding to symbols to be defined.

$$\begin{aligned}&\textbf{type}\ \mathit{Apply} :: \forall k\ a.\ k\ a \to a \to a\\&\textbf{type}\ \mathit{family}\ \mathit{Apply}\ f\ x\\&\textbf{type instance}\ \mathit{Apply}\ \mathit{IncrementByOne}\ n = S\ n\end{aligned}$$

Finally there is a type-family called *Map*. Rather than taking a type-level function directly, it takes a symbol associated with the type-level function and uses it to indirectly call the function using *Apply*. This means *Map IncrementByOne* can be used to increment each natural number in a list by one.

$$\begin{aligned}&\textbf{type}\ \mathit{Map} :: \forall k\ a.\ k\ a \to [a] \to [a]\\&\textbf{type}\ \mathit{family}\ \mathit{Map}\ f\ \mathit{xs}\ \textbf{where}\end{aligned}$$

$$
\begin{aligned}
&\mathit{Map}\ f\ [\,] = [\,] \\
&\mathit{Map}\ f\ (x : xs) = (\mathit{Apply}\ f\ x) : (\mathit{Map}\ f\ xs)
\end{aligned}
$$

Now consider the case where a scope is exited. This typically involves a function that uses each resource state of the context before the scope is exited, and potentially the corresponding resource state in the original context before the scope was even entered to produce a new resource state. It is typically the case that if a resource state in a context wasn't modified in the scope, then the resource state after the scope exits will be the same as the resource state, before the scope was entered.

In other words, it is a typical property that if the resource state, as the scope is about to be exited, is the same as it was, when the scope was entered, then the function to modify the resource state as it exits the scope cancels out with the function to modify the resource state as it enters the scope, so the resource state after exiting the scope will be the same as the resource state before entering a scope.

For these functions, there is another type family called *Reverse*. If a symbol s has a function that will produce x from $\mathit{Apply}\ s\ x$, given x and the output of $\mathit{Apply}\ s\ x$, then this function can be encoded as $\mathit{Reverse}\ s$. The type-checker plugin then knows to optimise $\mathit{Reverse}\ s\ x\ (\mathit{Apply}\ s\ x)$ to simply x.

There is a corresponding *MapR* function that is applied to two lists. It is intended to be used for two contexts, one before a scope is entered and another before a scope is exited. Again the type-checker plugin understands both how to optimise *MapR* if it is applied to *Map*, and how to optimise *MapR* with respect to other type-level functions such as *Append* and *Replace*.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Reverse} :: \forall k\ a.\ k\ a \to a \to a \to a \\
&\textbf{type}\ \mathit{family}\ \mathit{Reverse}\ f\ x\ y \\
&\textbf{type}\ \mathit{MapR} :: \forall k\ a.\ k\ a \to [a] \to [a] \to [a] \\
&\textbf{type}\ \mathit{family}\ \mathit{MapR}\ f\ xs\ ys\ \textbf{where} \\
&\quad \mathit{MapR}\ f\ [\,]\ \_ = [\,] \\
&\quad \mathit{MapR}\ f\ (x : xs)\ (y : ys) = (\mathit{Reverse}\ f\ x\ y) : (\mathit{MapR}\ f\ xs\ ys)
\end{aligned}
$$

To conclude this section, specific optimisations to improve type inference for scoped operations are introduced. These work by being able to identify type-level map expressions, and operations that 'cancel each other out'.

3.6 Alternative Designs

3.6.1 Injection of Effects

The typeclasses *Evolve* and *Contains* are used to relate combined states with effect states. The *Evolve* constraint states how a combined state could be used to form another combined state, by replacing the effect state at a particular index. The *Contains* constraint states that a particular effect state can be found at a given index in a combined state.

One unfortunate consequence of *Evolve* and *Contains* is that it means that the *inj* function used to inject an effect into a coproduct is not type-safe. This is not a major problem, since the *inj* function is not meant to be used directly. Instead the type-safe *inject* function that wraps around *inj* is used. However, one might imagine an alternative implementation of *Evolve* and *Contains* as type families rather than typeclasses.

For instance, there could be a type family that given the combined state, and a particular index, gives as output the effect state at that index. Then an equality between the effect state, that results from the type family, and another value could replace the *Contains* typeclass. A type family could also be used to replace the effect state at a particular index in a combined state, instead of the *Evolve* typeclass. An equality could similarly be used to equate the result from the type family with the new combined state.

In fact these constraints were originally encoded using type families rather than typeclasses in the original implementation of coproducts. The reason for switching to typeclasses, was that using type families meant that a type-checker plugin couldn't be used to perform simplifications.

Partial type signatures could be used to derive type signatures but instead of the generated signature consisting of *Evolve* and *Contains* constraints, a very long expression would be inferred directly in the postcondition of the computation. If type families are used, the variable representing the postcondition can directly be replaced with an expression that defines the variable in terms of a type family. There is no way for a plugin to integrate into the type-checking process to be able to simplify this generated expression.

Typeclasses on the other hand are constraints, and GHC will invoke the type-checker plugin to solve them. Therefore this gives the type-checker plugin the opportunity to simplify these constraints.

3.6.2 Encoding Operations

One decision was to use the freer monad representation rather than the free monad. One reason was the fact that the freer monad approach was conceptually more simple, and avoided having lots of *IFunctor* constraints, since the freer monad is always a functor. The other reason is that a parameterised version of the *Scope* data constructor is much easier to write in the style of *Freer* rather than *Free*.

3.6.3 Multiple Resources

An alternative representation for the state would be a type-level map between names and values. This would mean that resources are indexed by (hopefully) human-readable strings rather than a Peano representation of the natural numbers. The reason for choosing our representation, however, is that this would not be conducive to a natural style of programming in Haskell. Using numbers as pointers mean that the pointers can be given an identifier, and passed to functions as if they were normal values.

Chapter 4

Session Types

In this chapter, session types are implemented in the framework built. First the basic design is explained. Then more advanced features such as session delegation and recursion are added. Finally session types are used to give an evaluation of the framework built.

4.1 Development

Session types are a way of giving types to communication channels, between two processes (Dezani-Ciancaglini and de'Liguoro, 2010). Consider a protocol between a user and an ATM to withdraw money. The client might send the amount of money, that it wants to withdraw, and the server might respond by sending the balance. The type of the communication channel from the user's perspective is *Send Float (Recv Float End)*. The type of the communication channel from the ATM's perspective is *Recv Float (Send Float End)*. There is a symmetry between the user's and the ATM's types; they are called the *dual* of each other. It would be an error for the user to first try to receive data, or for the ATM to first try to send information. Session types are intimately related to linear logic (Caires and Pfenning, 2010), and we can encode session types using our framework.

Session types are enhanced by allowing processes to make different choices. For instance, the ATM might offer the choice to either withdraw or deposit money. This can be encoded as *Offer Deposit Withdraw*, where *Deposit* and *Withdraw* are session types for the respective cases. The user will have the type *Choice Deposit′ Withdraw′*, where *Deposit′* and *Withdraw′* are the du-

als of *Deposit* and *Withdraw* respectively. With *Choice*, it is the current process making the decision as to which of the two alternatives, should be taken. With *Offer*, it is the process being communicated with that makes the choice. We encode session types using the kind *ChannelType*.

```
data ChannelType where
  EndT    :: ChannelType
  SendT   :: a → ChannelType → ChannelType
  RecvT   :: a → ChannelType → ChannelType
  ChoiceT :: ChannelType → ChannelType → ChannelType
  OfferT  :: ChannelType → ChannelType → ChannelType
```

In our implementation, threads will be used instead of processes. We will allow new threads to be created using a *fork* operation. When a channel is constructed, the channel will be made accessible to the current thread and the next child thread to be created. Note that the channel cannot be used by the parent thread until it is passed to the child thread. We also need to make sure that the channel cannot be used elsewhere, and that the channel is used to completion by the two threads.

The effect state of the session effect will be a context of resource states for each channel. The resource state will be given by the type synonym *Channel*.

```
type Channel = Maybe (ChannelType, Bool)
```

There are three cases. If the resource state is *Nothing*, then this means the thread does not have access to the channel. It could be the case, that the channel has run till completion, or that the thread never had access to the channel in the first place. If the resource state is *Just* (c, True) it means that the current thread is the parent thread and that the channel has not yet been passed to the child thread. If the resource state is *Just* (c, False) it means that either the current thread is the parent thread and the channel has been passed to the child thread, or that the current thread is the child thread.

The type synonym *SesssionState* will refer to the effect state for the session effect.

```
type SessionState = [Channel]
```

The *fork* operation is defined as follows. Two higher-order maps will be defined using symbols. The *ScopeIn* symbol will be used to enter a scope,

and a *ScopeOut* symbol will be used to exit the scope. Both only depend on the state before entering the scope. If a resource state is transformed by entering a scope, and then exiting a scope, with nothing in between, this does not necessarily result in the original resource state. This is why *ScopeOut* isn't defined as the reverse of *ScopeIn* but as its own operation. Note that the *fork* operation has a constraint *MakeSureEmpty* that goes through the context, and ensures that it consists of solely *Nothing* values.

```
type SessionScope :: Scope SessionState
data SessionScope p p' q' q x x' where
  Fork
    :: (MakeSureEmpty q)
    ⇒ SessionScope p (Map ScopeIn p) q (Map ScopeOut p) () ()
```

As mentioned the symbol *ScopeIn* is used to define the context of resources at the beginning of the child thread. If the channel is *Nothing*, then it simply passes this on. If the channel is *Just* (t, False), then it also passes *Nothing* to the child thread, since two threads already have access to it. If the channel is *Just* (t, True), then it passes *Just* $(\mathit{Dual}\ t, \mathit{False})$ to it, as now the new thread will also have access to the channel.

```
data ForkScope a where
  ScopeIn  :: ForkScope Channel
  ScopeOut :: ForkScope Channel
type instance Apply ScopeIn (Just (p, True))
  = Just (p, False)
type instance Apply ScopeIn (Just (_, False))
  = Nothing
```

The symbol *ScopeOut* is used to define the context of resources for the parent thread after the *fork* operation. If the resource state isn't *Nothing*, it sets the second value of the pair to *False* regardless of its previous value. This is because if the previous value was *True*, then because a scope was just exited, that channel should have been passed to the scope.

```
type instance Apply ScopeOut (Just (p, _))
  = Just (p, False)
type instance Apply ScopeOut Nothing
  = Nothing
```

The type family *Dual* constructs the dual type and is defined as follows. An important thing to note is the injectivity annotation $o \to i$, which is possible thanks to the type family dependencies language extension (Stolarek et al., 2015). This improves type inference, since it means that the input to the type family can be derived from the output of the type family. In other words, Haskell can obtain the type of the channel from the parent's perspective from the type of the channel from the child's perspective, not just the other way around.

```
type Dual :: ChannelType → ChannelType
type family Dual i = o | o → i where
  Dual EndT          = EndT
  Dual (SendT a b)   = RecvT a (Dual b)
  Dual (RecvT a b)   = SendT a (Dual b)
  Dual (ChoiceT a b) = OfferT (Dual a) (Dual b)
  Dual (OfferT a b)  = ChoiceT (Dual a) (Dual b)
```

The *MakeSureEmpty* constraint is implemented with a typeclass that goes through and ensures every value is *Nothing*.

```
class MakeSureEmptyAux n ns | n → ns
instance MakeSureEmptyAux Z []
instance (MakeSureEmptyAux n xs)
  ⇒ MakeSureEmptyAux (S n) (Nothing : xs)
type MakeSureEmpty :: SessionState → Constraint
type MakeSureEmpty xs = MakeSureEmptyAux (Len xs) xs
```

Three of the operations are given below. The *Create* operation appends a new channel to the end of the context, creating a resource in the usual way. It naturally initialises the resource with *True* as the metadata, since only the parent thread has access to the channel. Also note that a will be ambiguous at the call site, since on creation, it is not possible to determine what the channel type with be. Every time an operation is performed on the channel, more of the channel type will be discovered. This is seen in the *Send* operation. The channel type is determined to be of the form *SendT a b*, but b will be unknown at that point. The *Close* operation ensures that the final part of the channel type is equal to *EndT*, and replaces the resource state of the channel with *Nothing* in the context.

```
type SessionOperation :: Operation SessionState
data SessionOperation p q x where
  Create
    :: SessionOperation p (Append p (Just (a, True))) (CNat (Len p))
  Close
    :: (Lookup p n ~ Just (EndT, False))
    => CNat n -> SessionOperation p (Replace p n Nothing) ()
  Send
    :: (Lookup p n ~ Just (SendT a b, False))
    => CNat n
    -> a
    -> SessionOperation p (Replace p n (Just (b, False))) ()
```

Recv is defined similarly to *Send*. There are two operations corresponding to *Choice*, *ChooseLeft* and *ChooseRight*. This is as *Choice* contains two channel types, for the rest of the communication through the channel.

```
type CNat :: Nat -> Type
data CNat n where
  CNat :: Chan Any -> Chan Any -> CNat n
```

The channels are represented by the *CNat* data type. At the type-level this is indexed by a natural number that references the resource state. At runtime this contains two values of the data type *Chan*, which will be referred to as physical channels. These physical channels will be used for sending and receiving data. Previously, *ENat* was used to represent a reference to a resource. At runtime *ENat* contained an integer which was the index where the reference could be found in a list. By encoding the physical channels directly in *CNat*, this means this extra indirection is unnecessary.

```
type SessionDualScope :: DualScope SessionState
data SessionDualScope p p1 q1 p2 q2 q where
  Offer
    :: (Lookup p n ~ Just (Offer p1 p2, False))
    => CNat n
    -> SessionDualScope p (Replace p n (Just (p1, False)))
         q (Replace p n (Just (p2, False))) q q
```

One operation that cannot be expressed in this fashion is *Offer*. The reason is that there are two possible types for the rest of the channel, depending

on what the other process chooses. This is an example of where a dual scoped operation can be used. The operation *offer* has two inner scopes, one for each of the two possible types, that could represent the rest of the communication channel. Only one of the scopes is executed after which execution passes back out of the scope. The state at the beginning of the scopes is the same as before the *Offer* operation, except that the type of the channel is replaced depending on the particular scope. Both of the scopes need to end in the same state, and this state is preserved once the scope is exited.

The semantics for the effect is implemented using the physical channels given by *Chan*, which is a concurrency mechanism provided by Haskell. We refer to the *CNat* values which are indexed by references to resource states in the context, as logical channels. Two physical channels underlie each logical channel. If a logical channel consists of two physical channels A and B, then one process will use A for sending and B for receiving. The other process will use B for sending and A for receiving. The physical channels will transfer values of type *GHC.Types.Any*, and *unsafeCoerce* will be used when interacting with the physical channels. While this is not type-safe, it is done exclusively in the handler, without affecting the type safety of the user's code.

Operations for *Chan* require the *IO* monad. In order to handle this, a new effect *EmbedIO* is introduced. This effect will be trivially parameterised, in that it will only be able to take the unit type as its effect state. It is intended to be at the end of an effect chain. The handler for *EmbedIO* will produce a value of *IO a*. Then the handler for the *Session* effect will simply reinterpret operations in *Session* in terms of *EmbedIO*.

```
type EmbedOperation :: Operation ()
data EmbedOperation p q x where
  EmbedOperation :: IO a → EmbedOperation () () a
```

As an example consider this snippet from the handler that is responsible for the *Create* operation. The handler uses *newChan* twice to create two physical channels. It will do this by calling *injectIO* which is the smart constructor for *EmbedOperation*. It will construct a *CNat* which is given to the continuation.

```
runSessionH b (Impure (Oinl Create) c)
  = do
```

```
chan ← injectIO (liftM2 CNat newChan newChan)
runSessionH b (c chan)
```

To conclude this section, the basic encoding of session types has been explained. It has been explained how the *Fork* operation is used to provide concurrency, how the semantics of the effect is implemented and how the encoding ensures exactly two threads have access to the channel. An interesting feature of this encoding is that regular, scoped and dual scoped operations are all used.

4.2 Delegation and Recursion

In this section, two improvements over the basic encoding of session types will be given: session delegation; and recursive session types.

One improvement that will be given to the treatment of session types is session delegation. This will allow a thread to send a channel of a given type over to another thread. The thread that sends the channel will lose access to it, while the thread that receives the channel is responsible for interacting with it. To allow for this, the *ChannelType* data type is extended with two new types *SendChannelT* and *RecvChannelT*. A thread communicating through a channel with type *SendChannelT a b* would have to send a channel to the other process with type a, with the current channel now having type b.

```
data ChannelType where
  SendChannelT :: ChannelType → ChannelType → ChannelType
  RecvChannelT :: ChannelType → ChannelType → ChannelType
```

This is implemented by adding operations *SendChannel* and *RecvChannel*. Like the normal *Recv* operation, the *RecvChannel* operation replaces the type of the channel being used with the type representing the rest of the communication, using the *Replace* type family. Additionally, it functions like the *Create* operation by adding a new channel to the context using the *Append* type family, which represents the channel received from the other process. Like the *Create* operation, the output type of the *RecvChannel* operation is the index of the new channel.

Meanwhile, the *SendChannel* functions like the *Send* operation, exception that it addition to replacing the type of the channel being used with the

channel type representing the rest of the communication, it also replaces the channel being sent with *Nothing*, reflecting the fact that it no longer has access to it.

```
data SessionOperation p q x where
  SendChannel
      :: (Lookup p n1 ~ Just (SendChannelT a1 a2, False)
         , Lookup p n2 ~ Just (a1, False))
      => CNat n1
      -> CNat n2
      -> SessionOperation p (Replace (Replace p n2 (Just (End, False)))
           n1 (Just (a2, False))) ()
  RecvChannel
      :: (Lookup p n ~ Just (RecvChannelT a1 a2, False))
      => CNat n
      -> SessionOperation p (Append (Replace p n (Just (a2, False)))
           (Just (a1, False))) (CNat (Len p))
```

Another improvement over the initial encoding is allowing recursive session types. This will make it possible to recurse back to a previous session type. To implement this de Bruijn indices will be used. The outermost expression that can be recursed back to will be referred to with the index 0, the next outermost expression that can be recursed back to will be referred to with the index 1, etc.

This leads to two new data constructors in *ChannelType*. The type *RecT* creates an expression that can be recursed back to. The data type *RecCallT* recurses back to the expression with the de Bruijn index provided as an argument.

```
data ChannelType where
  RecT     :: ChannelType -> ChannelType
  RecCallT :: Nat -> ChannelType
```

In order to be able to recurse to a previous *RecT* expression, the resource state is modified. Rather than storing the channel type, a stack of channel types is stored instead. The head of the stack is the current expression. When a *RecT* expression is encountered, the inner expression is pushed onto the stack. In order to recurse, expressions are popped off the stack, depending how far back you want to recurse.

```
type Channel = Maybe ([ChannelType], Bool)
```

This is implemented by two operations *Rec* and *RecCall*. In the *Rec* operation, the inner expression of *RecT* is pushed onto the stack. A value of type *NNat* (*Len xs*) is given. This has no runtime purpose, but acts as a type witness for *Len xs*, the index of the *RecT* expression in the stack, and can be given to *RecCall* to recurse back to the correct expression. The operation *RecCall* will use the type families *Len* and *Sub* to calculate how many items need to be popped from the stack, and this is done with the *Drop* type family.

```
data SessionOperation p q x where
  Rec
    :: (Lookup p n ~ Just ((RecT x) : xs, False))
    ⇒ CNat n
    → SessionOperation p
        (Replace p n (Just (x : (RecT x) : xs, False)))
        (NNat (Len xs))
  RecCall
    :: (Lookup p n ~ Just ((RecCallT t) : xs, False)
       , Sub (Len xs) (S t) ~ k)
    ⇒ CNat n
    → NNat t
    → SessionOperation p
        (Replace p n (Just (Drop xs k, False))) ()
```

Now a demonstration of session types will be given that takes advantage of recursive session types. There will be two threads: a server; and a client. The server will generate a random number between 1 and 30. If the number is greater than or equal to 3, it will send the number to the client and recurse. The client will keep printing the numbers it receives from the server. The server will only stop when a number less than 3 is generated.

```
server
  :: (Member Session u p ps i qs, _)
  ⇒ CNat n → ITree u ps qs ()
server channel = do
  v ← rec channel
  j ← injectIO (getStdRandom (randomR @Int (1, 30)))
```

```
  if j ⩾ 3 then do
    chooseLeft channel
    send channel j
    recCall channel v
    server channel
  else do
    chooseRight channel
    close channel
  return ()
```

The server begins by calling *rec*, so that it is able to recurse back to this point. The result is v which will be used to perform the recursion. Then the server uses *injectIO* to generate a random number. This will result in another membership constraint being added to the context of the function, ensuring that *EmbedIO* is part of the effect chain. The random number is stored as j.

If this is greater than or equal to 3, the function uses the operation *chooseLeft*. This indicates that it wants to choose the left alternative in a session type. Then it uses the *send* function to send the generated random number to the client. Finally it uses *recCall* so that the session type becomes what it was at the start of the function, at which point it is possible for the function to call itself. If the random number is less than 3, then it decides to pursue the right alternative in the session type with *chooseRight*, before ending the communication using the *close* operation.

```
client
  :: (Member Session u p ps i qs, _)
  ⇒ CNat n → ITree u ps qs ()
client channel = do
  v ← rec channel
  offer channel
    (do
      k ← recv channel
      injectIO (putStrLn (show k))
      recCall channel v
      client channel
    )
    (do
```

```
    injectIO (putStrLn "Goodbye!")
    close channel
)
```

The client also begins by calling *rec* so that it is able to get the session type back to what it was at the beginning of the function. It then uses *offer* to give the server a choice of two alternatives. When the server calls *chooseLeft* the first scope will run. When the server calls *chooseRight* the second scope will run. The first scope receives the random number from the server and prints it, using *injectIO*. It then restores the session type to its previous value with *recCall*, before calling itself. The second scope prints a farewell message, before closing the channel.

```
example = runIO ∘ runSession do
  channel ← create
  fork (client channel)
  server channel
```

The function *example* is responsible for setting up this scenario by running the handlers, creating the channel, and starting the client and server in different threads. To conclude this section, two extensions to the basic encoding of session types have been presented: session delegation; and recursive session types. In addition, an example of the session type encoding is given that takes advantage of the recursive session type feature.

4.3 Evaluation

In this section, the encoding of session types is used to evaluate the framework for encoding linearity.

The encoding of session types demonstrates the framework allows complex operations to be encoded in a straightforward way. An example of this is session delegation. The *SendChannel* operation was implemented using two *Replace* operations and the *RecvChannel* was implemented using a *Replace* operation and an *Append* operations. Another example of this is recursive session types. The *RecCall* operation used *Lookup* to find the resource state within the context, then compute the number of items to be popped from the stack, and then perform the removal using *Drop*.

This shows how the basic primitives provided by the framework can be combined to implement sophisticated behaviour. Moreover, most of the type-level functions used were those belonging to the framework, reflecting the utility of these functions. However, other type-level functions such as *Dual* and *MakeSureEmpty* were easy to integrate into the framework.

The encoding of session types also demonstrates the importance of having both scoped operations and dual scoped operations that could interact with the effect state. The scoped operation *fork* had to ensure the channel is only accessible from two threads. The dual scoped operation *offer* was essential in allowing session types where two alternatives can be picked from at runtime. Without it, the example could not be encoded.

The example presented shows that code written using the framework is generally very readable. It is straightforward to follow the logic of what is happening. One exception to this is the use of operations such as *rec* and *recCall*. These operations feel very unnatural. They resemble a type-level version of the goto control flow operator. However, constructions like this are typical of session type encodings. It is difficult to see how an explicit operator for recursion could be avoided. One way to improve the situation might be to build a fixed-point combinator that handles changing the session type, but also the actual recursion. This could hide the calls to *rec* and *recCall* under the hood.

Another strength of the encoding was the lack of intervention from the programmer to specify types. Aside from standard partial type signature, the only other piece of information provided in the example was a single type application to *randomR* which is not part of the session type encoding. The fact that Haskell is able to deduce the type signatures is both due to the effectiveness of the type-checker plugin and due to careful consideration about how to encode the operations.

A further strength of the encoding was the ability to provide the semantics by reinterpreting the operations in terms of *EmbedIO*. This demonstrates the power of basing the encoding of linearity on algebraic effects. Moreover, writing the handlers for the operations was very straightforward, as it mainly proxied operations on logical channels onto operations on the underlying physical channels.

One weakness of the encoding was the fact that *CNat*, which was used to encode the actual physical channels and to act as a type-level witness, has an implementation that is very specific to the desired semantics. This blurs the lines between syntax and semantics. Technically, this was not

necessary. An alternative would have been to use something similar to *ENat* seen previously, which is relatively generic, but this would mean performing a list lookup every time. One improvement to the framework could be to parameterise the type-level witnesses, so the specific implementation of the effect could choose which runtime data type should be used.

To conclude this chapter, the encoding of session types developed demonstrates the effectiveness and expressiveness of the framework.

Chapter 5

Mutable Arrays

In this chapter, we use our framework to model safe access to mutable arrays (Wadler, 1990). In Haskell, the standard data type used for collections is the list data type, which is implemented as a linked list. This means every time an update is made, a new list is created. This is called a destructive update, and it is a costly operation. An alternative data structure which avoids this is a mutable array. In order to maintain referential transparency, it is implemented using a monad.

Problems arise when two threads have access to the same array, as a data race could occur. This is where the output of the program depends on the sequencing of the individual threads. This non-deterministic behaviour is undesirable, and is not prevented by Haskell. The goal of this chapter is to model mutable arrays, such that only programs that are guaranteed not to have data races occurring from array accesses will compile.

We start by giving the intuition as to how data races can be prevented, by keeping track of the state of a mutable array. We then show how this can be implemented using our framework. We then explore how our encoding is too restrictive, and remedy the situation by introducing slices. We then give an encoding of quicksort using this framework. Finally we evaluate, mutable arrays as a demonstration of the capabilities of our framework.

5.1 Development

A read to an array is considered safe, if there are no other threads that could be writing to the array concurrently. Whereas a write to an array is

considered safe, if there are no other threads accessing the array at all. In other words, there can be multiple threads that can access the array, as long as they're all reading. However, if a thread is writing, then it must be the only thread that can access the array.

We model this situation using three *access levels*, N, R, X. If a thread has access level N to an array, then the thread cannot perform any operation at all. Whereas if a thread has an access level R to an array, then there could be other threads that have access to the array, so only reads can be performed. Finally if a thread has an access level X, then it has exclusive access to the array, and can write to the array. These access levels can be thought of as ordered with $N < R < X$, and are formulated in the Haskell type *AccessLevel*.

```
data AccessLevel = N | R | E
```

In our model, new threads will be constructed using a *fork* operation. This means the access level can actually change after a *fork* operation, since a thread might have access to a resource, but then lose it to a child thread. In the example below the parent thread begins with write access to the array, but then forks creating a child thread which has write access. After the fork operation, the parent thread has no access to the array.

```
f = do
  arr ← alloc True 10
  write a 0 False
  fork do
    write a 1 False
  return ()
```

The thread that runs f will be referred to as thread A. It creates a 10-element array of Booleans, with each element initialised to *True*. It creates a new thread with *fork*, which will be referred to as thread B. Before the *fork* operation, the access level of thread A was X. After the operation, the access level is N. Meanwhile, the access level of thread B is X.

Modelling this scenario is about encoding two kinds of constraints. The first is the constraint between the access levels of the parent thread prior to the *fork* operation, the parent thread after the *fork* operation, and the child thread. The second is ensuring that the access level is high enough to perform the operation.

To begin, an 'obvious' way of modelling this scenario will be considered and an explanation will be given for why it is inadequate. The straightforward approach would be to store the access level for each resource in the resource state, and to encode the two types of constraints using typeclasses. The problem is that it is easy to arrive in scenarios where there are ambiguous type variables, whose values Haskell is unable to deduce. If there is a *write* operation, in the child or the parent, then the type variables become concrete. However, if both threads perform read operations (or no operations at all), then there are multiple options for both of the access levels.

To avoid ambiguous type variables, a different resource state must be used. Before this is revealed, the concept of *extended access level* must be explained. The extended access level of a thread is the maximum access level of a thread and all its children. For instance, in the function f, after the *fork* operation, the thread A has access level N, but extended access level X, since its child thread has access level X.

Then the term *access bounds* will be defined as the upper bound of the access level of the current thread, and a lower bound of the extended access level. For instance, the access bounds after the *fork* operation finishes of the parent thread is (N, X) and the child thread is (X, X). This will be encoded with the Haskell type synonym *AccessBounds*.

$$\textbf{type}\ \mathit{AccessBounds} = (\mathit{AccessLevel}, \mathit{AccessLevel})$$

The resource state will consist of the type of the array, and its access bounds. The effect state will be the context, consisting of these resources states, and be encoded as the Haskell type synonym *ArrayState*.

$$\textbf{type}\ \mathit{ArrayState} = [(\mathit{Type}, \mathit{AccessBounds})]$$

Let us make this concrete with the example below. The function f constructs an array of Booleans *arr*. Interspersing the lines of code in the function, are annotations showing the access bounds that are part of the resource state for *arr* at any given time.

$f = \textbf{do}$
  $\mathit{arr} \leftarrow \mathit{alloc}\ \mathit{True}\ 10$
  (X, N)
  $\mathit{write}\ a\ 0\ \mathit{False}$
  (X, X)

```
fork do
  (X, N)
  write a 1 False
  (X, X)
(N, X)
return ()
```

When *arr* is created in thread A, the bounds are (X, N). The upper bound is X, since it could be the case that the thread has exclusive access to the array. The lower bound is N, since the array has not been accessed by the thread or its children (of which there are none). When the *write* operation is performed, the bounds becomes (X, X).

When thread B is constructed, the bounds become (X, N). The reasoning is similar here. The upper bound is X, because the thread has exclusive access to the array. The lower bound is N, because while thread A has accessed *arr*, thread B has not yet done so. When thread B does perform a write operation, the bounds become (X, X). Back in thread A, after the *fork* operation is performed, the bounds are (N, X). There is no situation in which thread A has any further access to the array.

Note that whenever a *read* operation is used, a maximum between R and the lower bound needs to be performed, to produce the new lower bound. Similarly, a less-than-or-equals-to comparison with the upper bound needs to be performed, to make sure R is less than or equal to the upper bound.

```
type Max :: AccessLevel → AccessLevel → AccessLevel
type family Max a b where
  Max X _ = X
  Max _ X = X
  Max R _ = R
  Max _ R = R
  Max _ _ = N
type Leq :: AccessLevel → AccessLevel → Constraint
class Leq a b
instance Leq a X
instance Leq R R
instance Leq N R
instance Leq N N
```

These constructs do not need to be used when a *write* operation is per-

formed. This is as when a write operation is performed, the lower and upper bound both have to be equal to X. This gives us the basis to formalise the three operations, *Alloc*, *Read* and *Write*.

The *Alloc* operation takes an integer, representing the size of the array, and a value which is used to initialise the elements of the array, and produces a reference to an array.

The *Read* operation take a reference to an array and an index as an integer, and produces the contents of the array. The *Write* operation takes a reference to an array, an index as an integer, and a value which is to be put into that index of the array.

```
type ArrayOperation :: Operation ArrayState
data ArrayOperation p q x where
  Alloc
    :: Int → t
    → ArrayOperation p (Append p (t, (X, N)))) (ANat (Len p))
  Read
    :: (Lookup p n ∼ (t, (u, l)), Leq R u)
    ⇒ ANat n → Int
    → ArrayOperation p (Replace p n (t, (u, Max R l))) t
  Write
    :: (Lookup p n ∼ (t, (X, l)))
    ⇒ ANat n → Int → t
    → ArrayOperation p (Replace p n (t, (u, X))) ()
```

The arrays are represented at value-level with an *ANat*. This is indexed with a type-level natural number which is the index of the resource state in the context. Internally the *ANat* consists of two integers, which delimit the range of indices that can be used to access the array, and the actual array as an *IOArray*.

```
type Bounds = (Int, Int)
type ANat :: Nat → Type
data ANat where
  ANat :: Bounds → IO.IOArray Int Any → ANat
```

We shall consider how to formalise the scoping operation *Fork*. As seen before, when a scope is entered the upper bound remains the same. This make sense intuitively, since the child thread can't expect to have a greater

level of access than its parent. However, the lower bound resets to N, since the child thread starts from a blank slate of no operations having been performed. To accomplish this the higher-order programming structures developed earlier will be used, to aid the type-checker plugin. The type family *Apply ForkScope a* resets the lower bound for a resource state a. Consequently the type family *Map ForkScope a* will do this for the entire context.

```
data Op a where
  ForkScope :: Op ArrayValue
type instance Apply ForkScope (r, (a, _))
  = (r, (a, N))
```

Similarly, when a scope is exited the context needs to be changed. The lower bound is formed by taking the maximum of the lower bound on exiting the scope, and the lower bound of the parent thread before the scope was entered. The change to the upper bound is slightly more complex. If the lower bound on exiting the scope is X, the upper bound becomes N. In other words, the child thread (or one of its descendants) writes to the array, so access is lost. If the lower bound is R, then the upper bound becomes R. In other words, the child thread reads to the array, so no writes are permitted. If the lower bound is N, i.e. there is no access to the array, then the upper bound reverts to what it was prior to entering the scope.

A key property is that if the resource state hasn't changed during a scope, then the lower bound which is reset to N will end the scope as N, and so the lower and upper bound will end up reverting to the same value, they had before entering the scope. This means the resource state change on exiting the scope can be modelled with *Reverse ForkScope a b* where a is the resource state before entering the scope, and b is the resource state before exiting. This means *MapR ForkScope a b* will apply the change for the whole context, and that the type-checker plugin can handle situations where resource states aren't changed at all by the context.

```
type instance Reverse ForkScope (r, (_, a)) (r, (_, X))
  = (r, (N, X))
type instance Reverse ForkScope (r, (_, a)) (r, (_, R))
  = (r, (R, Max a R))
type instance Reverse ForkScope (r, (t, a)) (r, (_, N))
  = (r, (t, a))
```

This means that the scoping operation can be defined as follows.

```
type ArrayScope :: Scope ArrayState
data ArrayScope p p' q' q x x' where
  Fork :: ArrayScope p (Map ForkScope p) q
    (MapR ForkScope p q) () ()
```

To conclude, this section a basic encoding of mutable arrays has been given in the framework.

5.2 Slice, Join and Wait

So far we have developed a way for safe concurrent access to arrays. However our model is overly restrictive, preventing safe programs from being able to be written. In this section, we consider the example of the quicksort algorithm (Hoare, 1962). This algorithm is recursive and can be naturally parallelised. We consider the problems that prevent it from being written using the model of mutable arrays developed so far. The first restriction is that sometimes concurrent writes to the same array can be safe. The second restriction is that there is no way to recover access to an array, that has been lost to a child thread. These shortcomings lead to the development of three new operations, *slice*, *join* and *wait*, that are crucial to an encoding of the quicksort algorithm.

The quicksort algorithm works by choosing a pivot, which is a particular element in the array, and elements in the array are split into two halves. The first halve consists of items smaller than or equal to the pivot element. The second halve consists of items greater than the pivot element. The quicksort algorithm then recurses on the two individual halves. One advantage of this algorithm is that it can be extended to a multithreaded environment, by running different recursive calls on different threads.

We cannot currently express a concurrent version of quicksort using our model. The first reason is that only one thread can write to an array concurrently. However simultaneous writes to the same array by different threads are safe, if the writes are to different memory locations. Languages such as Rust (Matsakis and Klock, 2014) enforce this through the concept of slices, which allow you to construct a reference into a contiguous sequence of elements within a data structure. We shall introduce a similar capability into our model.

This is accomplished by introducing a *Slice* operation, that allows you to construct two slices, by splitting at a certain index. Our context of resources will consist of arrays and slices. To allow this, the definition of *ArrayState* will be changed. In addition to keeping track of the type that a resource stores and the access bounds, the resource state also needs to store whether a resource is an array or a slice. If a resource has the value type *Slice* n it means that it is a slice, that was constructed from the resource at index n, which could itself be a slice.

```
data ValueType where
  Array :: ValueType
  Slice  :: Nat → ValueType
type ArrayState = [(Type, ValueType, AccessBounds)]
```

An example of slices are given below. The array i is split into two slices i_1 and i_2. It is sliced at index 4, meaning that the indices 0 to 4 of i_1 refer to indices 0 to 4 of i. Whereas the indices 0 to 4 of i_2 refer to indices 5 to 9 of i. After the slice occurs, no further operations can be performed on i, and i_1 and i_2 are treated as entirely separate resources. This means that i_1 can be written to in the child thread, and i_2 can be read from in the parent thread, while the child thread could be concurrently executing.

```
example = do
  i ← alloc True 10
  (i1, i2) ← slice i 4
  fork $ do
    write i1 0 False
  b ← read i2 0
  return b
```

We also need to consider how slices interact with access levels. Once a slice to a resource is created, you can no longer directly access that resource. Therefore to create a slice, you need exclusive access to the array, like the *write* operation. However, unlike the *write* operation after you create the slice, the upper bound of the original resource will become N, as access is lost.

```
data ArrayOperation p q x where
  Slice
```

```
:: (Lookup p n ~ (t, r, (X, a)), Len p ~ k)
⇒ ANat n
→ Int
→ ArrayOperation p
      (Append
        (Append
          (Replace p n (t, r, (N, X)))
          (t, Slice n, (X, N)))
        )
        (t, Slice n, (X, N))
      ) (ANat k, ANat (S k))
```

The integer argument to a *Slice* operation is the index at which you want to split, and the output of the operation are two indices, one for each of the slices. We can also imagine an operation to obtain the original resource, from the two slices. This would require exclusive access to both slices, and permanently destroy them.

```
Join
  :: (Lookup p n₁ ~ (t, Slice k, (X, a)),
      Lookup p n₂ ~ (t, Slice k, (X, b)),
      Lookup p k ~ (t, r, (N, c)))
  ⇒ ANat n₁
  → ANat n₂
  → ArrayOperation p (
      Replace (
        Replace (
          Replace p n₁ (t, Slice k, (N, X)))
        ) n₂ (t, Slice k, (N, X))
      ) k (t, r, (X, c))
    ) ()
```

The semantics are implemented using the *IOArray* which is stored inside *ANat*. The two pair of integers stored within it describe the bounds that can be used to access that array. When a *slice* operation is performed the array is copied over, and the bounds are changed. This means when a *read* or *write* operation is performed, the code is the same regardless of whether it is the original array or not. The handler reinterprets operations in terms

of the *EmbedIO* effect. If incorrect indices are given, then a runtime error will occur.

There is an additional problem preventing quicksort from being encoded. There is no way to wait for child threads to finish, or to recover access to arrays that have been given to them. For a sorting algorithm to be useful, it is essential that the sorted array can actually be used, after the algorithm's termination. This is accomplished by introduced the *wait* operation. This operation allows you to block the current thread till some other thread is finished, and retrieve the result of that thread. Once the *wait* operation is complete, it can be guaranteed that the thread being waited on, has finished.

The *fork* operation is changed so that it returns a future that can be passed to *wait*, signifying which child thread to wait for. It is important that a future can be used at most once. Otherwise multiple threads could get the same level of access that the finished thread had, which wouldn't necessarily be safe. To accomplish this the effect state of the array is extended so that it consists of two contexts. The first context is for the arrays and the second context is for the futures, which are another kind of resource. The resource state for the future is simply a boolean, indicating whether the resource state has been used or not.

Futures are encoded with the data type *Future*, which has three type parameters. The first is the index of its resource state in the context of futures. The second is the context of arrays prior to exiting the scope, that the future refers to. The third is the type of the result from the thread, represented by the future. *Future* serves a similar purpose to *ANat*, but all of the information associated with an *ANat* is stored in the resource state. Whereas for *Future* most of the information is stored in indices within the type, rather than the resource state. The reason for this is that *Future* can only be used once, so there is no need to store all of that information in the resource state.

Below the *Wait* operation is given. Its behaviour is particularly complex. Note that now the effect state consists of two contexts. The precondition for the *Wait* operation is (p_1, p_2) where p_1 is the context of arrays, and p_2 is the context of futures. The operation has the condition $\mathit{Lookup}\ p_2\ n \sim \mathit{False}$ making sure that the future hasn't been used. The change it makes is to replace this value with *True* to produce the new context of futures.

Meanwhile to produce the new context of arrays, it necessary to map through the current context and the context before the scope was exited, to produce the new context of arrays. The lower bounds will stay the same,

but the upper bounds will be equal to whichever upper bound is higher, the one in the current context or the one in the context before the scope was exited. Particular attention needs to be paid to the type-checker plugin. Is it possible to write this in a way that can be optimised by the type-checker plugin? The answer is almost.

The current context will be a function of the scope that exists after exiting the scope using 'nice' type-families such as *Append* and *Replace*. Note that because of *Map ForkScope* and *MapR ForkScope*, the context after exiting the scope will be a function of the scope that exists before entering the scope in terms of the same 'nice' type families. Meanwhile, the scope encoded by the future, is the context before exiting the scope, which is a function of the context before entering the scope in terms of 'nice' type families and also *Map ForkScope*. Finally note that if a resource state hasn't been changed by the scope, then it will return to the same value after the *wait* operation.

In other words, the *Wait* operation is an ideal case for a *MapR ForkScope* operation that the type-checker plugin can compress with *Map ForkScope*. However, there already exists a different *MapR ForkScope* operation. To deal with this, the *Reverse1* and *MapR1* type families are introduced, which the type-checker can optimise in the same way as *Reverse* and *MapR*. This allows the *Wait* operation to be encoded in a way that allows the type-checker plugin to infer code correctly.

$$
\begin{aligned}
&\textbf{data}\ \mathit{ArrayOperation}\ \textbf{where}\\
&\quad \mathit{Wait}\\
&\qquad :: (\mathit{Lookup}\ p_2\ n \sim \mathit{False})\\
&\qquad \Rightarrow \mathit{Future}\ n\ b\ a\\
&\qquad \rightarrow \mathit{ArrayOperation}\ (p_1, p_2)\\
&\qquad\qquad (\mathit{MapR1}\ \mathit{ForkScope}\ p_1\ b, \mathit{Replace}\ p_2\ n\ \mathit{True})\ a\\
&\textbf{type instance}\ \mathit{Reverse1}\ \mathit{ForkScope}\ (r, (a, c))\ (r, (b, \_))\\
&\quad = (r, (\mathit{Max}\ a\ b, c))
\end{aligned}
$$

In this section, a lot of progress has been made towards encoding quicksort. Three new operations have been added. Two of them deal with the problem of concurrent writes to different parts of the array. The other allows not just the result of a thread to be retrieved, but also the level of access it has.

5.3 Recursion and Quicksort

In this section, the final steps towards an implementation of quicksort are taken. This only obstacle left is how to implement recursion. The problem of encoding recursion where the a function calls itself with a different combined state is explained. In addition the solution to this problem is shown as explicitly giving the type signature. Finally two more additions, that are needed are explained: the *demandWrite* operation and *local* scoped operation. Finally a recursive implementation of quicksort is given.

The recursive nature of the quicksort algorithm leads to a problem, when trying to write it in a natural Haskell style. The problem is that the quicksort function is polymorphic, with the type variables including the effect state. Type inference with polymorphic recursion where type parameters change between invocations is undecidable in general. Haskell, in particular, makes the assumption that the type parameters are the same, i.e. the combined state does not change between invocations. However, this is not the case, since two slices are added. Haskell makes this assumption even before the type-checker plugin is invoked. The solution is to specify a type signature that constrains the input to the function.

$$
\begin{aligned}
&\mathit{quicksort} \\
&\quad :: (\mathit{Lookup}\ p_1\ n \sim (\mathit{Int}, r, (X, X)) \\
&\qquad , \mathit{Member}\ (\mathit{FindEffect}\ \mathit{ArrayEffect}\ u) \\
&\qquad\quad \mathit{ArrayEffect}\ u\ (p_1, p_2)\ i\ ps\ ps) \\
&\quad \Rightarrow \mathit{ANat}\ n \\
&\quad \rightarrow \mathit{ITree}\ u\ ps\ ps\ ()
\end{aligned}
$$

A problem is that the quicksort function will end creating new resources. It will create two slices, and also a future. This will need to be removed from the state, before the function exits. To do this a *local* scoped operation is added similar to the one created for the storage effect. This removes resources created inside the scope, so they are not accessible outside.

The final technicality to fix is that the function expects the access bounds for the array to be (X, X), since it needs to be the same as when the function ends. However, this is not the case when the slices are created. Since they haven't been written to before the resource state will be (X, N). To handle this the operation *demandWrite* is added, that essentially simulates a *write* operation at the type-level, taking (X, N) to (X, X). Its actual implementation is as a no-op.

```
quicksort arr = do
  len ← length arr
  if len ⩽ 2 then do
    when (len ≡ 2) do
      v0 ← read arr 0
      v1 ← read arr 1
      when (v0 > v1) do
        write arr 0 v1
        write arr 1 v0
  else local do
    i ← partition len arr
    (i1, i2) ← slice arr (max (i − 1) 0)
    t ← fork do
      demandWrite i1
      quicksort i1
    demandWrite i2
    quicksort i2
    wait t
    join i1 i2
    return ()
```

Quicksort is given above. First it checks the length of the array. If the length is less than two, then it exits. If it is equal to two, then it reads the two values of the array and swaps them if necessary. Otherwise, the function will call another function *partition*, that splits the function into two halves, returning i as the index of the pivot. Then *slice* is called to split the array, based on the pivot. Then a child thread is created to sort the left slice, while the parent thread sorts the right slice.

After the parent sorts the right slice, the parent invokes *wait* on the future representing the child thread. Once the child thread has finished, the parent thread is unblocked and now the parent has access to both slices. The parent can then invoke *join* to gain access to the initial array. Note that the recursive case takes place entirely within a *local* scope, meaning that the function ends in the same effect state as it began, which is important for recursion to work correctly.

The index at which the split should occur is given as the maximum of 0 and $i - 1$. Note that the index given to slice is included in the left split. It is important that the two subarrays are smaller than the initial array.

Otherwise, one subarray will be empty and the other will be the same as the initial array, leading to an infinite loop. Therefore if the pivot is the first item, the pivot is included in the left slice, and the rest of the array is in the right slice. Alternatively, if the pivot is the last item, then the pivot is included in the right slice, while every other element is in the left slice.

5.4 Evaluation

In this section, the use of the framework to encode mutable arrays will be used to evaluate the effectiveness of the framework.

The case study in this chapter presents a strong case that the framework developed is expressive and is a solid foundation upon which complex constraints can be encoded. One reason for this is that there could be different types of resources in a context, both regular arrays and slices, with the slices referring back to the original array. The operations *Join* and *Slice* manipulated these two types of resources in nuanced ways. For instance, the *Slice* operation uses has three *Lookup* constraints and performs three applications of *Replace* to produce the new effect state.

A second reason is the fact that in this example there were two different contexts. There was a context for arrays (and slices) and another context for futures. The *Wait* operation manipulated both of these contexts, as well as using the type-level information from the future. A third reason is that there were two scoping constructs, with different goals. The objective of *fork* was to create a new thread, while *local* cleans up the two contexts. Both were vital for encoding quicksort, and they demonstrate the usefulness of having scoping operations that manipulate effect state.

Despite the complexity of the effect, encoding the effect was relatively short. This speaks to both the framework and Haskell's type-level capabilities. A lot of the type-level utilities used in this model were the ones that are part of the core framework such as *Append*, *Map*, etc. A few additional type-level constraints were developed such as *Leq* and *Max* specifically for this model. This shows how one can tap into the power of Haskell's type-level programming machinery, and integrate that into the framework. By and large, the type-level code was written in a clean way, e.g. using data types to encode operations. The one exception to this is the awkward way of implementing type-level maps, but this is a small price to pay for the type-checker plugin to be able to perform simplifications.

The flexibility of the type-level tools, that make up the framework, was shown in this chapter, and suggests that they are at the right level of abstraction. They are not too low-level so as to make it difficult to encode complex constraints. On the other hand, they are not too high-level as to be to prescriptive with how constraints can be encoded, and which constraints can be encoded.

One particular weakness was the use of the *demandWrite* function, because it is unnatural and it exists to solve a problem at the type-level, while having no purpose at runtime. This is not a problem that limits the expressiveness of the framework, but it does limit the usability. While it is reasonable to expect a programmer to understand what constitutes well-typed behaviour e.g. concurrent writes to the same array are not permitted, it is unreasonable to expect the programmer to understand the minutiae behind how well-typed behaviour is verified.

In the formal definition of parameterised effects, it is possible to strengthen the precondition of an operation, using a morphism in the parameterising category. In the implementation of parameterised algebras in this project, the morphisms of the parameterising category have not been implemented. One solution to avoid *demandWrite* would be to encode the morphisms as a constraint, and require there to be a morphism between the postcondition of a computation and the precondition of another, for them to be sequenced. A consequence of this is that it would make the parameterised algebra machinery more complex.

Notwithstanding the presence of the *demandWrite*, the solution is very readable and similar to what a normal quicksort implementation would look like in Haskell. Operations such as *read* and *write* appear practically identical to what they would look like when interfacing with normal Haskell arrays. Indeed owing to the use of do-notation, it resembles what a quicksort algorithm might look like in an imperative programming language.

Another weakness is the fact that the type signature for a recursive function needs to be explicitly stated. This wasn't too problematic, since the function ends in the same state that it starts, so an *Evolve* constraint didn't have to be specified. This is typical for most recursive functions. Otherwise the state would not be knowable since every invocation might change the state, and it would not be possible to know how many invocations there are. However, a constraint on the initial state still had to be specified, which in some cases could be quite complicated. In the case of quicksort, it simply checked that the resource state was equal to a particular value.

Note even though the type signature had to be specified, the type-checker plugin still played a major, highlighted by the need for *Ref*. Without the type-checker plugin, Haskell would not be able to infer that the function starts and ends in the same state. This is unlike most functions, which could still be type-checked, even without the type-checker plugin.

Indeed, quicksort was an excellent demonstration of the effectiveness of the type-checker plugin in that it was able to correctly handle a function with multiple scoping constructs, branching and calls to both itself and another function (*partition*). Indeed aside from requiring a type signature for the function, there were no issues with type inference.

Another aspect that might be considered is performance. While performance was not a focus in this project, given that quicksort is a sorting algorithm, it is worth mentioning it briefly. The fact that the *ANat* data type contains a reference to the *IOArray* means that there are no performance costs in retrieving the underlying array from the *ANat*, like there was with the storage effect. Most of the work in this project exists at the type-level, and at runtime the framework operates as a typical algebraic effects library.

Several of the points mentioned in the previous case study also apply here. There is a slight issue in that *ANat* is very specific to the particular semantics desired in the example. However this was mainly for convenience, and there are ways to avoid this within the existing framework or through extending the framework. In addition, this is another demonstration of the useful pattern of reinterpreting effects in terms of the *EmbedIO* effect. The handlers were very simply to rewrite, simply proxying operations of the algebra onto *IOArray* operations.

Chapter 6

Conclusion

This report began with the objective of unifying linearity and computational effects to allow stronger guarantees on the correctness of computer programs. This chapter draws the report to a close, putting the work achieved in the wider context. It begins with a summary of the major contributions of the project, and an explanation of what distinguishes this work from what has come before. Then, a discussion of related work is given. Finally, some directions for future work are sketched out.

6.1 Contributions

In Chapter 3, an encoding of parameterised algebraic effects in Haskell was developed. These effects form the mainstay of the framework. This foundation was enhanced through: the introduction of contexts so that effects could interact with multiple resources; the introduction of scoped operations so more complex behaviour could be modelled; a type-checker plugin to allow good type inference.

In Chapter 4, the value of the type-level utilities, developed in the previous chapter is demonstrated through an encoding of session types. This means there can be static guarantees that two processes communicating through a shared channel are adhering to the same protocol.

In Chapter 5, the value of the framework is further established with an encoding of mutable arrays. This means there can be static guarantees that data races cannot occur, when two threads have access to the same mutable array. This is used to give a concurrent implementation of quicksort.

The outcome of this project is a framework that allows type-safe programming with resources. This allows errors that would not previously have been prevented by Haskell, to be caught at compile-time during type-checking. The examples demonstrate that the framework is capable of modelling sophisticated and varied constraints.

The work is novel in that the objective is a framework that can form the bedrock of a wide range of different examples of linearity in a uniform way. This project explores the design space of such as framework, so that it can be generic, extensible and expressive.

These considerations motivated the introduction of features that make this project unique: scoped operations that interact with type-level state; a type-checker plugin that significantly improves type inference.

The work is also novel in that it showcases the potential of parameterised algebraic effects. While algebraic effects have begun to enter the mainstream in the functional programming community in previous years, to the best of the author's knowledge, parameterised algebras have not previously been implemented in functional programming languages.

6.2 Future Work

6.2.1 Scoped Operations

One area for future work is how scoped operations belonging to an effect interact with other effects in an effect chain. Consider the *fork* operation used in a number of examples. The inner scope will run in a child thread, while the continuation will run in the parent thread. Consider another parameterised effect representing some resource e.g. network sockets. If the effect state is copied over into the inner scope, then there could be two threads with access to the same network socket. This would not be safe.

The current approach is to side-step this issue, by requiring that other effects in the effect chain do not change their effect state within a scoped operation. A more satisfying solution would be to only allow the coproduct of an effect, that includes scoped operations, with other effects, if the interaction of the scoped operation with the other effects is well defined. This could be done using a typeclass, stating how the effect state for the other effect will change when the inner scope is entered and exited.

That way rather than having a separate *Fork* operation for session types

and mutable arrays, there could be a single *Fork* operation, and the behaviour of session types and mutable arrays could be defined with a typeclass instance.

Some scoped operations might be well-defined with all effects. For instance, the *local* scoped operation in the storage effect will run the inner scope in the same thread before running the continuation. The only difference is that resources defined with respect to that effect are only accessible locally. There is no reason for this to impinge on other effects.

6.2.2 Exceptional Effects

Another area for future work is incorporating exceptions into the framework. Adding an operation such as *abort*, that terminates the computation immediately would not be safe, since it would mean that some resources do not end in the correct final state. A possible solution to this might be to parameterise computation trees with an additional parameter reflecting the state of early terminations. The use of the *abort* operation would require that the current state is the same as that encoded for early terminations.

A dual scoped operation *try*/*catch* might exist, with the early termination parameter of the computation tree in the *try* block being the same as the precondition of the *catch* block. This scoping construct will change the early termination parameter, since it will intercept an early termination and run the *catch* block instead.

Similarly there could exist a *try* / *finally* block. Similar to the *try* / *catch* construction, the early termination parameter for the *try* block will be the same as the precondition of the *finally* block. Unlike *catch* however, after intercepting the early termination and running the *finally* block, early termination occurs again.

6.2.3 Type-Level Rewrite Rules

This project developed a type-checker plugin to simplify constraints. While the type-checker plugin capabilities of GHC offer a lot of power, they also are very heavyweight tools. It would be advantageous to have more lightweight tools to influence the type-checking process. Allowing these to be written in source code would reduce the barrier-to-entry to accomplish this.

GHC has a feature called rewrite rules (Peyton Jones et al., 2001), where identities can be written between value-level expressions in source code. Dur-

ing compilation, Haskell will replace expressions that match the left-hand side of the rewrite rule, with the right-hand side. One might imagine a feature in GHC that allows writing identities that allow for the simplification of types. The *TcSimplify* module of GHC is internally responsible for simplifying type-level expressions, and this could be where these substitutions occur.

6.3 Related Work

6.3.1 Linear Haskell

There is currently on-going work to integrate linear types natively into Haskell (Bernardy et al., 2017). Their approach would be integrated into the language. It would require significant changes to GHC, since linearity has to be 'retrofitted' onto Haskell. In contrast, the work presented here takes advantage of the type-level programming features already present in GHC.

Linear Haskell uses a type of linearity called 'linearity on the arrow' as opposed to 'linearity on the kind'. This means rather than distinguishing between linear and unrestricted types, the primitive construction in Linear Haskell, is the linear arrow $A \multimap B$. Arrows are given multiplicities, stating whether the argument can be use once (1), or an unlimited number of times (ω).

In our approach, the lack of support for linearity within Haskell is sidestepped, by storing the type-level state within the parameterised algebra. In Linear Haskell, this can be stored directly within the resource. In our approach, the operations that modify the type-level state must be part of the parameterised algebra. In Linear Haskell, these would be linear functions, that modify the type-level state. Since the type-level state changes, these functions would have to produce a new copy of the resource, indexed by the new type-level state.

Linear Haskell supports multiplicity polymorphism, where multiplicities can be given as type parameters. In Linear Haskell, a generalisation of the *IO* monad is presented, which is indexed by the multiplicity of the result. For instance, if the returned value is a file handle then one might imagine the multiplicity would be 1 rather than ω. One could imagine that algebraic effects in Linear Haskell might offer the choice of which multiplicity should be used for the continuation of an operation.

6.3.2 The Linearity Monad

An embedding of linearity was presented by Paykin and Zdancewic (2017). Like our system, this was written in Haskell and used the type-level constructs in Haskell to ensure linearity. A major difference between this system and our system, is that the goal of this system is embedding a linear language within the Haskell host language as a domain-specific language.

This implementation was done using higher-order abstract syntax (Pfenning and Elliott, 1988), a technique for embedding a language in a host language while using the variable bindings of the host language. A disadvantage of this approach is the presence of combinators to implement function abstraction and application, that make the code difficult to read.

6.3.3 Session Types

An implementation of session types in Haskell was done using a parameterised monad (Pucella and Tov, 2008). While our implementation used parameterised algebras, both implementations use the idea of a pre and post-condition. Both implementations allow multiple channels. Their approach does so by using a stack, whereas we use a list. Like our implementation they use positional indices. A disadvantage of their implementation is that it does not support session delegation.

Another embedding of session types was done using a graded monad (Orchard and Yoshida, 2016). This approach allowed multiple channels using a type-level map, and allowed these channels to be indexed by type-level symbols. Like our implementation, their approach allows session delegation.

Note that, a graded monad is an alternative generalisation of a monad using a single type parameter, that has monoidal structure. The *return* operation takes a value of type a and produces a value of type $m\ e\ a$, where e is the unit element. While the *join* operation takes a value of type $m\ i\ (m\ j\ a)$ and produces a value of type $m\ (i\ \star\ j)\ a$, where $\star$ is the monoid operation. Note that recent work by Orchard et al. (2020) has unified parameterised monads and graded monads.

6.3.4 Granule

Another system is Granule (Orchard et al., 2019). This is an entirely new programming language, which attempts to combines linear, polymorphic and

indexed types.

A major difference between this system and our system, is that in this system values are linear by default. The approach taken by our framework is that linearity is a special case for resources. The system also differs from other implementations of linearity. While other implementations of linearity allow types to be given an unrestricted modality, in Granule there is a wide range of modalities that can be given to types. Not only that but these modalities are graded by elements from a resource algebra. From a theoretical perspective, these modalities are graded comonads.

Whereas monads allow you to go from a function $a \to F\ b$ to $F\ a \to F\ b$, comonads allow you to go from a function $C\ a \to b$ to $C\ a \to C\ b$. The modality ! in linear logic can be thought of as comonadic. If you have a resource of type $!A$, you can obtain a resource of type A. Similarly, if you have a resource of type $!!A$, you can obtain a resource of type $!A$. Graded comonads are a generalisation of comonads, similar to graded monads.

Granule has a number of resource algebras built into the language to choose from. While in Granule, a value is linear by default, the natural number grading allows values to be used an exact number of times. For example a value of type $t\ [2]$ can be used twice. To be more precise the value $t\ [2]$ is actually still a linear value, but unboxing syntax can be used to retrieve two instances of t. The natural number grading is similar to the storage effect, that was developed earlier. Granule also supports indexed types, allowing resources to be encoded with a type-level state. This allows resources to be constrained in a way that is similar to our system.

Bibliography

Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3–4):335–376, July 2009a. ISSN 0956-7968. doi: 10.1017/S095679680900728X. URL https://doi.org/10.1017/S095679680900728X.

Robert Atkey. Algebras for parameterised monads. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, pages 3–17, Berlin, Heidelberg, 2009b. Springer Berlin Heidelberg. ISBN 978-3-642-03741-2.

P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, page 75–90, Berlin, Heidelberg, 1993. Springer-Verlag. ISBN 3540565175.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer, 1981. URL http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, page 222–236, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642153747.

Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In Cosimo Laneve and Jianwen Su, editors, *Web Services and Formal Methods*, pages 1–28, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14458-5.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, January 1987. ISSN 0304-3975. doi: 10.1016/0304-3975(87)90045-4. URL https://doi.org/10.1016/0304-3975(87)90045-4.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 01 1962. ISSN 0010-4620. doi: 10.1093/comjnl/5.1.10. URL https://doi.org/10.1093/comjnl/5.1.10.

Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electron. Notes Theor. Comput. Sci.*, 172:437–458, April 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.02.019. URL https://doi.org/10.1016/j.entcs.2007.02.019.

Oleg Kiselyov. Typed tagless final interpreters. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*, SSGIP'10, page 130–174, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 9783642322013. doi: 10.1007/978-3-642-32202-0_3. URL https://doi.org/10.1007/978-3-642-32202-0_3.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *SIGPLAN Not.*, 50(12):94–105, August 2015. ISSN 0362-1340. doi: 10.1145/2887747.2804319. URL https://doi.org/10.1145/2887747.2804319.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, page 59–70, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323833. doi: 10.1145/2503778.2503791. URL https://doi.org/10.1145/2503778.2503791.

Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in haskell. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341706. URL https://doi.org/10.1145/3341706.

Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3): 103–104, October 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL `https://doi.org/10.1145/2692956.2663188`.

Conor McBride. Kleisli arrows of outrageous fortune, 2011.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, page 14–23. IEEE Press, 1989. ISBN 0818619546.

Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. *SIGPLAN Not.*, 51(1):568–581, January 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837634. URL `https://doi.org/10.1145/2914770.2837634`.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341714. URL `https://doi.org/10.1145/3341714`.

Dominic Orchard, Philip Wadler, and Harley Eades. Unifying graded and parameterised monads. *Electronic Proceedings in Theoretical Computer Science*, 317:18–38, May 2020. ISSN 2075-2180. doi: 10.4204/eptcs.317.2. URL `http://dx.doi.org/10.4204/EPTCS.317.2`.

Jennifer Paykin and Steve Zdancewic. The linearity monad. *SIGPLAN Not.*, 52(10):117–132, September 2017. ISSN 0362-1340. doi: 10.1145/3156695.3122965. URL `https://doi.org/10.1145/3156695.3122965`.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in ghc. *Haskell 2001*, 04 2001.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. volume 23, pages 199–208, 07 1988. doi: 10.1145/960116.54010.

Matthew Pickering, Nicolas Wu, and Boldizsár Németh. Working with source plugins. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 85–97, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342599. URL https://doi.org/10.1145/3331545.3342599.

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 809–818, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: 10.1145/3209108.3209166. URL https://doi.org/10.1145/3209108.3209166.

Gordon Plotkin and John Power. Computational effects and operations: An overview. *Electron. Notes Theor. Comput. Sci.*, 73:149–163, October 2004. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.08.008. URL https://doi.org/10.1016/j.entcs.2004.08.008.

Gordon Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), Dec 2013. ISSN 1860-5974. doi: 10.2168/lmcs-9(4:23)2013. URL http://dx.doi.org/10.2168/LMCS-9(4:23)2013.

Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. *SIGPLAN Not.*, 44(2):25–36, September 2008. ISSN 0362-1340. doi: 10.1145/1543134.1411290. URL https://doi.org/10.1145/1543134.1411290.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 51–62, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939197. doi: 10.1145/1411204.1411215. URL https://doi.org/10.1145/1411204.1411215.

Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. Injective type families for haskell. *SIGPLAN Not.*, 50(12):118–128, August 2015. ISSN 0362-1340. doi: 10.1145/2887747.2804314. URL https://doi.org/10.1145/2887747.2804314.

Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758. URL https://doi.org/10.1017/S0956796808006758.

Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.

Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-662-02880-3.

Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. Partial type signatures for haskell. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324*, PADL 2014, page 17–32, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 9783319041315. doi: 10.1007/978-3-319-04132-2_2. URL https://doi.org/10.1007/978-3-319-04132-2_2.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. *SIGPLAN Not.*, 49(12):1–12, September 2014. ISSN 0362-1340. doi: 10.1145/2775050.2633358. URL https://doi.org/10.1145/2775050.2633358.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 53–66, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311205. doi: 10.1145/2103786.2103795. URL https://doi.org/10.1145/2103786.2103795.