Notiz: Der Code für alle 3 Ansätze ist quasi fertig. Aktuell schreibe ich also fast ausschließlich an der Arbeit an sich. Dieses PDF ist eine unfertige Arbeitsversion. Einige bereits geschriebene Abschnitte sind noch unfertig oder an der falschen Position in der Arbeit.

Implementing a library for scoped algebraic effects in Agda

HTWK Leipzig

Jonas Höfer
Informatik
69555

2020

Abstract

Contents

Chapter 1

Introduction

1.1 Goals

Chapter 2

Preliminaries

2.1 Agda

Agda is a dependently typed functional programming language. The current version[1] was originally developed by Ulf Norell under the name Agda2 [Nor07]. Due to its type system Agda can be used as a programming language and as a proof assistant.

This section contains a short introduction to Agda, dependent types and the idea of "Propositions as types" under which Agda can be used for theorem proving.

2.1.1 Basic Syntax

Agdas syntax is similar to Haskells. Data types are declared with syntax similar to Haskells GADTs. Functions declarations and definitions are also similar to Haskell, except that Agda uses a single colon for the typing relation. In the following definition of ℕ, Set is the type of all (small) types.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Ordinary function definition are syntactically similar to Haskell. Agda allows the definition of mixfix operators. A mixfix operator is an nearly arbitrary list of symbols (builtin symbols like colons are not allowed as part of operators). Underscores in the operator name are placeholders for future parameters. A mixfix operator can be applied partially by writing underscores for the omitted parameters.

In the following definition of plus for natural numbers + is a binary operator and therefore containers two underscores.

```
_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)
```

2.1.2 Dependent Types

The following type theoretic definitions are taken from the homotopy type theory book [Uni13]. In type theory a type of types is called a universe. Universes are usually denoted $\mathcal{U}$. A function whose codomain is a universe is called a type family or dependent type.

$$F : A \to \mathcal{U} \quad \text{where} \quad B(a) : \mathcal{U} \quad \text{for} \quad a : A$$

To avoid Russell's paradox, a hierarchy of universes $\mathcal{U}_1 : \mathcal{U}_2 : ...$ is introduced. Usually the universes are cumulative i.e. if $\tau : \mathcal{U}_n$ then $\tau : \mathcal{U}_k$ for $k > n$. by default this is not the case in Agda. Each type is member of a unique universe, forcing us to do additional bookkeeping. Since Agda 2.6.1 an experimental `--cumulativity` flag exists.

[1] `https://github.com/agda/agda`

Dependent Function Types (Π-Types) are a generalization of function types. The codomain of a Π type is not fixed, but values with the argument the function is applied to. The codomain is defined using a type family of the domain, which specifies the type of the result for each given argument.

$$\prod_{a:A} B(a) \quad \text{with} \quad B : A \to \mathcal{U}$$

An element of the above type is a function which maps every $a : A$ to a $b : B(a)$. In Agda the builtin function type ⇒ is a Π-type. An argument can be named by replacing the type τ with $x : \tau$, allowing us to use the value as part of later types.

Dependent Sum Types (Σ-Types) are a generalization of product types. The type of the second component of the product is not fixed, but varies with the value of the first.

$$\sum_{a:A} B(a) \quad \text{with} \quad B : A \to \mathcal{U}$$

An element of the above type is a pair consisting of an $a : A$ and a $b : B(a)$. In Agda records represent n-ary Σ-types. Each field can be used in the type of the following fields.

Programming with Dependent Types A common example for dependent types are fixed length vectors. The data type depends on a type A and a value of type ℕ.

```
data Vec (A : Set) : ℕ → Set where
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
  []   : Vec A 0
```

Arguments on the left-hand side of the colon are called parameters and are the same for all constructors. Arguments on the right-hand side of the colon are called indices and can differ for each constructor. Therefore Vec A is a family of types indexed by ℕ.

The [] constructor allows us to create an empty vector of any type, but forces the index to be zero. The _::_ constructor appends an element to the front of a vector of the same element type and increases the index by 1. Only these two constructors can be used to construct vectors. Therefore the index is always equal to the amount of elements stored in the vector.

By encoding more information about data in its type we can add extra constraints to functions working with it. The following definition of head avoids error handling or partiality by excluding the empty vector as a valid argument.

```
head : ∀ {A n} → Vec A (suc n) → A
head (x :: _) = x
```

When pattern matching on the argument of head there is no case for []. The argument has type Vec A (suc n) and [] has type Vec A 0. Those to types cannot be unified, because suc and zero are different constructors of ℕ. Therefore, the [] case does not apply. By constraining the type of the function we were able to avoid the case, which usually requires error handling or introduces partiality.

We can extend this idea to type safe indexing. A vector of length n is indexed by the first n natural numbers. The type Fin n represents the subset of natural numbers smaller than n.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Because 0 is smaller than every positive natural number, zero can only be used to construct an element of Fin (suc n) i.e. for every type except Fin *0*.

If any number is smaller than n, then its successor is smaller than $n + 1$. Therefore, if any number is an element of Fin n then its successor is an element of Fin (suc n).

So we can construct a $k < n$ of type Fin n by starting with zero of type Fin (n - k) and applying suc k times. Using this definition of the bounded subsets of natural numbers we can define _!_ for vectors.

$$
\begin{aligned}
&\_!\_ : \forall \{A\ n\} \to \mathsf{Vec}\ A\ n \to \mathsf{Fin}\ n \to A \\
&(x :: \_)\ !\ \mathsf{zero}\ = x \\
&(\_ :: xs)\ !\ \mathsf{suc}\ i = xs\ !\ i
\end{aligned}
$$

Notice that similar to head there is no case for []. n is used as index for Vec A and Fin. The constructors for Fin only use suc, therefore the type Fin zero is not inhabited and the cases for [] do not apply.

By case splitting on the vector first we could have obtained the term [] $!i$. By case splitting on i we notice that no constructor for Fin zero exists. Therefore, this case cannot occur, because the type of the argument is uninhabited. It's impossible to call the function, because we cannot construct an argument of the correct type. In this example we can either omit the case or explicitly state that the argument is impossible to construct, by replacing it with (), allowing us to omit the definition of the right-hand side of the equation.

```
[]        ! () -- no right-hand side
```

The other two cases are straightforward. For index zero we return the head of the vector. For index suc i we call _!_ recursively with the smaller index and the tail of the vector. Notice that the types for the recursive call change. The tail of the vector xs and the smaller index i are indexed over the predecessor of n.

2.1.3 Propositions as Types

An more in depth explanation and an overview over the history of the idea can be found in Wadlers paper of the same name [Wad15].

FOL	MLTT	Agda
$\forall x \in A : P(x)$	$\Pi_{x:A} P(x)$	`(x : A) → P x`
$\exists x \in A : P(x)$	$\Sigma_{x:A} P(x)$	`Σ[ x ∈ A ] P x` mit `_,_ : (x : A) → P x → Σ A P`
$P \wedge Q$	$P \times Q$	`A × B`
$P \vee Q$	$P + Q$	`A ⊎ B`
$P \Rightarrow Q$	$P \to Q$	`A → B`
t	**1**	`tt : ⊤`
f	**0**	`⊥`

2.1.4 Notions of Equality and Equality Types

In the last section we saw how we can encode propositions from propositional and predicate logic as types. One of the most important proposition is equality i.e. the proposition that given two terms $a, b : A$ that a and b are equal. When using Agda for theorem proving we have to express propositions like $a+b = b+a$ and $2 = 1$, which could be true or false, to be able to prove or disprove them. In type theory and therefore in Agda we have to consider different notions of equality.

When defining a program rule like truth $=$ 42 we are making an *equality judgement*. The symbol truth is *definitional equal* to 42. A judgment is always true. We define one term to be equal to another one i.e. we allow Agda to reduce the left hand side of the equality to the right hand side.

The next notion of equality is *computational equality*. Two terms t_1 and t_2 are computational equal if they reduce to the same term. For example, given the above definition of $+$ the terms $0 + (0 + n)$ and n are computational equal, because using the first rule in the definition of plus $0+(0+n)$ β-reduces to n. On the other hand $n+0$ and n are not computational equal, because for a free variable n none of the program rules for $+$ can be used to reduce the term further. Therefore computational equality is not the right notion of equality, because the later equality should also hold.

To talk about the equality of two terms we have to use *propositional equality* i.e. we have to define a proposition representing the fact that two terms of type A are equal. This proposition is usually encapsulated in an *equality type* of A.

$$
\begin{aligned}
&\mathsf{infix}\ 4\ \_\equiv\_ \\
&\mathsf{data}\ \_\equiv\_\ \{A : \mathsf{Set}\}\ (x : A) : A \to \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{refl} : x \equiv x
\end{aligned}
$$

The type $x \equiv y$ represents the proposition that x and y are equal. The only way to construct evidence for the proposition is using the refl constructor i.e. if x and y are actually the same.

This notion of equality has the usualy expected properties like transitivity, symmetry and congruence. The definition of cong shows the typical way of working with equality proofs.

```
cong : ∀ {A B x y} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

We expect that $\equiv$ is a congruence relation i.e. if x and y are equal then fx and fy should also be equal. We cannot produce a value of type $f\,x \equiv f\,y$ because the two are not equal. By pattern matching in the argument of type $x \equiv y$ i.e. by inspecting the evidence that x and y are equal we obtain more information about the goal. Because refl can only be constructed if the two values are the same the two variables are unified. We therefore have to produce a proof that fx is equal to itself, which is given by reflexivity.

By pattern matching on variables used in the equality type we can obtain more information about the goal. Either because the constructors them self restrict the use of the variables or because the terms used in the equality type can be reduced further. Consider the following example.

```
+-identʳ : ∀ n → n + 0 ≡ n
+-identʳ 0       = refl
+-identʳ (suc m) = cong suc (+-identʳ m)
```

By pattern matching on the variable n we obtain two cases, one for each constructor (this is analogous to a proof by exhaustion). In both cases the term $n\ \ 0$ can now be reduced further. In the 0 case we obtain $0 + 0$ on the left hand side, which can be reduced to 0. The return type simplifies to $0 \equiv 0$ for which we can simply construct evidence using refl.

The second case is more complex. The left hand side still contains the free variable m, but reduces to suc $(m + 0)$ using the second rule for _+_. Using a recursive call we obtain evidence for $m + 0 \equiv m$. The recursive call to obtain evidence for a smaller case corresponds to the use of the induction hypothesis in an inductive proof. By applying suc on both sides of the equality we obtain a proof for the correct proposition.

We obtained a non obvious equality by using just definitional and computational equality together with the analogs of proofs by exhaustion and induction. This proof can now be used to rewrite arbitrary terms containing terms corresponding to or containing the left- or right-hand of the equality.

2.1.5 Termination Checking

The definition of non-terminating functions entails logical inconsistency. Agda therefore only allows the definition terminating functions. Due to the undecidability of the halting problem Agda uses a heuristic termination checker. The termination checker proofs termination by observing structural recursion. Consider the following definitions of List and map.

```
data List (A : Set) : Set where
  _::_ : A → List A → List A
  []   : List A

map : {A B : Set} → (A → B) → (List A → List B)
map f (x :: xs) = f x :: map f xs
map f []        = []
```

The [] case does not contain a recursive call. In the _::_ case the recursive call to map occurs on a structural smaller argument i.e. xs is a subterm of the argument $x :: xs$. Because elements of List A are finite the function map terminates for every argument.

Sized Types

In more complex recursive functions the structural recursion can be obscured. Agda does not inline functions containing pattern matches during termination checking and therefore cannot proof the termination. A common example are recursive calls in lambdas, which are passed to higher order functions like map and ≫=.

Consider a monad like List or Maybe. It is not obvious that the argument of the continuation of $\ggg$= is a subterm of the first argument.

TODO: explain better via [SEMI-CONTINUOUS SIZED TYPES AND TERMINATION ABEL]

A possibility to still proof termination are Sized Types. Agda has a special builtin, well-ordered type Size.

```
open import Agda.Builtin.Size public
  renaming ( SizeU to SizeUniv ) -- sort SizeUniv
  using ( Size                   -- Size : SizeUniv
        ; Size<_                 -- Size<_ : Size → SizeUniv
        ; ↑_                     -- ↑_ : Size → Size
        ; _⊔ˢ_                   -- _⊔ˢ_ : Size → Size → Size
        ; ∞ )                    -- ∞ : Size
```

By augmenting a data type with an index of type Size its size can be represented on the type level. If a value of type Size decreases with every recursive call, the functions has to termination, because there exist no infinitely decreasing chains on elements of Size.

A common idiom for data types is to mark all non-inductive constructors and recursive occurrences with an arbitrary size i and all inductive constructors with the next larger size $\uparrow i$. The size therefore corresponds to the height of the tree described by the term (+ the initial height for the lowest non-inductive constructor).

A common example for sized types are rose trees. The size index can be intuitively thought of as the height of the tree.

```
data Rose (A : Set) : Size → Set where
  rose : ∀ {i} → A → List (Rose A i) → Rose A (↑ i)
```

When fmap for rose trees is defined in terms of map the termination is obscured. The argument of the functions passed to map is not recognized as structurally smaller than the given rose tree. Using size annotations we can fix this problem.

```
map-rose : {A B : Set} {i : Size} → (A → B) → (Rose A i → Rose B i)
map-rose f (rose x xs) = rose (f x) (map (map-rose f) xs)
```

By pattern matching on the argument of type Rose A of size $\uparrow i$ we obtain a List of trees of size i. The recursive calls via map therefore occur on smaller trees. Therefore the functions has to terminates.

In this case inlining the call of map would also solve the termination problem. When generalizing the definition of the tree from List to an arbitrary functor this wouldn't be possible. In many cases inlining all helper functions is either not feasible or would lead to large and unreadable programs.

2.1.6 Strict Positivity

In a type system with arbitrary recursive types, it is possible to to implement a fixpoint combinator and therefore non terminating functions without explicit recursion. As explained in section 2.1.5 this entails logical inconsistency. Agda allows only strictly positive data types. A data type D is strictly positive if all constructors are of the form

$$A_1 \to A_2 \to \cdots \to A_n \to D$$

where A_i is either not inductive (does not mention D) or are of the form

$$A_1 \to B_2 \to \cdots \to B_n \to D$$

where B_j is not inductive. By restricting recursive occurrences of a data type in its definition to strict positive positions strong normalization is preserved.

Container

Because of the strict positivity requirement it is not allowed to apply generic type constructors to inductive occurrences of a data type in its definition. The reason for this restriction is that a type constructor is not required to use its argument only in strictly positive positions. To still work generically with type constructors or more precise functors we need a more restrictive representation, which only uses its argument in a strictly positive position. One representation of such functors are containers.

Containers are a generic representation of data type, which store values of an arbitrary type. They were introduced by Abbott, Altenkirch and Ghani [AAG03]. A container is defined by a type of shapes S and a type of positions for each of its shapes $P : S \to \mathcal{U}$. Usually containers are denoted $S \triangleright P$. A common example are lists. The shape of a list is defined by its length, therefore the shape type is $\mathbb{N}$. A list of length n has exactly n places or positions containing data. Therefore, the type of positions is $\Pi_{n:\mathbb{N}}\text{Fin } n$ where $\text{Fin } n$ is the type of natural numbers smaller than n. The extension of a container is a functor $[\![S \triangleright P]\!]$, whose lifting of types is given by

$$[\![S \triangleright P]\!]\ X = \sum_{s:S} P s \to X.$$

A lifted type corresponds to the container storing elements of the given type e.g. $[\![\ \mathbb{N} \triangleright \text{Fin}\ ]\!]\ A \cong$ List A. The second element of the dependent pair sometimes called position function. It assigns each position a stored value. The functors action on functions is given by

$$[\![S \triangleright P]\!]\ f\ \langle s, pf\rangle = \langle s, f \circ pf\rangle.$$

We can translate these definition directly to Agda. Instead of a data declaration we can use record declarations. Similar to other languages records are pure product types. A record in Agda is an n-ary dependent product type i.e. the type of each field can contain all previous values.

```
record Container : Set₁ where
  constructor _▷_
  field
    Shape : Set
    Pos : Shape → Set
open Container public
```

As expected, a container consists of a type of shapes and a dependent type of positions. Notice that Container is an element of Set_1, because it contains a type from Set and therefore has to be larger. Next we define the lifting of types i.e. the container extension, as a function between universes.

```
open import Data.Product using (Σ-syntax; _,_) -- TODO: define and explain earlier
open import Function using (_∘_)
⟦_⟧ : Container → Set → Set
⟦ S ▷ P ⟧ A = Σ[ s ∈ S ] (P s → A)
```

Using this definition we can define fmap for containers.

```
fmap : ∀ {A B C} → (A → B) → (⟦ C ⟧ A → ⟦ C ⟧ B)
fmap f (s , pf) = (s , f ∘ pf)
```

2.2 Curry

Curry [HKM95] is a functional logic programming language. It combines paradigms from functional programming languages like Haskell with those logical languages Prolog. Curry is based on Haskell i.e. its syntax and semantics not involving nondeterminism closely resemble Haskell. Curry integrates logical features, such as nondeterminism and free variables with a few additional concepts.

When defining a function with overlapping patterns on the left-hand side of equations all matching right-hand sides are executed. This introduces non determinism. Nondeterminism is integrated as an ambient effect i.e. the effect is not represented at type level. The simplest example of a nondeterministic function is the choice operator `?`.

```
(?) :: A -> A -> A
x ? _ = x
_ ? y = y
```

Both equations always match, therefore both arguments are returned i.e. `?` introduces a nondeterministic choice between its two arguments. Using choice we can define a simple nondeterministic program.

```
coin :: Int
coin = 0 ? 1

twoCoins :: Int
twoCoins = coin + coin
```

`coin` chooses non-deterministically between 0 and 1. Executing coin therefore yields these two results. When executing `twoCoins` the two calls of `coin` are independent. Both choose between 0 and 1, therefore `twoCoins` yields the results 0, 1, 1 and 2.

2.2.1 Call-Time-Choice

Next we will take a look at the interactions between nondeterminism and function calls.

```
double :: Int -> Int
double x = x + x

doubleCoin :: Int
doubleCoin = double coin
```

When calling `double` with a nondeterministic value two behaviors are conceivable. The first possibility is that the choice is moved into the function i.e. both `x` chose independent of each other yielding the results 0, 1, 1 and 2. The second possibility is choosing a value before calling the function and choosing between the results for each possible argument. In this case both `x` have the same value, therefore the possible results are 0 and 2. This option is called Call-Time-Choice and it is the one implemented by Curry.

Similar to Haskell, Curry programs are evaluated lazily. The evaluation of an expression is delayed until its result is needed and each expression is evaluated at most once. The later is important when expressions are named and reused via `let` bindings or lambda abstraction. The named expression is evaluated the first time it is needed. If the result is needed again the old value is reused. This behavior is called sharing. Usually function application is defined using the `let` primitive. Applying a non variable expression to a function introduces a new intermediate result, which bound using `let`.

$$(\lambda x.\sigma)\tau = \texttt{let}\ y = \tau\ \texttt{in}\ \sigma[x \mapsto y]$$

We therefore expect a variable bound by a `let` to behave similar to one bound by a function. This naturally extends Call-Time-Choice to `let`-bindings in lazily evaluated languages.

```
sumCoin :: Int
sumCoin = let x = coin in x + x
```

As expected this function yields the results 0 and 2.

2.2.2 Permutation Sort

Introduce Free Variables + Explain for Later Example

Chapter 3

Algebraic Effects

Algebraic effects are computational effects, which can be described using an algebraic theory. Examples of effects include I/O, exceptions, nondeterminism, state, delimited continuations and more [Bau18]. Handlers for algebraic effects were first presented by Gordon and Plotkin as a generalization of exception handlers []. Handlers can be used to describe non-algebraic operations, which include operations with scopes such as `catch` and `once`. Wu et al. describe a problem with this approach when multiple handlers interact. The ordering of handlers induces a semantic for the program, but simultaneously the handlers also delimit scopes. The correct ordering for scoping and semantics maybe not coincide. Wu et al. fix this problem by introducing new syntactic constructs to delimit scopes, removing the responsibility from the handler [WSH14; Pir+18].

Section 3.1 gives a concrete definition for algebraic effects. Section 3.2 describes the implementation using free monads in Agda. The following sections describe the implementation of unscoped and scoped effects in the first order setting. They focus on implementation details specific to Agda like termination checks. The scoped effects are implemented using explicit scope delimiters as described by Wu et al. [WSH14].

3.1 Definition

The following definition is similar to the one given by Bauer [Bau18].

An algebraic theory consists of a signature describing the syntax of the operations together with a set of equations. A signature is a set of operation symbols together with an arity set and an additional parameter. Operations of this form are usually denoted with a colon and $\rightsquigarrow$ between the three sets, suggesting that they describe special functions.

$$\Sigma = \{\mathrm{op}_i : P_i \rightsquigarrow A_i\}_{i \in \mathbf{N}}$$

Given a signature Σ we can build terms over a set of variables $\mathbb{X}$.

$$x \in \mathrm{Term}_\Sigma(\mathbb{X}) \quad \text{for} \quad x \in \mathbb{X} \qquad \mathrm{op}_i(p, \kappa) \in \mathrm{Term}_\Sigma(\mathbb{X}) \quad \text{for} \quad p \in P_i, \kappa : A_i \to \mathrm{Term}_\Sigma(\mathbb{X})$$

Terms can be used to form equations of the form $x \mid l = r$. Each equation consists of two terms l and r over a set of variables x. A signature Σ_T together with a set of equations $\mathcal{E}_T$ forms an algebraic theory T.

$$T = (\Sigma_T, \mathcal{E}_T)$$

An interpretation I of a signature is given by a carrier set $|I|$ and an interpretation for each operation $[\![\cdot]\!]_I$. An interpretation of an operation op_i is given by a functions to the carrier set $|I|$, that takes an additional parameter from P_i and $|A_i|$ parameters as a function from the arity set to the carrier set $|I|$.

$$[\![\mathrm{op}_i]\!]_I : P_i \times |I|^{A_i} \to |I|$$

Given a function $\iota : \mathbb{X} \to |I|$, assigning each variable a value, we can give an interpretation for terms $[\![\cdot]\!]_{(I,\iota)}$.

$$[\![x]\!]_{(I,\iota)} = \iota(x)$$
$$[\![\mathrm{op}_i(p, \kappa)]\!]_{(I,\iota)} = [\![op_i]\!]_I(p, \kappa \circ [\![\cdot]\!]_{(I,\iota)})$$

An interpretation for T is called T-model if it validates all equations in $\mathcal{E}_T$.

3.1.1 Free Model

For each algebraic theory T we can generate a so called free model $\mathrm{Free}_T(\mathbb{X})$, which validates all equations.

3.2 Free Monads

The syntax of an algebraic effect is described using the free monad. The usual definition of the free monad in Haskell is the following.

```haskell
data Free f a = Pure a | Impure (f (Free f a))

instance (Functor f) => Monad (Free f) where
  return = Pure

  Pura x    >>= k = k x
  Impure fa >>= k = Impure (fmap (>>= k) fa)
```

As the name suggests, the free monad is the free object in the category of monads, therefore the following holds.

1. For every (endo)functor F the functor Free F is a monad
2. For every natural transformation from an (endo)functor F to a monad G exists a monad homomorphism from Free F to G
3. As a consequence of (2) taking the natural transformation to be the identity on F, for every monad exists a monad homomorphism from a Free monad.
4. The Free functor is left adjoint and therefore preserves coproducts i.e. Free $F \oplus$ Free $G \cong$ Free $(F \oplus G)$

When defining the free monad in Agda we cannot use an arbitrary functor as in Haskell, because it would violate the strict positivity requirement. Instead we will represent the functor as the extension of a container as described in section 2.1.6.

```agda
data Free′ (C : Container) (A : Set) : Set where
  pure : A → Free′ C A
  impure : ⟦ C ⟧ (Free′ C A) → Free′ C A
```

The free monad represents an arbitrary branching tree with values of type A in its leafs. The pure constructor builds leafs containing just a value of type A. The impure constructor takes a value of the Free monad lifted by the container functor of C. Based on the choice of container the actual value could contain an arbitrary number of values i.e. Free monads or subtrees. Furthermore for each shape from the Shape type the number of subtrees can differ or contain additional arbitrary values. The container has no access to the type parameter of the free monad.

The $\gg=$ operator for free monads traverses the trees and applies the continuation to the values stored in the leafs, replacing them. $\gg=$ therefore substitutes leafs with new subtrees generated from the values stored in each leaf.

3.2.1 Functors à la carte

When modelling effects each functor represents the syntax i.e. the operations of an effect. For containers each shape corresponds to an operation symbol and the type of position for a shape corresponds to the arity set for the operation. The additional parameter of an operation is embedded in the shape. The free monad over a container describes a program using the effects syntax i.e. its the free model for the algebra without the equations. To combine the syntax of multiple effects we can combine the underlying functors, because the free monad preserves coproducts.

The approach described by Wu et al. is based on "Data types à la carte"[Swi08]. The functor coproduct is modelled as the data type `data (f :+: g) a = Inl (f a) | Inr (g a)`, which is again a `Functor` in `a`.

In Agda functors are represented as containers, a concrete data type not a type class. Containers are closed under multiple operations, coproducts being one of them [AAG03]. The coproduct of two containers F and G is the container whose shape is the disjoint union of Fs and Gs shapes and whose position function pos is the coproduct mediator of Fs and Gs position functions.

$$\_\oplus\_ : \mathsf{Container} \to \mathsf{Container} \to \mathsf{Container}$$
$$(Shape_1 \rhd Pos_1) \oplus (Shape_2 \rhd Pos_2) = (Shape_1 \uplus Shape_2) \rhd [\ Pos_1\ ,\ Pos_2\ ]$$

The functor represented by the coproduct of two containers is isomorphic to the functor coproduct of their representations. The container without shapes is neutral element for the coproduct of containers. This allows us to define n-ary coproducts for containers.

$$\mathsf{sum} : \mathsf{List\ Container} \to \mathsf{Container}$$
$$\mathsf{sum} = \mathsf{foldr}\ (\_\oplus\_)\ (\bot \rhd \lambda())$$

To generically work with arbitrary coproducts of functors we will define two utility functions. Given a value $x : A$ we want to be able to inject it into any coproduct mentioning A. Given any coproduct mentioning A we want to be able to project a value of type A from the coproduct, if A is the currently held alternative.

In the “Data types à la carte”[Swi08] approach the type class `:<:` is introduced. `:<:` relates a functor to a coproduct of functors, marking it as an option in the coproduct. `:<:`s functions can be used to inject values into or maybe extract values from a coproduct. The two instances for `:<:` mark F as an element of the coproduct if it’s an on the left-hand side of the coproduct (in the head) or if it’s already in the right-hand side (in the tail).

```
class (Syntax sub, Syntax sup) => sub :<: sup where
  inj :: sub m a -> sup m a
  prj :: sup m a -> Maybe (sub m a)

instance {-# OVERLAPPABLE #-} (Syntax f, Syntax g) => f :<: (f :+: g) where
  inj = Inl
  prj (Inl a) = Just a
  prj _ = Nothing

instance {-# OVERLAPPABLE #-} (Syntax h, f :<: g) => f :<: (h :+: g) where
  inj = Inr . inj
  prj (Inr ga) = prj ga
  prj _ = Nothing
```

The two instances overlap resulting in possible slower instance resolution. Furthermore, `:+:` is assumed to be right associative and only to be used in a right associative way to avoid backtracking.

Because in Agda the result of $\_\oplus\_$ is another container, not just a value of a simple data type, instance resolution using $\_\oplus\_$ is not as straight forward as in Haskell and in some cases extremely slow[1].

This implementation of the free monad uses an approach similar to the Idris effect library [Bra13]. The free monad is not parameterised over a single container, but a list ops of containers. This has the benefit that we cannot associate coproducts to the left by accident. The elements of the list are combined later using sum. To track which functors are part of the coproduct we introduce the new type $\_\in\_$.

$$\mathsf{data}\ \_\in\_\ \{\ell : \mathsf{Level}\}\ \{A : \mathsf{Set}\ \ell\}\ (x : A) : \mathsf{List}\ A \to \mathsf{Set}\ \ell\ \mathsf{where}$$
$$\quad \mathsf{instance}$$
$$\quad\quad \mathsf{here} : \forall\ \{xs\} \to x \in x :: xs$$
$$\quad\quad \mathsf{there} : \forall\ \{y\ xs\} \to \{\!\{\ x \in xs\ \}\!\} \to x \in y :: xs$$

The type $x \in xs$ represents the proposition that x is an element of xs. The two constructors can be read as rules of inference. One can always construct a proof that x is in a list with x in its head and given a proof that $x \in xs$ one can construct a proof that x is also in the extended list $y :: xs$.

[1] I encountered cases where type checking of overlapping instances involving $\_\oplus\_$ did not seem to terminate.

The two instances still overlap resulting in $\mathcal{O}(c^n)$ instance resolution. Using Agdas internal instance resolution can be avoided by using a tactic to infer _∈_ arguments. For simplicity the following code will still use instance arguments. This version can easily be adapted to one using macros, by replacing the instance arguments with correctly annotated hidden ones.

Using this proposition we can define functions for injection into and maybe projection out of coproducts.

```
inject : ∀ {C ops ℓ} {A : Set ℓ} → C ∈ ops → ⟦ C ⟧ A → ⟦ sum ops ⟧ A
inject here           (s , pf) = (inj₁ s) , pf
inject (there ⦃ p ⦄) prog    with inject p prog
... | s , pf = (inj₂ s) , pf

project : ∀ {C ops ℓ} {A : Set ℓ} → C ∈ ops → ⟦ sum ops ⟧ A → Maybe (⟦ C ⟧ A)
project here            (inj₁ s , pf)  = just (s , pf)
project here            (inj₂ _ , _ ) = nothing
project there           (inj₁ _ , _ ) = nothing
project (there ⦃ p ⦄) (inj₂ s , pf)  = project p (s , pf)
```

Both inject and project require a proof/evidence that specific container is an element of the list used to construct the coproduct.

Let us consider inject first. By pattern matching on the evidence we acquire more information about type sum *ops*. In case of here we know that *op* is in the head of the list i.e. that the given value C is the same as the one in the head of the list. Therefore the Shape types are the same and we can use our given s and pf to construct the coproduct. In case of there we obtain a proof that the container is in the tail of the list, which we can use to make a recursive call. By pattern matching on and repackaging the result we obtain a value of the right type.

project functions similarly. By pattern matching on the proof we either know that the value we found has the correct type or we obtain a proof for the tail of the list allowing us to make a recursive call.

3.2.2 The Free Monad for Effect Handling

Using the coproduct machinery we can now define a version of the free monad, suitable for working with effects. In contrast to the first definition, this free monad is parameterized over a list of containers. In the impure constructor the containers are combined using sum. The parameterization over a list ensures that the containers are not combined prematurely.

```
data Free (ops : List Container) (A : Set) : {Size} → Set where
  pure : ∀ {i} → A → Free ops A {i}
  impure : ∀ {i} → ⟦ sum ops ⟧ (Free ops A {i}) → Free ops A {↑ i}
```

Next we define utility functions for working with the free monad. inj and prj provide the same functionality as the ones used by Wu el al. inj allows to inject syntax into a program whose signature allows the operation. prj allows to inspect the next operation of a program, restricted to a specific signature. Furthermore we add the functions op and upcast. op generates the generic operation for any operation symbol. upcast transformes a program using any signature to one using a larger signature. Notice that upcast prserves the size of its input, because it just traverses the tree and repackages the contents.

```
inj : ∀ {C ops A} → ⦃ C ∈ ops ⦄ → ⟦ C ⟧ (Free ops A) → Free ops A
inj ⦃ p ⦄ = impure ∘ inject p

prj : ∀ {C ops A i} → ⦃ C ∈ ops ⦄ → Free ops A {↑ i} → Maybe (⟦ C ⟧ (Free ops A {i}))
prj ⦃ p ⦄ (pure x)    = nothing
prj ⦃ p ⦄ (impure x) = project p x

op : ∀ {C ops} → ⦃ C ∈ ops ⦄ → (s : Shape C) → Free ops (Pos C s)
op s = inj (s , pure)

upcast : ∀ {C ops A i} → Free ops A {i} → Free (C :: ops) A {i}
```

```
upcast (pure x) = pure x
upcast (impure (s , κ)) = impure (inj₂ s , upcast ∘ κ)
```

The free monad is indexed over an argument of Type Size. pure values have an arbitrary size. When constructing an impure value the new value is strictly larger than the ones produced by the containers position function. The size annotation therefore corresponds to the height of the tree described by the free monad. Using the annotation it's possible to proof that functions preserve the size of a value or that complex recursive functions terminate.

Consider the following definition of fmap for the free monad[2].

```
fmap _<$>_ : {F : List Container} {i : Size} → (A → B) → Free F A {i} → Free F B {i}
f <$> pure x          = pure (f x)
f <$> impure (s , pf) = impure (s , (f <$>_) ∘ pf)

fmap = _<$>_
```

fmap applies the given function f to the values stored in the pure leafs. The height of the tree is left unchanged. This fact is wittnessed by the same index i on the argument and return type.

In contrast to fmap, bind does not preserve the size. bind replaces every pure leaf with a subtree, which is generated from the stored value. The resulting tree is therefore at least as high as the given one. Because there is no + for sized types the only correct size estimate for the returned value is "unbounded". The return type is not explicitly indexed, because the compiler correctly inferes ∞.

```
_>>=_ : ∀ {ops} → Free ops A → (A → Free ops B) → Free ops B
pure x          >>= k = k x
impure (s , pf) >>= k = impure (s , (_>>= k) ∘ pf)

_>>_ : ∀ {ops} → Free ops A → Free ops B → Free ops B
ma >> mb = ma >>= λ _ → mb
```

To complete our basic set of monadic functions we also define ap.

```
_⊛_ _<*>_ : ∀ {ops} → Free ops (A → B) → Free ops A → Free ops B
pure f ⊛ ma = f <$> ma
impure (s , pf) ⊛ ma = impure (s , (_⊛ ma) ∘ pf)

_<*>_ = _⊛_
```

3.2.3 Properties

This definition of the free monad is a functor because it satisfies the two functor laws. Both properties are proven by structural induction over the free monad. Notice that to proof the equality of the position functions, in the induction step, the axiom of extensionality is invoked.

```
fmap-id : ∀ {ops} → (p : Free ops A) → fmap id p ≡ p
fmap-id (pure x)          = refl                                                          (1)
fmap-id (impure (s , pf)) = cong (impure ∘ (s ,_)) (extensionality (fmap-id ∘ pf))
```

```
fmap-∘ : ∀ {ops} (f : B → C) (g : A → B) (p : Free ops A) →
           fmap (f ∘ g) p ≡ (fmap f ∘ fmap g) p
fmap-∘ f g (pure x)          = refl                                                      (2)
fmap-∘ f g (impure (s , pf)) = cong (impure ∘ (s ,_)) (extensionality (fmap-∘ f g ∘ pf))
```

This definition of the free monad also satisfies the three monad laws.

```
bind-identˡ : ∀ {ops} (f : A → Free ops B) (x : A) → (pure x >>= f) ≡ f x              (3)
bind-identˡ f x = refl
```

[2]in the following code A, B and C are arbitrary types

$$
\begin{aligned}
&\mathsf{bind\text{-}ident^r} : \forall\ \{ops\}\ (x : \mathsf{Free}\ ops\ A) \to (x \gg\!= \mathsf{pure}) \equiv x \\
&\mathsf{bind\text{-}ident^r}\ (\mathsf{pure}\ x) \qquad\quad = \mathsf{refl} \\
&\mathsf{bind\text{-}ident^r}\ (\mathsf{impure}\ (s\ ,\ pf)) = \mathsf{cong}\ (\mathsf{impure} \circ (s\ ,\_))\ (\mathsf{extensionality}\ (\mathsf{bind\text{-}ident^r} \circ pf))
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
&\mathsf{bind\text{-}assoc} : \forall\ \{ops\}\ (f : A \to \mathsf{Free}\ ops\ B)\ (g : B \to \mathsf{Free}\ ops\ C)\ (p : \mathsf{Free}\ ops\ A) \to \\
&\qquad ((p \gg\!= f) \gg\!= g) \equiv (p \gg\!= (\lambda\ x \to f\ x \gg\!= g)) \\
&\mathsf{bind\text{-}assoc}\ f\ g\ (\mathsf{pure}\ x) \qquad\quad = \mathsf{refl} \\
&\mathsf{bind\text{-}assoc}\ f\ g\ (\mathsf{impure}\ (s\ ,\ pf)) = \mathsf{cong}\ (\mathsf{impure} \circ (s\ ,\_))\ (\mathsf{extensionality}\ (\mathsf{bind\text{-}assoc}\ f\ g \circ pf))
\end{aligned} \tag{5}
$$

3.3 Handler

An effect handler interprets and removes the syntax of an effect and injects corresponding code into the program. Some handlers manipulate syntax of other effects or the structure of the program itself. The handler for an algebraic effect defines it's semantics.

All handlers will have the same basic structure. They will take a program i.e. a variable of type Free C A , where the head of C is the effect interpreted by the handler. Each handler produces a program without the interpreted syntax i.e. just the tail of C in it's effect stack and potentially modify the type A to one modelling the result of the effect. For example a handler for exceptions would remove exception syntax and transform a program producing a value of type A to one producing a value of type $E \uplus A$, either an exception or a result.

A simple but important handler is the one handling the empty effect stack and therefore the Void effect. A program containing just Void syntax contains no impure constructors, because Void has no operations. Therefore, we can always produce a value of type A. This handler is important, because it can be used to escape the Free context after all other effects are handled.

```
run : Free [] A → A
run (pure x) = x
```

3.3.1 Nondeterministic Choice

The nondeterminism effect has two operations _⁇_ and fail. _⁇_ introduces a nondeterministic choice between two execution paths and fail discards the current path. We therefore have a nullary and a binary operation, both without additional parameters.

$$\Sigma_{\text{Nondet}} = \{⁇ : \mathbf{1} \rightsquigarrow \mathbf{2}, \text{fail} : \mathbf{1} \rightsquigarrow \mathbf{0}\}$$

Expressed as a container we have a shape with two constructors, one for each operation and both without parameters.

```
data Nondetˢ : Set where ⁇ˢ failˢ : Nondetˢ
```

When constructing the container we assign the correct arities to each shape.

```
Nondet : Container
Nondet = Nondetˢ ▷ λ where
  ⁇ˢ    → Bool
  failˢ → ⊥
```

We can now define smart constructors for each operation. These are not the generic operations, but helper functions based on them. The generic operations take no parameters an alaways use pure as continutation. These versions of the operations already process the continuations parameter.

```
_⁇_ : ∀ {ops} → ⦃ Nondet ∈ ops ⦄ → Free ops A → Free ops A → Free ops A
p ⁇ q = inj (⁇ˢ , (if_then p else q))

fail : ∀ {ops} → ⦃ Nondet ∈ ops ⦄ → Free ops A
fail = inj (failˢ , λ())
```

With the syntax in place we can now move on to semantics and define a handler for the effect. By introducing pattern declarations for each operations the handler can be simplified. Furthermore we introduce a pattern for other operations, i.e. those who are not part of the currently handled signature.

```
pattern Other s κ = impure (inj₂ s , κ)
pattern Fail κ = impure (inj₁ failˢ , κ)
pattern Choice κ = impure (inj₁ ??ˢ , κ)
```

The handler interprets Nondet syntax and removes it from the program. Therefore Nondet is removed from the front of the effect stack and the result is wrapped in a List. The List contains the results of all successful execution paths.

```
solutions : ∀ {ops} → Free (Nondet :: ops) A → Free ops (List A)
```

The pure constructor represents a program without effects. The singleton list is returned, because no nondeterminism is used in a pure calculation.

```
solutions (pure x)      = pure (x :: [])
```

The fail constructor represents an unsuccessful calculation. No result is returned.

```
solutions (Fail κ)      = pure []
```

In case of a Choice both paths can produce an arbitrary number of results. We execute both programs recursively using solutions and collect the results in a single List.

```
solutions (Choice κ)  = _++_ <$> solutions (κ true) ⊛ solutions (κ false)
```

In case of syntax from another effect we just execute solutions on every subtree by mapping the function over the container. Note that the newly constructed value has a different type. Since Nondet syntax was removed from the tree the proof for _∈_, which is passed to impure changes.

```
solutions (Other s κ) = impure (s , solutions ∘ κ)
```

3.3.2 State

The state effect has two operations get and put. The whole effect is parameterized over the state type s.

get simply returns the current state. The operation takes no additional parameters and has s positions. This can either be interpreted as get being an s-ary operation (one child for each possible state) or simply the parameter of the continuation being a value of type s.

put updates the current state. The operation takes an additional parameter, the new state. The operation itself is unary i.e. there is no return value, therefore tt is passed to the rest of the program.

$$\Sigma_{State} = \{\text{get} : \mathbf{1} \rightsquigarrow s, \text{put} : s \rightsquigarrow \mathbf{1}\}$$

As before we will translate this definition in a corresponding container.

```
data Stateˢ (S : Set) : Set where
  getˢ : Stateˢ S
  putˢ : S → Stateˢ S

State : Set → Container
State S = Stateˢ S ▷ λ where
  getˢ      → S
  (putˢ _) → ⊤

pattern Get κ = impure (inj₁ getˢ , κ)
pattern Put s κ = impure (inj₁ (putˢ s) , κ)
```

To simplify working with the State effect we add smart constructors. These correspond to the generic operations.

```
get : ∀ {ops S} → ⦃ State S ∈ ops ⦄ → Free ops S
get = inj (getˢ , pure)

put : ∀ {ops S} → ⦃ State S ∈ ops ⦄ → S → Free ops ⊤
put s = inj (putˢ s , pure)
```

Using these defintions for the syntax we can define the handler for State.

The effect handler for State takes an initial state together with a program containing the effect syntax. The final state is returend in addition to the result.

```
runState : ∀ {ops S} → S → Free (State S :: ops) A → Free ops (S × A)
```

A pure calculation doesn't change the current state. Therefore, the initial is also the final state and returned in addition to the result of the calculation.

```
runState s₀ (pure x)      = pure (s₀ , x)
```

The continuation/position function for get takes the current state to the rest of the calculation. By applying s_0 to κ we obtain the rest of the computation, which we can evaluate recursively.

```
runState s₀ (Get κ)       = runState s₀ (κ s₀)
```

put updates the current state, therefore we pass the new state s_1 to the recursive call of runState.

```
runState _ (Put s₁ κ)     = runState s₁ (κ tt)
```

Similar to the handler for Nondet we apply the handler to every subterm of non State operations.

```
runState s₀ (Other s κ) = impure (s , runState s₀ ∘ κ)
```

Example

Here is a simple example for a function using the State effect. The function tick returns tt and as side effect increases the state.

```
tick : ∀ {ops} → ⦃ State ℕ ∈ ops ⦄ → Free ops ⊤
tick = do i ← get ; put (1 + i)
```

Using the runState handler we can evaluate programs, which use the State effect.

```
(run $ runState 0 $ tick ≫ tick) ≡ (2 , tt)
```

Properties

```
module StateLaws (S : Set) (ops : List Container) (s₀ : S) where
  go : Free (State S :: ops) A → Free ops (S × A)
  go = runState {_} {ops} s₀

  put-put : {s₁ s₂ : S} → (go $ put s₁ ≫ put s₂) ≡ (go $ put s₂)
  put-put = refl

  put-get : {s : S} → (go $ put s ≫ get) ≡ (go $ put s ≫ pure s)
  put-get = refl

  get-get : {k : S → S → Free (State S :: ops) A}
    → (go $ get ≫= λ s → get ≫= k s) ≡ (go $ get ≫= λ s → k s s)
  get-get = refl

  get-put : (go $ get ≫= put) ≡ (go $ pure tt)
  get-put = refl
```

3.4 Scoped Effects

- Modularity - Combination of Effects - Semantics choosen by order of handlers - problem with scoping operations and syntax of different effects

To correctly handle operations with local scopes Wu et al. introduced scoped effects [WSH14]. They presented two solutions to explicitly declare how far an operations scopes over a program using arbitrary syntax. In the following section we will implement the scoped effect Cut using the first order approach in Agda. The central idea is to add new effect syntax, representing explicit scope delimiters. Whenever an opening delimiter is encountered the handler can be run again on the scoped program.

3.4.1 Cut and Call

First we will define the syntax for the new effect and its delimiters.

```
data Cutˢ : Set where cutfailˢ : Cutˢ
data Callˢ : Set where bcallˢ ecallˢ : Callˢ

pattern Cutfail = impure (inj₁ cutfailˢ , _)
pattern BCall κ = impure (inj₁ bcallˢ , κ)
pattern ECall κ = impure (inj₁ ecallˢ , κ)

Cut Call : Container
Cut  = Cutˢ ▷ λ _ → ⊥
Call = Callˢ ▷ λ _ → ⊤
```

The Cut effect has just a single operation, cutfail. cutfail can only be used in a context with nondeterminism. When cutfail is called it will prunes all unexplored branches and call fail. The Agda implementation of the handlers is identical to the one by Wu et al.

The handler itself calls the function go, which accumulates the unexplored alternatives in its second argument. fail is the neutral element for ⁇ and therefore the default argument. Since this handler is not orthogonal (i.e it interacts with another effect) Nondet is required to be in scope, but its position is irrelevant.

To prove termination we mark the second argument with an arbitrary but fixed size i. The position functions for each case return subterms indexed with a smaller size. Recursive calls to go with these terms as argument therefore terminate.

```
call : ⦃ Nondet ∈ ops ⦄ → Free (Cut :: ops) A → Free ops A
call = go fail
  where
    go : ⦃ Nondet ∈ ops ⦄ → Free ops A → Free (Cut :: ops) A {i} → Free ops A
```

In case of a pure value no cutfail happened. We therefore return a calculation choosing between the value and the earlier separated alternatives.

```
    go q (pure a)       = (pure a) ⁇ q
```

In case of a cutfail we terminate the current computation by calling fail and prune the alternatives by ignoring q.

```
    go _ Cutfail        = fail
```

To interact with Nondet syntax we have to find it. We have a proof that the Nondet effect is an element of the effect list. Whenever we find syntax from another effect we can therefore try to project the Nondet option from the coproduct. Notice that prj hides the structural recursion but decreases the Size index. We can therefore still proof that the function terminates.

```
    go q p@(Other s κ) with prj {Nondet} p
```

The case for ⁇ separates the main branch from the alternative. Using go the Cut syntax is removed from both alternatives, but the results are handled asymmetrically. The left option is direclty passed to the recursive call of go. The handeled right option is the new alternative for the left one and therefore could be pruned if left contains a cutfail call.

```
... | just (⁇ˢ , κ′) = go (go q (κ′ false)) (κ′ true)
```

When encountering a fail we continue with the accumulated alternatives.

```
... | just (failˢ , _ ) = q
```

Syntax from other effects is handled as usual.

```
... | nothing         = impure (s , go q ∘ κ)
```

With the handler for Cut in place we can define the handler for the scope delimiters. The implementation is again similar to the one presented by Wu et al., but to proof termination we again have to add Size annotations to the functions.

The bcall and ecall handler remove the scope delimiter syntax from the program and run call (the handler for cut) at the begining of each scope. Whenever a BCall is found the handler ecall is used to handle the rest of the program. ecall searches for the end of the scope and returns the program up to that point. The rest of the program is the result of the returned program.

A valid upper bound for the size of the rest of the program is i, the size of the program before separating the syntax after the closing delimiter. This fact is curcial to proof that the recursive calls to bcall and ecall using ≫= terminate.

Calling the handler on the extracted program guaranties that the handler does not interact with syntax outside the intended scope. Nested scopes are handled using recursive calls to ecall if BCall operations are encountered while searching a closing delimiter.

Since the delimiters could be placed freely it is possible to mismatch them. If we encounter a closing before and operening delimiter, we know that they are mismatched. Wu et al. use Haskells `error` function to terminate the program. In Agda we are not allowed to define partial functions, therefore we have to handle the error. We could either correct the error and just continue or short circuit the calculation using exceptions in form of e.g. a Maybe monad. For simplicity we will use the former approach. In a real application it would be advisable to inform the programmer about the error, either using exceptions or at least trace the error.

```
bcall : ⦃ Nondet ∈ ops ⦄ → Free (Call :: Cut :: ops) A {i} → Free (Cut :: ops) A
ecall : ⦃ Nondet ∈ ops ⦄ → Free (Call :: Cut :: ops) A {i}
  → Free (Cut :: ops) (Free (Call :: Cut :: ops) A {i})

bcall (pure x)      = pure x
bcall (BCall κ)     = upcast (call (ecall (κ tt))) ≫= bcall
bcall (ECall κ)     = bcall (κ tt) -- Unexpected ECall! We just fix the error.
bcall (Other s κ) = impure (s , bcall ∘ κ)

ecall (pure x)      = pure (pure x)
ecall (BCall κ)     = upcast (call (ecall (κ tt))) ≫= ecall
ecall (ECall κ)     = pure (κ tt)
ecall (Other s κ) = impure (s , ecall ∘ κ)
```

Using the handlers defined above we can define a handler for scoped Cut syntax, which removes Cut and Call syntax simultaneously. The delimiters and correctly scoped Cut syntax is removed using bcall and potential unscoped Cut syntax is removed with a last use of call. The function call′ is a smart constructor for the scope delimiters.

```
runCut : ⦃ Nondet ∈ ops ⦄ → Free (Call :: Cut :: ops) A → Free ops A
runCut = call ∘ bcall

call′ : ⦃ Call ∈ ops ⦄ → Free ops A → Free ops A
call′ p = do op bcallˢ ; x ← p ; op ecallˢ ; pure x
```

3.5 Call-Time Choice as Effect

Bunkenburg presented an approach to model call-time choice as a stack of scoped algebraic effects [Bun19]. In this section we will extend the nondeterminism effect from section 3.3.1 to one modelling call-time choice.

As explained in section 2.2.1, call-time choice semantics describe the interaction between sharing and nondeterminism. The current implementation of Nondet does not support sharing i.e. it is not possible for two choice to be linked. Based on Bunkenburgs implementation we will make two changes to the nondeterminism effect. Each choice is augmented with an optional identifier consisting of a triple of natural numbers, called *choice id*. The first two are used to identity the current scope and will be refereed to as *scope id*. The third number identifies the choice inside its scope. Furthermore, instead of producing a list the handler will now produce a tree of choices. This change allows to choose the evaluation strategy, e.g. depth first or breath first search, independent of the handler.

3.5.1 Effectful Data Structures

In Haskell and Curry ambient effects, like partiality, tracing and nondeterminism, can occur in components of data structures. Each argument of a constructor could be an effectful computation For example, in Curry the tail of a list could be a nondeterministic choice between two possible tails.

We want to simulate this behavior with algebraic effects. The effects have to be modelled explicitly using the Free type. We will lift data types using a standard construction, which is commonly used to simulate ambient effects [Abe+05; DCT19; CDB19]. The following example of an effectful List demonstrates the the general construction[3].

```
data Listᴹ (ops : List Container) (A : Set) : {Size} → Set where
  nil  : Listᴹ ops A {i}
  cons : Free ops A → Free ops (Listᴹ ops A {i}) → Listᴹ ops A {↑ i}
```

$\mathsf{List}^{\mathsf{M}}$ *ops* A represents a List A in whose components effects from the given effect stack *ops* can occur. To easily construct and work with lifted values we introduce smart constructors in the form of pattern synonyms.

```
pattern []ᴹ                = pure nil
pattern _::ᴹ_ mx mxs = pure (cons mx mxs)
```

The size annotations on the lifted data structures are needed to proof termination of structural recursive functions. To pattern match on a lifted value we have to use $\gg\!=$. Therefore the structural recursion is obscured.

```
_++ᴹ_ : Free ops (Listᴹ ops A {i}) → Free ops (Listᴹ ops A) → Free ops (Listᴹ ops A)
mxs ++ᴹ mys = mxs >>= λ where
  nil              → mys
  (cons mx mxs′) → mx ::ᴹ mxs′ ++ᴹ mys
```

Normalization of Effectful Data

Based on Bunkenburg's code [Bun19] we will introduce a type class for normalizing effectful data structures i.e. moving interleaved Free layers to the outside using $\gg\!=$.

```
record Normalform (ops : List Container) (A B : Set) : Set where
  field
    nf : A → Free ops B
open Normalform {{...}} public

!_ : {{ Normalform ops A B }} → Free ops A → Free ops B
! mx = mx >>= nf
```

[3] In the following code effectful data structures and lifted versions of functions are marked with a suffix $^{\mathrm{M}}$

The type class allow to normalize elements of type A (intuitively containing effectful calculations) to computations producing of elements of type B (intuitively a version of B without the effects). Instead of A and B we could have parameterized the type class over a type family of an effect stack, with nf producing an element of the type family applied to an empty stack. This implementation would allow us to restrict the normalizable types, but prohibit us from producing elements of standard data types.

In contrast to Bunkenburg's implementation we do not expect a lifted argument. Simplifying the type and introducing the helper function !_ removes the need for auxiliary normalization lemmas for pure and impure values in proofs. The extra degree of freedom, introduced by a monadic argument, was not used in the original implementation.

```
instance
  ℕ-normalform : Normalform ops ℕ ℕ
  Normalform.nf ℕ-normalform = pure

  Listᴹ-Normalform : ⦃ Normalform ops A B ⦄ →
    Normalform ops (Listᴹ ops A {i}) (List B)
  Normalform.nf Listᴹ-Normalform nil             = pure []
  Normalform.nf Listᴹ-Normalform (cons mx mxs) = ⦇ ! mx ∷ ! mxs ⦈
```

The data stored in an effectful list could also be effectful and therefore has to be normalized. We simply require a Normalform instance for the stored type. To allow normalization of general types and effectful data structures containing them, we have to implement dummy instances for data types like builtin natural numbers or booleans.

3.5.2 Sharing Handler

The idea of the following sharing handler is identical to the one presented by Bunkenburg. Thanks to the more flexible infrastructure described in the earlier sections we are able to define a more modular handler and avoid some inlining, necessary to prove termination in Coq. Similar to Cut the sharing handler is not orthogonal to other effects, but interacts with existing Nondet syntax.

The scoping operation share takes an additional argument, the unique identifier for the created scope. Similar to put the parameter is part of the container shape.

```
data Shareˢ : Set where bshareˢ eshareˢ : ℕ × ℕ → Shareˢ
pattern BShare n κ = impure (inj₁ (bshareˢ n) , κ)
pattern EShare n κ = impure (inj₁ (eshareˢ n) , κ)

Share : Container
Share = Shareˢ ▷ λ _ → ⊤


bshare : ⦃ NonDet ∈ ops ⦄ → Free (Share ∷ ops) A {i} → Free ops A
eshare : ⦃ NonDet ∈ ops ⦄ → ℕ → ℕ × ℕ → Free (Share ∷ ops) A {i}
  → Free ops (Free (Share ∷ ops) A {i})

bshare (pure x)        = pure x
bshare (BShare sid κ) = eshare 0 sid (κ tt) ≫= bshare
bshare (EShare sid κ) = bshare (κ tt) -- missmatched scopes, we just continue!
bshare (Other s κ)     = impure (s , bshare ∘ κ)

eshare next sid (pure x)           = pure (pure x)
eshare next sid (BShare sid′ κ) = eshare 0 sid′ (κ tt) ≫= eshare next sid
eshare next sid (EShare sid′ κ) = pure (κ tt) -- usually test that sid′ = sid
eshare next sid p@(Other s κ) with prj {NonDet} p
... | just (??ˢ _ , κ′) = inj $ ??ˢ (just $ sid , next) , eshare (1 + next) sid ∘ κ′
... | just (failˢ , κ′)  = inj $ failˢ , λ()
... | nothing             = impure (s , eshare next sid ∘ κ)
```

Next we will define the share operator as described by Bunkenburg. The operator generates new unique identifiers using a State effect.

```
record Shareable (ops : List Container) (A : Set) : Set₁ where
  field
    shareArgs : A → Free ops A
open Shareable {{...}} public

begin end : {{ Share ∈ ops }} → ℕ × ℕ → Free ops ⊤
begin id = op (bshareˢ id)
end   id = op (eshareˢ id)

share : {{ Shareable ops A }} → {{ Share ∈ ops }} → {{ State (ℕ × ℕ) ∈ ops }} →
  Free ops A → Free ops (Free ops A)
share p = do
  (i , j) ← get
  put (1 + i , j)
  pure do
    begin (i , j)
    put (i , 1 + j)
    x ← p
    x′ ← shareArgs x
    put (1 + i , j)
    end (i , j)
    pure x′

instance
  shareable-ℕ : Shareable ops ℕ
  Shareable.shareArgs shareable-ℕ = pure
```

3.5.3 Examples

```
CTC : List Container
CTC = State (ℕ × ℕ) :: Share :: NonDet :: []

runCTC : {{ Normalform CTC A B }} → Free CTC A → List B
runCTC p = dfs empty $ run $ runNonDet $ bshare $ evalState (0 , 0) (! p)

coin : {{ NonDet ∈ ops }} → Free ops ℕ
coin = pure 0 ??′ pure 1

doubleCoin : {{ Share ∈ ops }} → {{ State (ℕ × ℕ) ∈ ops }} → {{ NonDet ∈ ops }} →
  Free ops ℕ
doubleCoin = share coin >>= λ c → ⦇ c + c ⦈

runDoubleCoin : runCTC doubleCoin ≡ 0 :: 2 :: []
runDoubleCoin = refl
```

3.5.4 Laws of Sharing

Chapter 4

Higher Order

To address problems of the first order approach, like mismatched scope delimiters and explicit control over the continuation, Wu et al. introduced a second approach utilizing higher order syntax [WSH14]. Bunkenburg already tried to implement some scoped effects using this approach, but failed due to limitations of Coq [Bun19].

Chapter 3 demonstrated how the first order approach could be transferred to Agda utilizing container representations for functors and sized types. This chapter focuses on the implementation of effects using higher order syntax. It partially follows Bunkenburg's approach, because the limitation of Coq described by him does not exist in Agda. Therefore this chapter focuses on expanding his approach further. Due to other limitations, needed for the consistency of Agda, this approach ultimately failed again.

4.1 Higher Order Syntax

In the first order approach effects only store possible continuations. In the higher order approach by Wu et al. scoped effects store possible continuations as well as the part of the program in their scope.

In the first order approach the functors, which describe the syntax, are applied to types of the form `Free F A` i.e. programs consisting of syntax as described by `Free F`, which produce values of type `A`. In the higher order approach these functors are generalized to higher order functors i.e. in Haskell a data type of kind `(* -> *) -> * -> *`. These functors are applied to `Free F` and `A` separately.

Because the program type is separated in syntax and result type it is possible to store programs, which use the same signature but produce values of different types. Consider the following definition of the higher order exception syntax as given by Wu et al.

```
data HExc e m a = Throw e | forall x. Catch (m x) (e -> m x) (x -> m a)
```

The catch operation stores the computation in the scope of the catch block `m x`, the handler producing an alternative value in case of an error `e -> m x` as well as the continuation `x -> m a`. The three programs agree on an arbitrary, but per `catch` fixed type `x`.

4.2 Representing Strictly Positive Higher Order Functors

```
record Container : Set₂ where
  constructor _◁_/_
  field
    Shape : Set₁
    Pos : Shape → Set
    Ctx : (s : Shape) → Pos s → Set → Set
```

Chapter 5

Scoped Algebras

Due to the deep-rooted problem with the higher order approach from chapter 4 it seemed reasonable to search for another formulation of scoped effects, which does not rely on existential types. Piróg et al. [Pir+18] presented a novel formulation for scoped operations and their algebras, which fulfils this requirement. Their approach does not describe the combination of multiple effects, but due to the different structure the approach seemed worth exploring nonetheless.

This chapter transfers the basic implementation from Piróg et al. to Agda, derives a different implementation of fold, based on the work by Fu and Selinger [FS18], for their monad to aid termination checking and presents an idea for modularisation of their algebras.

5.1 The Monad E

Piróg et al. describe a monad, which is suited for modelling operations with scopes [Pir+18]. The monad is the result of rewriting a slightly modified version of the monad from “Effect Handlers in Scope” i.e. the monad used in chapter 4. They separate the signature into one describing effects with scopes Γ and one describing the algebraic effects Σ. The monad E can be described as the fixpoint of the following equation, where the coend (the integral sign) denotes an existential types.

$$EA \cong A + \Sigma(EA) + \int^{X \in \mathscr{C}} \Gamma(EX) \times (EA)^X$$

This representation looks appropriate considering our earlier definition. An element of the monad is either a value of A, an operation without scopes, i.e. a functor applied to `Free F A` or an operation with scopes. Operations with scopes (e.g. `catch`) always quantified over some type to store subprograms over the same signature but with different local result type. Furthermore they provided a continuation, which transformed values of the internal result type to the result type of the whole program. [GAMMA] The following Agda data type represents the above monad.

```
data ProgE (Σ Γ : Container) (A : Set) : Set₁ where
  var : A → ProgE Σ Γ A
  op  : ⟦ Σ ⟧ (ProgE Σ Γ A) → ProgE Σ Γ A
  scp : {X : Set} → ⟦ Γ ⟧ (ProgE Σ Γ X) → (X → ProgE Σ Γ A) → ProgE Σ Γ A
```

Notice again that this definition requires X to be a smaller type than E i.e. it’s not sufficient to model deep effects as explained in chapter 4. Piróg et al. also not size issues with this definition.

Piróg et al. rewrite this definition using a left Kan extension and derive the following equivalent monad without the existential type.

$$EA \cong A + \Sigma(EA) + \Gamma(E(EA))$$

An equivalent monad without the existential type is a promising candidate for modelling deep effects in Agda, because it avoids the size issues.

The monad models scopes using the double E layer. The outer layer corresponds to the part of program in scope. By evaluating it (i.e. the part of the program in scope) one obtains a value of type EA, the rest of the program producing a result of type A. This resulting program corresponds to the continuation in the higher order approach, which was a function mapping from the the result of the scoped program to the rest of the whole program.

5.2 The `Prog` Monad

In this section we will define the central monad for this approach. Because signatures are split into effects with and without scopes we will define effects as pairs of functors and therefore Containers.

```
record Effect : Set₁ where
  field
    Ops Scps : Container
open Effect public
```

The functions ops and scps combine the corresponding signatures of a List of Effects.

```
ops scps : List Effect → Container
ops  = sum ∘ mapᴸ Ops
scps = sum ∘ mapᴸ Scps
```

Using these helper functions we can define a modular version of the monad by Piróg et al [Pir+18]. It's equivalent to the fixpoint and the Haskell definition from the paper (assuming that signatures are given by strictly positive functors).

```
data Prog (effs : List Effect) (A : Set) : Set where
  var : A                                   → Prog effs A
  op  : ⟦ ops effs ⟧ (Prog effs A)          → Prog effs A
  scp : ⟦ scps effs ⟧ (Prog effs (Prog effs A)) → Prog effs A
```

Piróg et al. also derive algebras for the above monad, which correctly model scoped effects. To implement their recursion scheme we have to define $\gg=$ for Prog *effs* A. Notice that Prog is a truly nested or non-regular data type [BM98].

Defining recursive functions for these data types is more complicated. The "direct" implementations of fmap and $\gg=$ do not obviously terminate. Augmenting Prog with size annotations leads to problems when proving the monad laws[1]. Instead we will define a generic fold, which we can reuse for all functions traversing elements of Prog *effs* A.

5.2.1 Folds for Nested Data Types

Defining recursion schemes for complex recursive data types is a common problem. Fu and Selinger demonstrate a general construction for `fold`s for nested data types in Agda [FS18]. Following their construction and adapting it to arbitrary branching trees leads to a sufficiently strong `fold` to define all important functions for Prog *effs* A.

Index Type The first part of the construction by Fu and Selinger is to define the correct index type for the data structure. The index type describes the recursive structure of the data type i.e. how the type variables at each level are instantiated. For complex mutual recursive data types the index type takes the form of a branching tree. For Prog *effs* the one type variable is either instantiated with the value type A (pure) or some number of Prog *effs* layers applied to the current type variable (op, scp). Our index type therefore just counts the number of Prog *effs* layers. Natural numbers are sufficient.

Next Fu and Selinger define a type level function, which translates a value of the index type to its corresponding type. For Prog *effs* A this function just applies Prog *effs* n times to A. The operator _^_ applies a function n-times to a given argument x i.e. it represents n-fold function application, usually denoted $f^n(x)$.

```
_^_ : ∀ {ℓ} {C : Set ℓ} → (C → C) → ℕ → C → C
(f ^ 0)     x = x
(f ^ suc n) x = f ((f ^ n) x)
```

[1]As in chapter 4 the monad laws only hold for size ∞, but working with size ∞ in proofs leads to termination problems. The scp constructor can produce an arbitrary number of Prog *effs* layers. This forces us to use our induction hypothesis on not obviously smaller terms, but the size annotation is assumed to be ∞.

Recursion Scheme Next we will define fold for Prog *effs* A. The fold produces an arbitrary $\mathbb{N}$ indexed value. The type family P determines the result type at each index. The fold produces a value of type $P\,n$ for a given value of type (Prog *effs* ^ n) A. Furthermore the `fold` takes functions for processing substructures. As usual, the arguments of these function correspond to those of the constructors they processes with recursive occurrences of the type itself already processed. The cases are obtained by pattern matching on the index type and the value of type (Prog *effs* ^ n) A i.e. the value of the type depending on the index.

```
fold : (P : ℕ → Set) → ∀ n →
  (A                                        → P 0      ) →
  (∀ {n} → P n                              → P (suc n) ) →
  (∀ {n} → ⟦ ops  effs ⟧ (P (suc n))        → P (suc n) ) →
  (∀ {n} → ⟦ scps effs ⟧ (P (suc (suc n))) → P (suc n) ) →
  (Prog effs ^ n) A → P n
fold P 0       a v o s x       = a x
fold P (suc n) a v o s (var x) = v (      fold P n             a v o s  x)
fold P (suc n) a v o s (op  x) = o (map ( fold P (suc n)       a v o s) x)
fold P (suc n) a v o s (scp x) = s (map ( fold P (suc (suc n)) a v o s) x)
```

Notices that for the op and spc constructors the recursive occurrences and potential additional values are determined by the Container. The functions therefore processor arbitrary Container extensions, which contain solutions for the corresponding index. When implementing those two cases we cannot apply fold directly to recursive occurrences, because those depend on the choice of Container. map for Containers exactly captures the correct behavior.

The fold function, based on the approach by Fu and Selinger, is similar to the one by Piróg et al. Piróg et al. derived their fold by deriving algebras for a second monad. In “Syntax and Semantics for Operations with Scopes” [Pir+18] they show that the two monads are equivalent and therefore transfer the fold from the second monad to the one used in their Haskell implementation. Following Fu and Selinger we derived essential the same fold and implemented it directly i.e. without the use of $\gg=$.

Example Using fold we can implement $\gg=$ for Prog *effs* A. The continuations k passed to $\gg=$ should just effect the inner most layer. Intuitively, when using bind an a value constructed with scp we traverse the outer layer and call bind on the inner one recursively. To implement $\gg=$ we fold over the first argument. We could fix the arbitrary n to be 1, but defining a version generic in n is useful for later proofs. The $\mathbb{N}$ indexed type bind-P defines the type for the intermediate result at every layer.

```
bind-P : ∀ A B effs → ℕ → Set
bind-P A B effs 0 = A
bind-P A B effs (suc n) = Prog effs ^ (suc n) $ B
```

$\gg=$ corresponds to variable substitution i.e. it replaces the values stored in var leafs with subtrees generated from these values. Calling k on a program with zero layers (a value of type A) would produce a program with one layer. Therefore, $\gg=$ can only be called on programs with at least one layer. The lowest layer is extended with the result of k. For all numbers greater than zero we produce a program with the same structure but a different value type. For layer zero (i.e. values) we could either produce a value of type Prog *effs* B or leave the value of type A unchanged. In both cases we have to handle the lowest var constructors differently.

```
bind : ∀ n → (Prog effs ^ suc n) A → (A → Prog effs B) →
  (Prog effs ^ suc n) B
bind {effs} {A} {B} n ma k = fold (bind-P A B effs) (suc n) id (λ where
  {0}     x → k x
  {suc n} x → var x
  ) op scp ma
```

We left the values unchanged, therefore the var constructor at layer zero has to produce value of type Prog *effs* B by calling k. In all other cases we replace each constructor with itself to leave these parts unchanged.

Using $\gg\!=$ for a program with one layer and var as return we can define a monad instance for Prog *effs*. In contrast to Haskell this automatically defines a functor and an applicative instances, because these can be defined in terms of $\gg\!=$ and return[2].

```
instance
  Prog-RawMonad : RawMonad (Prog effs)
  Prog-RawMonad = record { return = var ; _>>=_ = bind 0 }
```

The record (which is used as a type class) is part of the Agda standard library. It is named RawMonad, because it does not enforce the monad laws. We will prove the laws in section 5.2.3 using a technique from section 5.2.2.

5.2.2 Induction Schemes for Nested Data Types

Following the examples by Fu and Selinger [FS18] we can generalize the fold from section 5.2.1 to a dependently type version, an induction principle. In the induction principle the result type/proposition P is generalized to a dependent type. Therefore the induction principle allows for proofs of predicates by induction without explicit recursion.

The types for the four functions, corresponding to the three constructors and the base case, also have to be generalized.

```
ind : (P : (n : ℕ) → (Prog effs ^ n) A → Set) → ∀ n →
```

In the base case the given value of type A is the handled value. Therefore the value is bound and passed to the proposition.

```
( (x : A) → P 0 x ) →
```

The var can be handled similar to the examples by Fu and Selinger. An additional hidden parameter x for the smaller, recursively handled value is introduced. Because x represents the recursively handled type its part of the smaller results type. Furthermore, the parameter is used to describe the currently handled value var x. var x is therefore part of the functions result type.

Under the Curry Howard corresponds this function (ignoring the index type) can be read as the proposition $\forall x.P(x) \Rightarrow P(\text{var } x)$ i.e. the induction step for the constructor var.

```
(∀ {n x}       → P n x → P (suc n) (var x) ) →
```

Similar to fold dealing with op and scp is more complicated, because they represent arbitrarily branching nodes. Remember that a containers position function maps from a type of positions, which depends on the values shape, to the contained values. Hence, assuming the containers values corresponds to assuming a position function. Analogues to the fold the result for the recursively handled value is function, mapping from positions, for the assumed shape, to proofs for the proposition for the there contained values. The value for the position is obtained using the assumed position function κ. The currently handled value is constructed using s and κ. Notice that the each of the four parameters depends on all its predecessors.

```
(∀ {n} s {κ} → ((p : Pos (ops  effs) s) → P (1 + n) (κ p)) → P (suc n) (op  (s , κ)) ) →
(∀ {n} s {κ} → ((p : Pos (scps effs) s) → P (2 + n) (κ p)) → P (suc n) (scp (s , κ)) ) →
(x : (Prog effs ^ n) A) → P n x
```

The actual implementation of the induction principle is straight forward and similar to the one for fold. The composition with the position function corresponds to the call to map.

```
ind P 0 a v o s x                  = a x
ind P (suc n) a v o s (var x)      = v   (ind P n             a v o s x)
ind P (suc n) a v o s (op  (c , κ)) = o c (ind P (suc n)       a v o s ∘ κ)
ind P (suc n) a v o s (scp (c , κ)) = s c (ind P (suc (suc n)) a v o s ∘ κ)
```

[2] We automatically obtain the functions <$>, <*>, pure and ≫.

5.2.3 Proving the Monad Laws

Left Identity First we will prove the left identitiy law. Because the value passed $\gg=$ constructed using return i.e. var both sides of the equality can be directly evaluated to $k\ a$.

$$\begin{aligned}&\mathsf{bind\text{-}ident}^{\mathsf{l}} : \forall\ \{A\ B : \mathsf{Set}\}\ \{a\}\ \{k : A \to \mathsf{Prog}\ \mathit{effs}\ B\} \to\\&\quad (\mathsf{return}\ a \gg= k) \equiv k\ a\\&\mathsf{bind\text{-}ident}^{\mathsf{l}} = \mathsf{refl}\end{aligned}$$

Right Identity When proving the right identity the value passed to $\gg=$ can be an arbitrarily complex program. Therefore the proposition is not "obvious" as it was the case with bind-ident$^{\mathsf{l}}$.

$$\mathsf{bind\text{-}ident}^{\mathsf{r}} : \{A : \mathsf{Set}\}\ (ma : \mathsf{Prog}\ \mathit{effs}\ A) \to (ma \gg= \mathsf{return}) \equiv ma$$

$\gg=$ is recursive in its first argument, therefore we will prove the proposition by induction on ma. To use the induction scheme we have to define a proposition for every layer. Since $\gg=$ does not change values we simply produce a value of type $\top$ at layer zero. For all other we proof the proposition for the appropriate bind i.e. the one called by fold to handle a value for the given n.

$$\mathsf{bind\text{-}ident}^{\mathsf{r}}\ \{\mathit{effs}\}\ \{A\} = \mathsf{ind}\ (\lambda\{\ 0\ \_ \to \top\ ;\ (\mathsf{suc}\ n)\ p \to \mathsf{bind}\ n\ p\ \mathsf{var} \equiv p\ \})$$

We call the induction scheme with the given value with one layer, therefore the initial n is 1. The "proof" for values can be inferred, because for each value it's just a value of the unit type.

$$1\ \_$$

To proof that the proposition holds for values constructed using var we have to case split on the number of layers n. For 0 we have to proof that var x is equal to itself, because $\gg=$ leaves values unchanged. This proof forms the basis for our induction.

For the second case we are given a program x with suc n layers and an induction hypothesis IH that the proposition holds for a call to bind n. We have to prove that the proposition holds for bind (suc n) and the new program var x. By applying the $\gg=$ rule for var (for n greater than zero) we can move var to the outside. The new proposition is var (bind n) $\equiv$ var x. This proposition is equal to the induction hypothesis with var applied to both sides. Therefore we can proof it using congruence.

$$(\lambda\{\ \{0\}\ (\mathsf{tt}) \to \mathsf{refl}\ ;\ \{\mathsf{suc}\ n\}\ \mathit{IH} \to \mathsf{cong}\ \mathsf{var}\ \mathit{IH}\ \})$$

The proofs for the op and scp are identical and similar to the var case. For each shape s we are given a proof for each position of the given shape. We have to proof that the proposition holds for a new program constructed using either op or scp. For both constrcutors bind n calls itself recusrivly on the all contained values i.e. the results of the position function. Using congruence we can simplify the equality to the equality of the position functions. By invoking the axiom of extensionality we can prove the equality for each position. This equality is exactly the one given by the induction hypothesis.

$$\begin{aligned}&(\lambda\ s\ \mathit{IH} \to \mathsf{cong}\ (\mathsf{op}\ \circ (s\ ,\_))\ (\mathsf{extensionality}\ \mathit{IH}))\\&(\lambda\ s\ \mathit{IH} \to \mathsf{cong}\ (\mathsf{scp} \circ (s\ ,\_))\ (\mathsf{extensionality}\ \mathit{IH}))\end{aligned}$$

Associativity The proof for associativity follows the same pattern as the one for the right identity. $\gg=$ is defined by recursion on its first argument. We therefore proof the proposition by induction on ma. In all cases the left-hand side reduces to the induction hypothesis in the same manner as before. Therefore these cases look identical.

$$\begin{aligned}&\mathsf{bind\text{-}assoc} : \forall\ \{A\ B\ C\}\\&\quad (f : A \to \mathsf{Prog}\ \mathit{effs}\ B)\ (g : B \to \mathsf{Prog}\ \mathit{effs}\ C)\ (ma : \mathsf{Prog}\ \mathit{effs}\ A) \to\\&\quad (ma \gg= f \gg= g) \equiv (ma \gg= \lambda\ a \to f\ a \gg= g)\\&\mathsf{bind\text{-}assoc}\ f\ g = \mathsf{ind}\\&\quad (\lambda\ \mathsf{where}\end{aligned}$$

```
    0 p          → ⊤
    (suc n) p → bind n (bind n p f) g ≡ bind n p λ a → bind 0 (f a) g
  ) 1 _
  (λ{ {0} _ → refl ; {suc n} IH → cong var IH})
  (λ s IH → cong (op  ∘ (s ,_)) (extensionality IH))
  (λ s IH → cong (scp ∘ (s ,_)) (extensionality IH))
```

Functor and Applicative Laws The monad laws imply the functor and applicative laws. We can proof them without using explicit induction by rewriting equations involving ≫= using the above law. These proofs are simpler than the ones explicitly involving the definition of ≫= using fold. We can write them using the chain reasoning operators as described by Norell [Nor07].

```
fmap-id : ∀ {effs} {A B : Set} → (ma : Prog effs A) → (id <$> ma) ≡ ma
fmap-id ma = begin
  (id <$> ma)                ≡⟨⟩ -- definition of <$>
  (ma ≫= λ a → var (id a)) ≡⟨⟩ -- definition of id and η-conversion
  (ma ≫= var)               ≡⟨ bind-identʳ ma ⟩
  ma                         ∎
```

5.3 Combining Effects

Similar to chapter 4 we reuse parts of the infrastructure from chapter 3. Each Effect consists of a pair of Containers, on representing scoped and one representing unscoped effects. Therefore, an Effect stack now stores pairs of containers, not containers directly. The type _∈_ together with inj and prj can be reused to model effect constraints.

Due to the component wise combination of Effects, a proof that an Effect is an element of an effect stack implies that an operations from one of the two signatures is part of the corresponding signature of the combined effect.

```
opsInj : e ∈ effs → Ops e ∈ mapᴸ Ops effs
opsInj (here refl) = here refl
opsInj (there p) = there (opsInj p)

scpsInj : e ∈ effs → Scps e ∈ mapᴸ Scps effs
scpsInj (here refl) = here refl
scpsInj (there p) = there (scpsInj p)
```

Using the above proofs we can implement smart constructors/inject functions for scoped and unscoped operations without requiring other evidence.

```
Op : ⦃ e ∈ effs ⦄ → ⟦ Ops e ⟧ (Prog effs A) → Prog effs A
Op ⦃ p ⦄ = op ∘ inj (opsInj p)

Scp : ⦃ e ∈ effs ⦄ → ⟦ Scps e ⟧ (Prog effs (Prog effs A)) → Prog effs A
Scp ⦃ p ⦄ = scp ∘ inj (scpsInj p)
```

To escpe the monad after interpreting all effects we again add the run handler.

```
run : Prog [] A → A
run (var x) = x
```

5.4 Nondet

As first example we will implement the nondeterminism effect. In section 5.4.1 we will implement the example by Piróg et al. to demonstrate that the fold definition from section 5.2.1 is sufficient. Section 5.4.2 will present an idea for modular effects and handlers.

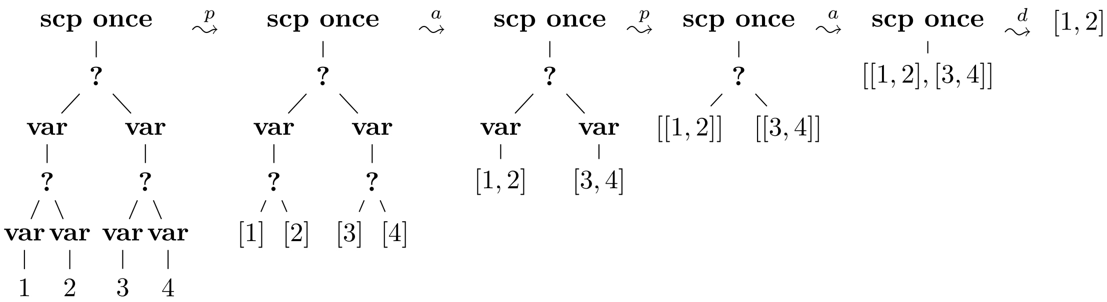

Figure 5.1: Interpretation of **once** (**var** 1 ? **var** 3) $\gg\!=$ $\lambda x.$**var** x ? **var** $(x + 1)$

5.4.1 As Scoped Algebra

The signature for the two algebraic operations is the same as before. Furthermore we define a signature for the scoped operations once.

```
data Choiceˢ : Set where ⁇ˢ : (Maybe CID) → Choiceˢ ; failˢ : Choiceˢ
data Onceˢ   : Set where onceˢ : Onceˢ
```

Nondet$^\mathsf{P}$ corresponds to the data type from the example by Piróg et al. [Pir+18, sec. 6]. The effect has the unary, scoped operation once and the nullary and binary, algebraic operations ⁇ and fail.

```
Nondetᴾ : Effect
Ops  Nondetᴾ = Choiceˢ ▷ λ{ (⁇ˢ _) → Bool ; failˢ → ⊥ }
Scps Nondetᴾ = Onceˢ   ⤳ ⊤
```

A scoped algebra for an $\mathbb{N}$ indexed carrier type C consists of three operations. a the *algebra* for the unscoped operations. p for *promoting* the carrier type when entering a scope. d for *demoting* the carrier type when leaving a scope and interpreting the syntax. Figure 5.4.1 demonstrates how the handler interprets Nondet syntax using the scoped algebra. It corresponds to an image for the other monad from "Syntax and Semantics for Operations with Scopes", but represents programs using the Prog *effs* monad.

The indices for d and a are offset by one compared to the definition by Piróg et al. We won't define our carrier data type in two stages as Piróg et al. but as an n-fold version of a simpler type. This change allows a simpler implementation using the fold from section 5.2.1, but should be insignificantly enough to present the similarities between the to recursion schemes.

```
record ScopedAlgebra (E : Effect) (C : ℕ → Set) : Set where
  constructor ⟨_,_,_⟩
  field
    p : ∀ {n} → C n                        → C (1 + n)
    d : ∀ {n} → ⟦ Scps E ⟧ (C (2 + n)) → C (1 + n)
    a : ∀ {n} → ⟦ Ops  E ⟧ (C (1 + n)) → C (1 + n)
open ScopedAlgebra
```

Given a ScopedAlgebra for an Effect we can interpret its syntax. fold$^\mathsf{P}$ corresponds to slightly modified version of the fold by Piróg et al. Their function works for an arbitrary n to allow recursion. Since, this recursion is abstracted in fold this is not needed. Furthermore they have to inject values explicitly into the carrier type. This happens implicitly due to the handling of the 0-th layer. Notice that the functions from the ScopedAlgebra correspond exactly to the ones expected by the fold from section 5.2.1. The values them self aren't preprocessed and will just get promoted, hence id as passed as the first of the four functions.

```
foldᴾ : ∀ {C : Set → ℕ → Set} → ScopedAlgebra e (C A) →
  Prog (e ∷ []) (C A 0) → C A 1
foldᴾ {C = C} ⟨ p , d , a ⟩ = fold (C _) 1 id p
  (λ{ (inj₁ s , κ) → a (s , κ) }) λ{ (inj₁ s , κ) → d (s , κ) }
```

Just porting the algebra for Nondet by Piróg et al. yields a correct handler for the effect in isolation. The carrier type are iterated Lists.

The implementation of the scoped algebra is straight forward. The algebraic operations are handled as before and the promotion operation corresponds to the earlier handling of values. The demotion operation i.e. the handler for once has to produce an n-fold list, given an $n+1$-fold list. Before applying the handler, the programs in the var nodes are the results for all possible continuations for the program in scope. The elements of the given list are the results of these programs. By acting just on the given list of lists the handler can separate between the results of the different branches in the scoped program.

```
NondetAlg : ScopedAlgebra Nondetᵖ (λ i → List ^ i $ A)
p NondetAlg            = _:: []
a NondetAlg (??ˢ _ , κ) = κ true ++ κ false
a NondetAlg (failˢ , κ) = []
d NondetAlg (onceˢ , κ) with κ tt
... | []     = []
... | x :: _ = x

runNondetᴾ : Prog (Nondetᵖ :: []) A → List A
runNondetᴾ = foldᴾ {C = λ A i → List ^ i $ A} NondetAlg
```

The carrier type can be thought of as the context for the computation. By having contexts of contexts it's possible to differentiate between the state of the computation in scope and the whole computation.

Lastly we will define a smartconstructor for the once operation. To capture the program p in scope, pure is mapped over it. The orignial program is now the outher Prog $effs$ layer i.e. the first and second layers of the first term in figure 5.4.1. The original var x nodes are now var(var x) nodes. Since ≫= just effects the lowest layer it won't change the captured program and just act on it's results.

```
onceᵖ : ⦃ Nondetᵖ ∈ effs ⦄ → Prog effs A → Prog effs A
onceᵖ p = Scp (onceˢ , λ _ → pure <$> p)
```

5.4.2 As Modular Handler

Piróg et al. just present handlers for single effects. This section presents an idea for adapting the handlers from the paper to modular ones.

As in the earlier chapters, when working with modular effects, the general approach is to execute the handlers for each effect one after another [Sch+19]. Each handler interprets the syntax for its effect and leaves the rest in place (or in case of non-orthogonal effects manipulates syntax of certain other effects). The carrier type for these handlers consists of the usual carrier type for these handlers, which is post composed with the type of the program without the interpreted syntax. For example, a handler for exceptions would produce values of type $E \uplus A$, therefore the modular handler produces value of type Prog $effs$ $(E \uplus A)$.

In the context of scoped algebras, the $\mathbb{N}$ indexed carrier type is usually the n-fold of some simpler type C. It seems reasonable to choose an n-fold of Prog $effs \circ C$ as carrier type, because the structure of n layers of Cs has to be preserved to reuse the non modular handler. Simply lifting the n-fold carrier type in the monad is not sufficient. The interleaved monad layers are needed to preserve non-interpreted syntax.

Consider the following implementation of modular handler for nondeterminism. This version of the nondeterminism effect does not support the once operation, because the semantics of the above definition is not the expected one when interacting with other effects. Similar to earlier chapter we introduce patterns to simplify the handler.

```
Nondet : Effect
Ops  Nondet = Choiceˢ ▷ λ{ (??ˢ _) → Bool ; failˢ → ⊥ }
Scps Nondet = Void

pattern Other s κ     = (inj₂ s , κ)
```

```
pattern Choice cid κ = (inj₁ (⁇ˢ cid) , κ)
pattern Fail         = (inj₁ failˢ , _)
```

The new handler has the expected signature, which follows from the above described carrier type.

```
runNondet′ : Prog (Nondet ∷ effs) A → Prog effs (Tree A)
runNondet′ {effs} {A} = fold (λ i → (Prog effs ∘ Tree) ^ i $ A) 1 id
```

Values are now not only injected into the context, but also into the monad.

```
(pure ∘ leaf)
```

The algebraic operations are interpreted as before, except that all results are lifted into the monad. When an operation from a foreign effect is encountered the syntax is just reconstructed. The outer container coproduct is removed by interpreting the Nondet syntax, therefore s already has the correct type (note that Other hides the inj_2). κ produces result for the positions of the given shape, because the recursion is handled by fold. Therefore the implementation is the same as in chapter 3 except that the recursion is hidden.

```
(λ where
  (Choice id κ) → branch id <$> κ true <*> κ false
  Fail          → pure failed
  (Other s κ)   → op (s , κ)
```

The interesting case is the one for scoped operations of foreign effects. Similar to the algebraic operations we have to reconstruct the scoped operation, but the position function has the wrong type. It produces values of type Prog $effs$ ∘ Tree ^ suc n \$ A but expected are values of type Prog $effs$ \$ Prog $effs$ ∘ Tree ^ n \$ A.

To remove the intermediate Tree layer we map the functionhdl over the result of the postion function. hdl traverses the outer tree and recombines the inner trees under the monad i.e. by interleaiving binds it orders the results. On a term level, we are given a list of possible continuations, which we execute on after another, appending their reults.

```
) λ where
  (Other s κ) → scp (s , λ p → hdl <$> κ p)
where
  hdl : ∀ {A} → Tree (Prog effs (Tree A)) → Prog effs (Tree A)
  hdl (branch cid l r) = branch cid <$> hdl l <*> hdl r
  hdl (leaf v)         = v -- no recursive call due to fold
  hdl failed           = var failed
```

In terms of Haskell typeclass the above implementation generalizes if C is a traversable monad, because hdl corresponds to `fmap join . sequence`. This is quite a strong condition, but it does not seems necessary, because we will see another example in section 5.6, which does not follow this pattern. Furthermore, the carrier types of effects are often simple data structures like products, coproducts, lists or trees, which are usually traversable and often monads. Its also unclear if a monad or just a notion of flattening is needed.

Notice that the hdl is similar to the handler function from chapter 4. Because the fold already interpreted parts of the program we do not call the handler recursively, but just join the results.

```
runNondet : Prog (Nondet ∷ effs) A → Prog effs (List A)
runNondet p = dfs empty <$> runNondet′ p

fail : ⦃ Nondet ∈ effs ⦄ → Prog effs A
fail = Op (failˢ , λ())

_⁇_ : ⦃ Nondet ∈ effs ⦄ → Prog effs A → Prog effs A → Prog effs A
p ⁇ q = Op (⁇ˢ nothing , (if_then p else q))
```

5.5 Exceptions

As our first example for a scoped effect in the modular setting we will take a look at exceptions. The syntax for throw is the same as before. Similar to chapter 4 catch has two sub-computations, the program in scope and the handler. The boilerplate code for the syntax is given below.

```
data Throwˢ (E : Set) : Set where throwˢ : (e : E) → Throwˢ E
data Catchˢ           : Set where catchˢ : Catchˢ
data Catchᵖ (E : Set) : Set where
  mainᵖ   : Catchᵖ E
  handleᵖ : (e : E) → Catchᵖ E

Exc : Set → Effect
Ops  (Exc E) = Throwˢ E ⇝ ⊥
Scps (Exc E) = Catchˢ   ⇝ Catchᵖ E

pattern Throw e = (inj₁ (throwˢ e) , _)
pattern Catch κ = (inj₁ catchˢ , κ)
```

The first part of the handler is the same as in the higher order setting. To interpret the scoped operation catch we first execute scoped program in the main^p position. It produces either an exception or the result for the rest of the program. In the later case we just return the result. Notice that the recursive call was taken care of by the fold. In the other case we obtain an exception e with which we can obtain the result of the continuation in the handle^p position i.e. handle the exception. The result of the exception handler is again wrapped in _⊎_. We pass the result along by either unwrapping the program or re-injecting the exception in the program using pure. The same function is used to traverse foreign scopes. Again the function corresponds to the handler from chapter 4, but without the recursive call.

```
runExc : Prog (Exc E :: effs) A → Prog effs (E ⊎ A)
runExc {E} {effs} {A} = fold (λ i → (Prog effs ∘ (E ⊎_)) ^ i $ A) 1 id
  (pure ∘ inj₂)
  ( λ where
    (Throw e)   → pure (inj₁ e)
    (Other s κ) → op (s , κ)
  ) λ where
    (Catch κ) → κ mainᵖ >>= λ where
      (inj₁ e) → κ (handleᵖ e) >>= [ pure ∘ inj₁ , id ]
      (inj₂ x) → x
    (Other s κ) → scp (s , λ p → [ pure ∘ inj₁ , id ] <$> κ p)
```

The smart constructors for the operations follow the known pattern.

```
throw : ⦃ Exc E ∈ effs ⦄ → E → Prog effs A
throw e = Op (throwˢ e , λ())

_catch_ : ⦃ Exc E ∈ effs ⦄ → Prog effs A → (E → Prog effs A) → Prog effs A
p catch h = Scp $ catchˢ , λ where
  mainᵖ       → pure <$> p
  (handleᵖ e) → pure <$> h e
```

5.6 State

To implement the sharing effect as described by Bunkenburg we also need to implement the State effect. The syntax is the same as before.

```
data Stateˢ (S : Set) : Set where
  putˢ : (s : S) → Stateˢ S
  getˢ : Stateˢ S
```

```
State : Set → Effect
Ops (State S) = Stateˢ S ▷ λ{ (putˢ s) → ⊤ ; getˢ → S }
Scps (State S) = Void

pattern Get κ     = (inj₁ getˢ , κ)
pattern Put s₁ κ = (inj₁ (putˢ s₁) , κ)
```

When traversing a foreign scope we have to eliminate the intermediate context. κ produces a value of type Prog *effs* $(S \times (S \to$ Prog *effs* ...)). The lifted pair consists of the final state of the program in scope and the function mapping an initial state to the result of the corresponding result of the continuation. To remove the inner context we simply pass the state along, by applying the function to the given state using eval.

```
runState : Prog (State S :: effs) A → S → Prog effs (S × A)
runState {S} {effs} {A} = fold (λ i → (λ X → S → Prog effs (S × X)) ^ i $ A) 1 id
  (λ x s₀ → pure (s₀ , x))
  (λ where
    (Put s₁ κ) _  → κ tt s₁
    (Get κ)    s₀ → κ s₀ s₀
    (Other s κ) s₀ → op (s , λ p → κ p s₀)
  ) λ where
    (Other s κ) s₀ → scp (s , λ p → eval <$> κ p s₀)
  where
    eval : ∀ {A B} → A × (A → B) → B
    eval (a , f) = f a

evalState : Prog (State S :: effs) A → S → Prog effs A
evalState s₀ p = π₂ <$> runState s₀ p
```

The operations are just the usual generic operations for the effect.

```
get : ⦃ State S ∈ effs ⦄ → Prog effs S
get = Op (getˢ , pure)

put : ⦃ State S ∈ effs ⦄ → S → Prog effs ⊤
put s = Op (putˢ s , pure)
```

5.7 Share

Lastly we will implement the sharing effect. The signature contains just the single, unary, scoped operation, share which creates new sharing scope.

```
data Shareˢ : Set where shareˢ : SID → Shareˢ

Share : Effect
Ops Share = Void
Scps Share = Shareˢ ⇝ ⊤

pattern ShareScp sid κ = (inj₁ (shareˢ sid) , κ)
```

In the handler the passed along choice id is handled similar to the initial state in the State effect. It's part of the iterated carrier type, which allows it to be changed between layers[3]. When handling a sharing scope the captured program is handled with the id of the stored in the shape of the scoping operation. The continuation is accessed via ⋙= and continuous with the outside i.e. the initial id. Scopes of foreign effects don't effect sharing. In contrast to State, the captured program does not influence the continuation, therefore both use the same, initial id.

[3]This is a general pattern with this approach. Handler arguments are turned into function arguments for the carrier type, because the recursion is done via fold. The same can be observed in a setting without algebraic effects and folding handlers [Sch+19].

```
runShare′ : ⦃ Nondet ∈ effs ⦄ → Prog (Share ∷ effs) A → SID → ℕ → Prog effs A
runShare′ {effs} {A} ⦃ p ⦄ = fold (λ i → ((λ X → SID → ℕ → Prog effs X) ^ i) A) 1 id
  (λ z _ _ → var z)
  (λ{ (Other s pf) → case prj (opsInj p) (s , pf) of λ where
    nothing          sid n → op (s , λ p → pf p sid n)
    (just (failˢ  , κ)) sid n → fail
    (just (⁇ˢ _ , κ)) sid n → Op (⁇ˢ (just (sid , n)) , λ p → κ p sid (suc n))
  }) λ where
    (ShareScp sid′ κ) sid n → κ tt sid′ 0 >>= λ r → r sid n
    (Other s κ) sid n → scp (s , λ p → (λ k → k sid n) <$> κ p sid n)

runShare : ⦃ Nondet ∈ effs ⦄ → Prog (Share ∷ effs) A → Prog effs A
runShare p = runShare′ p (0 , 0) 0
```

For simplicity the handler is initial invoked with choice id $(0,0,0)$. This should not be a problem if the numbering of scopes starts with a higher ID, because after calling the Share handler no scopes are duplcaited. If such a situation should arrise the ID passed around by the handler can simply be made optional via Maybe, allowing no IDs for the outside choices.

The Shareable and Normalform infrastructure from chapter 3 can be reused to define the Ágda operator.

The program in the scope is again captured by mapping pure over it. The constrcution of scope ids follow again the implementation by Bunkenburg.

```
share⟨_⟩ : ⦃ Share ∈ effs ⦄ → SID → Prog effs A → Prog effs A
share⟨_⟩ {Scps} {Ops} sid p = Scp (shareˢ sid , λ _ → pure <$> p)

share : ⦃ State SID ∈ effs ⦄ → ⦃ Share ∈ effs ⦄ → ⦃ Shareable effs A ⦄ →
  Prog effs A → Prog effs (Prog effs A)
share {Ops} {Scps} p = do
  (i , j) ← get
  put (i + 1 , j)
  let p′ = do
        put (i , j + 1)
        x ← p
        x′ ← shareArgs x
        put (i + 1 , j)
        pure x′
  pure $ share⟨ i , j ⟩ p′
```

Using Nondet, State and Share we can again simulate call-time choice semantics. As expected, doubling a shared coin yields the results 0 and 2.

```
runCTC : ⦃ Normalform (State SID ∷ Share ∷ Nondet ∷ []) A B ⦄ →
  Prog (State SID ∷ Share ∷ Nondet ∷ []) A → List B
runCTC p = run $ runNondet $ runShare $ evalState (! p) (1 , 1)

coin : ⦃ Nondet ∈ effs ⦄ → Prog effs ℕ
coin = pure 0 ⁇ pure 1

doubleCoin : ⦃ Nondet ∈ effs ⦄ → ⦃ Share ∈ effs ⦄ → ⦃ State SID ∈ effs ⦄ →
  Prog effs ℕ
doubleCoin = do c ← share coin
                ⦇ c + c ⦈

runDoubleCoin : runCTC doubleCoin ≡ 0 ∷ 2 ∷ []
runDoubleCoin = refl
```

In contrast to chapter 4 this approach can also model deep effects without involving universe levels. The definition of an effectful list as well as its typeclass instances from chapter 3 can be reused.

```
doubleHead : ⦃ Nondet ∈ effs ⦄ → ⦃ Share ∈ effs ⦄ → ⦃ State SID ∈ effs ⦄ →
  Prog effs ℕ
```

```
doubleHead = do mxs ← share (coin ::ᴹ []ᴹ)
                ⦇ headᴹ mxs + headᴹ mxs ⦈

runDoubleHead : runCTC doubleHead ≡ 0 :: 2 :: []
runDoubleHead = refl
```

Chapter 6

Conclusion

6.1 Summary

Bibliography

[AAG03] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Categories of Containers". In: *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Andrew D. Gordon. Vol. 2620. Lecture Notes in Computer Science. Springer, 2003, pp. 23–38. DOI: `10.1007/3-540-36576-1\_2`. URL: `https://doi.org/10.1007/3-540-36576-1%5C_2`.

[Abe+05] Andreas Abel et al. "Verifying haskell programs using constructive type theory". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*. Ed. by Daan Leijen. ACM, 2005, pp. 62–73. DOI: `10.1145/1088348.1088355`. URL: `https://doi.org/10.1145/1088348.1088355`.

[Bau18] Andrej Bauer. "What is algebraic about algebraic effects and handlers?" In: *CoRR* abs/1807.05923 (2018). arXiv: `1807.05923`. URL: `http://arxiv.org/abs/1807.05923`.

[BM98] Richard S. Bird and Lambert G. L. T. Meertens. "Nested Datatypes". In: *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*. Ed. by Johan Jeuring. Vol. 1422. Lecture Notes in Computer Science. Springer, 1998, pp. 52–67. DOI: `10.1007/BFb0054285`. URL: `https://doi.org/10.1007/BFb0054285`.

[Bra13] Edwin C. Brady. "Programming and reasoning with algebraic effects and dependent types". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 133–144. DOI: `10.1145/2500365.2500581`. URL: `https://doi.org/10.1145/2500365.2500581`.

[Bun19] Niels Bunkenburg. "Modeling Call-Time Choice as Effect using Scoped Free Monads". MA thesis. Germany: Kiel University, 2019.

[CDB19] Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. "Verifying effectful Haskell programs in Coq". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*. Ed. by Richard A. Eisenberg. ACM, 2019, pp. 125–138. DOI: `10.1145/3331545.3342592`. URL: `https://doi.org/10.1145/3331545.3342592`.

[DCT19] Sandra Dylus, Jan Christiansen, and Finn Teegen. "One Monad to Prove Them All". In: *Art Sci. Eng. Program.* 3.3 (2019), p. 8. DOI: `10.22152/programming-journal.org/2019/3/8`. URL: `https://doi.org/10.22152/programming-journal.org/2019/3/8`.

[FS18] Peng Fu and Peter Selinger. "Dependently Typed Folds for Nested Data Types". In: *CoRR* abs/1806.05230 (2018). arXiv: `1806.05230`. URL: `http://arxiv.org/abs/1806.05230`.

[HKM95] Michael Hanus, Herbert Kuchen, and Juan José Moreno-Navarro. *Curry: A Truly Functional Logic Language*. 1995.

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[Pir+18] Maciej Piróg et al. "Syntax and Semantics for Operations with Scopes". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 809–818. DOI: 10.1145/3209108.3209166. URL: https://doi.org/10.1145/3209108.3209166.

[Sch+19] Tom Schrijvers et al. "Monad transformers and modular algebraic effects: what binds them together". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*. Ed. by Richard A. Eisenberg. ACM, 2019, pp. 98–113. DOI: 10.1145/3331545.3342595. URL: https://doi.org/10.1145/3331545.3342595.

[Swi08] Wouter Swierstra. "Data types à la carte". In: *J. Funct. Program.* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758. URL: https://doi.org/10.1017/S0956796808006758.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[Wad15] Philip Wadler. "Propositions as types". In: *Commun. ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407. URL: https://doi.org/10.1145/2699407.

[WSH14] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect handlers in scope". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 1–12. DOI: 10.1145/2633357.2633358. URL: https://doi.org/10.1145/2633357.2633358.