Implementing a library for scoped algebraic effects in Agda

HTWK Leipzig

Jonas Höfer
Informatik
69555

2020

Abstract

Contents

Chapter 1

Introduction

Algebraic effects were first introduced by Plotkin and Powers [16] as an alternative representation for algebraic computational effects. Effect handlers were introduced by Plotkin and Pretnar [18] as a generalization of exception handlers. They allow modelling non-algebraic effects and are used in conjunction with algebraic effects [19]. Together they form a theoretical well understood representation for a large class of computational effects.

Algebraic effects and handlers have a wide field of applications e.g. in programming as a less boiler plate intensive alternative to monad transformers, as well as to simulate programming semantics in proof assistants [10, 9].

Wu et al. noticed a lack of modularity when combining computational effects with scoping operations [24]. They introduce several solutions for the problem, by introducing new operations and generalizing syntax to delimit scopes.

1.1 Goals

The goal of this thesis is the implementation of an effect library based on the work by Wu et al. [24] and Piróg et al. [15] in the dependently typed, functional programming language Agda [14]. To allow use as a proof assistant, Agda programs are subject to constraints not present in other functional languages such as Haskell. All Agda programs have to be total (and therefore terminate), all data types have to be strictly positive and all universe levels have to be consistent. These restrictions prevent a direct translation of Haskell code to Agda.

This thesis explores multiple Haskell implementations of scoped effects and studies how well these translate to Agda. Furthermore we presents a set of standard solutions for common problems, arising during the implementation of the Haskell approaches in Agda.

Agda's builtin theorem proving capabilities allow direct, machine checked correctness proofs for the implemented library. Alongside the implementation of scoped effects, the library contains proofs for common properties of these effects. Furthermore, the library itself can be used to simulate semantics of other programming languages, such as Curry's [13] call-time choice, and allow verification of equivalent programs written in these languages in Agda. To allow the latter, the library has to be able to represent deep effects i.e. data structures with effectful components.

1.2 Structure

Chapter 2 contains short introductions to Agda and dependent types, Curry and its call-time choice semantics (which are a central example for a scoped effect throughout the thesis) and algebraic effects.

Chapter 3 and 4 explore the two approaches based on "Effect handlers in scope" by Wu et al. [24]. The former of the two approaches was already partially explored by Bunkenburg [8] in Coq and can also be implemented easily in Agda. The latter of the two approaches tries to fix some inherent problems of the first one, but sadly cannot fully be implemented in Agda due to some other problems with universe consistency.

Chapter 5 describes an implementation of a novel representation for scoped effects by Piróg et al. [15] which also fixes the problems of the first approach, while avoiding the universe inconsistencies of

the second one. Furthermore, the implementation explores some ideas to modularize the exemplary Haskell implementation by Piróg et al.

Chapter 6 summaries and compares the three approaches. Furthermore, it gives an overview over related and possible future work.

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 [14]. 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

Agda's 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 $\mathbb{N}$, Set is the type of all (small) types.

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

Ordinary function definitions 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 contains 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 [22]. 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 a dependent type.

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

To avoid Russell's paradox, a hierarchy of universes $\mathcal{U}_1 : \mathcal{U}_2 : \ldots$ is introduced. In Agda the universes are named Set_n, where Set_0 can be abbreviated as Set. 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, but it's possible to lift a type to a higher universe manually. 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 $\to$ 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 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 a non partial version of _!_ for vectors.

```
_!_ : ∀ {A n} → Vec A n → Fin n → A
(x :: _ ) ! zero  = x
(_ :: xs) ! suc i = xs ! i
```

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, because it's impossible to reach this case
```

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.

The above, quite simple examples can also be implemented in Haskell, for example using type level natural numbers. One of the advantages of Agda is the seamless and complete implementation of dependent types. For example, in contrast to a Haskell implementation we could simply reuse the normal type of natural numbers to index other types. Furthermore, we will later see more complex uses of dependent types, which cannot be translated as easily to Haskell.

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 [23].

The idea of propositions as types, also know as Curry-Howard correspondence, relates type theory to logic. As described by Wadler, it's a deep correspondents, relating *propositions to types*, *proofs to programs* and *simplification of proofs to evaluation of programs*. We will first consider the connection between the simple typed lambda calculus (with product and sum types) and *intuitionistic* propositional logic. The intuitionistic version of propositional logic uses weaker axioms than classical propositional logic. For example, the law of the excluded middle and double negation elimination do not hold in general. Intuitively speaking, in intuitionistic logic every proof has to be witnessed. When proving a disjunction one has to state which of the alternatives holds i.e. one cannot just proof existence. With this view point in mind it should be clear why intuitionists refute the law of the excluded middle, if $A \vee \neg A$ were true in general one would already have to know which alternative holds.

A proposition corresponds to a type, while the elements of the type correspond to the proofs of the proposition. A proposition is true iff the type representing it is inhabited. Constructing a proof for a proposition means constructing a value of its type. Unit and bottom type correspond to true and false, because one can always deduce true (construct a value of the unit type $\top$) and there exist no proof of falsity i.e. $\bot$ the type representing false is not inhabited. Under the above interpretation the usual logical connectives like $\wedge$, $\vee$ and $\Rightarrow$ correspond product, sum and function types.

For example, by the introduction rule for conjunctions, to construct a proof of $A \wedge B$ one has to construct a proof of A and a proof of B. This corresponds to the construction of an element of the product type, because to construct an element of $A \times B$ one has to provide an element of A and B. The elimination rules for conjunction, i.e. one can deduce a proof of A and a proof of B from the conjunction $A \wedge B$, corresponds to the projection functions for products π_1 and π_2. The argumentation for disjunction and sum/coproduct types is dual.

Functions from A to B map element of A to elements of B i.e. they construct proofs of B from proofs of A. They correspond to implications. Lambda abstracting of a value of type A corresponds to assuming a proof of A i.e. implication introduction, while application function corresponds to implication elimination.

As explained in section 2.1.2, Agda's type system has dependent types. With dependent types the construction of a type (and therefore whether the type is inhabited) can change based on a

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`
$\neg A$	$A \to 0$	`A → ⊥`
$P \wedge Q$	$P \times Q$	`A × B`
$P \vee Q$	$P + Q$	`A ⊎ B`
$P \Rightarrow Q$	$P \to Q$	`A → B`
t	**1**	$\top$
f	**0**	$\bot$

Table 2.1: correspondence between first order logic and types in mathematical and Agda notation

value. Dependent types therefore correspond to predicates. Σ and Π types correspond to existential and universal quantification. To proof an existential one has to provide a value and a proof that the proposition holds for this value. These two correspond to the two elements of a the Σ type. The type of the second element depends on the first one i.e. it's a proof for a predicate on the first value. The prove a universal quantified proposition one has to proof that it holds for every possible value. A Π type is a function whose return type depends on the given value i.e. given any value it returns a proof for the predicate on its argument.

Extending the type system with dependent types corresponds to extending intuitionistic propositional logic to intuitionistic predicate logic. Using the above corresponds Agda can be used to state and proof theorems in predicate logic. An overview over the correspondences between certain types and connectives, as well as the corresponding syntax in Agda can be found in table 2.1.3.

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.

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.

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

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 usually 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.

The above definition of propositional equality defines so called *intentional* propositional equality. Intentional equality respects how objects are defined. Extensional equality just observes how objects behave. This distinction is important when comparing functions, as we will do in nearly all later proofs. Two functions which behave identical but are defined differently are extensional but not intentional equal. In Agda one can use extensional equality be arguing in another setoid (i.e. using a different equivalence relation, which assumes that extensionality holds) or by adding an axiom to modify the underlying theory. For simplicity we will do the later and postulate the axiom of extensionality.

```
postulate
  extensionality : ∀ {A : Set} {B : A → Set}
    {f g : (x : A) → B x} → (∀ (x : A) → f x ≡ g x) → f ≡ g
```

The axiom simply states that the two functions f and g are considered equal if they are equal pointwise. It extends the intentional propositional equality to the extensional one, while preserving Agda's consistency.

2.1.5 Termination Checking

Non-terminating functions correspond to circular arguments. Therefore, allowing the definition of non-terminating functions entails logical inconsistency. When defining functions Agda allows general recursion, but only terminating functions are valid Agda programs. 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. For example, Agda does not inline functions containing pattern matches during termination checking. A common example are recursive calls in lambdas, which are passed to higher order functions like map and $\gg=$.

To still proof termination in such cases we can resort to type-based termination checking as described by Abel [2]. The fundamental idea is to represent the size of a recursive data structure in its type. Termination of recursive functions can be proven by observing a reduction in size on the type level. Abel noted as main advantages robustness with respect to small/local changes in code (which do not impact the type) as well as scaling to higher-order functions and polymorphism [2]. The latter will be essential in future chapters, since representations of computational effects involve complex constructs, which obscure recursion on the value level.

Type-based termination is available in Agda in the form of *sized types*. The special, builtin type Size can be used to annotate data types with a value which represents an upper bound on there size. A set of builtin functions and types can be used to establish relations between different sizes. The elements of Size are well ordered. $\uparrow\_$ is the successor operation on Size i.e. $i < \uparrow i$. $\_\sqcup^s\_$ is the supremum w.r.t. the relation. ∞ is the top element in $Size$. Given an i the type of all smaller sizes can be constructed using Size<_[2]. If the size decreases across all recursive call of a function termination follows from the well-ordering on 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.

[2] We will not use this construction. It's usually used when working with coinductive data types.

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 [1]. 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 $S \rhd P$. 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 $\Pi_{n:\mathbb{N}}\mathrm{Fin}\ n$ where $\mathrm{Fin}\ n$ is the type of natural numbers smaller than n. The extension of a container is a functor $[\![S \rhd P]\!]$, whose lifting of types is given by

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

A lifted type corresponds to the container storing elements of the given type e.g. $[\![\ \mathbb{N} \rhd \mathrm{Fin}\ ]\!]\ A \cong \mathrm{List}\ A$. 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

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

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 [13] is a functional logic programming language. It combines paradigms from functional programming languages like Haskell with those from logical languages like 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.

Like in Haskell, functions are defined using equations, but curry assigns different semantics to overlapping clauses. When calling a function all right-hand sides of matching left-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 programs.

```
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. The later option is called Call-Time choice and 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.

2.3 Algebraic Computational Effects

Giving a concrete definition for computational effects is hard. Examples include I/O, exceptions, nondeterminism, state and delimited continuations [5]. Algebraic effects are computational effects, which can be described using an algebraic theory. They were first presented by Plotkin and Power [17].

Handlers for algebraic effects were first presented by Plotkin and Pretnar as a generalization of exception handlers [18]. 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 [24, 15].

2.3.1 Algebraic Theories

The following definition is similar to the one given by Bauer [5].

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 $[\![\cdot]\!]_{(I,\iota)}$.

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

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

Free Model

For each algebraic theory T we can generate a so called free model $\mathrm{Free}_T(\mathbb{X})$. The free model is the optimal model for the theory i.e. it validates just equations from $\mathcal{E}_T$ and the ones directly following from them and no more.

The set $\mathrm{Tree}_T(\mathbb{X})$ contains term trees, built from the variables in $\mathbb{X}$ and the operations of the signature Σ_T.

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

The free model for T is given by

$$\mathrm{Free}_T(\mathbb{X}) = \mathrm{Tree}_T(\mathbb{X}) \ / \ \sim_T$$

where $\sim_T$ is the least equivalence relation, such that for all equations $x \mid l = r \in \mathcal{E}_T$, $l \sim_T r$. Furthermore $\sim_T$ is required to be a congruence relation for the operations of the signature i.e.

$$\forall i \in \{1, \ldots, n\}.x_i \sim_T y_i \Rightarrow \mathrm{op}(x_1, \ldots, x_i) \sim_T \mathrm{op}(y_1, \ldots, y_i)$$

In section 2.3.3 we will see that the free model forms the basis for implementing algebraic effects.

2.3.2 Effect Handler

2.3.3 Free Monad

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

```
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)
```

The free monad corresponds to the free model for an algebraic theory T without equations i.e. the equivalence relation $\sim_T$ is just the identity relation on $\mathrm{Tree}_T(\mathbb{X})$. As a result some trees which should be equal aren't. When the syntax is interpreted by a correct handler these trees should be equal again and the equations should hold.

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.

```
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.

Containers correspond to operations. The constructors for Shape correspond to the operations op_i and simultaneously store the parameter P_i. A constructor without extra arguments corresponds to the an operation with parameter set $\mathbb{1}$. The type of positions corresponds to the arity set A_i. The Pos type depends on a value of type Shape i.e. it assigns an arity to each shape/operation. In the free model the parameters/child trees are given by a function mapping from the arity to the carrier set. This function corresponds to the position function in the Container extension, which maps from the type of positions to the held type, in this case the free monad i.e. the carrier type.

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. It corresponds to variable substitution.

In there categorical formulation, Plotkin et al. algebraic operations are required to have a certain natural condition. On the level of the free monad algebraic operations are those which commute with $\gg\!=$ i.e. for an n-ary algebraic operations the following holds.

$$\mathbf{op}(x_1, \ldots, x_n) \gg\!= \kappa = \mathbf{op}(x_1 \gg\!= \kappa, \ldots, x_n \gg\!= \kappa)$$

For common scoping operations like `once`, `catch` or `fork` this does not hold. Dealing with this limitation leads to the general notion of scoped effects.

2.3.4 Scoped Effects

Not all operations, know from common monads can be represented as an algebraic effects. The most common example is `catch` as it was noted early by Plotkin and Powers [16].

Chapter 3

First Order

The following sections describe the implementation of algebraic and scoped effects in the first order setting. We focus on implementation details specific to Agda like termination and positivity checks. The scoped effects are implemented using explicit scope delimiters as described by Wu et al. [24].

3.1 Functors à la carte

When modelling effects each functor represents the syntax i.e. the operations of an effect. For containers each shape constructor 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 effect's syntax i.e. it is 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[1].

The approach described by Wu et al. [24] is based on "Data types à la carte" by Swierstra [21]. 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 section `??` we defined the free monad over a strictly positive functor, which was represented by a Container. We could use a similar data type to combine the extensions of two containers, but we would lose the advantages of containers i.e. that container extension is a strictly positive functor. Containers are closed under multiple operations, coproducts being one of them [1]. Therefore, we can combine two container functors by combining the underlying containers. The coproduct of two containers F and G is the container whose shape is the disjoint union of F's and G's shapes and whose position function pos is the coproduct mediator of F's and G's position functions[2].

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

The functor represented by the coproduct of two containers is isomorphic to the functor coproduct of their representations. The container without shapes is the neutral element for the coproduct of containers[3]. This allows us to define n-ary coproducts for containers.

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

To generically work with arbitrary coproducts of functors we 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 the coproduct was constructed using a value of type A. Therefore, we produce a value of type Maybe A and return nothing in case the coproduct was constructed differently.

In the "Data types à la carte" [21] 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

[1] This follows from the facts that `Free` is the left half of a free-forgetful adjunction and that all left-adjoints preserve colimits.

[2] The function $[\_,\_]$ corresponds to the Haskell function `either :: (a -> c) -> (b -> c) -> Either a b -> c`.

[3] It is the initial element in the category of containers over set, because $\mathbb{0}$ is initial in set.

be used to inject or potentially extract values from a coproduct. The two instances for `:<:` mark F as an element of the coproduct if it's on the left-hand side of the coproduct (in the head) or if it's already in the right-hand side (in the tail). Usually one would add a third instances `f :<: f`, relating `f` to itself, but this instance is not needed if the last functor is always `Void`.

```
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 _⊕_ is another container, not just a value of a simple data type, instance resolution using _⊕_ is not as straight forward as in Haskell and in some cases extremely slow[4].

This implementation of the free monad uses an approach similar to the Idris effect library [7]. 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 _∈_.

```
data _∈_ {ℓ : Level} {A : Set ℓ} (x : A) : List A → Set ℓ where
  instance
    here  : ∀ {xs} → x ∈ x :: xs
    there : ∀ {y xs} → ⦃ x ∈ xs ⦄ → x ∈ y :: xs
```

The type $x \in xs$ represents the proposition that x is an element of xs. The two constructors can be read as inference rules. 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 $y :: xs$.

The two instances still overlap resulting in an exponential slowdown in instance resolution[5]. 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 hidden ones with tactic annotations.

Using this proposition we can define functions for injection into and 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 that specific container is an element of the list used to construct the coproduct.

[4] We encountered cases where type checking of overlapping instances involving _⊕_ did not seem to terminate.
[5] https://agda.readthedocs.io/en/v2.6.1.1/language/instance-arguments.html#overlapping-instances

Let us consider inject first. By pattern matching on the evidence we acquire more information about the 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 either make a recursive call or stop the search.

3.1.1 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}
```

The free monad is indexed over an argument of Type Size. We follow the pattern described in section 2.1.5, to annotate the data type. pure values have an arbitrary size. When constructing an impure value the new value is strictly larger than the ones produced by the container's position function. The size annotation therefore corresponds to the height of the tree described by the free monad. Using the annotation it is possible to prove that functions preserve the size of a value or that complex recursive functions terminate.

Next we define utility functions for working with the free monad. inj and prj provide the same functionality as the ones used by Wu et al. [24]. 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 preserves the size of its input, because it just traverses the tree and repackages the contents. prj decreases the input of its argument by one, because it deconstructs the root of the given tree. These size preserving functions are essential when we later define handlers.

```
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)
```

Next we define the basic monadic operations for the above monad. Usually >>= and return would be enough to define all the other functions, but >>= is not size preserving, while some other functions can be defined such that they are. For example, later we will rely on the fact that <$> preserves the size of its argument. Consider the following definition of fmap for the free monad[6].

```
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)
```

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

```
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 witnessed 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 addition for sized types the only correct size estimate for the returned value is unbounded (∞). This means that by using $\gg=$ on a value we lose our size esstimate. This is a problem in recursive functions, because in terms of termination checking it renders the annotation useless. For later programs using effects this is not a problem, because should not rely on the size annotation of the free monad, because they should not recurse on a value of type Free. The return type is not explicitly indexed, because the compiler correctly infers ∞.

```
_>>=_ : ∀ {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)
```

3.1.2 Properties

The definition of the free monad from section 3.1.1 is a functor because it satisfies the two functor laws. The two laws state that the lifting of morphisms via fmap preserves identity and the composition of morphisms.

Both properties are proven by structural induction over the free monad. Let us consider the first proof. It is written using chain reasoning operators to display the general structure of the later proofs In the definition of <$>, in the case for pure the given function is simply applied to the stored value. Therefore, the base case is proven by refl. In the induction step we reach a point were we have to show the equivalence of the two continuations. Using cong we reduce the equality on the whole terms to the equality on the continuations. Next we invoke extensionality to show point-wise equivalence. This equivalence is exactly the induction hypothesis, which we obtain using a recursive call.

Because all steps using ≡⟨⟩ are proven by refl they can be omitted. The other proofs follow the same pattern and can therefore are written more concise.

```
fmap-id : (p : Free ops A) → fmap id p ≡ p
fmap-id (pure x)       = refl
fmap-id (impure (s , pf)) = begin
  fmap id (impure (s , pf))          ≡⟨⟩ -- definition of fmap
  impure (s , fmap id ∘ pf)          ≡⟨⟩ -- definition of ∘
  impure (s , λ p → fmap id (pf p)) ≡⟨ cong (λ t → impure (s , t))
                                        (extensionality λ p → fmap-id (pf p)) ⟩
  impure (s , λ p → pf p)            ≡⟨⟩ -- η-conversion
  impure (s , pf)                    ■
```

(1)

```
fmap-∘ : ∀ (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
fmap-∘ f g (impure (s , pf)) = cong (impure ∘ (s ,_)) (extensionality (fmap-∘ f g ∘ pf))
```

(2)

Later we will need the interchange law for applicative functors. Based on the definition of <*> it allows us to reduce a call to <*> with a pure second argument to one to <$>.

```
interchange : ∀ a (f : Free ops (A → B)) →
  (f <*> pure a) ≡ (pure (_$ a) <*> f)
interchange a (pure f)          = refl
interchange a (impure (s , κ)) = cong (impure ∘ (s ,_)) (extensionality (interchange a ∘ κ))
```
(3)

This definition of the free monad also satisfies the three monad laws. Written using the kleisli composition (>=>) they reduce to the laws for morphism composition in a category i.e. pure is the left and right identity for the associative composition. We are more interessted in this formulation, because it is the one occuring more directly in programs.

```
bind-identˡ : ∀ {ops} (f : A → Free ops B) (x : A) → (pure x >>= f) ≡ f x
bind-identˡ f x = refl
```
(4)

```
bind-identʳ : ∀ {ops} (x : Free ops A) → (x >>= pure) ≡ x
bind-identʳ (pure x)          = refl
bind-identʳ (impure (s , pf)) = cong (impure ∘ (s ,_)) (extensionality (bind-identʳ ∘ pf))
```
(5)

```
bind-assoc : ∀ {ops} (f : A → Free ops B) (g : B → Free ops C) (p : Free ops A) →
  ((p >>= f) >>= g) ≡ (p >>= (λ x → f x >>= g)) -- inner parens can be omitted
bind-assoc f g (pure x)          = refl
bind-assoc f g (impure (s , pf)) = cong (impure ∘ (s ,_)) (extensionality (bind-assoc f g ∘ pf))
```
(6)

3.2 Handler

Usually, an effect handler removes and interprets the syntax of an effect. In the simplest case the handler replaces the effect syntax with a corresponding implementation in the host language. It maps from the free model to a model in the language i.e. it transforms an impure to a pure computation. Therefore, the handler induced semantics for the effects. Some handlers manipulate syntax of other effects or the structure of the program itself. These handlers are usually called *non-orthogonal handlers*.

All handlers in the following sections 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 i.e. its Shape type has no constructors. Pattern matching on the impure constructor yields ().

Therefore, we can always produce a value of type A. The handler for Void is important, because it can be used to escape the Free context after all other effects have been handled.

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

The following two sections describe the implementation of handlers for State and Nondeterminism. By means of examples, they explain the construction of handlers as well as lay ground work for more complex handlers.

3.2.1 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 Nondet^s : Set where ⁇^s fail^s : Nondet^s
```

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

```
Nondet : Container
Nondet = Nondet^s ▷ λ where
  ⁇^s   → Bool
  fail^s → ⊥
```

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 additional parameters and always use pure as continuation. These versions of the operations already process the continuations parameter. The generic ⁇ operation nondeterministicly produces a boolean, but usually we want to use it analogous to Curry's `?` operator, therefore we implement this refined version. op fail^s produces a computation with result type ⊥ i.e. the operation has no continuations. To avoid continuing with λ() after every call of fail (to produce a computation of the correct type) we add the continuation to the smart constructor.

```
_⁇_ : ∀ {ops} → ⦇ Nondet ∈ ops ⦈ → Free ops A → Free ops A → Free ops A
p ⁇ q = op ⁇^s >>= λ b → if b then p else q

fail : ∀ {ops} → ⦇ Nondet ∈ ops ⦈ → Free ops A
fail = op fail^s >>= 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 fail^s , 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.

⦇ and ⦈ are *idiom brackets* and denote applicative functor style function application i.e. pure f <⊛>... <⊛>... <⊛>.... In this case the function f is the mixfix operator _++_ which can still be written in infix notation.

```
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. Other hides the inj₂ constructor, therefore the newly constructed impure value is of type Free *ops* (List *A*).

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

Properties

As explained in section 2.3.3, the free monad represents the free model for the theory without equations i.e. some terms should be equal but are not. The handler for an algebraic effect defines its semantics by removing and interpreting its syntax. The result of a handler should again be a model for the effect i.e. the equations characterizing the effect should hold. We can proof that our handler produces a valid (but not necessarily free) model for our equations by verifying that the laws holds after interpreting the terms. The ability to rigorously proof these laws about the program is one of the main advantages of Agda.

We have four laws governing our Nondet effect. The first law states that const fail should be an absorbing element for the Kleisli composition.

```
bind-fail : {k : A → Free (Nondet :: ops) B} →
  solutions (fail >>= k) ≡ solutions fail
bind-fail = refl
```
(7)

The proof is trivial, because fail has no continuations. Simplifying both sides yields fail.

The other three laws state that ⁇ and fail form a monoid. The proofs for all three are similar. By using functor, applicative and monad laws one can reduce the proof to one for pure lists and simply use the corresponding equality from the standard library.

```
⁇-identˡ : (q : Free (Nondet :: ops) A) → solutions (fail ⁇ q) ≡ solutions q
⁇-identˡ q = begin
  solutions (fail ⁇ q)                    ≡⟨⟩ -- definition of handler for ⁇
  _++_ <$> solutions fail <*> solutions q ≡⟨⟩ -- definition of handler for fail
  _++_ <$> pure []        <*> solutions q ≡⟨⟩ -- definition of <$>
  pure ([] ++_)           <*> solutions q ≡⟨⟩ -- definition of <*> / ap-ident-left
  [] ++_                  <$> solutions q ≡⟨⟩ -- definition of ++ / ++-ident-left
  id                      <$> solutions q ≡⟨ fmap-id (solutions q) ⟩
  solutions q                             ■
```

The proof for the left identity is straight forward. No standard library equality is needed, because ++ is recursive in its first argument. To eliminate <$> a functor law is needed.

```
⁇-identʳ : (p : Free (Nondet :: ops) A) → solutions (p ⁇ fail) ≡ solutions p
⁇-identʳ p = begin
  solutions (p ⁇ fail)                          ≡⟨⟩ -- definition of handler
  _++_ <$> solutions p   <*> pure []            ≡⟨ interchange [] (_++_ <$> solutions p) ⟩
  pure (_$ []) <*> (_++_ <$> solutions p)       ≡⟨⟩ -- definition of <*> / ap-ident-left
  (_$ [])      <$> (_++_ <$> solutions p)       ≡⟨ sym (fmap-∘ (_$ []) _++_ (solutions p)) ⟩
  (_$ [])      ∘   _++_  <$> solutions p        ≡⟨⟩ -- definition of ∘ and $
  (_++ [])           <$> solutions p           ≡⟨ cong (_<$> solutions p)
                                                    (extensionality ++-identityʳ) ⟩
  id                 <$> solutions p           ≡⟨ fmap-id (solutions p) ⟩
  solutions p                                  ■
```

Proving the right identity is more complicated, because fail is on the right-hand side of ⁇. All monad functions from section 3.1.1 are recursive in their left argument, therefore most propositions involving right arguments do not follow trivially from the definition and need own proofs.

This is a common pattern. Because ++ is recursive in its first argument, the left identity is given by definitional equality, while the right one is not, but follows from the definition. Notice that partially applying ++ with [] is only equal to id on the left without invoking extensionality.

Using the proofs from section 3.1.2 we can simplify the equations and reduce them to a problem for normal List. The proof for associativity is longer, but follows the same patterns.

3.2.2 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. the continuation is called with tt to start 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 and define patterns to simplify the handler.

```
data State^s (S : Set) : Set where
  get^s : State^s S
  put^s : S → State^s S

State : Set → Container
State S = State^s S ▷ λ where
  get^s     → S
  (put^s _) → ⊤

pattern Get κ   = impure (inj₁ get^s , κ)
pattern Put s κ = impure (inj₁ (put^s s) , κ)
```

To simplify working with the State effect we add smart constructors. These correspond to the generic operations, which we can implement using op.

```
get : ∀ {ops S} → ⦃ State S ∈ ops ⦄ → Free ops S
get = op get^s

put : ∀ {ops S} → ⦃ State S ∈ ops ⦄ → S → Free ops ⊤
put s = op (put^s s)
```

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
```

3.2.3 Handling Combined Effects

As described in section 3.1, signatures can be combined by combining the underlying functors. Handlers can be used to partially evaluate programs over a combined signature. Consider the following program using nondeterminism and state.

```
chooseTick : {| Nondet ∈ ops |} → {| State ℕ ∈ ops |} → Free ops ℕ
chooseTick = tick ≫ ((tick ≫ get) ⁇ (tick ≫ tick ≫ get))
```

For the interaction of State and Nondet two semantics are conceivable.

Local State Each branch in the nondeterministic calculation has its own state. When ⁇ is used to choose between two reuslts both branches continue with own states, which initialy have the same value.

Global State All branches share a single, global state. Based on the evaluation strategy for the nondeterminism (first result/all results; depth first search/breath first search/fair search) an ordering for all operations in the tree is induced. The state operations are executed in this order an all act on the same state.

The handlers for the two effects define semantics for their operations. Operations of the other siganture are just traversed. The interaction smeantics are induced by the ordering of the handlers. We obtain two combined handlers, each corresponding to one of the two cases above.

```
localState : S → Free (State S :: Nondet :: []) A → List (S × A)
localState s₀ = run ∘ solutions ∘ runState s₀

globalState : S → Free (Nondet :: State S :: []) A → S × List A
globalState s₀ = run ∘ runState s₀ ∘ solutions
```

Let us consider the above example under both combined handlers with initial state 0.

Local State The first tick increases the state to 1. Both branches are executed independently with initial state 1. The first ticks once, therefore the get yields 2. The second ticks twice, therefore the get yields 3.

```
localState 0 chooseTick ≡ (2 , 2) :: (3 , 3) :: []
```

Global State The nondet handler executes the banches from left to right. The final state of one branch is the inital one for the next branch. Therefore, the second branch continues with state 2 increasing it twice.

```
globalState 0 chooseTick ≡ (4 , 2 :: 4 :: [])
```

3.3 Effects with Scopes using Brackets

To correctly handle operations with local scopes Wu et al. introduced scoped effects [24]. 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. Thanks to a closing delimiters the end of a scope can be found. Therefore, calling $\gg=$ on a scoped operation to extend the continuation isn't a problem, because the continuation can be separated in the part inside and outside the scope.

3.3.1 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 calls 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 $\gg=$ 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
```

3.4 Call-Time Choice as Effect

Bunkenburg presented an approach to model call-time choice as a stack of scoped algebraic effects [8]. In this section we will extend the nondeterminism effect from section 3.2.1 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 Bunkenburg's 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.

3.4.1 Deep Effects

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. Such effects are sometimes called *deep effects*.

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 deep effects [3, 10, 9]. The following example of an effectful List demonstrates the the general construction[7].

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

$\text{List}^M\ ops\ A$ 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 (List^M ops A {i}) → Free ops (List^M ops A) → Free ops (List^M 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 [8] 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
```

[7] In the following code effectful data structures and lifted versions of functions are marked with a suffix ^{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 A 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

  List^M-Normalform : ⦇ Normalform ops A B ⦈ →
    Normalform ops (List^M ops A {i}) (List B)
  Normalform.nf List^M-Normalform nil        = pure []
  Normalform.nf List^M-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.

3.4.2 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 Share^s : Set where bshare^s eshare^s : ℕ × ℕ → Share^s
pattern BShare n κ = impure (inj₁ (bshare^s n) , κ)
pattern EShare n κ = impure (inj₁ (eshare^s n) , κ)

Share : Container
Share = Share^s ▷ λ _ → ⊤
```

The handler has the same structure as other handlers for scoped effects like Cut. The bshare handler searches for opening delimiters, which are subsequently handled by eshare. ehsare handles the part of the program in scope and returns the unhandled continuations, which is again handled using bshare. Nested scopes are handled uisng recursive eshare calles. To implement sharing the handler modifies the program in scope. Using prj, NonDet syntax is extracted and reinjected. Choice nodes are modified to now include the unique choice id. The identifier is generated from the scope id of the current shareing scope and a counter, which is managed by the handler. When ever a choice is tagged with an id the counter is increased. The uniquenes of the scope identifiers is not managed by the handler, but the share operator.

Due to the size annotations termination is proven on the type level. To implement the sharing handler it's not necessary to combine the two cases, as done by Bunkeburg in his Coq implementation.

```
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)
```

3.4.3 Share Operator

Next we will define the share operator as described by Bunkenburg. The operator generates new unique scope identifiers using a State effect. Furthermore, it shares all choices in the components of effectful data structures.

To map the operator over arbitrary structures the Shareable type class is introduced. The shareArgs function calls share recursively on the components of the shared data structure. Similar to Normalform, trivial instances for simple data types are introduced.

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

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

Using the instances for data we can implement Bunkenburg's version of the operator. The operator itself relies on a State effect, which is later executed as first effect in the stack, to generate the identifiers. Given a computation producing values of type A the operator returns a computation of computations of type A i.e. the result has two Free ops layers. The outer layer is usually execute (injected at into the larger program at the current point) immediately using $>>=$. The outer computation generates a new identifier. The inner computation captures the identifier, therefore binding the inner computation multiple times yields multiple scopes with the same id. An in depth explanation can be found in "Modeling Call-Time Choice as Effect using Scoped Free Monads" [8].

```
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 op $ bshareˢ (i , j)
                     put (i , 1 + j)
                     x  ← p
                     x′ ← shareArgs x
                     put (1 + i , j)
                     op $ eshareˢ (i , j)
                     pure x′
```

Using share we can implement a shareable instance for effectful lists. In the original Haskell implementation shareArgs is a higher order function, which takes share as an additional argument. For simplicity and to avoid termination problems this is not the case in the Agda implementation.

```
instance
  Listᴹ-shareable : {{ Shareable ops A }} → {{ Share ∈ ops }} → {{ State (ℕ × ℕ) ∈ ops }}
    → Shareable ops (Listᴹ ops A {i})
  Shareable.shareArgs Listᴹ-shareable nil          = []ᴹ
  Shareable.shareArgs Listᴹ-shareable (cons mx mxs) = cons <$> share mx <*> share mxs
```

3.4.4 Examples

Using the handler we can simulate call-time choice semantics using effects in Agda. Call-time choice can be simulated with the following effect stack

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

Before calling any handlers the data is normalized, so that not only surface, but also deep effects are evaluated. State is evaluated first to generate the identifiers for sharing. Sharing is evaluated next to tag all choices with their scopes identifiers. At last nondeterminism is evaluated to collect the results of all possible branches, taking the scope ids into account.

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

Now we can implement the doubleCoin example from section 2.2.

```
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 |)
```

By calling share the choice in coin is shared. By binding the result of share the computation tree generating the ids (i.e. the outer layer) is expanded. The argument of the continuation c is itself a program, it's the original coin function wrapped into a sharing scope, which captures the generated id of the outer scope. By binding c this computation is injected into the whole computation. Binding c multiple times yields the same scope, because the scope identifier is captured. Equal scope identifier yield equal decisions during the evaluation of NonDet.

```
runCTC doubleCoin ≡ 0 :: 2 :: []
```

Thanks to the Shareable type class it's also possible to share choice, which are embedded in data structures. The following simple example demonstrates this possibility.

```
head^M : {| NonDet ∈ ops |} → Free ops (List^M ops A) → Free ops A
head^M mxs = mxs >>= λ where
  nil → fail'
  (cons mx _) → mx

doubleHead : {| Share ∈ ops |} → {| State (ℕ × ℕ) ∈ ops |} → {| NonDet ∈ ops |} →
  Free ops ℕ
doubleHead = share (coin ::^M []^M) >>= λ mxs → (| head^M mxs + head^M mxs |)
```

As expected, observing the value in the head twice yields the same value, because the choice was shared.

```
runCTC doubleHead ≡ 0 :: 2 :: []
```

Using the Normalform type class it's possible to embed effects deep into data structures. This allow to write programs operating on recursive data structures, which closely resemble their Curry version. Consider the example of nondeterministic insertion into a list.

```
insertND : {| NonDet ∈ ops |} →
  Free ops A → Free ops (List^M ops A {i}) → Free ops (List^M ops A {↑ i})
insertND mx mxs = mxs >>= λ where
  nil              → mx ::^M []^M
  (cons my mxs) → (mx ::^M my ::^M mxs) ?' (my ::^M insertND mx mxs)
```

When evaluating the result of insertND the List^M is normalized using the type class. As expected, insert 1 deterministically in the list with 2 and 3 yields three possible results.

```
runCTC (insertND (pure 1) (pure 2 ::^M pure 3 ::^M []^M)) ≡
(1 :: 2 :: 3 :: []) :: (2 :: 1 :: 3 :: []) :: (2 :: 3 :: 1 :: []) :: []
```

$$\text{pure} x \gg\!= k = k\ x \qquad \text{(ident-left)}$$
$$mx \gg\!= \text{pure} = mx \qquad \text{(ident-right)}$$
$$mx \gg\!= f \gg\!= g = mx \gg\!= \lambda x \to fx \gg\!= g \qquad \text{(assoc)}$$

$$\text{fail} \gg\!= k = \text{fail} \qquad \text{(fail-absorb)}$$
$$(p \mathbin{??} q) \gg\!= k = (p \gg\!= k) \mathbin{??} (q \gg\!= k) \qquad (??\text{-algebraic})$$

$$\text{share fail} = \text{pure fail} \qquad \text{(share-fail)}$$
$$\text{share} \perp = \text{pure} \perp \qquad (\text{share-}\perp)$$
$$\text{share}\ (p \mathbin{??} q) = \text{share}\ p \mathbin{??} \text{share}\ q \qquad (\text{share-}??)$$
$$\text{share}\ (\text{c}\ x_1 \dots x_n) = \text{share}\ x_1 \gg\!= \lambda y_1 \dots \text{share}\ x_n \gg\!= \lambda y_n \to \text{pure}\ (\text{pure}\ (c\ y_1 \dots y_n)) \qquad \text{(HNF)}$$

Figure 3.1: laws for a monad with sharing by Fischer et al.

3.4.5 Laws of Sharing

Fischer et al. introduced a set of equations characterizing call-time choice semantics [11]. The equations can be found in figure 3.1. The first three laws are just the monad laws, which were proven in section 3.1.2. Section 3.2.1 contains a proof for the fourth law. The fifth law is the algebraicity of ⁇, which follows for any operation directly from the definition of ≫=.

```
⁇-algebraic : {{ _ : Normalform CTC B C }} → {{ _ : Shareable CTC B }} →
  (p q : Free CTC A) (k : A → Free CTC B) →
  runCTC ((p ⁇′ q) >>= k) ≡ runCTC ((p >>= k) ⁇′ (q >>= k))
⁇-algebraic p q k = refl
```

In addition to the monad laws and equations for ⁇ they introduced equations involving share.

```
share-fail : {{ _ : Normalform CTC A B }} → {{ _ : Shareable CTC A }} →
  runCTC {A} {B} (share fail′ >>= id) ≡ runCTC {A} {B} (pure fail′ >>= id)
share-fail = refl
```

The second law involving share does not apply to our effect stack, because we have no notion of `undefined`. Partiality can be implemented identical to fail with carrier Maybe. The effect would be placed last in the stack to allow canceling all running calculations. The law can be proven analogous to the one for fail.

The third law involving share states that share distributes over ⁇. This law doesn't hold for this implementation, because there is no guarantee that get or put aren't called outside the sharing operator. Developing another implementation satisfying the laws in all cases is outside the scope of this thesis.

The last law states that share distributes over the components of effectful data structures. Since our representation for effectful data isn't uniform this law has to be proven on a case by case basis. One could try to automate this proves using tactics, find an equivalent formulation which doesn't refer to constructors directly or use a different representation for effectful data (e.g. fixed points of polynomial functors, since they can be made effectful generically).

Chapter 4

Higher Order Syntax

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 abstract syntax [24]. Bunkenburg already tried to implement some scoped effects using this approach, but failed due to limitations of Coq [8].

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. Section 4.5 presents a limited implementation in Agda to discusses some implementation details and highlight the occurring problems.

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 described by `Free F`, which produce values of type `A`. In the higher order approach these functors are generalized to higher order functors, that is in Haskell data types 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 alternate result 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` as well as a common signature, defined by `m`.

Calling $\gg=$ on a higher order term just extends the continuation i.e. just substitutes the variables in the trees of type `m a`. See figure 4.1 for an example. Therefore these operations aren't algebraic but also capture the notation of scopes.

Figure 4.1: calling $\gg=$ on operations with and without scopes (TODO)

4.2 Representing Strictly Positive Higher Order Functors

In Haskell the generalization to higher order syntax is straight forward. In Agda we again have to find an appropriate representation for functor to guaranty strict positivity, which leads to the idea of an indexed container as described by Altenkirch et al. [4] [1]. Indexed containers from the paper as well in the Agda standard library are defined as follows[2].

```
record IndexedContainer (I O : Set₁) : Set₁ where
  field
    Shape : O → Set
    Pos : ∀ {o} → Shape o → Set
    Ctx : ∀ {o} → (s : Shape o) → Pos s → I

  ⟦_⟧ : (I → Set) → (O → Set)
  ⟦_⟧ F A = Σ[ s ∈ Shape A ] ((p : Pos s) → F (Ctx s p))
```

Allowing indicies of arbitrary levels and instantiating I and O with Set gives rise to a container extension of the correct shape.

Bunkenburg also tried using indexed container. The above definition is similar to his first definition of a container representation for a higher order functor. His definition is given below.

```
record HContainer : Set₂ where
  field
    Shape : Set₁
    Pos : Shape → Set
    Ctx : (s : Shape) → Pos s → Set → Set

  ⟦_⟧ : (Set → Set₁) → (Set → Set₁)
  ⟦_⟧ F A = Σ[ s ∈ Shape ] ((p : Pos s) → F (Ctx s p A))
```

In the second definition the Shape and Pos types cannot depend on the given index i.e. the argument for the returned functor. Otherwise, both definitions are quite similar. All stored elements are lifted by the given functor F. F's argument is generated using Ctx. Ctx has access to the chosen shape, accessed position and the argument type A. Conceptually, Ctx is used to choose between the stored X and the argument type A i.e. programs inside or outside the operations scope.

Bunkenburg noted that it is not possible to define $\gg=$ bind generically for the definition above [8]. Similar to the Haskell implementation by Wu et al. [24] we could define a type class to implement a emap for all valid higher-order functors, which would allows the definition $\gg=$. Both representations share another problem, the produced type constructor is not necessarily strictly positive. This is a necessary requirement to define recursive, effectful data structures.

There are multiple ways to fix this problem. Following the pattern from chapter 3 we could replace the returned functor with normal Container. This option is the most expressive, but also quite complex. Furthermore, for most effects this level of control over syntax is not needed. In most effects the argument A is only used directly i.e. the Haskell functor contains `a` only as `m a`.

Working under this assumption, it is possible to replace the returned functor with a value of type Maybe Set and replace nothing with A in the container extension. This also guaranties strict positivity for A. Furthermore, it allows a generic definition of $\gg=$, because it is possible to pattern match on the result of Ctx and therefore differentiate between container extension which do or do not contain A[3].

Bunkenburg fixed this problem by switching to version of a Bi-Container. A Bi-Container has two position types for each shape. Its extension is a Bifunctor, which is strictly positive in both arguments. The definition of a Bi-Container and its extension is given below.

```
record Effect : Set₂ where
  constructor _◁_+_/_
```

[1] Note that this is not the first publication related to indexed containers. Papers of the same name, by the same author(s) exist, as well as data types, known as "Interaction Structures" with essentially the same shape.

[2] The names of the fields are changed and level polymorphism is omitted to highlight differences between the different definitions.

[3] An example implementation can be found in the repository.

```
    field
      Shape : Set₁
      Pos : Shape → Set
      PosX : Shape → Set
      Ctx : ∀ s → PosX s → Set
  open Effect

  ⟦_⟧ : Effect → (Set → Set₁) → Set → Set₁
  ⟦ S ◁ P + PX / C ⟧ F A = Σ[ s ∈ S ] (((p : P s) → F A) × ((p : PX s) → F (C s p)))
```

A Bi-Container has two sets of positions for each shape. Its extension consists of a shape and two position functions, one for each set of positions. In the above definition, the second set of positions is augmented with a context function, identical to the second indexed container definition. In Agda this definition is recognized as strictly positive in A. The analogous definition in Bunkeburg's Coq implementation is not [8].

Ultimately the choice of representation for the functor doesn't matter, because all approaches based on storing a type directly encounter the problem described in section 4.3. Going forward we will use Bunkeburg's indexed Bi-Container for examples, because it allows us to continue where he left of and it is the easiest to work with.

Using our choice of functor representation we can define the Prog monad for higher order syntax as described by Wu et al. [24]. It is similar to the free monad in the first order case, except that the functor has access to the type parameter an the type constructor, not just the combined type.

```
  ⟦_⟧′ : Effect → (Set₁ → Set₁) → Set₁ → Set₁
  ⟦ S ◁ P + PX / C ⟧′ F A = Σ[ s ∈ S ] (((p : P s) → F A) × ((p : PX s) → F (Lift _ $ C s p)))

  data Prog (C : Effect) (A : Set₁) : Set₁ where
    pure   : A → Prog C A
    impure : ⟦ C ⟧′ (Prog C) A → Prog C A
```

Notice that due to the potentially captured type in the Shape is an element of Set_1. Therefore, Prog is also an element of Set_1. Because Prog is an element of Set_1 we can also increase the universe level for A to 1, which is needed to define effectful data structures. Unfortunately, because the universe level of the captured type is smaller than Prog 's it is not possible to work with effectful data as in Haskell. Section 4.3 contains a detailed explanation of the problem.

4.3 Effectful Data and Existential Types

A central problem with this approach is that it cannot model deep effects without involving variable levels. Consider the the following definition of an effectful list. Although it is valid, in contrast to the Coq version, it cannot be used as expected with Prog

```
  data ListM (E : Effect) (A : Set₁) : Set₁ where
    nil  : ListM E A
    cons : Prog E A → Prog E (ListM E A) → ListM E A
```

$\mathsf{ListM}\ C\ A$ is an element of Set_1, because it stores elements of type $\mathsf{Prog}\ C\ X$, which is an element of Set_1. $\mathsf{Prog}\ C\ X$ is an element of Set_1, because it holds values of the container extension i.e. elements of the Shape type, which is an element Set_1, because it has to store the result type of the subcomputation X. Therefore, $\mathsf{ListM}\ C\ A$ is to large to be stored as result type of a subcomputation i.e. it is not possible to call a scoped operation (such as share) on an effectful data structure.

More generally speaking, let us fix the universe level of the result type X of the subcomputation $\ell_X = l$. The type is stored in the Shape, therefore the universe level of the Shape type is given by $\ell_S = l + 1$. Because the Prog stores an element of the Shape its level is given by $\ell_P = \ell_S$. Effectful data structures live in the same universe as Prog, but are simultaneously required to be stored in the subcomputation. We obtain the contradictory requirements $\ell_E = \ell_P = \ell_S = l + 1$ and $\ell_E \leqslant l$.

Holding on to the idea of storing types in the Shape, the only way to fix this problem is to let all levels vary as needed. This leads to different levels across the program, which are difficult to unify. Furthermore, when mapping a scoped operation over a recursive data structure (e.g. share) the increase in the universe level depends on the depth of the structure. Both of these phenomena are hard to deal with and in a complex program potentially unsolvable.

4.4 The Problem with Indexing

One could try solving the problem of existential types by indexing the program over a typing context. The level of the index is independent from the level of the structure, therefore the chain of reasoning from section 4.3 does not apply. The typing context needs to have the same structure as the program. This is possible by reusing the arbitrary branching mechanism, known from the free monad.

A central problem with this approach is that every call of $\gg\!=$ changes the structure of the program and therefore the context. These changes now have to be reflected on the type level, because the typing context has to reflect the programs shape. If $\gg\!=$ substitutes leaves differently, the changes to the context aren't simple substitutions, but depend on the stored value. As with increasing universe levels, this complications are increasingly hard to deal with and are hard to hide from the user of the library.

4.5 Limited Implementation

Although, this implementation makes working with deep effects difficult, it can still implement scoped effects in a setting without them. For completeness the following section will show key points of a simplified implementation using this approach. To write concise code we will work with Bunkenburg's indexed Bi-Containers, because they model effects with a simple continuation accurately. In a more complex implementation, where more control over the type of the continuation is needed, a container with a Ctx for A might be needed[4].

First we port over the infrastructure for modular effects. The coporduct carries over seamlessly. We combine the position functions pairwise and handle the Ctx functions using the coproduct mediator.

```
_⊕_ : Effect → Effect → Effect
(S₁ ◁ P₁ + PX₁ / C₁) ⊕ (S₂ ◁ P₂ + PX₂ / C₂) =
  (S₁ ⊎ S₂) ◁ [ P₁ , P₂ ] + [ PX₁ , PX₂ ] / [ C₁ , C₂ ]
```

To easily work in the higher order setting we adapt the Prog data type similarly to Free monad from chapter 3 i.e. we parameterize over a list of effects and add a Size parameter. To avoid excessive use of universe levels and avoid the `--cumulativity` flag we work with a version of Prog which takes a type parameter from Set[5].

```
data Prog (effs : List Effect) (A : Set) : {Size} → Set₁ where
  pure   : A → Prog effs A {i}
  impure : ⟦ sum effs ⟧ (λ X → Prog effs X {i}) A → Prog effs A {↑ i}
```

To implement $\gg\!=$ we define the emap function. The function has the same signature as the one by Wu et al. [24]. In contrast to the implementation by Wu et al. [24] the function can be defined generically for all effects, because the Bi-Container representation is more restrictive than the their representation in Haskell. Switching to a Bi-Container with a Ctx for the first position function, which returns a Container, preserves this general definition.

```
emap : (F A → F B) → ⟦ E ⟧ F A → ⟦ E ⟧ F B
emap f p = π₁ p , f ∘ π₁ (π₂ p) , π₂ (π₂ p)
{-# INLINE emap #-}

_>>=_ : Prog effs A → (A → Prog effs B) → Prog effs B
pure x  >>= k = k x
impure x >>= k = impure (emap {F = λ X → Prog _ X} (_>>= k) x)
```

The definition for $\gg\!=$ is identical to the one in Haskell. Using the INLINE pragma, as explained in section 2.1.6, we can define emap separately without obscuring the termination.

[4] To guarantied strict positivity one of the solutions mentioned in section 4.2 is needed.

[5] To be monad this parameter has to be an element of Set_1. Such a structure is sometimes known as a *relative monad*, because we can still define a $\gg\!=$ for which the usual laws hold. Adapting this structure to a normal monad is possible, but involves more level annotations and it is easier in a universe polymorphic setting.

The infrastructure from chapter carries over with minimal changes. We reuse the _∈_ proposition from chapter 3. The inj and op functions have to be modified slightly for the new program type.

```
inj : E ∈ effs → ⟦ E ⟧ F A → ⟦ sum effs ⟧ F A
inj here (s , κ) = inj₁ s , κ
inj {F = F} (there ⦃ p ⦄) prog with inj {F = F} p prog
... | s , κ = inj₂ s , κ

op : ⦃ E ∈ effs ⦄ → ⟦ E ⟧ (λ X → Prog effs X) A → Prog effs A
op ⦃ p ⦄ = impure ∘ inj {F = λ X → Prog _ X} p
```

Usually, when an effect is handled the value type is lifted in a context, represented by a functor (e.g. List for Nondet).

Following the approach by Wu et al. we will define a type class Syntax, which, given a handler, allows evaluating a computation in a context to a new computation producing values in the context. To define such a type class we first have to define a context. The normal definition of a functor does not suffice, because it lifts all type to the same Set_n. This is problematic, because scoped operations can only capture elements of Set, while computations are elements of Set_1. Therefore values and computations in context have different universe levels, but during the evaluating we have to convert between the two. Even with `--cumulativity` this behavior is not handled automatically by Agda[6].

To simulate the above behavior we define the Context type class. A context is essentially a functor which respects the universes structure. Given a type from Set_n it lifts it to another type in the same Set_n. Furthermore, given a function between two data types in different universes it lifts the function to a function between the lifted types i.e. between universes. An instances should satisfy the functors laws, even though they are not enforced here. Notice that Context works with arbitrary Levels ℓ and therefore is an element of $\mathsf{Set}\omega$.

```
record Context (F : ∀ {ℓ} → Set ℓ → Set ℓ) : Setω where
  field _<$>_ : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → F A → F B
  _<$_ : ∀ {a b} {A : Set a} {B : Set b} → A → F B → F A
  a <$ mb = const a <$> mb
open Context ⦃...⦄
```

With the context in place we define the Syntax type class by Wu et al. By generalizing to higher order abstract syntax it is possible to represent computations in scope and continuations explicitly. The order in which they are evaluated and therefore how an evaluation context should be moved through the computation is a semantic question. The type class explains how an arbitrary handler can be thread through an effects syntax. The first argument is the initial context. Using <$ it is possible to move an arbitrary value or computation into the context. The second argument is a handler as defined by Wu et al. The handler takes a computation in a context and evaluates it. The result computation has a different effect stack, usually it is the tail of the given stack. By partially evaluating the computation the context was moved through the computation. As a result the values are now lifted into the resulting context. Using the initial context and the handler, the type class explains how an arbitrary operation from its signature is evaluated.

Notice that the given programs have an arbitrary size. – TODO: explain here or later?

```
record Syntax (E : Effect) : Setω where
  field handle : ⦃ Context C ⦄ → C ⊤ →
                 (∀ {X} → C (Prog effs′ X {i}) → Prog effs″ (C X)) →
                 ⟦ E ⟧ (λ X → Prog effs′ X {i}) A → ⟦ E ⟧ (λ X → Prog effs″ X) (C A)
open Syntax ⦃...⦄


instance
  Void-Syntax : Syntax Void
  Syntax.handle Void-Syntax _ _ ()
```

[6] https://agda.readthedocs.io/en/v2.6.1.1/language/cumulativity.html#limitations

4.5.1 Exceptions

An an example for an effect in the higher order setting we define exceptions. Exceptions support two operations, the usual algebraic operation

Syntax The two operations are the equivalent of the Haskell definition from section 4.1. throw as well as the scoped operation catch. throw takes the thrown exception as an additional parameter an has no continuations, because it terminates the computation. catch's Shape stores the result type of the computations in scope.

```
data Exc^s (E : Set) : Set₁ where
  throw^s : (e : E) → Exc^s E
  catch^s : (X : Set) → Exc^s E
```

We use Bunkenburg's indexed Bi-Container. The extension of the container is equipped with two positions functions, one for computations resulting in Prog *effs* A and one for computations in scope i.e. such resulting in computations of type Prog *effs* Y where Y is defined by the context. Let us first consider the computations in scope.

```
data ExcCatch^P (E X : Set) : Set where
  main^P   : ExcCatch^P E X
  handle^P : (e : E) → ExcCatch^P E X

Exc^px : ∀ E → Exc^s E → Set
Exc^px E (throw^s e) = ⊥
Exc^px E (catch^s X) = ExcCatch^P E X

Exc^c : ∀ E s → Exc^px E s → Set
Exc^c E (catch^s X) main^P      = X
Exc^c E (catch^s X) (handle^P e) = X
```

throw has no computations in scope and therefore no positions. catch's positions are given by ExcCatch^P. The operation has two kinds of continuations in scope. The first single continuation is given by main^P, the program in scope whose exceptions should be caught. handle^P provides a continuation for each error i.e. corresponds to the exception handler. Both computations result in a value of type X, which is expressed using Exc^c

Next we define the positions for the effects continuation i.e the computation producing values of type A.

```
Exc^P : ∀ E → Exc^s E → Set
Exc^P E (throw^s e) = ⊥
Exc^P E (catch^s X) = X
```

Again, throw has no continuations, because it produces no results. catch provides a continuation which processes the result of the computations in scope i.e. it provides a continuation for each value of X.

Using the above definitions we define the Exc effect. Furthermore, we introduce pattern synonyms to easily pattern match on values of the container extension in the handler.

```
Exc : Set → Effect
Exc E = Exc^s E ◁ Exc^P E + Exc^px E / Exc^c E

pattern Throw e     = impure (inj₁ (throw^s e) , _ , _)
pattern Catch κ σ   = impure (inj₁ (catch^s _) , κ , σ)
pattern Other s κ σ = impure (inj₂ s           , κ , σ)
```

Lastly we define smart constructors for the operations. throw has no continuations. catch captures the program as well as the exception handler in scope, therefore they are returned by σ. The effects continuation κ is initially given by pure. It is later extended using $\gg\!=$ this is analogous to the generic operations in the first order approach.

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

_catch_ : ⦃ Exc B ∈ effs ⦄ → Prog effs A → (B → Prog effs A) → Prog effs A
p catch h = op $ catchˢ _ , pure , λ{ mainᵖ → p ; (handleᵖ e) → h e }
```

Handler Before we implement the handler we have to provide a Context instance for $E \uplus \_$ to use handle to traverse Other operations. The instance is nearly identical to the functor instance, except that the newly constructed values maybe are element of another Set.

```
instance
  _⊎-Context : ∀ {E : Set} → Context (E ⊎_)
  Context._<$>_ _⊎-Context f (inj₁ e) = inj₁ e
  Context._<$>_ _⊎-Context f (inj₂ x) = inj₂ (f x)
```

As explained above, the Syntax instance is needed to thread the handler through the syntax of other effects i.e. the instance for Exc is not needed for the Exc handler. We still define a Syntax instance for Exc now to explain the usual structure.

```
Exc-Syntax : ∀ {E} → Syntax (Exc E)
Syntax.handle Exc-Syntax _ _ (throwˢ e , _ , _ ) = throwˢ e , (λ()) , λ()
Syntax.handle Exc-Syntax {C = C} ctx hdl (catchˢ X , κ , σ ) = catchˢ (C X) ,
  (λ x → hdl (κ <$> x )) , λ where
  mainᵖ → hdl (σ mainᵖ <$ ctx )
  (handleᵖ e) → hdl (σ (handleᵖ e) <$ ctx )
```

The case for throw^s is trivial, because the operation has no continuations, one which to evaluate the handler. The case for catch^s is more complex.

The idea behind the Syntax type class is to thread the handler through the computation i.e. evaluate it on every subtree. The given handler hdl is slightly different from the final handler in that it only accepts computations inside its context. This is needed to thread the handlers current state through the computation. The continuation for the effects final continuation (the one producing a value of the actual result type A) takes an argument of type $C\ X$. By mapping κ over the given x the continuation is injected in the context of the argument x. This make sense, because the given value is either the result of the sucessfull captured computation or the handler. The rest of the computation should continue where these compuations left of. For example, in case of Nondet, κ would continue on each result of the program in scope, because $<\$>$ applies the given function simply to each element of the list.

Next we will look at the computations in scope i.e. the computations stored by σ. As in chapter 3 and explained in 2.3.4, the handler should be evaluated on the computation in scope in isolation i.e. not reuse the previous state. In the first order case this was accomplished by calling the handler again when finding an opening scope delimiter. To call the handler the computations, given by the functions κ and σ, have to be inject into a context. In case of κ we were given a context, now we have to use the given context. The given value cxt holds a value of type $\top$ i.e. no information just handler state. cxt should be the initial handler context or state for the handler. Using $<\$$ we can inject any computation into it. Injecting a computation into ctx and calling hdl should be the same as calling the actual handler for the effect on the computation.

The cases for the three computations are identical to the ones presented by Wu et al. After evaluating the handler on the sub-trees, values are lifted into the context, therefore the captured intermediate result type changes. In Agda we have to manage the type explicitly. In contrast to the Haskell implementation by Wu et al. we have to explicitly change the captured type. Using the curly brackets we access the hidden type constructor C for the context and simply evaluate it on the given type.

Lastly we define the handler for Exc. Notice that the handler takes a computation of an arbitrary size i. This is needed to prove termination, because the handler itself is threaded through the Other operations using handle. To use handle we require a Syntax instance for the rest of the effect stack. The cases for pure and Throw are trivial.

```
runExc : ⦃ Syntax (sum effs) ⦄ → Prog (Exc B :: effs) A {i} → Prog effs (B ⊎ A)
runExc (pure x)    = pure (inj₂ x)
runExc (Throw e) = pure (inj₁ e)
```

To handle Catch we first evaluate the operation in scope by calling runExc on the computation in the main^P position. Using $\gg\!=$ we can inspect the result and either call the rest of the computation κ with the result or the exception handler using handle^P. In the latter case we again inspect the result and in case of a value also continue with κ.

```
runExc (Catch κ σ) = runExc (σ main^P) >>= λ where
  (inj₁ e) → runExc (σ (handle^P e)) >>= λ where
    (inj₁ e) → pure (inj₁ e)
    (inj₂ x) → runExc (κ x)
  (inj₂ x) → runExc (κ x)
```

Lastly we handle Other operations. As in chapter 3 we reconstruct the operation using impure. Moving the handler along now requires us to use handle. We call handle on the given value of the container extension, constructed using s, κ and σ. Notice that Other hides the inj$_2$, therefore the newly constructed container extension has a different type. It is an element of the signature without the exception syntax. handle requires two additional arguments, the initial context as well has the handler. The initial context is given by inj$_2$ tt i.e. a successful result of type $\top$. The handler has to evaluate Exc syntax for a computation in the context. We handle the two cases using [_,_]. If the argument is already an error we simply produce a pure calculation which produces the error. Otherwise it is a computation, which we can evaluate using runExc. Notice that the initial context always leads to the second case.

```
runExc (Other s κ σ) = impure (handle (inj₂ tt) [ (λ x → pure (inj₁ x)) , runExc ] (s , κ , σ))
```

Even tough we pass runExc to handle, Agda accepts this definition as terminating. Agda can prove termination, because the argument is of an arbitrary size i, therefore the values produced by κ and σ have a smaller size. Note that in the definition of handle the size of the handler argument is the same as the size of the last argument i.e. the operation, which is handled. Therefore, the argument passed to the coproduct mediator, which calls runExc, is smaller than the currently handled value.

Example Using the Exc effect we can define simple programs, which throw and catch exceptions. The first function simply raises an exception.

```
testExc : {! Exc String ∈ effs !} → Prog effs ℕ
testExc = pure 1 >> throw "Foo"
```

As expected, evaluating it yields the error message.

```
run (runExc testExc) ≡ inj₁ "Foo"
```

The second function calls the first one, but catches the exception, returning an alternative result.

```
catchTestExc : {! Exc String ∈ effs !} → Prog effs ℕ
catchTestExc = testExc catch λ _ → pure 42
```

Evaluating the second function yields the value returned by the exception handler.

```
run (runExc catchTestExc) ≡ inj₂ 42
```

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. [15] 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 [12], 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 [15]. 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.

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

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 [15]. 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 [6].

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 [12]. 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 $\_\hat{}\_$ 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 (suc n) a v o s (op  x) = o (map (
fold P (suc n) a v o s (scp x) = s (map (
                                 fold P n                a v o s  x)
                                 fold P (suc n)          a v o s) x)
                                 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" [15] 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 [12] 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 $k\ a$.

```
bind-identˡ : ∀ {A B : Set} {a} {k : A → Prog effs B} →
  (return a >>= k) ≡ k a
bind-identˡ = refl
```

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-identˡ.

```
bind-identʳ : {A : Set} (ma : Prog effs A) → (ma >>= return) ≡ 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.

```
bind-identʳ {effs} {A} = ind (λ{ 0 _ → ⊤ ; (suc n) p → bind n p var ≡ 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.

```
(λ{ {0} (tt) → refl ; {suc n} IH → cong var 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.

```
(λ s IH → cong (op  ∘ (s ,_)) (extensionality IH))
(λ s IH → cong (scp ∘ (s ,_)) (extensionality IH))
```

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.

```
bind-assoc : ∀ {A B C}
  (f : A → Prog effs B) (g : B → Prog effs C) (ma : Prog effs A) →
  (ma >>= f >>= g) ≡ (ma >>= λ a → f a >>= g)
bind-assoc f g = ind
  (λ where
```

```
  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 [14].

```
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.

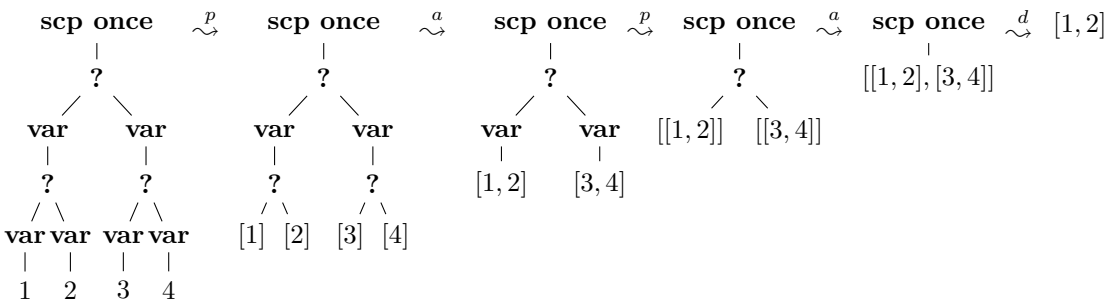

Figure 5.1: Interpretation of **once** (**var** 1 ? **var** 3) $\gg\!=$ $\lambda x.$**var** x ? **var** $(x + 1)$

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
```

Nondetᵖ corresponds to the data type from the example by Piróg et al. [15, 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. foldᵖ 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 $\gg=$ 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 [20]. 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 function hdl over the result of the continuation. 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 second 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 mainᵖ 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 handleᵖ 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) = State^s S ▷ λ{ (put^s s) → ⊤ ; get^s → S }
Scps (State S) = Void

pattern Get κ     = (inj₁ get^s , κ)
pattern Put s₁ κ = (inj₁ (put^s 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 (get^s , pure)

put : {{ State S ∈ effs }} → S → Prog effs ⊤
put s = Op (put^s 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 Share^s : Set where share^s : SID → Share^s

Share : Effect
Ops  Share = Void
Scps Share = Share^s ⇝ ⊤

pattern ShareScp sid κ = (inj₁ (share^s 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 $\gg=$ 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 [20].

```
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

6.2 Related Work

Bibliography

[1] 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.

[2] Andreas Abel. "Semi-Continuous Sized Types and Termination". In: *Log. Methods Comput. Sci.* 4.2 (2008). DOI: 10.2168/LMCS-4(2:3)2008. URL: https://doi.org/10.2168/LMCS-4(2:3)2008.

[3] 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.

[4] Thorsten Altenkirch et al. "Indexed containers". In: *J. Funct. Program.* 25 (2015). DOI: 10.1017/S095679681500009X. URL: https://doi.org/10.1017/S095679681500009X.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. "Purely functional lazy non-deterministic programming". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 2009, pp. 11–22. DOI: 10.1145/1596550.1596556. URL: https://doi.org/10.1145/1596550.1596556.

[12] 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.

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

[14] 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.

[15] 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.

[16] Gordon D. Plotkin and John Power. "Algebraic Operations and Generic Effects". In: *Appl. Categorical Struct.* 11.1 (2003), pp. 69–94. DOI: 10.1023/A:1023064908962. URL: https://doi.org/10.1023/A:1023064908962.

[17] Gordon D. Plotkin and John Power. "Notions of Computation Determine Monads". In: *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Mogens Nielsen and Uffe Engberg. Vol. 2303. Lecture Notes in Computer Science. Springer, 2002, pp. 342–356. DOI: 10.1007/3-540-45931-6_24. URL: https://doi.org/10.1007/3-540-45931-6%5C_24.

[18] Gordon D. Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 80–94. DOI: 10.1007/978-3-642-00590-9_7. URL: https://doi.org/10.1007/978-3-642-00590-9%5C_7.

[19] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: *Log. Methods Comput. Sci.* 9.4 (2013). DOI: 10.2168/LMCS-9(4:23)2013. URL: https://doi.org/10.2168/LMCS-9(4:23)2013.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.