HTWK Hochschule für Technik, Wirtschaft und Kultur Leipzig

HTWK Leipzig
Fakultät Informatik und Medien

Bachelor-Thesis

Implementing a Library for Scoped Algebraic Effects in Agda

Autor: Jonas Höfer

Betreuer : Prof. Dr. Johannes Waldmann HTWK Leipzig
M. Sc. Niels Bunkenburg Christian-Albrechts-Universität zu Kiel

Leipzig, 18. November 2020

Abstract

Computational effects are a central concept in functional programming and formal verification. Different representations for computational effects exist, notably monads and algebraic effects. The composability of the representation is important to allow the development of modular programs and reasoning about them. Algebraic effects and handlers are generally a more modular approach than traditional monad transformers. Furthermore, algebraic effects are convenient representation of computational effects in total, dependently typed languages. For example, Dylus et al. (2019) and Christiansen et al. (2019) used them to explicitly represent some ambient effects, such as Haskell's partiality, in the proof assistant Coq.

In the traditional approach, operations with scopes can only be implemented as an effect handler. Wu et al. (2014) noted a lack of modularity, compared to monad transformers, when multiple effect with scoping operations are used. This is an important limitation because most commonly used effects have associated operations with scopes. To fix this limitation Wu et al. introduced scoped effects. Regarding verification, scoped effects are needed to model more complex ambient effects, such as sharing.

In this thesis we explore how well three different Haskell implementations of scoped effects by Wu et al. (2014) and Piróg et al. (2018) translate in the dependently typed, functional programming language Agda. We implement a library for scoped effects that allows the implementation of effectful programs as well as formal reasoning using Agda's theorem proving capabilities. To test the applicability of our implementations, we implement effects related to modelling Curry's (Hanus et al., 1995) ambient non-strict non-determinism. Our main example for a scoped effect is sharing of nondeterminism, as modelled by Bunkenburg (2019), to simulating Curry's call-time choice semantics. To model ambient effects in data structures, the implementation also supports deep effects.

Zusammenfassung

Nebenwirkungen von Programmen sind ein zentrales Konzept in der funktionalen Programmierung und in der formalen Verifikation. Es existieren verschiedene Repräsentationen von Nebenwirkungen, insbesondere Monaden und algebraische Wirkungen. Um modulare Entwicklung von Programmen und Beweise über diese zu erlauben, ist die Modularität der Repräsentation entscheidend. Algebraische Wirkungen und Wirkungs-Behandler sind generell ein modularerer Ansatz als traditionelle Monaden-Transformer. Des Weiteren sind algebraische Wirkungen eine bequeme Repräsentation für Nebenwirkungen in totalen Sprachen mit Dependent-Types. Beispielsweise haben Dylus u. a. (2019) und Christiansen u. a. (2019) diese verwendet, um ambiente Wirkungen, wie Haskells Parzialität, im Beweisassistenten Coq zu modellieren.

Im traditionellen Ansatz können Operationen mit Sichtbarkeitsbereichen nur als Wirkungs-Behandler implementiert werden. Wu u. a. (2014) bemerkten fehlende Modularität im Vergleich zu Monaden-Transformern, wenn mehrere Wirkungen mit Sichtbarkeitsbereichen verwendet werden. Dies ist eine wichtige Einschränkung, da die meisten Wirkungen zugehörige Operationen mit Sichtbarkeitsbereichen haben. Um diese Limitation zu beheben, haben Wu u. a. (2014) Wirkungen mit Sichtbarkeitsbereichen vorgestellt. In Bezug auf Verifikation sind Wirkungen mit Sichtbarkeitsbereichen nötig, um komplexere ambiente Nebenwirkungen wie Sharing zu modellieren.

In dieser Arbeit erkunden wir, wie gut sich drei Haskell-Implementierungen von Wirkungen mit Sichtbarkeitsbereichen durch Wu u. a. (2014) und Piróg u. a. (2018) nach Agda, eine funktionale Programmiersprache mit Dependent-Types, übertragen lassen. Wir implementieren eine Bibliothek für Wirkungen mit Sichtbarkeitsbereichen. Diese erlaubt sowohl die Implementierung von wirkungsbehafteten Programmen, als auch deren Verifikation durch Agda als Beweisassistenten. Um die Anwendbarkeit unserer Implementierung zu testen, implementieren wir Wirkungen, welche verwendet werden können um Currys (Hanus u. a., 1995) ambienten, nicht strikten Nichtdeterminismus zu modellieren. Unser Hauptbeispiel für eine Wirkung mit Sichtbarkeitsbereich ist Sharing von Nichtdeterminismus, nach Bunkenburgs (2019) Modellierung, um Currys Call-Time Choice Semantik zu simulieren. Um ambiente Wirkungen in Datenstrukturen zu modellieren, unterstützt die Implementierung tiefe Wirkungen.

Contents

Chapter 1

Introduction

Computational effects such as state, nondeterminism and partiality are a central concepts in functional programming and program verification A formal representation for computational effects is essential to allow reasoning about programs using them.

Algebraic effects were first introduced by Plotkin and Power [24] as an alternative representation for computational effects. They represent a large class of effects using a set of operations and an algebraic theory relating them. Furthermore, they provide ways of constructing, reasoning and combining effects [27]. Although they can represent a large set of effects, not all operations from the corresponding monad for the effect fit in the algebraic effect framework. More precisely, they cannot represent non-algebraic operations, for example, operations with scopes, such as `catch` [24].

Effect handlers were introduced by Plotkin and Pretnar [26] as a generalization of exception handlers. They allow modelling of non-algebraic operations and are used in conjunction with algebraic effects [27]. Algebraic effects and handlers have a wide field of applications. In functional programming they are used as a less boiler plate intensive and more composable alternative to monad transformers. They can be used as a theoretical foundation for reasoning about computational effects. Furthermore, they can be used to model semantics of programming languages. Most functional programming languages allow a limited set of side effects which are not represented in the type system of the language. These effects are usually called *ambient effects*. For example, to allow easier development, Haskell programs can contain traces and functions are allowed to be partial. Dylus et al. [11] and Christiansen et al. [10] presented a modelling of ambient effect using free monads in Coq. They showed that forms of algebraic effects and handlers are a convenient basis for implementing effects in languages with dependent types and allow reasoning about Haskell programs taking ambient effects into account.

Wu et al. [34] noticed a lack of modularity when combining multiple effects with scoping operations because handlers delimit scopes as well as define semantics. They introduce two different solutions for the problem. They fix the problem by introducing new operations to mark scopes and by generalizing syntax to model scopes directly. Scoped effects are usually used to implement effect systems in Haskell because they provide a similar interface to normal monad transformers. Furthermore, they are needed to simulate more complex semantics of programming languages, such as sharing.

1.1 Goals

The goal of this thesis is the implementation of an effect library based on the work by Wu et al. [34] and Piróg et al. [23] in the dependently typed, functional programming language Agda [22]. Due to its stronger type system, Agda can be not only be used as a programming language, but also as a proof assistant. Therefore, Agda does not only allow writing programs with more expressive types, but also allows verifying their properties. 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 and therefore complicate porting of Haskell code to Agda.

This thesis explores multiple Haskell implementations of scoped effects and studies how well these translate to Agda. Furthermore, we present 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 [14] call-time choice, and allow verification of equivalent programs written in these languages in Agda. To model ambient effects in data structures correctly, 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. [34]. The former of the two approaches was already partially explored by Bunkenburg [9] 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 it 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. [23]. This approach also fixes the problems of the first approach and avoids 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[1] is a dependently typed functional programming language. The current version (2.6.1.1 as of Nov 2020) was originally developed by Norell [22] under the name Agda2. 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 Haskell's [16]. Data types are declared with syntax similar to Haskell's 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 a nearly arbitrary list of symbols (builtin symbols like colons are not allowed as part of operators). An arbitrary number of 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 addition 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

Agda has a stronger type system than Haskell. Agda supports dependent types, i.e. types that can depended on values. In this section we give a short theoretical introduction to dependent types and show how they can be used to write programs in Agda.

The following type theoretic definitions are taken from the book "*Homotopy Type Theory: Univalent Foundations of Mathematics*" [31]. 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.

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

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

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 is possible to lift a type to a higher universe manually. Since Agda 2.6.1 an experimental `--cumulativity` flag exists.

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, that 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, that 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 Pair Types (Σ-Types) are a generalization of product types. A value of a Σ-type is a pair, but 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. The Agda standard library provides a data type called Σ, that represents normal dependent product types with two elements. Furthermore, it provides a more convenient syntax for writing Σ-types. The type above can be written as $\Sigma[\ a \in A\ ]\ (B\ a)$.

Programming with Dependent Types Next we introduce some common patterns for programming with dependent types. The example of fixed length vectors and indexing using bounded subsets of natural numbers is one of the examples presented by Norell [21].

The data type Vec depends on a type A and a value of type $\mathbb{N}$.

```
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, $\mathsf{Vec}\ A$ is a family of types indexed by $\mathbb{N}$.

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 one. 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 the idea of a type safe head function to type safe indexing. A vector of length n is indexed by the first n natural numbers. The type $\mathsf{Fin}\ n$ 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).

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 is 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.

```
[]       ! () --impossible to reach case, no right-hand side
```

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

As last example we implement the function replicate.

```
replicate : ∀ {A} → A → (n : ℕ) → Vec A n
replicate a 0       = []
replicate a (suc n) = a :: replicate a n
```

Given an element of type A and a natural number n the function produces a vector of the given length. Therefore, the function is a Π-type in its second argument. The type of the returned value depends on the value of n. By pattern matching on the given argument we obtain more information about the return type.

The above, quite simple examples can also be implemented in Haskell, for example using GADTs and 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. The last example of the function replicate, which simply uses a Π-type is more difficult to implement in Haskell, because it requires translating between a value and type level number. Later we will see more complex uses of dependent types, which cannot be translated as easily to Haskell.

2.1.3 Propositions as Types

The idea of propositions as types, also known as Curry-Howard correspondence, relates type theory to logic and is the basis for theorem proving in dependently typed languages, like Agda. In this section we give a short overview over the idea. A more in depth explanation and an overview over the history of the idea can be found in Wadler's paper of the same name [33].

As described by Wadler [33], the idea of propositions as types is a deep correspondence, relating *propositions to types*, *proofs to programs* and *simplification of proofs to evaluation of programs*. First we will 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

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)`
$\neg A$	$A \to \mathbb{0}$	`A → ⊥`
$P \wedge Q$	$P \times Q$	`A × B`
$P \vee Q$	$P + Q$	`A ⊎ B`
$P \to Q$	$P \to Q$	`A → B`
t	$\mathbb{1}$	$\top$
f	$\mathbb{0}$	$\bot$

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

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 prove 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 $\to$ 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 function application 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 value. Therefore, dependent types correspond to predicates. Σ and Π types correspond to existential and universal quantification. To prove 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 is a proof for a predicate on the first value. To prove a universal quantified proposition one has to prove that it holds for every possible value. A Π type is a function whose return type depends on the given value. 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 prove theorems in this 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 *definitionally equal* to 42. A judgment is always true. We define one term to be equal to another one and as result 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 computationally equal if they reduce to the same term. For example, given the above definition of $+$ the terms $0 + (0 + n)$ and n are computationally 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*. We 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 [31, 21, 17].

```
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 are also equal. We cannot produce a value of type $f\ x \equiv f\ y$, because the two are not equal, because x and y are arbitrary variables. 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 return type of the function. Because refl can only be constructed if the two values are the same the two variables are unified. Therefore, we 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, because the first argument of $+$ is now 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 in larger Proofs. For example, using cong we can rewrite arbitrary terms 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 latter 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

Under the idea of propositions as types, non-terminating functions correspond to ill-formed. For example, the term loop = loop has an arbitrary type, i.e. it should be a proof for any proposition. Although, evaluating loop does not yield evidence for the proposition, but a term without normal form or a circular argument. 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 proves 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 allow inlining of functions containing pattern matches during termination checking. A common example are recursive calls in lambdas, that are passed to higher order functions like map and >>=. Consider the following definition of mapRose. The recursive call is made by the higher-order function map for lists. An equivalent definition in Agda does not obviously terminate, because the structural recursion is obscured.

```
data Rose a = Branch a [Rose a]

mapRose :: (a -> b) -> Rose a -> Rose b
mapRose f (Branch x rs) = Branch (f x) (map (mapRose f) rs)
```

To still prove 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 or 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 checking 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 the size of the types values. A set of builtin functions and types can be used to establish relations between different sizes and therefore assist the termination checker. The elements of Size are well ordered. ↑_ is the successor operation on Size, i.e. $i < \uparrow i$. _⊔ˢ_ is the supremum with respect to the relation. ∞ is the top element in Size. Given a Size i, the type of all smaller sizes can be constructed using the Size<_ family of types[2]. If the size decreases across all recursive calls of a function, termination follows from the well-ordering on Size.

[2]We will not need this construction. It is usually used when working with coinductive data types.

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).

Let us again consider the rose tree example. The size index can be intuitively thought of as the height of the tree.

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

When mapRose 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.

```
mapRose : {A B : Set} {i : Size} → (A → B) → (Rose A i → Rose B i)
mapRose f (rose x xs) = rose (f x) (map (mapRose 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 tree of type Rose A ($\uparrow i$). Therefore, the functions has to terminate.

In this case inlining the call of map also solves the termination problem. When generalizing the definition of the tree from List to an arbitrary functor, this would not 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 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 each A_i does not mention D or is of the form

$$B_1 \to B_2 \to \cdots \to B_n \to D$$

where B_j does not mention D. By restricting recursive occurrences of a data type in its definition to strict positive positions strong normalization is preserved [32].

Container

Because of the strict positivity requirement, it is not allowed to apply generic type constructors to recursive 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 strictly positive functors are containers.

Containers are generic representations of data types, that store values of an arbitrary type. They were introduced by Abbott et al. [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 given 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 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

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

We can translate these definitions 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
```

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.

```
⟦_⟧ : 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)
```

As mentioned in Section 2.1.5, Agda does not inline functions containing a pattern match, even if the INLINE pragma is used. Because container extensions are just record types, we can define fmap slightly different to assist the termination checker. By using the projection functions to access the components of the Σ-type we can avoid the pattern match. By forcing Agda to inline the function using the pragma we define a version of fmap, which is transparent for the termination checker. This allows us to use this small auxiliary function in recursive functions.

```
fmap' : ∀ {A B C} → (A → B) → (⟦ C ⟧ A → ⟦ C ⟧ B)
fmap' f c = π₁ c , f ∘ π₂ c
{-# INLINE fmap' #-}
```

2.2 Curry

Curry [14] 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. 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 nondeterminism. 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, `?` introduces a nondeterministic choice between its two arguments. Using `?` we can define a simple nondeterministic program.

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

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

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

2.2.1 Call-Time Choice

Next we 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 behaviours are conceivable. The first possibility is that the choice is moved into the function, i.e. both `x` choose 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 latter option is called call-time choice and it is the one implemented by Curry[3]. The rationale behind call-time choice is that deterministic functions behave the same for deterministic and non-deterministic inputs. A variable bound by a lambda or `let` always refers to just a single value.

Similar to Haskell, Curry programs are evaluated lazily. The evaluation of an expression is delayed until its result is demanded 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 behaviour is called sharing. Usually function application is defined using the `let` primitive [18]. Applying a non-variable expression to a function introduces a new intermediate result, which is bound using `let`.

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

As noted by Hanus and Teegen [15], in Haskell sharing is used to evaluate programs more efficiently, but in Curry sharing is needed for correct evaluation. Sharing allows delaying the evaluation of a function arguments as well as linking all their uses. Sharing a nondeterminism computations means making the same choices in all places the shared term is used. If the choice is demanded, the resulting behaviour is identical to choosing before calling the function. Furthermore, because the `let` primitive introduces sharing, we expect a variable bound by a `let` to behave similar to one bound by a function.

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

As expected this function yields the results 0 and 2.

Because the program is evaluated lazily, it is possible to never demanded a choice. In this case the number of branches does not increase. Consider the following example of the length and the sum of a nondeterministic list.

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

[3]Similar to Haskell, these semantics provide referential transparency (one can freely replace a term with its definition) and definiteness (each variable has exactly one value), but not unfoldability [29]. That is, one cannot simply substitute in non-value function arguments, because they could contain choices. Instead, one expects the above behaviour of conceptually choosing before calling the function.

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs

coins :: [Int]
coins = [coin, coin]
```

When summing up the list `coins`, all choices in the components are demanded. Therefore, we obtain the four results 0, 1, 1 and 2. In case of length this does not happen. The function does not demand the values stored in the list. Therefore, we obtain just the single result 2.

2.3 Algebraic Computational Effects

Computational effects are a wide concept and usually introduced by example. They include I/O, exceptions, nondeterminism, state and continuations [25, 6]. Algebraic effects are computational effects, that can be described using an algebraic theory. They were first presented Plotkin and Power [25], as an alternative to monads for describing computational effects. They capture a wide class of computational effects. Due to their uniform representation they provide a general way of defining, combining and reasoning about computational effects [27]. Furthermore, by describing computational effects in terms of operations and equations, they directly allow equational reasoning about effectful programs.

Handlers for algebraic effects were first presented by Plotkin and Pretnar as a generalization of exception handlers [26]. They describe how to transform effectual computations and can be used to describe non-algebraic operations, that is operations which did not fit in the general theory of algebraic effects. Examples include operations with scopes such as `catch` and `once`.

Wu et al. [34] describe a problem with this approach when multiple handlers interact. The ordering of handlers induces a semantic for the program, but simultaneously handlers which implement scoping operations 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 [34, 23].

In the following sections we give a short introduction to the classical notion of algebraic effects and handlers as described by Bauer [6] and used in Chapter 3 as well as scoped effects as described by Wu et al. [34].

2.3.1 Algebraic Theories

The following definition is similar[4] to the one given by Bauer [6]. Bauer uses abstract algebra not arbitrary categories. We use his work to define some basic concepts and terminology regarding algebraic effects.

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 symbol and the two sets, suggesting that they describe special functions.

$$\Sigma = \{\mathrm{op}_i : P_i \rightsquigarrow A_i\}$$

Given a signature Σ we can build terms over a set of variables $\mathbb{X}$. When describing operations of computational effect, the set of variables corresponds to the set of values produced by the computation.

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

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.

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

[4] We use a slightly different notation for the assignment of variables when interpreting terms.

An interpretation I of a signature is given by a carrier set $|I|$ and an interpretation for each operation, given by $[\![\cdot]\!]_I$. 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 that are 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.

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

Example Consider the theory for a monoid. The signature for a monoid contains two operations, the binary multiplication $\cdot$ and the nullary operation defining the neutral element e. For a given set of values A the set $\mathrm{Tree}_{\mathrm{Mon}}(A)$ describes term trees build using values from A, the constant e and the operation $\cdot$. The laws for a monoid state that the binary operations is associative, therefore, the equivalence relation $\sim_{\mathrm{Mon}}$ equates all trees that are equal up to rotation. The constant e is the left and right identity for $\cdot$, therefore, $\sim_{\mathrm{Mon}}$ also equates (sub)trees describing the terms $e \cdot t = t = t \cdot e$.

Because trees can be rotated freely, the only important information is the order of elements. Identities can be freely omitted, since the only important term involving identity is e. It is easy to show that $\mathrm{Tree}_{\mathrm{Mon}}(A) \,/\, \sim_{\mathrm{Mon}} = \mathrm{Free}_{\mathrm{Mon}}(A) \cong A^*$, i.e. the free model for the theory of a monoid over a set of values A are words over A. The empty word ε corresponds to the identity element and $\cdot$ corresponds to concatenation $\circ$. This is of course the usual definition of a free monoid over A.

2.3.2 Effect Handlers

Lastly we introduce the notion of an effect handler, again as described by Bauer [6]. Effect handlers were first presented by Plotkin and Pretnar [26]. They are a generalization of exception handlers and can be used to model a wide range of non-algebraic effects. A handler

$$h : \mathrm{Free}_T(V) \Rightarrow \mathrm{Free}_{T'}(V')$$

is a map between free models for theories T and T' over value sets V and V'. A handler is given by an additional T-model on $|\mathrm{Free}_{T'}(V')|$ and a T-homomorphism from $\mathrm{Free}_T(V)$ to the additional model. Notice that the set $|\mathrm{Free}_{T'}(V')|$ is equipped with two models, the free T' model it was generated by, as well as an additional (not necessarily free) T model.

Because a handler is a homomorphism from a free model it is uniquely determined by a map $f : V \to |\mathrm{Free}_{T'}(V')|$ for the underlying values and the interpretations $[\![op_i]\!] : P_i \times |\mathrm{Free}_{T'}(V')|^{A_i} \to$

$|\text{Free}_{T'}(V')|$ for each operation op_i in the target T-model. The handler is then given by the following equations.

$$h([\text{pure } x]_{\sim_T}) = f(x)$$
$$h([\text{op}_i(p, \kappa)]_{\sim_T}) = h_i(p, h \circ \kappa)$$

Usually, when defining a handler, the operations of the theory of the right-hand side are a subset of the ones from the left-hand side. The operations are interpreted by extending the structure on the values, i.e. mapping the impure model, given by $\text{Free}_T(V)$, to a pure one. For example, a handler for exceptions could be given by

$$\begin{aligned} \text{handleExc} : \text{Free}_{\text{Exc E}}(A) &\Rightarrow \text{Free}_{\emptyset}(E \uplus A) \cong E \uplus A \\ \text{pure } x &\mapsto \text{pure } (inj_2\ x) \\ \text{throw}(e, \kappa) &\mapsto \text{pure } (inj_1\ e) \end{aligned}$$

where the source is a free model for the theory of exceptions E and the target is a free model for the empty theory. The model for the theory of exceptions on the right-hand side is given by the extended set of values, which corresponds to the usual monad modelling the effect. Another handler for the theory of exceptions is the catch operation. The handler is parameterized over a function mapping exceptions to alternative results. The handler simply maps occurrences of throw e to the alternative result for e.

If the source and target theory share operations, it is possible to interpret an operations by itself, i.e. not handle it. This allows us to handle more complex combined theories using multiple modular handlers. It is also possible to interpret operations using other operations, i.e. to simulate an effect in the source theory using effects in the target theory. For example, one could simulate a logging effect (in Haskell `Writer String a`) using a state effect (in Haskell `State String a`).

2.3.3 Free Monads

The syntax of an algebraic effect is described using the *free monad* [30]. Given any functor `f`, the data type `Free f a` is a monad. Furthermore, the natural transformations from any functor f to any monad m (in Haskell functions of type `forall a. f a -> m a`) are in bijective correspondence with monad homomorphisms from `Free f a` to `m`. Therefore, each monad can be described as the image of a monad homomorphisms from a free monad. The usual definition of the free monad in Haskell is given below.

```
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 essential represents computation trees. Computations are either `pure`, they yield a value without calling other operations, or `impure`. An `impure` computation is given by a value of the free monad that was lifted using the functor. The constructors for `f a` correspond to operations and arguments of type `a` correspond to subcomputations or arguments for the operation. For example, the functor `data Mon a = Mul a a | E` corresponds to a signature with a binary operation and a constant. `Free Mon a` is the type of computation trees using variable of type `a`, the constant `E` and the binary operation `Mul`.

`return` lifts a value to a computation by creating a `Pure` computation that returns the value. The $\gg=$ operation for the free monad corresponds to variable substitution. When calling $\gg=$ on an `Impure` value, $\gg=$ is called recursively on all subcomputations using `fmap` for the functor. `Pure` leafs are substituted with the subtree, generated from their stored value. Therefore, $\gg=$ allows processing the resulting values of an effectful program with another effectful program.

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 $\text{Tree}_T(\mathbb{X})$. When implementing algebraic effect one simply uses theories without equations, that is, just the signatures. By moving

from free models to term trees we obtain a uniform representation of syntax as free monad over a functor. As a result, for theories with equation some trees that should be equal are not. These terms are just syntax without semantics[5]. Semantics are completely determined by the handler that interprets the syntax by mapping it to an actual model. When interpreting the syntax using a handler falsely different values can again be equated by treating them the identically. Therefore, if syntax is interpreted by a correct handler the equations should hold again and working directly on trees, not equivalence classes of trees, is correct as long as the handlers are correct. Furthermore, by representing effectful programs as computations trees one delays the actual evaluation. This allows a clear separation into syntax and semantics and allows choosing the semantics at a later point in time.

Free Monads in Agda 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. Replacing arbitrary functors with container extension, especially to implement the free monad, is a well known approach [11, 10, 20].

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

Because we constrain the generator from arbitrary to strictly positive functors, we cannot model all free monads with this data type. Using Free we can model a class of free monads, which are accepted by Agda. The definition of Containers correspond closely to the one for signatures. The constructors for Shape correspond to the operation symbols op_i and the constructor arguments correspond to the parameter P_i. A constructor without extra arguments corresponds to an operation with parameter set $\mathbb{1}$. The type of positions for a shape 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 operation. In the free model the parameters/child trees are given by the function κ, which maps from the arity set to the carrier set. This function corresponds to the position function in the Container extension, which maps from the type of positions to the stored values. In this case the stored values are exactly the subcomputations of type Free C A.

Algebraicity In there categorical formulation by Plotkin and Power [25], algebraic operations are required to have a certain natural condition. In the earlier definition this requirement is captured by restricting the shape of equations. When described by the free monad, algebraic operations are those which commute with $\gg\!=$, i.e. for an n-ary algebraic operations the following holds [23].

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

For common operations with scopes, such as `once`, `catch` or `fork`, this does not hold. Dealing with this limitation in a modular way leads to the general notion of scoped effects.

2.3.4 Scoped Effects

As shown by Plotkin and Power [25] many computational effects and therefore their associated monads can be described as algebraic theories, but not all operations know from these monads are algebraic. The most common example is `catch`, as it was noted early by Plotkin and Power [24]. `catch` takes two arguments, a computation `p` which maybe produces an exception and a handler `h`, which produces an alternate result for each exception. If `catch` were algebraic, the following equation would hold.

$$\mathbf{catch}\ p\ h \gg\!= k = \mathbf{catch}\ (p \gg\!= k)\ (\lambda e.h\ e \gg\!= k)$$

The equations obviously does not hold because if it would hold, `catch` could not differentiate between an exception thrown inside or outside its scope. The same pattern can be observed with other operations that delimit scopes.

[5] For some effects these two are the same, i.e. their corresponding monad is free. For example, in case of exceptions the `throw` already has the correct semantics. Because it has no sub-computations, it aborts the program.

As noted in Section 2.3.2, non-algebraic and in particular operations with scopes can be implemented as a handler. As noted by Piróg et al. [23], this means that non-algebraic operations always have associated semantics, while algebraic operations are just syntax and are assigned meaning later by a handler. Furthermore, Wu et al. [34] note a general problem when working with multiple handlers describing scoped operations. The order of handlers determines the semantics of a program, but simultaneously the handlers are used to delimit scopes. Therefore, their position are fixed. Problems arise if the ordering for certain semantics and the forced positions for delimiting scopes do not coincide. It is either impossible to model certain semantics or one has to evaluate handlers multiple times, reinjecting intermediate results into the computation. Wu et al. [34] note that the root of this problem is that handlers combine syntax and semantics by delimiting scopes as well as interpreting syntax.

To fix this problem Wu et al. [34] introduced two solutions for syntactically delimiting scopes. Using this new syntax, the handlers are freed from delimiting scopes and are again purely semantics constructs, which can be moved freely to the outside of the program. By expanding the syntax, the handling of effects becomes more difficult. In Chapter 3 and Chapter 4 we explain these two approaches and discuss how well they translate from Haskell to Agda.

Chapter 3

Implementing Scoped Effects in a First Order Setting

In this chapter we present our first implementation of algebraic effects in Agda. The implementation uses a traditional approach, i.e. it represents effects as functors and uses the free monad, as described in Section 2.3.3. We focus on implementation details specific to Agda, for example termination and positivity checks.

In Section 3.1 we introduce our general infrastructure for modular effects, which provides a similar interface to the one by Wu et al. [34]. The basic idea of operating on lists of effects is similar to the Idris effect library by Brady [8]. In Section 3.2 we implement some of the examples by Wu et al. [34] to explain the general construction of effects and handlers using this approach. In Section 3.3 we implement scoped effects using explicit scoped delimiters as described by Wu et al. [34]. To define handlers for effects with and without scopes we use a general construction using sized types to prove termination. This allows us to conveniently implement handlers in Agda, i.e. without changing the structure of handlers to prove termination. As a more complex example, we implement the sharing handler by Bunkenburg [9], providing a way of simulating call-time choice semantics in Agda.

3.1 Functors à la Carte

When modelling effects each functor represents the signature, i.e. the operations for an effect. Theoretically, it is possible to represent each operation as a functor. Usually, operations which are handled together are implemented as a single functor. 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 algebraic theory without the equations. To combine the syntax of multiple effects we can combine the underlying functors because the free monad preserves coproducts[1]. This method of modularization of syntax was presented by Swierstra [30].

The approach for modularization functors described by Wu et al. [34] is based on "Data types à la carte" by Swierstra [30]. 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 2.3.3, 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 given by the coproduct mediator[2] of F's and G's position functions.

[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`.

```
_⊕_ : Container → Container → Container
(Shape₁ ▷ Pos₁) ⊕ (Shape₂ ▷ Pos₂) = (Shape₁ ⊎ Shape₂) ▷ [ Pos₁ , Pos₂ ]
```

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

```
sum : List Container → Container
sum = foldr _⊕_ (⊥ ▷ λ())
```

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" [30] approach the type class `:<:` is introduced. `:<:` relates a functor to a coproduct of functors, marking it as an option in the coproduct. `:<:`'s functions can be used to inject or potentially extract values from a coproduct. The two instances for `:<:` mark F as an element of the coproduct if it is on the left-hand side of the coproduct (in the head) or if it is 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 slow[3].

Instead, we used an similar approach similar to the Idris effect library by Brady [8]. 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[4] 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 instance keyword marks the two constructors as possible solution for instance arguments. Instance arguments are special hidden arguments which are denoted using ⦃ and ⦄. Instances and instance

[3] We encountered cases where type checking of overlapping instances involving _⊕_ did not seem to terminate.

[4] This proposition is just a special case of the Any proposition on Lists. For simplicity, we implement the proposition here directly. Types like _∈_ are an example of a common pattern in dependently typed languages called *view*. The purpose view type is just to pattern match on its values, to reveal more information about its indices [21].

arguments are resolved using a special instance resolution algorithm and can be used similar to Haskell's type classes and type class constraints.

The two instances still overlap resulting in a potentially exponential slowdown in instance resolution[5]. Using Agda's 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. The repository contains an implementation using tactics, based on the work by Norell.

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 effect is an element of the list used to construct the coproduct.

Let us consider inject first. By pattern matching on the evidence we acquire more information about the type sum *ops*. In case of here we know that C is in the head of the list of containers. Because the return type is constructed using sum over the given list, we know that the given value s , pf has the same type as the left alternative in the returned coproduct. In case of there we obtain a proof that the container is in the tail of the list, which we use to make a recursive call. The with keyword allows us to call a function and add its result to the left-hand side of our function equation. By pattern matching on and repackaging the result we obtain a value of the correct 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. Notice that in both cases we do not handle cases related to the empty list, because it is impossible to prove that the empty list has a member.

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 our first definition in section 2.3.3, this version 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. We could store a proof that some effect E is an element of *effs* directly. This would allow us to use E directly, allowing us to omit the sum, but also raise the universe level of Free.

```
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. Therefore, a calculation is larger than its continuations or a computation is finite. Using the annotation it is possible to prove that functions preserve or decrease the size of a computation and therefore 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. [34]. inj allows injecting syntax into a program whose

[5] https://agda.readthedocs.io/en/v2.6.1.1/language/instance-arguments.html#overlapping-instances

signature allows the operation. prj allows to inspect the next operation of a program, restricted to the signature for a specific effect. Furthermore, we add the functions op and upcast.

Writing operations with explicit continuations is unintuitive. Furthermore, $\gg=$ and do-notation provide a more convenient syntax for extending the continuation. op generates the *generic operation* for any operation symbol. The generic operation for $\mathsf{op} : P \rightsquigarrow A$ is the operation with pure as continuation. Therefore, it is a function of type $P \to \mathrm{Free}_T(A)$, justifying the $\rightsquigarrow$-notation.

upcast transforms 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 $\gg=$ and return would be enough to define all the other functions, but $\gg=$ is not size preserving, while some other functions can be defined such that they are. For example, later we 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)

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, $\gg=$ does not preserve the size. $\gg=$ 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 estimate. 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 they 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 {i} → (A → Free ops B) → Free ops B
pure x       >>= k = k x
impure (s , pf) >>= k = impure (s , (_>>= k) ∘ pf)

_>>_ : ∀ {ops} → Free ops A {i} → 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)
```

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

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 functions via fmap preserves the identity and the composition of functions.

Both properties are proven by structural induction over the free monad. Let us consider the first proof. It is written using chain reasoning operators [22] to display the general structure of the later proofs. The operators provide a convenient way of writing down transitive equalities. If the equality of two terms is not proven by refl, we use _≡⟨_⟩_, which allows us to give a proof of their equality as second argument. 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 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. The law essentially allows us to swap the arguments of <*> if one of the is pure. Because <*> is defined in terms of <$>, it allows us to reduce some <*> calls to <$>.

```
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 (f >=> g = g ∘ (_>>= f)) 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 interested in this formulation using >>=, because it is the one occurring 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 Implementing Algebraic Effects and Handlers

In section 2.3.2 we introduced effect handlers more formally. In this section we implement algebraic effect and their handlers in Agda.

All handlers in the following sections have the same basic structure. They take a program, 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 its effect stack. Furthermore, they potentially modify the type A to one modelling the result of the effect. They transform an impure computation partially to a pure one. By interpreting the syntax of an effect (mapping to a model for the theory) the handler defines the semantics for the effect's syntax. For example, a handler for exceptions removes exception syntax and transforms a program producing a value of type A to one producing a value of type $E \uplus A$, either an exception or a result. Internally the handler maps the impure throw operation to a pure computations, witch simply yields the exception inside an either monad. Some later handlers will also modify syntax of other effects. These handlers are usually called *non-orthogonal handlers*.

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, because its Shape type has no constructors. Pattern matching on the impure constructor yields (). The program does not call impure operations and always yields 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 groundwork 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. Nondeterministically choosing between two paths means that ⁇ nondeterministically produces a boolean or has two continuations. Therefore, we have a nullary and a binary operation, both without additional parameters.

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

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

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

When constructing the container we assign the correct arities to each shape, by defining a Π-type which produces the correct type for each shape.

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

We can now define smart constructors for each operation. These are not the generic operations, but helper functions based on them. The generic operations take no additional parameters and always use pure as continuation. These versions of the operations already process the continuations parameter. The generic ⁇ operation nondeterministically produces a boolean, but usually we want to use it analogous to Curry's `?` operator. Therefore, we implement this refined version. op failˢ 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 failˢ >>= λ()
```

With the syntax in place we can now move on to semantics and define a handler for the effect. Wu et al. [34] use Haskell language extensions like pattern synonyms and view patterns to simplify their implementations. Agda also has pattern synonyms, which we use to simplify at least simple effect handlers, i.e. those which do not rely on prj. Similar to Haskell, the left-hand side of a pattern defines an abbreviation for the right-hand side, which can be used on the left- and right-hand side of function rules. We introduce patterns for each operation from the signature as well as for other operations, i.e. those who are not part of the currently handled signature.

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

The handler interprets Nondet syntax and removes it from the program. Therefore, Nondet is removed from the front of the effect stack. For the handler different semantics are conceivable. For example, a handler could implement backtracking, that is, search for the first result that is not fail. The result type of such a handler could be modelled using a Maybe. In our case, we implement a handler that returns all results. Therefore, the result is wrapped in a List. The List contains the results of all successful execution paths.

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

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

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

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

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

In case of a Choice both paths can produce an arbitrary number of results. We execute both programs recursively using runNondet and collect the results in a single List. Note that we invoked the continuation of the operation with all possible arguments. A handler just searching for a single result would potentially discard one execution path.

⦇ 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.

```
runNondet (Choice κ)  = ⦇ runNondet (κ true) ++ runNondet (κ false) ⦈
```

In case of syntax from another effect we just execute runNondet on every subtree by mapping the function over the container. Note that the newly constructed value has a different type. Other hides the inj_2 constructor. Therefore, the newly constructed impure value is of type Free ops (List A). Furthermore, notice that this function rule corresponds to one of the equations for handlers, given in section 2.3.2.

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

Even though, this handler is defined recursively the definition from section 2.3.2 can still be recognized. The case for pure corresponds to the function mapping values to computations. The cases for each operation correspond to the maps h_i, which process the parameter and the results of the continuation. The handler is not explicitly composed with the continuation before calling h_i, but in the cases above all results of κ are processed by runNondet.

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's syntax 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 prove that our handler produces a valid (but not necessarily free) model for our equations by verifying that after interpreting the syntax the laws hold. The ability to rigorously prove properties of a program, such as these laws for our handler, 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. Writing the law using bind leaves us with the following equality.

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

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) → runNondet (fail ⁇ q) ≡ runNondet q
⁇-identˡ q = begin
  runNondet (fail ⁇ q)                                ≡⟨⟩ -- def. of handler for ⁇
  _++_ <$> runNondet fail <*> runNondet q             ≡⟨⟩ -- def. of handler for fail
  _++_ <$> pure []        <*> runNondet q             ≡⟨⟩ -- def. of <$>
  pure ([] ++_)           <*> runNondet q             ≡⟨⟩ -- def. of <*> / ap-ident-left
  [] ++_                  <$> runNondet q             ≡⟨⟩ -- def. of ++ / ++-ident-left
  id                      <$> runNondet q             ≡⟨ fmap-id (runNondet q) ⟩
  runNondet 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 required.

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

$$\Sigma_{State} = \{\text{get} : \mathbb{1} \rightsquigarrow S, \text{put} : S \rightsquigarrow \mathbb{1}\}$$

As before we 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 definitions 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 returned in addition to the result.

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

A pure calculation does not 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 for get maps a state to the rest of the computations, which somehow processes the given state. By applying s_0 to κ we obtain the rest of the computation for the current state, which we evaluate recursively. Notice that this operation technical has $|S|$ continuations, but we only call a single one. The handler for Nondet always uses all of its continuations. In reality the continuations are of course implemented as a single function, which takes a parameter of type S.

```
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. The current state is passed to the recursive calls of the handler for each continuation.

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

Lastly we define the usual helper function evalState, which simply drops the final state.

```
evalState : ∀ {ops S} → S → Free (State S :: ops) A → Free ops A
evalState s₀ p = proj₂ <$> runState s₀ p
```

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

To show that our implementation of State is correct, i.e. that it is a model, we have to show that the equations characterizing the effect hold. Because some equations use just get and put, *ops* cannot be automatically inferred.

```
go : Free (State S :: ops) A → Free ops (S × A)
go = runState {_} {ops} s₀
```

Given an arbitrary initial state s_0 and rest effect stack *ops* the following equations have to hold.

```
∀ {s₁ s₂ : S} → go (put s₁ ≫ put s₂) ≡ go (put s₂)
```

```
∀ {s : S} → go (put s ≫ get) ≡ go (put s ≫ pure s)
```

```
∀ {k : S → S → Free (State S :: ops) A} →
  go (get >>= λ s → get >>= k s) ≡ go (get >>= λ s → k s s)
```

```
go (get >>= put) ≡ go (pure tt)
```

All four equalities are proven trivially using refl. Our model for State is just the usual state monad. Plotkin and Power [25] showed that the four equalities above are enough to determine the state effect and that the free model is the usual state monad.

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))
```

As explained by Wu et al. [34], 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 programs, both branches continue with their own state. The initial state for both branches is the current state of the branch calling ⁇.

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 and all act on the same state.

The handlers for the two effects define semantics for their operations. Operations of the other signature are just traversed. The interaction semantics are induced by the ordering of the handlers [34]. 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 ∘ runNondet ∘ runState s₀

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

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. introduce scoped effects [34]. They presented two syntactic approaches to explicitly declare how far an operations scopes over an arbitrary program. In the following section we implement the scoped effect Cut following Wu et al. [34] in Agda.

The central idea by Wu et al. [34] is to add new effect syntax, representing explicit scope delimiters. When handling the new syntax, whenever an opening delimiter is encountered another handler, that implements the non-algebraic operation, evaluates the program in scope. Whenever the second handler encounters a closing delimiter it stops the evaluation and returns the rest of the program. Due to a closing delimiters the end of a scope can be found. Therefore, calling $\gg=$ on a scoped operation, to extend the continuation, is not a problem, because the continuation can be separated in the parts inside and outside the scope. By introducing scope delimiters Wu et al. [34] move the problem of scoping from the handler to the syntax and therefore allows a flexible ordering of the handlers and the implementation of operations with scopes. Furthermore, this change allows calling all handlers for the program at once, similar to `runM` functions for monad transformers. A more in depth explanation, as well as an example were the handler orders for semantics and scoping do not coincide can be found in the original paper by Wu et al. [34] as well as in the repository.

Separating the program into the parts inside and outside the scope requires a more complex recursive handler, because scopes could be nested. Using our size annotated free monad we are able to transfer the implementation by Wu et al. [34], without making large changes. In the following sections we will implement scoped effects and exemplarily present how the Haskell implementation can be transferred to Agda.

3.3.1 Cut and Call

As first example for an effect with explicit scope delimiters we implement cut and call following Wu et al. [34]. The effect is implemented in two stages. First we implement the traditional effect Cut. It allows to prune branches from nondeterministic computations. The handler for the effect determines the scope of the pruning operation. It corresponds to the usual call operation, which restricts the cut's scope. Second we implement an effect which marks scopes, when handled isolates the programs in scope and evaluates the handler for Cut on each of them. This allows us to implement a version of call, which is not a handler. First we define the syntax for the actual 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 prunes all unexplored branches and calls fail. Using cutfail we can implement the usual cut operation. The structure of the Agda implementation of the handlers is identical to the one by Wu et al. [34], but to prove termination we have to use the size annotation.

Internally, the handler calls the function go, which accumulates the unexplored alternatives in its second argument using ⁇. fail is the neutral element for ⁇ and therefore the default argument.

Since this handler is non-orthogonal (it interacts with another effect's syntax) Nondet is required to be in the effect stack, but its position is irrelevant. We require a proof that Nondet is in the effect stack. Using prj we can inspect Nondet syntax without explicitly knowing how the container coproduct was constructed.

To prove termination we mark the second argument of go with an arbitrary size i. The continuations for each operation return subterms indexed by a smaller size. Therefore, recursive calls to go with these terms as argument 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. Therefore, we return a program that chooses 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 discarding 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. Therefore, we can still prove 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 directly passed to the recursive call of go. The handled 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 earlier accumulated alternatives.

```
... | just (failˢ , _ ) = q
```

Syntax of 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. [34], but to prove termination we again add Size annotations to the functions.

The bcall and ecall handlers remove the scope delimiter syntax from the program and run call (the handler for cut) at the beginning 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 crucial to prove that the recursive calls to bcall and ecall using ≫= terminate. Remember that bind discards the size of its argument. For handlers following this pattern this is not a problem, because the returned program (i.e. the inner Prog layer) is size annotated.

Calling the handler on the extracted program guaranties that the handler does not interact with syntax outside the intended scope and the handlers state, in this case the separated branches, is isolated. Nested scopes are handled using recursive calls to ecall if BCall operations are encountered while searching a closing delimiter.

Since the delimiters can be placed freely it is possible to mismatch them. If we encounter a closing before an opening delimiter, we know that they are mismatched. Wu et al. [34] use Haskell's `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 some form of exceptions, for example the Maybe monad. For simplicity, we 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. Semantically it is identically to the earlier call handler, but this version of the function can be moved freely, because it is not a handler.

```
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 [9]. In this section we extend the nondeterminism effect from section 3.2.1 to one modelling call-time choice.

As explained in section 2.2.1, in Curry call-time choice semantics are realized using sharing of nondeterministic choices. 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 make two changes to the nondeterminism effect from Section 3.2.1. In the following, the new effect and all its associated functions have names ending in ′. Each choice is augmented with an optional identifier consisting of a triple of natural numbers, called *choice id*. The first two are used to identify 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 values. The modified handler is identical to the handler in Section 3.2.1, except that it stores values using the leaf constructor and concatenates results using branch. This change allows to choose the evaluation strategy, e.g. depth first or breath first search, independent of the handler.

```
SID CID : Set
SID = ℕ × ℕ
CID = SID × ℕ

data Tree (A : Set) : Set where
  branch : Maybe CID → (l r : Tree A) → Tree A
  leaf   : A → Tree A
  failed : Tree A
```

3.4.1 Deep Effects

In Haskell and Curry, ambient effects (e.g. 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 or in Haskell the list's tail could be `undefined`. These effects inside data structures are sometimes called *deep effects*. Because Haskell and Curry are evaluated lazily, it is possible to never demand the effectful data. In this case the effect should not occur. For example, in Haskell it is possible to calculate the length of a list, that stores `undefined`s.

We want to simulate this behaviour with algebraic effects. The effects are modelled explicitly using the Free type. We lift data types using a standard construction, which is commonly used to simulate ambient deep effects [3, 11, 10]. The following example of an effectful List demonstrates 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}
```

List^M *ops* *A* represents a List *A* in whose components effects from the given effect stack *ops* can occur. Notice that the lifted list directly stores programs, i.e. values of type Free *ops* *A*. This makes it possible to ignore parts of the data structure and their associated effects. This is not possible with simple lifted lists, i.e. values of type Free *ops* (List *A*). Furthermore, this construction allows us evaluate the effects of the different components at different points in time, by calling $\gg=$ on them separately. 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 prove that structural recursive functions terminate. To pattern match on a lifted value we have to use $\gg=$. Therefore, the structural recursion is obscured. Due to the size annotation it is still possible to prove termination.

```
_++^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
```

By calling $\gg=$ we extend the given computation tree at its leafs with the binds continuation. This corresponds to demeaning the value in a lazily evaluated language. The operations in the continuation are placed explicitly after the operations in the bound program. For example, in the above program, we bind the effects in the first lists constructors from head to tail. The values stored by each $::^M$ and therefore their effects are ignored.

Normalization of Effectful Data Based on the code by Bunkenburg [9] we 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
```

The type class allows normalizing 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 the arbitrary types *A* and *B* we could have parameterized the type class over a type

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

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, is not used in the original implementation.

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

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

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

3.4.2 Sharing Handler

In this section we implement the sharing effect by Bunkenburg [9]. The effect provides the share operation, which can be used to share nondeterministic choices in a program. As explained in Section 2.2, sharing of nondeterminism is the basis for call-time choice semantics in Curry. Because only the choices in scope of the share operation are shared, it should be obvious that this is a scoped effect.

The basic structure of the following sharing handler is identical to the one presented by Bunkenburg [9]. Due 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 non-orthogonal, because it 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's shape.

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

Share : Container
Share = Shareˢ ▷ 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 are again handled using bshare. Nested scopes are handled using recursive eshare calls. To implement sharing the handler modifies the program in scope. Using prj, Nondet′ syntax is extracted, modified and reinjected. Choice operations are modified to include the unique choice id. The id is generated from the scope id of the current sharing scope and a counter, that is managed by the handler. When ever a choice is tagged with an id, the counter is incremented. The uniqueness of the scope identifiers is not managed by the handler, but the share operator.

```
bshare : ⦃ Nondet′ ∈ ops ⦄ → Free (Share :: ops) A {i} → Free ops A
eshare : ⦃ Nondet′ ∈ ops ⦄ → ℕ → SID → 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) -- mismatched 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)
```

Due to the size annotations, termination is proven on the type level. To implement the sharing handler it is not necessary to combine the two cases, as done by Bunkenburg [9] in Coq.

3.4.3 Share Operator

Next we define the share operator as described by Bunkenburg [9]. 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) : Set where
  field shareArgs : A → Free ops A
open Shareable ⦃...⦄

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 to generate the unique ids. State is later executed as first effect in the stack, to generate concrete identifier values for the sharing handler. 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 executed immediately (i.e. injected into the larger program at the current position) using >>=. The outer computation generates a new identifier. The inner computation actually evaluates the shared program and manages its sharing scope. 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" [9].

```
share : ⦃ Shareable ops A ⦄ → ⦃ Share ∈ ops ⦄ → ⦃ State SID ∈ 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 of shareArgs is a higher order function, that takes the used sharing operator as an additional argument. For simplicity and to avoid termination problems this is not the case in the Agda implementation. Note that we implement Shareable for lists of size i to guaranty termination of shareArgs.

```
instance
  ListM-shareable : {! Shareable ops A !} → {! Share ∈ ops !} → {! State SID ∈ 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, such that not only surface, but also deep effects are evaluated. State is evaluated first to generate the identifiers for the sharing scopes. Sharing is evaluated next to tag all choices with their scopes identifier. At last nondeterminism is evaluated to collect the results of all possible branches into a tree. The tree is traversed using depth first search. When ever a choice with a choice id is encountered its decision is remembered.

```
dfs : {A : Set} → Map Bool → Tree A → List A
dfs mem failed                = []
dfs mem (leaf x)              = x :: []
dfs mem (branch nothing  l r) = dfs mem l ++ dfs mem r
dfs mem (branch (just id) l r) with lookup id mem
... | nothing = dfs (insert id true mem) l ++ dfs (insert id false mem) r
... | just d  = if d then dfs mem l else dfs mem r

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 is the original coin function inside a sharing scope, that 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 :: []
```

Due to the Shareable type class it is also possible to share choices, which are embedded into data structures. The following simple example demonstrates this possibility.

```
headM : {! Nondet′ ∈ ops !} → Free ops (ListM ops A) → Free ops A
headM 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 → (! headM mxs + headM mxs !)
```

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

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

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

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

Figure 3.1: Laws for a monad with sharing by Fischer et al. [12]

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

Using the Normalform type class it is 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 mys) → (mx ::^M my ::^M mys) ??' (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 :: []) :: []
```

3.4.5 Laws of Sharing

Fischer et al. [12] introduce a set of equations characterizing call-time choice semantics. 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 does not hold for this implementation, because there is no guarantee that get or put are not 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 is not 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 does not 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).

3.5 Results

In this chapter we implemented usual notion of algebraic effects and handlers. Based on the work by Brady [8], we introduced our general infrastructure for working with modular effects in Agda. In addition to the implementation of effects and the handlers, we used Agda's theorem proving capabilities to reason about effectful programs. Furthermore, we used Wu et al. [34] method of explicit scope delimiters to provide modular versions of operations with scopes. Our sized annotated version of the free monad allowed us to implement the more complex, recursive handlers for scope delimiters, without changing the general structure of the implementation by Wu et al. [34]. Following Bunkenburg [9] we were able to implement a general infrastructure for deep effects as well as a handler for sharing, allowing us to simulate call-time choice semantics in Agda.

Because we simply implemented the Haskell version of this approach in Agda, the problems from the Haskell implementation also carry over. The problem noted by Wu et al. [34] is malformed syntax. Because the beginning and end of each scope are marked using different operations, the handler has to deal with potentially mismatched scope delimiters. This problem is amplified in Agda, because mismatched scope delimiters could potentially also be encountered in proofs. Furthermore, when handling operations with scopes the handlers has no direct access to the continuation of the operation. This prevents potentially optimizations. For example, direct evaluation of the continuation with memorized result of a shared computation.

Chapter 4

Implementing Scoped Effects using Higher Order Syntax

To address problems of the first order approach, noted in Section 3.5, Wu et al. introduce a second approach using higher order abstract syntax [34] to directly represent scoping constructs. By changing the syntax, this approach moves away from the exact definition of algebraic effects, but the general concept of defining syntax and inducing semantics using a handler is still present. This approach is usually used in Haskell to implement scoped effects (e.g. polysemy [19], fused-effects [35]). Bunkenburg already tried to implement some scoped effects using this approach in Coq. His approach failed due to issues with Coq's positivity checker [9]. Due to sizes issues, needed for the consistency of Agda, this approach ultimately failed again.

In Section 4.1, we introduce the concept of higher order syntax as described by Wu et al. [34]. We partially follow Bunkenburg's approach because the limitation of Coq described by him does not exist in Agda. In Section 4.1.1, we discuss different representation for higher order syntax in Agda. We explain a size issue with all of these representations that generally prohibits us from implementing deep effects without making excessive changes. In Section 4.1.3, we discuss some potential changes to the representation and their consequences. Because deep effects are an uncommon requirement, in Section 4.2 we still present a limited implementation to highlight some details specific to Agda and lay groundwork for Chapter 5.

4.1 Higher Order Syntax

In the first order approach operations only store their continuation. If the operation has a local scope, the result of the continuation is expected to contain a closing scope delimiter, which the handler uses to reconstruct the scope. In the higher order approach by Wu et al. [34] scoped operations store the part of the program in their scope and their continuation separately. Therefore, the handler can directly access the program in scope as well as the rest of the program. In the following, we explain the basic construction of higher order syntax in Haskell by Wu et al. [34]. Afterwards, we present results regarding our Agda implementation.

In the first order approach the functors, that describe the syntax, are applied to types of the form `Free F A`. These type represent 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 `(* -> *) -> (* -> *)`. In the implementation by Wu et al. [34], the `Prog sig a` data type represents syntax. It is a slight generalization from the free monad in the first order approach. In the `Impure` constructor the higher order functor is applied to the syntax (represented by `Prog sig`) and the result type `a` separately.

```
data Prog sig a = Pure a | Impure (sig (Prog sig) a)
```

Because the program type is separated in syntax and result type it is possible to store programs, that use the same signature but produce values of different a type. Consider the following definition of the higher order exception syntax as given by Wu et al. [34].

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

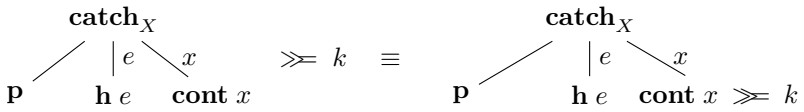

Figure 4.1: Calling ≫= on an operation with a local scope

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 rest of the program `x -> m a`, in the following called continuation. The continuation maps from the intermediate result, produced by the programs in scope, to the rest of the computation. The three programs agree on an arbitrary but per `catch` fixed type `x` as well as on syntax, defined by `m`.

For this generalized syntax the behaviour of ≫= is changed slightly. Calling ≫= on a higher order term just extends the continuation, i.e. just substitutes the variables in terms of type `m a`. Figure 4.1 demonstrates the action of ≫= on catch. The program in scope and the e indexed subprogram (the exception handler) both produce values of the captured type X, therefore ≫= does not affect them. Only the third computation produces values of type A and therefore is affected by ≫=. In terms of operation, the programs in scope are not modified and just the actual rest of the program, described by the continuation, is affected. Therefore, by generalizing the syntax Wu et al. [34] allow a direct representation of operations with scopes. Before we can implement the infrastructure, we have to choose an appropriate representation for higher-order functors in Agda.

4.1.1 Representing Strictly Positive Higher Order Functors

In Haskell the generalization to higher order syntax is straight forward. Wu et al. [34] switch to the new representation and introduce type constructor classes to define higher-order functors as well as the notion of syntax in this implementation of scoped effects.

In Agda we again have to find an appropriate representation for higher-order functors 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) : Set₂ where
  field
    Shape : O → Set₁
    Pos : ∀ {o} → Shape o → Set
    Ctx : ∀ {o} → (s : Shape o) → Pos s → I

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

Similar to the Haskell data type `HExc e m a` we have to store an arbitrary type, when defining scoping operations. Therefore, the higher-order functor has to produce elements of the larger universe Set_1. Instantiating I and O with Set gives rise to a container extension of the correct shape.

Bunkenburg's approach also uses indexed containers [9]. 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
```

[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 definitions.

```
⟦_⟧ : (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 type X and the argument type A i.e. programs inside or outside the operations scope.

Bunkenburg [9] notes that it is not possible to generically define $\gg=$ for the resulting monad using the above representation. Similar to the Haskell implementation by Wu et al. [34] we could define a type constructor class to implement `emap :: (Monad m) => (m a -> m b) -> (sig m a -> sig m b)` for all valid higher order functors, which would allow the definition $\gg=$. Both representations share another problem. The produced type constructor is not necessarily strictly positive, because Ctx can produce arbitrary types containing A. 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 cases 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 type returned by Ctx 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 [9] fixes 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, that is strictly positive in both arguments. Bunkenburg extends the usual bi-functor to an indexed bi-functor by adding a context, similar to the above definition, to one of the positions. The definition of an indexed bi-container and its extension is given below.

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

An indexed bi-container has two sets of positions for each shape, Pos and PosX. 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. The definition is similar to the second indexed container definition, except that the context has no access to the given type A. Therefore, the first position function always produces values of type $F\ A$, and the second position function always produces values of type $F\ X$, where X does not contain A. Later they produce the continuation and the programs in scope respectively. In Agda this definition is recognized as strictly positive in A. The analogous definition in Bunkenburg's Coq implementation is not [9].

Ultimately the choice of representation for the functor does not matter because all approaches based on storing a type directly encounter the problem described in Section 4.1.2. Going forward we will use Bunkenburg's indexed bi-container for examples because they are easiest to work with and allow us to continue the approach. As a side note, some operations with more complex syntax, like `pass :: m (a, w -> w) -> m a` and `listen :: m a -> m (a, w)` for the `Writer` monad, cannot be implemented with this representation.

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

Using our choice of functor representation we can define the Prog monad for higher order syntax as described by Wu et al. [34]. It is analogous to the Haskell definition above.

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

Notice that due to the potentially captured type the Shape is an element of Set_1. Therefore, Prog^W is also an element of Set_1. Because Prog^W 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^W's, it is not possible to work with effectful data as in Haskell. Section 4.1.2 contains a detailed explanation of the problem.

4.1.2 Effectful Data and Existential Types

A central problem with this approach is that it cannot model deep effects without involving variable levels. Consider the following definition of an effectful list.

```
data List^M (E : Effect) (A : Set₁) : Set₁ where
  nil  : List^M E A
  cons : Prog^W E A → Prog^W E (List^M E A) → List^M E A
```

In contrast to the Coq version this definition is valid, but it cannot be used as expected with Prog^W. $\mathsf{List}^\mathsf{M}\ C\ A$ 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 and therefore values of the Shape type. The Shape type is an element Set_1 because it has to store the result type of the subcomputation $X : \mathsf{Set}$. Therefore, $\mathsf{List}^\mathsf{M}\ C\ A$ 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. Calling a scoped operation on an effectful data structure containing computations with level ℓ produces a computation on level $\ell+1$. This leads to different levels across the program, which are difficult to unify because they require manual lifting of types. Similar to Size, Level is a builtin type with a limited set of associated functions. 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 lead in our implementations to an impractical library interface. We were not able to solve the problem of data dependent levels in recursive data structures.

4.1.3 Indexing and Integrating Data

To solve the problem from Section 4.1.2 we explore two approaches. For an effect system it seems reasonable to not let universe level vary or in the best case not increase it, that is, either make Prog^W an element of Set or represent effectful data in Set. In this section we mention two ideas, one for each goal. Unfortunately, in our tests non of them lead to a concise solution.

Indexing We could try to solve the problem of increasing universe levels by not storing the intermediate result types directly. Instead, we index the program over a data type storing all intermediate result types, that is, we introduce a typing context. The level of the index is independent of the level of the structure. Therefore, the chain of reasoning from Section 4.1.2 does not

apply and we can decrease the universe level of Prog. The typing context needs to have the same structure as the program. This is possible by reusing the arbitrary branching mechanism, which is known from the free monad.

A central problem with this approach is that every call of $\gg\!=$ potentially grows the tree described by Prog and therefore the context. These changes now have to be reflected on the type level because the context is embedded in the type of Prog If $\gg\!=$ calls different operations for different values, leafs in the context are substituted with different new contexts. As with increasing universe levels, these complications are increasingly hard to deal with and are hard to hide from the user of the library. For example, in our tests the changes to the context required the definition of helper functions that describe the modification. This lead to an impractical interface.

Integrating Data and Computations We cannot model deep effects because effectful data structures are an element of Set_1, but data is required to be an element of Set. But, these special data structures are only too large because they contain values of the computation type Prog itself, not arbitrary values from Set_1. Conceptually these data structures continue the computation tree. In the above definitions this fact is unused. Instead, we try to store arbitrary elements of Set_1, i.e. we over approximate the class of types. We tried using this fact by unifying data and computations. For example, the following data type for computations can also store effectful tuples.

```
data Prog^D (Effs : Effect) : Set → Set₁ where
  var     : A → Prog^D Effs A
  impure  : ⟦ Effs ⟧ (Prog^D Effs) A → Prog^D Effs A
  _,_     : ⟦ Effs ⟧ (Prog^D Effs) A → ⟦ Effs ⟧ (Prog^D Effs) B → Prog^D Effs (A × B)
```

By representing data structures using a small type and creating their actual structure using a new Prog constructor it is possible to avoid the size issues. We tried implementing this idea but failed when trying to represent generic, recursive structures. Furthermore, working with data described this way is unintuitive.

Following the same idea, i.e. storing just representations of effectful data, one could try to store either type variables or data type representation (e.g. polynomial functors) in the Shape of effects and build an effectful representation in the container extension.

4.2 Limited Implementation

Translating the approach by Wu et al. [34] without major changes to Agda makes working with deep effects difficult. However, it can still be used to implement scoped effects without deep effects. For completeness, the following section will show key points of a simplified implementation using this approach. To write concise code we 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]. Furthermore, we do not define a level polymorphic version of Prog as its only advantage is a limited representation of deep effects. Instead, we limit value types to elements of Set, i.e. those which can always be scoped.

First we port over the infrastructure for modular effects. The coproduct carries over seamlessly. We combine the position functions pairwise and handle the Ctx functions using the coproduct mediator. Void is also defined analogous to the first order setting.

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

Void : Effect
Void = ⊥ ◁ (λ()) + (λ()) / λ()
```

To easily work in the higher order setting we adapt the Prog data type similarly to the 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 to avoid the `--cumulativity` flag we work with a version of

[4] To guarantied strict positivity one of the solutions mentioned in Section 4.1.1 is needed.

Prog with a Set value type[5]. We try to avoid the flag in this exemplary implementation because it weakens type inference, forcing us to write more implicit parameters explicitly.

```
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. [34]. In contrast to the implementation by Wu et al. [34] the function can be defined generically for all effects because the indexed bi-container representation is more restrictive than their Haskell representation.

```
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 of $\gg\!=$ is identical to the one in Haskell. Using the INLINE pragma and projections, as explained in Section 2.1.6, we can define emap separately without obscuring the termination. Note that emap only applies the given function to the result of the first position function because only it has access to A. The programs in scope are ignored. Therefore, the recursive calls to $\gg\!=$ also only affect to the continuation.

The infrastructure from Chapter 3 carries over with minimal changes. We reuse the $\_\in\_$ proposition. 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, which is represented by a functor (e.g. List for Nondet).

Following the approach by Wu et al. we define the Syntax type class for effects. Given a handler, the type class explains how to move handlers and evaluation contexts for other effects through syntax of the given effect. A handler is given by a function that transforms a program in a context to a new program producing values inside the context, i.e. in Haskell a function of type `c (Prog sig a) -> Prog sig' (c a)`. We first have to define contexts. The normal definition of a functor does not suffice because it lifts all types to the same Set_n. Because we have to lift programs as well as data, this means Set_1. Later, we will have to lift the intermediate result type of scoped operations in the context. This is problematic, because scoped operations can only capture elements of Set. Therefore, the context should preserve the universe level of its argument, i.e. lift values to types in Set and programs to types in Set_1. Even with `--cumulativity` this behavior is not handled automatically by Agda[6].

To simulate the above behaviour we define the Context type constructor class. A context is essentially a functor that respects the universes structure. Given a type from Set_n, it lifts it to another type in the same Set_n. Furthermore, a function between two data types in different universes is lifted to a function the lifted types, i.e. between universes. An instance should satisfy the functors laws even though they are not enforced here. Notice that Context quantifies over arbitrary Levels ℓ and therefore is an element of $\mathsf{Set}\omega$, the universe which is larger than all natural number indexed universes. $\mathsf{Set}\omega$ represents the end of Agda's universe hierarchy. There

[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 is easier in a universe polymorphic setting.

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

are no universes indexed by larger ordinals and therefore to preserve consistency there are some restrictions on terms of this type. In our case this should not be a problem because we only ever parameterize over contexts.

```
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 programs in scope and continuations explicitly. The order in which they are evaluated and therefore how an evaluation context should be moved through the program is a semantic question. The type class explains how an arbitrary handler can be threaded through an effect's syntax using the function handle. 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 program in a context and interprets parts of the syntax. The resulting program has a different effect stack, usually the tail of the given stack. By partially evaluating the program the context is moved through the program. As a result, the values are now lifted into the context. Using the initial context and the handler, the type class explains how an arbitrary operation from the signature E is traversed.

Notice that the given operations contain program of an arbitrary size. Furthermore, the given handler works on programs of the same size. This will be important in Section 4.2 when we define handlers which pass them self to handle.

```
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 {{...}}
```

To finish the basic infrastructure we define two Syntax instances. The instance for the empty signature Void is trivial because the type of container extensions is empty. Combined signatures are handled by inspecting the given operation, calling the handle for its signatures and injecting the result in the combined signature.

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

  ⊕-syntax : {{ Syntax E }} → {{ Syntax E′ }} → Syntax (E ⊕ E′)
  Syntax.handle ⊕-syntax ctx hdl (inj₁ s , κ , σ) with handle ctx hdl (s , κ , σ)
  ... | s′ , κ′ , σ′ = (inj₁ s′) , κ′ , σ′
  Syntax.handle ⊕-syntax ctx hdl (inj₂ s , κ , σ) with handle ctx hdl (s , κ , σ)
  ... | s′ , κ′ , σ′ = (inj₂ s′) , κ′ , σ′
```

4.2.1 Exceptions

As an example for an effect in the higher order setting we implement exceptions.

Syntax Exceptions support two operations, the usual algebraic operation throw as well as the scoped operation catch. The two operations are identical to the Haskell definition from Section 4.1. throw takes the thrown exception as an additional parameter. catch has no additional parameters, but its Shape stores the result type of the computations in scope.

```
data Excˢ (E : Set) : Set₁ where
  throwˢ : (e : E)   → Excˢ E
  catchˢ : (X : Set) → Excˢ E
```

The extension of an indexed bi-container is equipped with two positions functions, one for programs resulting in Prog *effs* A and one for programs in scope, i.e. such resulting in programs of type Prog *effs* Y where Y is defined by the context. Let us first consider the positions for programs in scope.

```
data ExcCatchᴾ (E X : Set) : Set where
  mainᴾ   : ExcCatchᴾ E X
  handleᴾ : (e : E) → ExcCatchᴾ E X

Excᵖˣ : ∀ E → Excˢ E → Set
Excᵖˣ E (throwˢ e) = ⊥
Excᵖˣ E (catchˢ X) = ExcCatchᴾ E X

Excᶜ : ∀ E s → Excᵖˣ E s → Set
Excᶜ E (catchˢ X) mainᴾ       = X
Excᶜ E (catchˢ X) (handleᴾ e) = X
```

throw has no programs in scope and therefore no positions. catch's positions are given by ExcCatchᴾ. The operation has two kinds of programs in scope. The first single program is given by mainᴾ, the program whose exceptions should be caught. The program in the handleᴾ position provides a continuation for each exception. It corresponds to the exception handler. Both programs produce results of type X. The result types for the programs in scope are defined using the context Excᶜ.

Next we define the positions for the effect's continuation, i.e the programs producing values of type A.

```
Excᵖ : ∀ E → Excˢ E → Set
Excᵖ E (throwˢ e) = ⊥
Excᵖ E (catchˢ X) = X
```

Again, throw has no continuations because it produces no results. catch provides a continuation that processes the result of the program in scope, i.e. it provides a continuation for each value of type 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 = Excˢ E ◁ Excᵖ E + Excᵖˣ E / Excᶜ E

pattern Throw e     = impure (inj₁ (throwˢ e) , _ , _)
pattern Catch κ σ   = impure (inj₁ (catchˢ _) , κ , σ)
pattern Other s κ σ = impure (inj₂ s          , κ , σ)
```

Lastly we define smart constructors for the operations. Because throw has no continuations, we define the container extensions as $\lambda()$. catch captures a program p as well as the exception handler in scope. Therefore, they are returned by σ. The effect's 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 }
```

Instances 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 values that are constructed using f could be elements of another Set. Because E ⊎_ is polymorphic in the level of its argument type, this is not a problem.

```
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. Therefore, the instance for Exc is not needed for the Exc handler. Nevertheless, we define a Syntax instance for Exc next 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 cx → hdl (κ <$> cx )) , λ where
  mainᴾ → hdl (σ mainᴾ <$ ctx )
  (handleᴾ e) → hdl (σ (handleᴾ e) <$ ctx )
```

The case for throwˢ is trivial because the operation has no continuations on which to evaluate the handler. The case for catchˢ is more complex.

The idea behind the Syntax type class is to thread the handler through the operations, i.e. to evaluate the syntax of another effect in all subprograms while the context moves through the computation. 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. Moving the context correctly through the syntax is a semantic problem and therefore has to be defined on a case by case basis. Therefore, we have to consider how catch is evaluated. The catch operation is evaluated by evaluating the program in the mainᴾ position first. In case of an exception we produce an alternative result using the program in the handleᴾ position. At last, we feed the result in the continuation. Handlers for other effects should traverse the effect similarly. Therefore, the intermediate result is lifted into the handler's context.

The effect's continuation κ now takes an argument of type $C\ X$, i.e. the result of a program in scope inside the foreign handler's context. By mapping the given κ over the given cx the continuation is applied to the argument of type X and lifted inside its context. By applying hdl we partially evaluate the syntax and move the context through the continuation of the operation. For example, in case of threading Nondet through exception syntax the context is a list. 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 programs in scope, i.e. the programs produced by σ. To correctly thread its state through the syntax of other operations the handler requires that all computations are lifted into the context. In case of κ we were given an argument in a context. Because the programs in scope are evaluated first, they are lifted into the initial context. The given value cxt holds a value of type $\top$, i.e. no information just handler state. Using <$ we can inject any program into it. Injecting a program into ctx and calling hdl should be the same as calling the actual handler for the effect on the computation.

The three cases are identical to the ones presented by Wu et al. [34]. After evaluating the handler on the sub-trees, values are lifted into the context. Therefore, the captured intermediate result type changes. In Agda we explicitly store the type. Therefore, in contrast to the Haskell implementation by Wu et al. [34], 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. Furthermore, injecting a program into a context is only possible because we defined special versions of the functor operators that respect the universe structure. If we had used a normal functor we either could not have stored the lifted intermediate result type or could not have injected results of κ and σ into the context.

Handler Lastly we define the handler for Exc. Notice that the handler takes a program of an arbitrary size i. The size 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 program in the mainᴾ position. Using >>= we can inspect the result and either call the rest of the program 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 mainᴾ) >>= λ where
  (inj₁ e) → runExc (σ (handleᴾ 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 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 program in the context. We handle the two cases using [_,_]. If the argument is already an exception, we simply produce a pure program which produces the exception. Otherwise, it is a program that 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 programs 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 traversed operation. Therefore, the argument passed to the coproduct mediator that calls runExc is smaller than the currently handled value.

Example Using the Exc effect we can define simple programs that 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, catches the exception and returns 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
```

4.2.2 Lifting First Order Syntax

Wu et al. [34] present a construction for lifting first order syntax to the higher order setting. A similar construction is possible with the container representations. In this section we present the corresponding construction and use it to lift nondeterminism syntax into the higher order setting.

Given the two fields of a normal container we construct an indexed bi-container. The shapes and positions for A correspond exactly to the first two arguments for the indexed bi-container. The second position type and the context are precisely the generalization from the first-order to the higher-order setting. Therefore, we simply use $\bot$ to ignore them.

We did not use the cumulativity flag to simplify this exemplary implementation. This simplified the earlier implementation, but forces us now to do some manual lifting. The Lift data type is element of an arbitrary universe and stores a value of an arbitrary type from a smaller universe. Using Lift we can pass the given shape type to the constructor. To apply the position function we unwrap the lifted shape.

```
_▷_ : (Shape : Set) → (Pos : Shape → Set) → Effect
S ▷ P = Lift _ S ◁ (λ (lift s) → P s) + (λ _ → ⊥) / λ _ ()
```

As shown by Wu et al. [34], we can define a general Syntax instance for lifted first order syntax. The second type of positions is empty. Therefore, we can omit the definition of σ. Notice that this definition applies to all effects constructed using $\triangleright$. Therefore, we avoid defining parts of the infrastructure for effects without scoping operations.

```
instance
  ▷-Syntax : ∀ {S P} → Syntax (S ▷ P)
  Syntax.handle ▷-Syntax ctx hdl (s , κ , σ) = s , (λ p → hdl (κ p <$ ctx)) , λ()
```

Example: Nondet Using the above function and instance we can lift the nondeterminism syntax from Chapter 3 to the higher-order setting.

```
Nondet : Effect
Nondet = Nondet^s ▷ λ where fail^s → ⊥ ; ??^s → Bool

pattern Fail      = impure (inj₁ (lift fail^s) , _ , _)
pattern Choice κ = impure (inj₁ (lift ??^s)  , κ , _)
```

We still have to define a Context instance and a custom handler for Nondet because we have to define the interaction with other higher order syntax. Again, the Context instance corresponds to the normal functor instance.

```
instance
  List-Context : Context List
  Context._<$>_ List-Context f []       = []
  Context._<$>_ List-Context f (x :: xs) = f x :: (f <$> xs)
```

The first three cases of the handler are identical to the implementation in Section 3.2.1. The interesting new case is the one for Other syntax. We must use handle to traverse the arbitrary operation. Analogous to exceptions we evaluate handle on the reconstructed given operation. The initial context has to be singleton list, because empty lists would stop the computation, while lists with two or more elements would duplicate parts of the program.

hdl has to map from a List of program with Nondet syntax to a program without Nondet syntax and a List as value type. Consider the Syntax instance for Exc, given in Section 4.2.1. The two program in scope use the initial context, i.e. the given list is a singleton. The continuation uses <$> to apply the rest of the program to the given results of the computation in scope and evaluates them using hdl. Conceptually, the argument of hdl corresponds to the programs that produce results for each result of some earlier computation. We therefore use the obvious (and usually given) implementation of the hdl function i.e. we sequence the computations from left to right and concatenate the results.

```
runNondet : ⦃ Syntax (sum effs) ⦄ → Prog (Nondet :: effs) A {i} → Prog effs (List A)
runNondet (pure x)       = pure (x :: [])
runNondet Fail           = pure []
runNondet (Choice κ)     = ⦇ runNondet (κ true) ++ runNondet (κ false) ⦈
runNondet (Other s κ σ) = impure (handle (tt :: []) hdl (s , κ , σ))
  where hdl : ⦃ Syntax (sum effs) ⦄ →
              List (Prog (Nondet :: effs) B {i}) → Prog effs (List B)
        hdl []          = pure []
        hdl (mx :: mxs) = ⦇ runNondet mx ++ hdl mxs ⦈
```

Notice that the argument of the handler as well as the hdl function are of size i to guaranty termination, following the pattern explained in Section 4.2.1.

```
_??_ : ⦃ Nondet ∈ effs ⦄ → Prog effs A → Prog effs A → Prog effs A
p ?? q = op (lift ??^s , (if_then p else q) , λ())
```

Ordering of Operations It is important to note that the above handler induces unexpected semantics when combined with certain other effects. Interpreting Nondet syntax yields the results in depth first ordering.

If a part of a nondeterministic program is captured in a scope, the order of operations changes. The program in scope is evaluated, yielding its results in depth first ordering. Afterwards, using hdl the continuation is evaluated on each intermediate result and the final results are concatenated. The results are in the expected depth first order, but the operations not. The order of operations is introduced by $\gg=$. A scoping operation is evaluated by first evaluating the part of the computation in scope and afterwards the continuations outside the scope. Therefore, the unevaluated operations are now also in this order. The potential unexpected semantics of this implementation is that introducing a semantically irrelevant scoping operation (e.g. `local id`) influences the ordering of operations of other globally evaluated effects. This behaviour is known, because it also occurs in some Haskell implementations of scoped effects[7].

Encountering the above behaviour is quite uncommon, because the program has to evaluate a scoping operation globally over a nondeterministic program, while simultaneously using global operations whose ordering can be observed. Most effects only provide one of the two. The most common examples are global state to collect results, global writer to log and global partiality to stop all branches, which do not encounter the above behavior. When catching global exceptions over a nondeterministic program, the above behaviour seems more reasonable. More complex semantics (e.g. leaving and re-entering the scoped computation) are conceivable, but not captured by the standard representation of higher order syntax.

4.2.3 Modular Higher Order Effects

In the last to sections we presented effects in isolation. Similar to the combined semantics described in Section 3.2.3 we can induce different semantics by reordering the handlers. Consider the following program, which chooses between a pure value and an exception.

```
throwNondet : ⦃ Nondet ∈ effs ⦄ → ⦃ Exc ⊤ ∈ effs ⦄ → Prog effs ℕ
throwNondet = pure 42 ⁇ throw tt
```

Evaluating Exc first runs it locally in each branch of the nondeterministic computation i.e. each branch either returns a result or an exception.

```
run (runNondet (runExc throwNondet)) ≡ inj₂ 42 ∷ inj₁ tt ∷ []
```

Evaluating Exc last evaluates it globally. Therefore, throwing an exception stops the current, by the Nondet handler induced, ordered evaluation of nondeterministic branches and just returns the exception.

```
run (runExc (runNondet throwNondet)) ≡ inj₁ tt
```

4.3 Results

In this chapter, we tired to implement scoped effects a higher order syntax representation approach by Wu et al. [34]. We continued Bunkenburg's [9] investigation of this approach and uncovered a deep-seated problem while implementing it in Agda, which arises due to more restrictive requirements for programs compared to Haskell. The problem occurs because we have to work in a consistent hierarchy of universes to preserve Agda's consistency. When using higher order syntax, computations potentially have to store the type of data they produce and therefore are larger than the data. This prevents scoping operations from capturing other computations without switching to a larger type of computations. This specifically includes effectful data structures, which are needed for the implementation of deep effects. In case of deep sharing this leads to a dependence between universe levels and the depth of data structures. We discussed two general approaches to avoid increasing universe levels when working with effectful data. However, both of them require a more complex representation for either types or data, probably leading to a more complex implementations that is harder to work with in real programs.

[7] https://github.com/polysemy-research/polysemy/issues/246

Because deep effects are an uncommon requirement, we still presented key points of a limited implementation. Although, we removed deep effects as a requirement, we still had to work with different universe levels, while implementing the Syntax instances. This lead to manual lifting between universes and structures involving Setω, which should generally be avoided. A level polymorphic implementation, needed for increasing the size of the computation types, uses both constructs more frequently, but can be used to work with effectful data to a limited extend.

Chapter 5

Implementing Scoped Effects using 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, that does not rely on existential types. Piróg et al. [23] present a novel formalization for scoped operations and their algebras, that fulfils this requirement. In their paper they do 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 Haskell by Piróg et al. [23] to Agda. To implement scoped algebras in an obviously terminating way, we derive a different implementation of the fold by Piróg et al. [23], based on the work by Fu and Selinger [13]. To test our implementation, we implement the nondeterminism example by Piróg et al. [23], exceptions, state and sharing as defined by Bunkenburg [9]. Furthermore, we present a new implementation for naive modularisation of handlers.

5.1 The Monad E

Piróg et al. [23] describe a monad, in their paper called E, that is suited for modelling operations with scopes. 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.

In the following we describe the original and the resulting monad in Agda. Piróg et al. [23] separate the signature into one describing effects with scopes Γ and one describing the algebraic effects Σ. Limiting our self to just strictly positive functors we can define the following data type in Agda.

```
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
```

This data type looks appropriate considering our earlier definition in Chapter 4. An element of the monad is one of the following three. A value of type A, construed using var. An operation without scopes, that is the functor Σ applied to the monad itself. Notice that this case corresponds to the impure constructor in the free monad. Or lastly, an operation with a local scope, represented by the second signature Γ. Operations with a local scope always quantifies over some type X, which is the type for the intermediate result. Furthermore, they provided a continuation, which transforms values of the internal result type to the result type of the whole program. The operations with scopes from Chapter 4 followed exactly this pattern. The shape of the by bi-container stored the intermediate result type and the two position functions provided continuations resulting in programs with value type A or X.

Notice that this definition also requires X to be a smaller type than ProgE, i.e. it is not sufficient to model deep effects as explained in Chapter 4. Piróg et al. [23] also note size issues with this definition. Piróg et al. [23] rewrite this monad using a left Kan extension and derive the following equivalent monad without the existential type.

```
data ProgE′ (Σ Γ : Container) (A : Set) : Set where
  var : A                                    → ProgE′ Σ Γ A
  op  : ⟦ Σ ⟧ (ProgE′ Σ Γ A)                 → ProgE′ Σ Γ A
  scp : ⟦ Γ ⟧ (ProgE′ Σ Γ (ProgE′ Σ Γ A))    → ProgE′ Σ Γ A
```

An equivalent monad without the existential type is a promising candidate for modelling deep effects in Agda because it avoids the size issues. Compared to the higher order approach, by using this monad we lose explicit control over the parts of the program, which is important for potential optimizations like memorization.

The monad models scopes using the double ProgE′ 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 computation, the rest of the program producing a result of type A. This resulting program corresponds to the continuation in the higher order approach, which is a function mapping from the result of the scoped program to the rest of the whole program. In their paper, Piróg et al. [23] develop a theory of *scoped algebras* for a second equivalent monad and us them to define a sensible evaluation for this monad. In Section 5.2, we define a more practical version of the monad and translate part their practical results form Haskell to Agda.

5.2 The Prog Monad

In this section we define the central monad for this approach. Because effects are split into signatures with and without scopes and most computational effect have both, we define effects as pairs of signatures 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. map$^{\mathsf{L}}$ corresponds to the Haskell function `map` for lists.

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

Using these helper functions we define a modular version of the monad by Piróg et al. [23]. It is equivalent to the fixpoint and the Haskell definition from the paper (assuming that signatures are given by functors that can be represented as container).

```
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. [23] define a new kind of algebra for operations with scopes, i.e. a recursion scheme for the above monad. To implement their recursion scheme we have to define <$> for Prog *effs* A. Notice that in the scp constructor the type variable A is instantiated with the type Prog *effs* A itself. Such a data type is called a truly nested or non-regular data type [7].

Defining recursive functions for these data types is more complicated. The "direct" implementations of <$> and >>= for Prog *effs* A do not obviously terminate. Augmenting Prog *effs* A with size annotations leads to problems while proving the monad laws[1]. Instead, we define a generic fold, which we reuse for all functions traversing values of type 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 [13] demonstrate a general construction for `folds` for nested data types in Agda. 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.

[1] As in Chapter 4 the monad laws only hold for size ∞, but working with size ∞ in proofs leads to termination problems. Repeated occurrences of 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.

Index Type The first part of the construction by Fu and Selinger [13] 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 tree with potentially different branch and leaf constructors. 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.

Afterwards, Fu and Selinger [13] define a type level function, that translates a value of the index type to the type it describes. 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)
```

Recursion Scheme Next we define fold for Prog *effs* A. The fold produces value of an arbitrary $\mathbb{N}$ indexed type. 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 process with recursive occurrences of the type itself already processed. In contrast to a normal recursion scheme, these recursive occurrences respect the index. For example, the function processing values constructed using scp takes an argument of type $P\,(2+n)$ because the constructor takes a value of type Prog *effs* (Prog *effs* A). The cases are obtained by pattern matching on the index type and the value of type (Prog *effs* ^ n) A, i.e. the value of the type depending on the index.

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

Notices that for the op and spc constructors the recursive occurrences and potential additional values are determined by the Container. Therefore, the functions processes arbitrary Container extensions, which contain solutions for the corresponding $P\,n$. When implementing those two cases we cannot apply fold directly to recursive occurrences because those depend on the choice of Container. Because container extensions are functors, we map over the given value.

The fold, based on the approach by Fu and Selinger [13], is similar to the one by Piróg et al. [23]. They derive their fold by constructing algebras for a second monad. Then Piróg et al. [23] show that the two monads are equivalent and transfer the fold from the second monad to the one used in their Haskell implementation. Following Fu and Selinger [13] we derived essentially the same fold and implemented it directly, without using $\gg\!=$.

Example Using fold we can implement $\gg\!=$ for Prog *effs* A. The continuations k passed to $\gg\!=$ should just affect the innermost layer. Intuitively, when using bind 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 given program. 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 for each 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 var leafs with subtrees generated from their stored 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 constructors at layer zero have to produce values 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 instance 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 }
```

In contrast to Chapter 3 this is not a problem because we do not have the meta problem of preserving sizes. The record is part of the Agda standard library. It is named RawMonad because it does not enforce the monad laws. We 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 [13] we generalize the fold from Section 5.2.1 to a dependently typed version, an induction principle. In the induction principle the proposition P is generalized to a dependent type i.e. a relation on values of type Prog *effs* A. Therefore, the induction principle allows for proofs of these predicates by induction without the use of 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. We bind it to x and pass it to the proposition.

```
( (x : A) → P 0 x ) →
```

The var case can be handled similar to the examples by Fu and Selinger [13]. We assume values for all constructor arguments, as well as proofs for the smaller cases. For var an additional, hidden argument x is introduced. x represents a recursive occurrence, therefore it is part of the type of the proof for the smaller case $P\ n\ x$. x is used to describe the currently handled value var x. Therefore, var x is part of the functions result type. Under the Curry Howard corresponds this function (ignoring the index type) can be read as the proposition $\forall x.P(x) \to P(\text{var } x)$ 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 container's position function maps from a type of positions,

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

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 a function from positions (for the assumed shape) to proofs for the proposition for the values contained there. The value for each position is obtained using the assumed position function κ. The currently handled value is constructed using s and κ.

```
(∀ {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 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)
```

5.2.3 Proving the Monad Laws

To demonstrate the use of the induction principle from Section 5.2.2 and to verify that Prog *effs* A is a monad we prove the monad laws.

Left Identity First we prove the left identity law. Because the value passed $\gg\!=$ is constructed using return i.e. var, both sides of the equality evaluate to $k\ a$.

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

Right Identity Next we prove the right identity law. 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 prove the proposition by induction on ma. To use the induction principle 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 prove 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 principle 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 prove that the proposition holds for values constructed using var we case split on the number of layers n. For 0 we prove 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 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 on both sides. Therefore, we prove 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 shape. We have to prove that the proposition holds for a new program constructed using either op or scp. For both constructors bind n calls itself recursively 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 prove the equality point wise i.e. 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. Therefore, we 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 $\gg\!=$ using the above laws. These proofs are simpler than the ones explicitly involving the definition of $\gg\!=$ using fold. We can write them using the chain reasoning operators [22].

We just prove a single law to demonstrate the general structure. By defining the monad instance the applicative and functor operators are defined in terms of $\gg\!=$. Therefore, we replace $<\$>$ with its definition. By simplifying the term we can apply the right identity law for monads, which yields the correct result.

```
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                         ■
```

Proofs for the other laws can be found in the repository. They are used to prove properties of effect handlers.

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, one representing scoped and one representing algebraic 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 operation from one of the two signatures is part of the corresponding signature of the combined effect.

```
opsInj : e ∈ effs → Ops e ∈ map^L Ops effs
opsInj here          = here
opsInj (there ⦃ p ⦄) = there ⦃ opsInj p ⦄

scpsInj : e ∈ effs → Scps e ∈ map^L Scps effs
scpsInj here          = here
scpsInj (there ⦃ p ⦄) = there ⦃ scpsInj p ⦄
```

Using the above proofs we can implement smart constructors for scoped and algebraic operations without requiring additional 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 escape the monad after interpreting all effects we again add the handler for the Void effect.

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

5.4 Nondeterministic Choice

As first example, we implement the nondeterminism effect. In Section 5.4.1, we implement the example by Piróg et al. [23] to demonstrate that the fold definition from Section 5.2.1 is sufficient. Section 5.4.2 presents an idea for modular effects and handlers.

5.4.1 Implementation as Scoped Algebra

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

```
data Choice^s : Set where ⁇^s : (Maybe CID) → Choice^s ; fail^s : Choice^s
data Once^s   : Set where once^s : Once^s
```

Nondet^p corresponds to the data type from the example by Piróg et al. [23, sec. 6]. The effect has the unary, scoped operation once and the nullary and binary, algebraic operations ⁇ and fail.

```
Nondet^p : Effect
Ops  Nondet^p = Choice^s ▷ λ{ (⁇^s _) → Bool ; fail^s → ⊥ }
Scps Nondet^p = Once^s   ⇝ ⊤
```

A scoped algebra for an ℕ indexed carrier type C consists of three operations. a the *algebra* for the algebraic operations. p for *promoting* the carrier type when entering a scope. d for *demoting* the carrier type when leaving a scope and interpreting the scoped operation. Figure 5.4.1 demonstrates how the handler interprets Nondet syntax using the scoped algebra. The carrier type are n-fold lists. The example is taken from Piróg et al. [23], but notated using Prog *effs* A. **var** nodes are handled using p. The value is promoted by lifting it into a singleton list. The algebraic operations **fail** and **?** are handled using a. Notice that in the definition of fold they are guaranteed an index $n \geqslant 1$ i.e. at least one list layer. They are implement identical to earlier versions of Nondet. **?** concatenates the left and right list. In the third step the lists are promoted again because they enter the scope created by **once**. This is key because for algebraic operations in the scope the multiple results from outside the scope appear as a single value. **once** is evaluated using d, removing the scope. **once** keeps just the first result inside the scope i.e. the list containing the results of the continuation applied to the result of **once** (**var** 1 ? **var** 3) ≡ [1].

The indices for d and a are offset by one compared to the definition by Piróg et al. We do not 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 two recursion schemes.

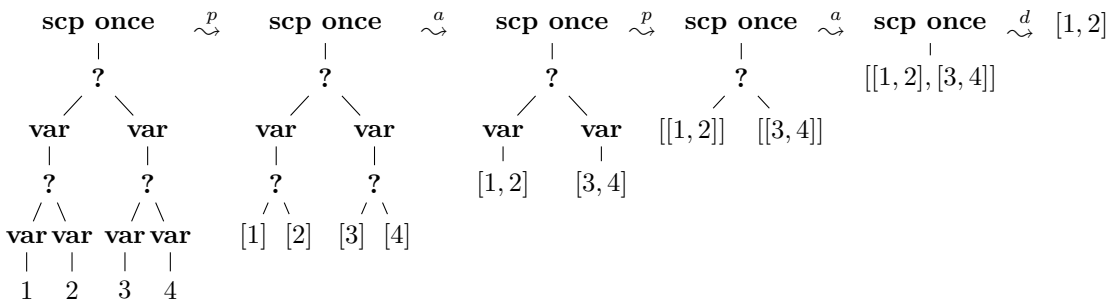

Figure 5.1: Interpretation of **once** (**var** 1 ? **var** 3) $\gg\!=$ $\lambda x.$**var** x ? **var** $(x + 1)$

```
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 interpret its syntax. fold$^{\mathsf{P}}$ corresponds to a slightly modified version of the fold by Piróg et al. [23]. Their function works for an arbitrary n because it is defined recursively. In this implementation the recursion is abstracted in fold, therefore 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 are not preprocessed and just get promoted, hence id is passed as the first of the four arguments.

```
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. [23] 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. once is implemented by taking the head of the list if possible i.e. the results of the continuation applied to the first result of the program in scope.

```
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) = case κ tt of λ where
  []      → []
  (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 is possible to differentiate between the state of the computation in scope and the whole computation.

Lastly we define a smart constructor for the once operation. To capture the program p in scope, pure is mapped over it. The original program is now the outer Prog *effs* layer i.e. the first and second layers of the first term in Figure 5.4.1. The original var x leafs are now var (var x) leafs. Because $\gg=$ just affects the lowest layer, it does not change the captured program and just act on its results.

```
once^P : {[ Nondet^P ∈ effs ]} → Prog effs A → Prog effs A
once^P p = Scp (once^S , λ _ → pure <$> p)
```

5.4.2 Implementation as Modular Handler

Piróg et al. [23] just present handlers for single effects and make some remarks regarding modularity in a theoretical context for their equivalent monad. We work in a more practical context, i.e. we are just concerned with a small number of possible effects and can therefore impose more restrictions on them. This section presents a naive idea for adapting the handlers from the paper to modular ones by following patterns from earlier chapters. As before, we will be working with the Prog monad, i.e. the one used by Piróg et al. [23] to implement effects, not the one used to define scoped algebras.

As in the earlier chapters, when working with modular effects, the general approach is to execute the handlers for each effect one after another [28]. 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 a 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. Because the results of all branches are evaluated and just some are kept, just the result and not other operations in the branch are pruned. It is still possible to implement once using a version of cut and call. Similar to earlier chapters we introduce patterns to simplify the handler.

```
Nondet : Effect
Ops  Nondet = Choice^S ▷ λ{ (⁇^S _) → Bool ; fail^S → ⊥ }
Scps Nondet = Void

pattern Other s κ     = (inj₂ s , κ)
pattern Choice cid κ = (inj₁ (⁇^S cid) , κ)
pattern Fail          = (inj₁ fail^S , _)
```

The new handler has the expected signature and uses the carrier type described above.

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

First we have to handle the var case. 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 the 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. By interleaving binds it orders the results. On a term level, we are given a tree of possible continuations. We execute them from left to right and store their results.

Similar to the handler from Section 4.2.2, this implementation orders the results correctly, but not the operations of other effects, which entails possible unexpected interaction semantics.

```
  ) 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 type classes 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 seem 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. It is also unclear if a monad or just a notion of flattening, or even additional coherence conditions are needed.

Notice that hdl is similar to the handler function from Chapter 4. This is partially expected because, as explained in Section 5.1, the Prog monad is equivalent to another monad which is similar to the one from Chapter 4. Because the fold already interpreted parts of the program, we do not call the handler recursively and 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 (fail^s , 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 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 Throw^s (E : Set) : Set where throw^s : (e : E) → Throw^s E
data Catch^s           : Set where catch^s : Catch^s
data Catch^p (E : Set) : Set where
  main^p   : Catch^p E
  handle^p : (e : E) → Catch^p 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
```

Examples Using the smart constructors, we can define programs using exceptions as well as nondeterminism. Because Nondet has no scoping operation, only one of the two cases for traversing foreign scopes is used, the one for Nondet in the example with global exceptions. More examples for interacting of different effects with scoping operations can be found in the repository.

```
interaction : {! Nondet ∈ effs !} → {! Exc ⊤ ∈ effs !} → Prog effs ℕ
interaction = (throw tt ?? pure 2) catch λ tt → pure 1
```

Running the handler for exceptions first results in local exceptions, i.e. each nondeterministic computation branch returns either an exception or a value and the catch interacts with each branch in isolation. Therefore, the catch provides the alternative result 1 on the first branch and does nothing on the second branch.

```
run (runNondet (runExc interaction)) ≡ inj₂ 1 :: inj₂ 2 :: []
```

Because the program in scope of the catch encounters an exception, all results are discarded and replaced with the result of the handler. The handler for Nondet is run before the handler for exceptions and therefore has to traverse the scope of catch.

```
run (runExc (runNondet interaction)) ≡ inj₂ (1 :: [])
```

5.6 State

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

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

In Chapter 3 the state handler was defined recursively and the current state was passed as an additional parameter to the handler. When implementing the handler as a fold this is not an option. Instead, we choose the usual carrier for state (transformer) as carrier for our handler i.e. function from the current state to a (lifted) pair, consisting of the final state and the result. Handler with a function as carrier type are called *parameter passing handler* [26]. When implementing the handler as a fold, the function argument appears as an additional parameter alongside the currently handled operation and as additional parameter to the continuation. Parameter passing handlers are a common pattern when implementing handler because they allow threading a state through the handler.

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 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 = Shareˢ ⇝ ⊤

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

The handler is implemented as a parameter passing handler. The carrier type is a function that maps from the current choice id to the shared program. In contrast to State, where a function was the intuitive solution, runShare′ follows the explanation by Plotkin and Pretnar [26] more closely. It simulates handler for Share that passes additional parameters around. runShare supplies initial parameters and hides this internal state from the caller.

When handling a sharing scope, the captured program is handled with the id stored in the shape of the scoping operation. The continuation is accessed via ≫= and continuous with the outside id. Scopes of foreign effects do not affect sharing. The foreign scope is a subscope of the current sharing scope. Therefore, the choices are labelled with the same scope id. To guaranty the uniqueness of ids we have to thread the current choice id through the inner scope. We add the current choice id as an additional return value. After traversing a foreign scope, we continue with the last id from the inner scope. In case of a sharing scope we number choices inside the scope using the scopes id and continue with the current outer id after the scope.

```
runShare′ : {! Nondet ∈ effs !} → Prog (Share :: effs) A →
  Maybe CID → Prog effs (Maybe CID × A)
runShare′ {effs} {A} {! p !} = fold
  (λ i → ((λ X → Maybe CID → Prog effs (Maybe CID × X)) ^ i) A) 1 id
  (λ z cid → var (cid , z))
  (λ{ (Other s pf) → case prj (opsInj p) (s , pf) of λ where
    nothing          sid                → op (s , λ p → pf p sid)
    (just (failˢ , κ)) sid              → fail
    (just (??ˢ _ , κ)) nothing          → Op (??ˢ nothing , λ p → κ p nothing)
    (just (??ˢ _ , κ)) cid@(just (sid , n)) → Op (??ˢ cid , λ p → κ p (just (sid , suc n)))
  }) λ where
    (ShareScp sid′ κ) sid → κ tt (just $ sid′ , 0) ≫= λ (_ , r) → r sid
    (Other s κ) sid → scp (s , λ p → (λ (sid′ , k) → k sid′) <$> κ p sid)

runShare : {! Nondet ∈ effs !} → Prog (Share :: effs) A → Prog effs A
runShare p = π₂ <$> runShare′ p nothing
```

We introduce the helper function runShare to conveniently call the actual sharing handler. The handler is initially called without an id because by default choices are not shared. Furthermore, after evaluating all sharing syntax we discard the current id because it is not needed anymore.

The Shareable and Normalform infrastructure from Chapter 3 can be reused to define the share operator. The program in the scope is again captured by mapping pure over it. The construction of scope ids follows again the implementation by Bunkenburg [9].

```
share⟨_⟩ : {! Share ∈ effs !} → SID → Prog effs A → Prog effs A
share⟨_⟩ 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 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′
```

Examples Using Nondet, State and Share we can again simulate call-time choice semantics. We introduce the function runCTC to easily evaluate programs using normalizable data and call-time choice semantics.

```
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 ⟫
```

As expected, doubling a shared coin yields the results 0 and 2.

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

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 type class instances from Chapter 3 are reused. The definition of doubleHead stays the same, while the new underlying representation for the effects eliminates the problem of potentially mismatched scope delimiters.

```
doubleHead : {⟦ Nondet ∈ effs ⟧} → {⟦ Share ∈ effs ⟧} → {⟦ State SID ∈ effs ⟧} →
  Prog effs ℕ
doubleHead = do mxs ← share (coin ::ᴹ []ᴹ)
                ⟪ headᴹ mxs + headᴹ mxs ⟫
```

Using the Shareable instance we can share choices inside data structures. As expected, sharing a coin inside a list, unwrapping and doubling it yields the results 0 and 2.

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

5.8 Results

In this chapter we presented an implementation of scoped algebras [23] in Agda. To generally guaranty termination of recursive functions on values of Prog, we defined its recursion scheme based on the work by Fu and Selinger [13]. Our recursion scheme, based on Fu and Selinger [13]'s idea of introducing an index type, has essentially the same shape as the one needed for scoped algebras.

Using the recursion scheme we were able to define the nondeterminism example by Piróg et al. [23] in Agda, without resorting to sized types. Furthermore, using the corresponding induction scheme for Prog we were able to prove propositions by induction on Prog. With this approach we could easily implement deep effects because the Prog monad does not increase the universe level. Based on the patterns from the earlier chapters, i.e. the general idea of partially removing syntax [28], we implemented a naive modularization for handlers. In our current limited, practical setting the examples in the thesis as well as the ones in the repository seem promising. The cases for traversing foreign sharing scopes still bear resemblance to the functions passed to handle in Chapter 4. This is partially expected because the monad in this chapter is derived from a monad similar to the one in Chapter 4. However, due to rewriting the monad, we again loose explicit control over the continuation. Using our modular approach we could again implement Bunkenburg's [9] sharing handler.

Chapter 6

Conclusion

In this last chapter we summarize the work done in the thesis and discuss the results as well as possible future work.

6.1 Summary

The main goal was to implement a library for scoped algebraic effects by porting the two haskell implementations from "Effect handlers in scope" to Agda.

Of particular significance was to work under Agda's constraints for consistency, which are needed for theorem proving. These included defining only functions which Agda could prove terminate, defining only strictly positive recursive types and work with a consistent universe structure. As a running example we tried implementing effects related to simulating Curry's call-time choice semantics, namely nondeterminism and the sharing effect by Bunkenburg [9]. In the chapters 3, 4 and 5 we discussed how three different approaches translate to Agda.

6.2 Results

In Chapter 3, we first implemented ordinary algebraic effects in Agda. The implementation only depends on the definition of the free monad, which we implemented using the well known approach of replacing functors with containers. To prove termination we used sized types. They proved effective because they allowed the definition of Wu et al. [34] handlers for explicit scope delimiters as well as avoided excessive inlining. Furthermore, it was possible to implement the sharing handler by Bunkenburg [9] in Agda, which makes this approach a valid choice for simulating call-time choice semantics. A problem with this approach is that it inherently allows mismatched scope delimiters and therefore malformed syntax, which potentially complicates proofs.

In Chapter 4, we tried implementing effects using higher order syntax as described by Wu et al. [34]. This approach turned out to be problematic because it relies on existential types to capture subcomputations. Holding onto the idea of storing types directly, we showed that it is not possible to implement deep effects in a setting with bounded universe levels. Furthermore, we discussed potential alternative approaches although non of them seem promising. The repository contains a sketch of an implementation with variable universe levels, which showcases the practical difficulties. Even thought this approach cannot model deep effects and therefore is not expressive enough to model Curry's ambient nondeterminism in data structures, we still presented the key points of a limited implementation of normal scoped effects using higher order syntax.

Due to the deep-seated problem of the approach from Chapter 4 we experimented with an novel representation by Piróg et al. [23] which allowed us to remove the existential type. The naive Agda implementation of the monad Piróg et al. [23] proved to be problematic in terms of theorem proving. We derived an alternative underlying implementation, based on the work by Fu and Selinger [13], which allowed the implementation of complex recursive functions and proofs without the use of sized types. The approach allows the implementation of deep effects and also can be used to simulate sharing. Furthermore, we experimented with a naive modularisation of scoped effects in this setting. The naive implementation seems promising in our limited, practical setting and shows parallels to the higher-order approach.

6.3 Related Work and future Directions

Different versions of effects and handlers are becoming more popular in the functional programming community. To still provide the usual structure and expressiveness of normal monad transformers, most Haskell implementations (e.g. polysemy [19], fused-effects [35]) internally use higher order syntax to implement scoped effects.

In dependently typed language much less work has been done. Brady [8] presented an implementation of resource dependent effects in Idris. Resource dependent effects are a version of algebraic effects in a dependently typed setting. An Agda version of the library by Norell exists. Baanen [5] also implement algebraic effects in Agda. His work focuses more on program verification.

6.3.1 Future Work

For each of the three approach, the basic infrastructure as well as some exemplary effects were presented. Based on them, the corresponding implementation could be extended with more effects. In case of the approaches from Chapter 4 and Chapter 5 a simplified syntax was used. Some scoped operations, for example `pass` and `listen` for the `Writer` effect, have fundamentally different shape which cannot be represented at the moment. Implementing these operations requires generalizing the syntax. For higher-order approach alternative container representations were mentioned. For scoped algebras Piróg et al. [23] also note a generalization for more expressive syntax.

The modularization of effect in the Chapter 5 is quite experimental. Implementing more tests for the current set of handlers as well as adding more effects with scoping operations to test the modularization should be the next step. Because we only explored an practical implementation of modular effects, we should also examine the approach in a more formal setting.

Another area which could be explored further is reasoning about programs using effects. All proofs in the thesis either used a fixed effect stack or just hold if the handlers for the effect was called last. Most real program use multiple effects, which differ between functions. Properties of effects may not hold if a non-orthogonal handler is called first. Exploring this issue and defining structures for proving and using propositions about effects in a modular setting could be a useful addition to the library.

The approaches in this thesis are based on the work by Wu et al. [34] and Piróg et al. [23]. Norell ported the Idris effect library by Brady [8] to Agda. The library does not allow scoped effects, but it uses a fundamentally different structure. Extending this approach with scoped effects could provide an alternative to the approaches implemented in this thesis. Vice versa, one could try extending one of the approaches presented here with recourse dependent effects.

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] Tim Baanen. "Algebraic effects, specification and refinement". MA thesis. 2019.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. "Purely functional lazy nondeterministic 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.

[13] 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.

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

[15] Michael Hanus and Finn Teegen. "Memoized Pull-Tabbing for Functional Logic Programming". In: *CoRR* abs/2008.11999 (2020). arXiv: 2008.11999. URL: https://arxiv.org/abs/2008.11999.

[16] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, Sept. 2002, p. 277. URL: http://haskell.org/definition/haskell98-report.pdf.

[17] Wen Kokke, Jeremy G. Siek, and Philip Wadler. "Programming language foundations in Agda". In: *Sci. Comput. Program.* 194 (2020), p. 102440. DOI: 10.1016/j.scico.2020.102440. URL: https://doi.org/10.1016/j.scico.2020.102440.

[18] John Launchbury. "A Natural Semantics for Lazy Evaluation". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. Ed. by Mary S. Van Deusen and Bernard Lang. ACM Press, 1993, pp. 144–154. DOI: 10.1145/158511.158618. URL: https://doi.org/10.1145/158511.158618.

[19] Sandy Maguire. *polysemy: Higher-order, low-boilerplate free monads*. 2019. URL: https://github.com/polysemy-research/polysemy#readme.

[20] Conor McBride. "Turing-Completeness Totally Free". In: *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Lecture Notes in Computer Science. Springer, 2015, pp. 257–275. DOI: 10.1007/978-3-319-19797-5_13. URL: https://doi.org/10.1007/978-3-319-19797-5%5C_13.

[21] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, 2008, pp. 230–266. DOI: 10.1007/978-3-642-04652-0_5. URL: https://doi.org/10.1007/978-3-642-04652-0%5C_5.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Harald Søndergaard and Peter Sestoft. “Referential Transparency, Definiteness and Unfoldability”. In: *Acta Informatica* 27.6 (1990), pp. 505–517. DOI: 10.1007/BF00277387. URL: https://doi.org/10.1007/BF00277387.

[30] 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.

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

[32] P. Wadler. “Recursive types for free”. In: 1991.

[33] 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.

[34] 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.

[35] Nicolas Wu et al. *fused-effects: A fast, flexible, fused effect system.* 2018. URL: https://github.com/fused-effects/fused-effects.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, an noch keiner anderen Stelle zur Begutachtung eingereicht, keine anderen als die angegebenen Hilfsmittel verwendet und alle wörtlichen oder sinngemäßen Übernahmen von Aussagen bzw. Ergebnissen Dritter an der jeweiligen Stelle meiner Ausführungen kenntlich gemacht sowie alle benutzten Quellen in der Bibliografie aufgelistet habe.

Ort, Datum

Unterschrift