Implementing a library for scoped algebraic effects in Agda

HTWK Leipzig

Jonas Höfer
Informatik
69555

2020

Abstract

Contents

Chapter 1

Introduction

1.1 Goals

Chapter 2

Preliminaries

2.1 Agda

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

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

2.1.1 Basic Syntax

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

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

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

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

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

2.1.2 Dependent Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1.3 Propositions as Types

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

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

2.1.4 Termination Checking

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

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

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

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

Sized Types

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1.5 Strict Positivity

In a type system with arbitrary recursive types, it is possible to to implement a fixpoint combinator and therefore non terminating functions without explicit recursion. As explained in section 2.1.4 this entails logical inconsistency. Agda allows only strictly positive data types. A data type D is strictly positive if all constructors are of the form

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

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

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

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

Container

Because of the strict positivity requirement it is not allowed to apply generic type constructors to inductive occurrences of a data type in its definition. The reason for this restriction is that

a type constructor is not required to use its argument only in strictly positive positions. To still work generically with type constructors or more precise functors we need a more restrictive representation, which only uses its argument in a strictly positive position. One representation of such functors are containers.

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

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

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

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

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

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

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

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

Using this definition we can define fmap for containers.

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

2.2 Curry

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

When defining a function with overlapping patterns on the left-hand side of equations all matching right-hand sides are executed. This introduces non determinism. The simplest example of such a function is the choice operator ?.

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

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

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

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

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

Call-Time-Choice

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

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

doubleCoin :: Int
doubleCoin = double coin
```

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

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

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

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

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

As expected this function yields the results 0 and 2.

Permutation Sort

Introduce Free Variables + Explain for Later Example

Chapter 3

Algebraic Effects

Algebraic effects and handlers were first presented by Gordon and Plotkin as a generalization of exception handlers []. Algebraic effects are computational effects, which can be described using an algebraic theory. Examples include I/O, exceptions, nondeterminism, state, delimited continuations and more [Bau18].

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

3.1 Definition

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

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

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

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

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

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

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

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

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

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

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

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

3.1.1 Free Model

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

3.2 Free Monads

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

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

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

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

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

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

When defining the free monad in Agda we cannot use an arbitrary functor as in Haskell, because it would violate the strict positivity requirement. Instead we will represent the functor as the extension of a container as described in section 2.1.5.

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

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

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

3.2.1 Functors à la carte

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

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

In Agda functors are represented as containers, a concrete data type not a type class. The coproduct of two containers F and G is the container whose shape is the disjoint union of Fs and Gs shapes and whose position function pos is the coproduct mediator of Fs and Gs position functions.

```
_⊕_ : 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 is 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 will define two utility functions. Given a value $x : A$ we want to be able to inject it into any coproduct mentioning A. Given any coproduct mentioning A we want to be able to project a value of type A from the coproduct, if A is the currently held alternative.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.2 The Free Monad for Effect Handling

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

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

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

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

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

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

upcast : ∀ {C ops A i} → Free ops A {i} → Free (C :: ops) A {i}
upcast (pure x) = pure x
upcast (impure (s , κ)) = impure (inj₂ s , upcast ∘ κ)
```

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

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

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

fmap = _<$>_
```

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

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

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

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

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

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

_<*>_ = _⊛_
```

3.2.3 Properties

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

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

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

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

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

```
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))
```
(4)

```
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))
bind-assoc f g (pure x)       = refl
bind-assoc f g (impure (s , pf)) = cong (impure ∘ (s ,_)) (extensionality (bind-assoc f g ∘ pf))
```
(5)

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

3.3 Handler

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

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

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

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

3.3.1 Nondet

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2 State

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

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

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

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

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

```
data 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.

```
get : ∀ {ops S} → {{ State S ∈ ops }} → Free ops S
get = inj (get^s , pure)

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

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

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

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

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

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

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

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

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

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

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

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

Example

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

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

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

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

Properties

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

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

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

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

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

3.4 Scoped Effects

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

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

3.4.1 Cut and Call

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

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

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

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

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

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

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

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

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

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

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

```
    go _ Cutfail         = fail
```

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

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

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

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

When encountering a fail we continue with the accumulated alternatives.

```
    ... | just (fails , _ ) = q
```

Syntax from other effects is handled as usual.

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

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

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

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

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

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

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

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

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

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

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

call : ⦃ Call ∈ ops ⦄ → Free ops A → Free ops A
call p = do op bcall^s ; x ← p ; op ecall^s ; pure x
```

3.5 Call-Time Choice as Effect

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

As explained in section 2.2, call-time choice semantics describe the interaction between sharing and nondeterminism. The current implementation of Nondet does not support sharing i.e. it is not possible for two choice to be linked. Based on Bunkenburgs implementation we will make two changes to the nondeterminism effect. Each choice is augmented with an an identifier consisting of a triple of natural numbers, called *choice id*. The first two are used to identity the current scope and will be refereed to as *scope id*. The third numbers identifies the choice inside it's scope. Furthermore, instead of producing a list the handler will now produce a tree of choices. This change allows to choose the evaluation strategy, e.g. depth first or breath first search, independent of the handler.

Chapter 4

Higher Order

- working implementation using (simplified) indexed containers
- scoped effects alone work
- does only partially work with monadic data structures (size problems when using scoped effects like sharing)
- sharing is the “worst case scenario”
- size problems generally occur with this approach thanks to existential types (which again hold programs)

Chapter 5

Hybrid Approach

- potential worth pursuing (no explicit scope delimiters, but control over continuation (seems) limited in contrast to HO approach (important for memorization of sharing))
- based on Syntax and Semantics for Scoped Effects by Wu et al.
- rewrites higher order monad using Kan Extension ⇒ removes coend ⇒ no existential types ⇒ no size issue (marks scope using double monad layer)
- algebras from paper don't seem to compose easily, but traditional handlers seem to work
  - Carrier types are type families ⇒ Agda implementation could be easier than Haskell implementation thanks to better dependent type support
- Current Problem: proof of monad laws seems impossible due to termination issues thanks to double monadic layer in the `Scope` constructor

Chapter 6

Conclusion

6.1 Summary

Bibliography

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

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

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

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

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

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

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

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

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

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