Implementing a library for scoped algebraic effects in Agda

HTWK Leipzig

Jonas Höfer
Informatik
69555

Juli 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 proofing.

[TODO: General Introduction]

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

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

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

2.1.1 Dependent Types

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

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

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

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 $b : B(a)$.

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

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 $b : B(a)$.

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 an 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 type, increasing the index in the process. 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 the 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 i.e. it's impossible to call the function, because we cannot construct such an argument. 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 hard 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.

2.1.2 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**	$\perp$

2.1.3 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

```
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

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

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)
```

2.1.4 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 2.1.3 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. 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

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

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.

$$\mathsf{fmap} : \forall \{A\ B\ C\} \to (A \to B) \to (\llbracket\ C\ \rrbracket\ A \to \llbracket\ C\ \rrbracket\ B)$$
$$\mathsf{fmap}\ f\ (s\ ,\ pf) = (s\ ,\ f \circ pf)$$

2.2 Curry

Curry [HKM95] is a functional logic programming language.

When defining a function with overlapping patterns on the left-hand side of the 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 1.

Chapter 3

Algebraic Effects

Algebraic effects are computational effects, which can be described using an algebraic theory.

Section 3.1 gives a concrete definition for algebraic effects. Section 3.2 describes the implementation using free monads in Agda.

3.1 Definition

Needed? A la Bauer? Adapts nicely to containers, but I'm not sure how well it works wth scoped syntax and the HO approach

3.2 Free Monads

The syntax of an algebraic effect is described using the free monad.

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

⟦_⟧ : {ℓ : Level} → Container → Set ℓ → Set ℓ
⟦ S ▷ P ⟧ A = Σ[ s ∈ S ] (P s → A)
```

In the approach described by Wu et la. 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 F and Gs shapes and whose position function pos is the coproduct mediator of F and Gs position functions. The functor represented by the coproduct of two containers is isomorphic to the functor coproduct of their representations.

```
_⊕_ : Container → Container → Container
(Shape₁ ▷ Pos₁) ⊕ (Shape₂ ▷ Pos₂) = (Shape₁ ⊎ Shape₂) ▷ [ Pos₁ , Pos₂ ]
-- ⟦ F ⊕ G ⟧ A ≅ (⟦ F ⟧ + ⟦ G ⟧) A Should i proof this?
```

Later each container will represent the syntax (the operations) of an effect. To combine syntax of effects Wu et al. use a “Data types à la carte”[Swi08] approach. The type class `:<:` marks a functor as an option in a coproduct. `:<:` can be used to inject values into or maybe extract values from a coproduct. The two instances for `:<:` overlap and use `:+:`. Since 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.

Therefore 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 representing an n-ary coproduct. Whenever the functor would be used, an arbitrary container *op* together with a proof $op \in ops$ is used.

```
infix 4 _∈_
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.

```
data Free {ℓ : Level} (ops : List Container) (A : Set ℓ) : {Size} → Set (ℓ ⊔ suc zero) where
  pure : ∀ {i} → A → Free ops A {i}
  impure : ∀ {i op} → ⦃ op ∈ ops ⦄ → ⟦ op ⟧ (Free ops A {i}) → Free ops A {↑ i}
```

[GENERAL EXPLANATION]

[EXPLANATION FOR LEVEL]

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[1].

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

fmap = _<$>_
```

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

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

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

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

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

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

3.2.1 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.

[1] in the following code A, B and C are arbitrary types from arbitrary type universes

$$\begin{aligned}
&\mathsf{fmap\text{-}id} : \forall\ \{ops\} \to (p : \mathsf{Free}\ ops\ A) \to \mathsf{fmap}\ \mathsf{id}\ p \equiv p \\
&\mathsf{fmap\text{-}id}\ (\mathsf{pure}\ x) \qquad\quad = \mathsf{refl} \\
&\mathsf{fmap\text{-}id}\ (\mathsf{impure}\ (s\ ,\ pf)) = \mathsf{cong}\ (\mathsf{impure} \circ (s\ ,_))\ (\mathsf{extensionality}\ (\mathsf{fmap\text{-}id} \circ pf))
\end{aligned} \tag{1}$$

$$\begin{aligned}
&\mathsf{fmap\text{-}\circ} : \forall\ \{ops\}\ (f : B \to C)\ (g : A \to B)\ (p : \mathsf{Free}\ ops\ A) \to \\
&\qquad \mathsf{fmap}\ (f \circ g)\ p \equiv (\mathsf{fmap}\ f \circ \mathsf{fmap}\ g)\ p \\
&\mathsf{fmap\text{-}\circ}\ f\ g\ (\mathsf{pure}\ x) \qquad\quad = \mathsf{refl} \\
&\mathsf{fmap\text{-}\circ}\ f\ g\ (\mathsf{impure}\ (s\ ,\ pf)) = \mathsf{cong}\ (\mathsf{impure} \circ (s\ ,_))\ (\mathsf{extensionality}\ (\mathsf{fmap\text{-}\circ}\ f\ g \circ pf))
\end{aligned} \tag{2}$$

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

$$\begin{aligned}
&\mathsf{bind\text{-}ident}^{\mathsf{l}} : \forall\ \{ops\}\ (f : A \to \mathsf{Free}\ ops\ B)\ (x : A) \to (\mathsf{pure}\ x \gg= f) \equiv f\ x \\
&\mathsf{bind\text{-}ident}^{\mathsf{l}}\ f\ x = \mathsf{refl}
\end{aligned} \tag{3}$$

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

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

3.3 Handler

```
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 = impure (??ˢ , (if_then p else q))
```

```
fail : ∀ {ops} → ⦃ Nondet ∈ ops ⦄ → Free ops A
fail = impure (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 ⦃ there ⦄ (s , κ)
pattern Fail κ = impure ⦃ here ⦄ (failˢ , κ)
pattern Choice κ = impure ⦃ here ⦄ (??ˢ , κ)
```

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

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

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

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

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

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

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

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

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

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

3.3.2 State

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

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

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

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

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

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

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

$$\text{pattern Get } \kappa = \text{impure } \{\!\!\{ \text{ here } \}\!\!\} \ (\text{get}^{\mathsf{s}} \ , \kappa)$$
$$\text{pattern Put } s \ \kappa = \text{impure } \{\!\!\{ \text{ here } \}\!\!\} \ ((\text{put}^{\mathsf{s}} \ s) \ , \kappa)$$

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

$$\text{get} : \forall \ \{ops \ S\} \rightarrow \{\!\!\{ \text{ State } S \in ops \ \}\!\!\} \rightarrow \text{Free } ops \ S$$
$$\text{get} = \text{impure } (\text{get}^{\mathsf{s}} \ , \text{pure})$$

$$\text{put} : \forall \ \{ops \ S\} \rightarrow \{\!\!\{ \text{ State } S \in ops \ \}\!\!\} \rightarrow S \rightarrow \text{Free } ops$$
$$\text{put } s = \text{impure } (\text{put}^{\mathsf{s}} \ s \ , \text{pure})$$

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.

$$\text{runState} : \forall \ \{ops \ S\} \rightarrow S \rightarrow \text{Free } (\text{State } S :: ops) \ A \rightarrow \text{Free } ops \ (S \times A)$$

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 caluclation.

$$\text{runState } s_0 \ (\text{pure } x) \quad = \text{pure } (s_0 \ , x)$$

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.

$$\text{runState } s_0 \ (\text{Get } \kappa) \quad = \text{runState } s_0 \ (\kappa \ s_0)$$

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

$$\text{runState } \_ \ (\text{Put } s_1 \ \kappa) \quad = \text{runState } s_1 \ (\kappa \ \text{tt})$$

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

$$\text{runState } s_0 \ (\text{Other } s \ \kappa) = \text{impure } (s \ , \text{runState } s_0 \circ \kappa)$$

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.

$$\text{tick} : \forall \ \{ops\} \rightarrow \{\!\!\{ \text{ State } \mathbb{N} \in ops \ \}\!\!\} \rightarrow \text{Free } ops$$
$$\text{tick} = \text{do } i \leftarrow \text{get} \ ; \text{put } (1 + i)$$

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

$$(\text{run } \$ \text{ runState } 0 \ \$ \text{ tick} \gg \text{tick}) \equiv (2 \ , \text{tt})$$

Properties

3.4 Scoped Effects

3.4.1 Cut

3.5 Call-Time-Choice as Effect

Chapter 4

Higher Order

Chapter 5

Conclusion

5.1 Summary

Bibliography

[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`.

[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`.