

Implementing a library for scoped algebraic effects in Agda

HTWK Leipzig

Jonas Höfer
Informatik
69555

Juli 2020

Abstract

Contents

1	Introduction	3
1.1	Goals	3
2	Preliminaries	4
2.1	Agda	4
2.1.1	Dependent Types	4
2.1.2	Propositions as Types	5
2.1.3	Strict Positivity	5
2.1.4	Termination Checking	5
2.2	Curry and Call-Time-Choice	6
3	Algebraic Effects	7
3.1	Definition	7
3.2	Free Monads	7
3.2.1	Properties	8
3.3	Handler	9
3.3.1	Nondet	9
3.3.2	State	10
3.4	Scoped Effects	11
3.4.1	Cut	11
3.5	Call-Time-Choice as Effect	11
4	Higher Order	12
5	Conclusion	13
5.1	Summary	13

Chapter 1

Introduction

1.1 Goals

Chapter 2

Preliminaries

2.1 Agda

Agda is a dependently typed functional programming language. The current version¹ 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.

Agda’s syntax is similar to Haskell’s. 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 `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.

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

To avoid Russell’s paradox, i.e. there cannot exist a type of all types, a hierarchy of universes is introduced.

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.

$$\Pi_{a:A} B(a) \quad \text{with} \quad B : A \rightarrow \mathcal{U}$$

An element of the above type is a function which maps every $a : A$ to a $b : B(a)$.

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.

$$\Sigma_{a:A} B(a) \quad \text{with} \quad B : A \rightarrow \mathcal{U}$$

¹<https://github.com/agda/agda>

In Agda Consider the following definition of a vector. The data type depends on a type A and a value of type \mathbb{N} . Therefore, it's a family of types indexed by \mathbb{N} . Each $n : \mathbb{N}$ is mapped to the type $\text{Vec } A \ n$.

```
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.

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 $\text{Vec } A \ (\text{suc } n)$ and `[]` has type $\text{Vec } A \ 0$. Those two types cannot be unified, because `suc` and `zero` are different constructors of \mathbb{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.

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
$\forall x \in A : P(x)$	$\prod_{x:A} P(x)$	$(x : A) \rightarrow P \ x$
$\exists x \in A : P(x)$	$\Sigma_{x:A} P(x)$	$_,_ : (x : A) \rightarrow P \ x \rightarrow \Sigma A \ P$
$P \wedge Q$	$P \times Q$	$A \times B$
$P \vee Q$	$P + Q$	$A \uplus B$
$P \Rightarrow Q$	$P \rightarrow Q$	$A \rightarrow B$
\mathbf{t}	$\mathbf{1}$	$\mathbf{tt} :$
\mathbf{f}	$\mathbf{0}$	\perp

2.1.3 Strict Positivity

Container

2.1.4 Termination Checking

The definition of non-terminating functions entails logical inconsistency. Agda therefore only allows the definition of terminating functions. 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 structurally 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)

```

2.2 Curry and Call-Time-Choice

```
let x = coin in x + x
```

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 with 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 al. 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 G s shapes and whose position function `pos` is the coproduct mediator of F and G s 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 $_ \in _$ 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 prove that functions preserve the size of a value or that complex recursive functions terminate. Consider the following definition of **fmap** for the free monad¹.

```

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 infers ∞ .

```

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

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

```

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

```

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

```

3.2.1 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.

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

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

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

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

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

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

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

3.3 Handler

$$\begin{aligned}
\text{run} &: \text{Free } [] \ A \rightarrow A \\
\text{run } (\text{pure } x) &= x
\end{aligned}$$

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.

$$\text{data Nondet}^s : \text{Set where } ??^s \text{ fail}^s : \text{Nondet}^s$$

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

$$\begin{aligned}
\text{Nondet} &: \text{Container} \\
\text{Nondet} &= \text{Nondet}^s \triangleright \lambda \text{ where} \\
&\quad ??^s \rightarrow \text{Bool} \\
&\quad \text{fail}^s \rightarrow \perp
\end{aligned}$$

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 and always use `pure` as continuation. These versions of the operations already process the continuations parameter.

$$\begin{aligned}
?? &: \forall \{ops\} \rightarrow \llbracket \text{Nondet} \in ops \rrbracket \rightarrow \text{Free } ops \ A \rightarrow \text{Free } ops \ A \rightarrow \text{Free } ops \ A \\
p \ ?? \ q &= \text{impure } (??^s, (\text{if_then } p \text{ else } q))
\end{aligned}$$

```
fail :  $\forall \{ops\} \rightarrow \llbracket \text{Nondet} \in ops \rrbracket \rightarrow \text{Free } ops \ A$ 
fail = impure (fails,  $\lambda()$ )
```

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  $\kappa$  = impure  $\llbracket \text{there} \rrbracket (s, \kappa)$ 
pattern Fail  $\kappa$  = impure  $\llbracket \text{here} \rrbracket (\text{fail}^s, \kappa)$ 
pattern Choice  $\kappa$  = impure  $\llbracket \text{here} \rrbracket (??^s, \kappa)$ 
```

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 :  $\forall \{ops\} \rightarrow \text{Free } (\text{Nondet} :: ops) \ A \rightarrow \text{Free } ops \ (\text{List } A)$ 
```

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  $\kappa$ ) = 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  $\kappa$ ) = _+_<$> solutions ( $\kappa$  true)  $\otimes$  solutions ( $\kappa$  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  $\kappa$ ) = impure (s, solutions  $\circ$   $\kappa$ )
```

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.

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

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

```
data States (S : Set) : Set where
  gets : States S
  puts : S  $\rightarrow$  States S

State : Set  $\rightarrow$  Container
State S = States S  $\triangleright$   $\lambda$  where
  gets  $\rightarrow$  S
  (puts _)  $\rightarrow$ 
```

```

pattern Get  $\kappa$  = impure  $\llbracket \text{here} \rrbracket$  (gets ,  $\kappa$ )
pattern Put  $s$   $\kappa$  = impure  $\llbracket \text{here} \rrbracket$  ((puts  $s$ ) ,  $\kappa$ )

```

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

```

get :  $\forall \{ops\} S \rightarrow \llbracket \text{State } S \in ops \rrbracket \rightarrow \text{Free } ops\ S$ 
get = impure (gets , pure)

put :  $\forall \{ops\} S \rightarrow \llbracket \text{State } S \in ops \rrbracket \rightarrow S \rightarrow \text{Free } ops$ 
put  $s$  = impure (puts  $s$  , pure)

```

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

```

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 calculation.

```

runState  $s_0$  (pure  $x$ ) = 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.

```

runState  $s_0$  (Get  $\kappa$ ) = 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**.

```

runState _ (Put  $s_1$   $\kappa$ ) = runState  $s_1$  ( $\kappa$  tt)

```

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

```

runState  $s_0$  (Other  $s$   $\kappa$ ) = impure ( $s$  , 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.

```

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

```

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

```

(run $ runState 0 $ tick  $\gg$  tick)  $\equiv$  (2 , 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>.
- [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>.