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 functional language with dependent types. The current version[1] was originally developed by Ulf Norell [Nor07]. Due to its type system Agda can be used as a programming language and as a proof assistant.

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

2.1.1 Dependent Types

Dependent types are types which depend on values. Types can have both other types and values as arguments. Arguments on the left-hand side of the colon are called parameters and are the same for all constructors of an algebraic data type. Arguments on the right-hand side of the colon are called indices an can differ for each constructor.

Consider the following definition of a vector. The data type depends on a type A and a value, a natural number.

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

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 [TODO: static dynamic semantics / promote to type error / ...]. 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.

2.1.2 Propositions as Types

[Wad15]

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

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 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
        ; ∞ )                    -- ω : 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 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 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 i} → Free ops A {i} → (A → Free ops B) → Free ops B
pure x        >>= k = k x
impure (s , pf) >>= k = impure (s , (_>>= k) ∘ pf)

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

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.

```
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)

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

```
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)

3.3 Handler

3.3.1 Nondet

3.3.2 State

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.

[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.