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
        ; ∞ )                    -- ∞ : 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

3.1 Free Monads

3.2 Handler

3.2.1 Nondet

3.2.2 State

3.3 Scoped Effects

3.3.1 Cut

3.4 Call-Time-Choice as Effect

Chapter 4

Higher Order

Chapter 5

Conclusion

5.1 Summary

Bibliography

[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.

[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`.