λ-Calculus in Agda

Joris Klaasse Bos

February 9, 2024

Preface

This document is a literate Agda program that implements and explains the λ-calculus. Of course, I recognise the tremendous irony that herein lies, seeing as we explain λ-calculus through what is essentially not much more than an implementation of dependently typed λ-calculus; it is unlikely that someone who knows Agda should not know λ-calculus already—although they need not be familiar with Church encodings per se. This document should be seen as (very overkill) Theory of Functional Programming lecture notes by someone who is already well versed in the subject.

Contents

Prelude

```
module lc where
  open import Agda.Builtin.Equality
  open import Agda.Primitive using (Level; lsuc; lzero; _⊔_) renaming (Set to 𝒰)
  import Relation.Binary.PropositionalEquality as Eq
  open import Relation.Nullary using (¬_)
  open import Relation.Nullary.Negation using (¬?)
  open Eq using (_≡_; refl; cong; cong₂; cong-app; sym; trans)
  open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
  open import Function.Base using (_∘_; _∘₂_; id; _∋_; flip)
  open import Agda.Builtin.String renaming (primStringEquality to _=ₛ_)
  open import Data.String using (String; _≟_)
  open import Data.List using (List; []; _∷_; _++_; filter)
open import Data.Bool.Base using (not; if_then_else_; Bool; true; false; _∧_; _∨_)
open import Data.Maybe using (Maybe; just; nothing; map)

variable
  ℓ ℓ₁ ℓ₂ ℓ₃ ℓ₄ ℓ₅ ℓ₆ ℓ₇ ℓ₈ ℓ₉ : Level
  A B C D : 𝓤 ℓ

-- Finds the first element satisfying the boolean predicate
findᵇ : (A → Bool) → List A → Maybe A
findᵇ p [] = nothing
findᵇ p (x ∷ xs) = if p x then just x else findᵇ p xs

-- Check if a list of strings contains a certain string
contains : List String → String → Bool
contains l x with findᵇ (_=ₛ x) l
contains l x | nothing = false
contains l x | just _ = true

liftM2 : (A → B → C) → Maybe A → Maybe B → Maybe C
liftM2 f (just x) = map (f x)
liftM2 f nothing _ = nothing
```

1 λ-Calculus

1.1 λ-Terms

The λ-calculus is an incredibly simple Turing-complete language, i.e. it can express any computation a Turing machine can. It has only three introduction rules:

```
data λ-Term : 𝓤 where
  `     : String → λ-Term
  _⌣_   : λ-Term → λ-Term → λ-Term
  _↦_   : String → λ-Term → λ-Term
```

These three types of terms are known as *variables*, *applications*, and *abstractions* respectively. An example of a λ-term could be

```
_ = λ-Term ∋ ((("a" ↦ ("b" ↦ ` "a")) ⌣ ` "x") ⌣ ` "y")
```

There are a lot of parentheses there. To make it a little easier to read, we will add some precedence rules to Agda. Since interpreting `("a" ↦ "b" ↦ ` "a")` as `(("a" ↦ "b") ↦ ` "a")` results in a malformed expression, we will make _↦_ right-associative. We will make application left-associative so we can read chains of applications from left to right without needing parentheses.

```
infixl 20 _⌣_
infixr 15 _↦_
infix 20 `
```

We can now rewrite the previous expression as follows

```
_ = λ-Term ∋ ("a" ↦ "b" ↦ ` "a") ⌣ ` "x" ⌣ ` "y"
```

1.2 Bound and Free Variables

We distinguish two types of variables: *bound* and *free* or *unbound*. When we have an abstraction, all occurrences of variables in the body of an abstraction formed from the same string as the first element of said abstraction are called bound. When a variable is not bound by any abstraction, it is called free.

We can write the following function, which returns the names of all the free variables in a λ-term:

```
freeVars : λ-Term → List String
freeVars (` x) = x :: []
freeVars (x ⌣ y) = freeVars x ++ freeVars y
freeVars (x ↦ y) = filter (¬? ∘ (_=? x)) (freeVars y)
```

For example:

```
_ = freeVars (("a" ↦ "b" ↦ ` "a") ⌣ ` "x" ⌣ ` "y") ≡ "x" :: "y" :: [] ∋ refl
_ = freeVars ("a" ↦ ` "b" ⌣ ` "a") ≡ "b" :: [] ∋ refl
```

We can also write a function to find all the names of the bound variables.

```
boundVars : λ-Term → List String
boundVars (` x) = []
boundVars (x ⌣ y) = (boundVars x) ++ (boundVars y)
boundVars (x ↦ y) = x :: (boundVars y)
```

```
_ = boundVars (("a" ↦ "b" ↦ ` "a") ‿ ` "x" ‿ ` "y") ≡ "a" :: "b" :: [] ∋ refl
_ = boundVars ("a" ↦ ` "b" ‿ ` "a") ≡ "a" :: [] ∋ refl
```

Of course, there may be overlap between the results of `freeVars` and `boundVars`, because we are only looking for the names of variables that are bound or free; a name may be used for a bound variable in one subexpression, but free in another.

1.3 Substitution

When we give computational meaning to λ-terms, we will make use of substitution, so we will invent a function for performing substitutions. Do keep in mind that we are not adding something new to the definition of the λ-calculus, but just defining a function in the meta-language Agda to be able to define the computation rules we will see hereafter. We disallow substitutions that change the binding of a variable.

```
_[_:=_] : λ-Term → String → λ-Term → Maybe λ-Term
` v [ x := y ] = if v =ₛ x then just y else just (` v)
e₁ ‿ e₂ [ x := y ] = liftM2 _‿_ (e₁ [ x := y ]) (e₂ [ x := y ])
(v ↦ e) [ x := y ] =
  if x =ₛ v
    then just (v ↦ e) -- Don't do anything when we have an inner rebind
    else if contains (freeVars y) v
      then nothing
      else map (v ↦_) (e [ x := y ])
```

Examples:

```
_ = ("a" ↦ "b" ↦ ` "a") ‿ ` "x" ‿ ` "a" [ "a" := ` "c" ]
    ≡ just (("a" ↦ "b" ↦ ` "a") ‿ ` "x" ‿ ` "c") ∋ refl
_ = ("a" ↦ "b" ↦ ` "a" ‿ ` "c") ‿ ` "c" ‿ ` "a" [ "c" := ` "x" ]
    ≡ just (("a" ↦ "b" ↦ ` "a" ‿ ` "x") ‿ ` "x" ‿ ` "a") ∋ refl
_ = ("a" ↦ "b" ↦ ` "c") ‿ ` "c" ‿ ` "a" [ "c" := ` "a" ]
    ≡ nothing ∋ refl
```

1.4 α-Equivalence

α-Equivalence, also known as *α-conversion* and *α-renaming*, states that the name of a variable in a λ-abstraction does not matter; the name is only used to identify which variable is bound to which λ-abstraction. It states we should be able to rename the variable of a λ-abstraction and be left with an expression that is somehow "the same". Of course, the restrictions imposed on substitution still apply. We will also add some recursive definitions so it applies α-equivalence to the first λ-abstraction it encounters for ease of use.

```
α-equiv : String → λ-Term → Maybe λ-Term
α-equiv x (v ↦ y) = map (x ↦_) (y [ v := ` x ]) -- Main definition
α-equiv x (` x₁) = nothing
α-equiv x (e₁ ‿ e₂) with α-equiv x e₁
α-equiv x (e₁ ‿ e₂) | nothing = map (e₁ ‿_) (α-equiv x e₂)
α-equiv x (e₁ ‿ e₂) | just e₁ ′ = just (e₁ ′ ‿ e₂)
```

Example:

```
_ = α-equiv "z" ("a" ↦ "b" ↦ ("c" ↦ ` "a" ‿ ` "c") ‿ ` "b" ‿ ` "a")
    ≡ just ("z" ↦ "b" ↦ ("c" ↦ ` "z" ‿ ` "c") ‿ ` "b" ‿ ` "z") ∋ refl
_ = α-equiv "b" ("a" ↦ "b" ↦ ("c" ↦ ` "a" ‿ ` "c") ‿ ` "b" ‿ ` "a")
    ≡ nothing ∋ refl -- Name clash
```

1.5 β-Reduction

Now we get to the crux of the matter: *β-reduction.* β-Reduction explains how we compute λ-terms, namely, if we apply a λ-abstraction to a λ-term, we can obtain a new λ-term by substituting the term we are applying to for the bound variable in the body of the abstraction. We will also add recursive calls for β-reduction when talking about expressions other than functions, so we will just reduce the first application we encounter.

```
β-reduc : λ-Term → Maybe λ-Term
β-reduc ((v ↦ e₁) ‿ e₂) = e₁ [ v := e₂ ] -- Main definition
β-reduc (` v) = nothing
β-reduc (v ↦ e) = map (v ↦_) (β-reduc e)
β-reduc (e₁ ‿ e₂) with β-reduc e₁
β-reduc (e₁ ‿ e₂) | nothing = map (e₁ ‿_) (β-reduc e₂)
β-reduc (e₁ ‿ e₂) | just e₁ ′ = just (e₁ ′ ‿ e₂)
```

Examples:

```
_ = β-reduc (("a" ↦ "b" ↦ ` "b") ‿ ` "x")
    ≡ just ("b" ↦ ` "b") ∋ refl
_ = β-reduc (("a" ↦ "b" ↦ ` "a") ‿ ` "x")
    ≡ just ("b" ↦ ` "x") ∋ refl
_ = β-reduc (("a" ↦ "b" ↦ ` "a") ‿ ` "b")
    ≡ nothing ∋ refl -- Name clash
```

1.6 η-Reduction

1.7 Equational Reasoning

```
data _→Mλ_ (e₁ : λ-Term) : Maybe λ-Term → 𝓤 where
  α       : {v : String} → e₁ →Mλ α-equiv v e₁
  β       : e₁ →Mλ β-reduc e₁
  λ-refl  : e₁ →Mλ just e₁
  λ-trans : {e₂ e₃ : λ-Term}
          → e₁ →Mλ just e₂
          → e₂ →Mλ just e₃
          → e₁ →Mλ just e₃

_→λ_ : λ-Term → λ-Term → 𝓤
e₁ →λ e₂ = e₁ →Mλ just e₂

step→λ : ∀ (x {y z} : λ-Term) → y →λ z → x →λ y → x →λ z
step→λ _ = flip λ-trans

syntax step→λ x y→λz x→λy = x →⟨ x→λy ⟩ y→λz

_■λ : (e : λ-Term) → e →λ e
e ■λ = λ-refl

infixr 5 step→λ
infix 6 _■λ
```

Example:

```
_ =         ("a" ↦ "b" ↦ ` "b") ‿ ` "x" ‿ ` "y"
    →⟨ β ⟩ ("b" ↦ ` "b") ‿ ` "y"
    →⟨ β ⟩ ` "y" ■λ
```

2 Syntax

2.1 Parser Combinators

2.2 Core Syntax

2.3 Extended Syntax

3 Combinatory Logic

4 Church Encodings