λ-Calculus in Agda

Joris Klaasse Bos

February 12, 2024

Preface

This document is a literate Agda program that implements and explains the λ-calculus. 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

The two most important subsections of this section are subsections 1.1 and 1.5. These two subsections define the terms of the λ-calculus and the one computation rule respectively. However, since we use named variables in our definition of λ-terms, and since the name of a variable should not matter for the meaning of a λ-term, we need to add some extra scaffolding to deal with these variable names and possible clashes, and spend some more time on the interpretation of a λ-term. There are other representations of λ-terms possible which do not suffer from these same setbacks, such as *de Bruijn representation*, but these are harder for humans to read. If you want to get a quick overview of λ-calculus, it would suffise to just read the main two subsections, but you will be missing some context which would be necessary to have a solid grasp of λ-calculus.

Strictly speaking we will not be defining the syntax of the λ-calculus in this section. Rather, we represent λ-terms as Agda data types. Agda syntax is quite flexible, so we can make it relatively legible, but really we will just be programming in the abstract syntax tree (AST) directly (for now). We will actually define the syntax in the next section (2).

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 : 𝒰 where
  `     : String → Λ
  _‿_   : Λ → Λ → Λ
  _↦_  : String → Λ → Λ
```

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

```
_ = Λ ∋ ((("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 ∋ ("a" ↦ "b" ↦ ` "a") ‿ ` "x" ‿ ` "y"
```

More examples (from the slides):

```
_ = Λ ∋ ` "v"
_ = Λ ∋ ` "v" ‿ ` "v"
_ = Λ ∋ ` "v" ‿ ` "v'"
_ = Λ ∋ "v" ↦ ` "v" ‿ ` "v'"
_ = Λ ∋ ` "v" ‿ ("v" ↦ ` "v" ‿ ` "v'")
```

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 → 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
-- From the slides:
_ = freeVars (` "x" ‿ (("x" ↦ ` "x" ‿ ` "y") ‿ ` "x")) ≡ "x" ∷ "y" ∷ "x" ∷ [] ∋ refl
```

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

```
boundVars : Λ → 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
-- From the slides:
_ = boundVars (` "x" ‿ (("x" ↦ ` "x" ‿ ` "y") ‿ ` "x")) ≡ "x" ∷ [] ∋ 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 a free one 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 to avoid name clashes later on. When a variable is bound multiple times, by convention we say it is bound by the inner most binding abstraction. If we are substituting a variable, but it gets rebound in a subexpression, we do not substitute it in that expression.

```
_[_:=_] : Λ → String → Λ → Maybe Λ
` 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    -- Would change the binding of a variable
      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
-- From the slides:
_ = ( ∀ {n : Λ} →
        ` "x" ‿ (("x" ↦ ` "x" ‿ ` "y") ‿ ` "x") ["x" := n ]
        ≡ just (n ‿ (("x" ↦ ` "x" ‿ ` "y") ‿ n))
    ) ∋ 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, and these restrictions avoid name clashes which would actually changing the meaning of a λ-term. 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 → Maybe Λ
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)
```

Examples:

```
_ = 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 terms other than abstractions so we will just reduce the first application we encounter.

```
β-reduc : Λ → Maybe Λ
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
-- From the slides:
_ = β-reduc (("x" ↦ ` "y" ‿ ` "x") ‿ (` "z" ‿ ` "z"))
  ≡ just (` "y" ‿ (` "z" ‿ ` "z")) ∋ refl
```

Now that we have given computational meaning to applications of λ-abstractions, we will interchangeably call λ-abstractions (λ-)functions, since now they actually "function". They are actually *anonymous pure unary functions*: we can write them without having to give them a name, they take only one input, and the output only depends on the input (which is true in mathematics per se, but not in computer science).

1.6 η-Reduction

η-Reduction is an optional second reduction rule some people choose to extend the λ-calculus with. We will discover its purpose through a little example.

Consider the following two lambda expressions:

```
_ = ` "a"
_ = "x" ↦ ` "a" ‿ ` "x"
```

These λ-terms are obviously not the same, nor can either be β-reduced. But what if we apply both to the same argument `` ` "b"``?

```
_ = ` "a" ‿ ` "b"
_ = β-reduc (("x" ↦ ` "a" ‿ ` "x") ‿ ` "b") ≡ just (` "a" ‿ ` "b") ∋ refl
```

They compute to the same expression `` ` "a" ‿ ` "b"``. In a sense, the first two expressions are the same, because they do the same. This is an instance of *function extensionality*. The second λ-term just wraps `` ` "a"`` in a function, but we cannot unwrap such a wrapped value, or “reduce” it in a sense. It may be desireable to be able to do this, and that is what η-reduction enables.

```
η-reduc : ∀ String → Λ → Maybe Λ
η-reduc x (v ↦ e ‿ ` v′) = -- Main definition
  if x =ₛ v ∧ x =ₛ v′ ∧ not (contains (freeVars e) v)
    then just e
    else map (λ e′ → v ↦ e′ ‿ ` v′) (η-reduc x e)
η-reduc x (v ↦ e) = map (v ↦_) (η-reduc x e)
η-reduc x (` v) = nothing
η-reduc x (e₁ ‿ e₂) with η-reduc x e₁
η-reduc x (e₁ ‿ e₂) | nothing = map (e₁ ‿_) (η-reduc x e₂)
η-reduc x (e₁ ‿ e₂) | just e₁′ = just (e₁′ ‿ e₂)
```

Like with α-equivalence and β-reduction we add recursive calls to apply it to the first subexpression where η-reduction applies.

Example:

```
_ = η-reduc "x" ("x" ↦ ` "y" ‿ ` "x") ≡ just (` "y") ∋ refl
```

1.7 Equational Reasoning

We will create a type which encodes the proposition that some λ-term is reducible to another. These reductions can be done by α-equivalence, β-reduction, or η-reduction. Since `α-equiv`, `β-reduc`, and `η-reduc` return `Maybe`'s, we will make the type have a `Maybe` in its right argument.

```
data _→Mλ_ : Λ → Maybe Λ → 𝓤 where
```

α-Equivalence and β-reduction:

```
α : ∀ {e : Λ} {v : String} → e →Mλ α-equiv v e
β : ∀ {e : Λ}              → e →Mλ β-reduc e
η : ∀ {e : Λ} (v : String) → e →Mλ η-reduc v e
```

We will add transitivity so we can chain reductions into one larger reduction, where we keep unwrapping the `Maybe` in between. This also means that we cannot form reductions using `nothing`'s. We also add reflexivity, so doing nothing is also a valid reduction. This will be useful when defining equational reasoning with the type.

```
λ-trans : ∀ {e₁ e₂ e₃ : Λ} → e₁ →Mλ just e₂ → e₂ →Mλ just e₃ → e₁ →Mλ just e₃
λ-refl  : ∀ {e₁ : Λ}       → e₁ →Mλ just e₁
```

With our definition thus far we can only apply α-equivalence and β-reduction to the first (sub)expression to which they are applicable (as that is how we have defined `α-equiv` and `β-reduc`) in *lazy evaluation order*. This is not so much a problem for β-reduction as it is for α-equivalence, since we might want to apply α-equivalence to a subexpression to show that the whole expression is α-equivalent to another—with β-reduction we can just repeatedly apply it until we are "done", where the order does not matter all that much (except if possible infinite computations are involved). To avoid this problem with α-equivalence, we will say that applying a reduction to a subexpression is also a valid reduction.

```
λ-left  : ∀ {e₁ e₂ e₃ : Λ}              → e₁ →Mλ just e₂ → (e₁ ‿ e₃) →Mλ just (e₂ ‿ e₃)
λ-right : ∀ {e₁ e₂ e₃ : Λ}              → e₁ →Mλ just e₂ → (e₃ ‿ e₁) →Mλ just (e₃ ‿ e₁)
λ-body  : ∀ {e₁ e₂ : Λ} {v : String} → e₁ →Mλ just e₂ → (v ↦ e₁) →Mλ just (v ↦ e₂)
```

This completes our definition of this type.

We will add a shorthand for `e₁ →Mλ just e₂`.

```
_→λ_ : Λ → Λ → 𝓤
e₁ →λ e₂ = e₁ →Mλ just e₂
```

We can now define equational reasoning analogously to equational reasoning with propositional equality in the Agda standard library.

```
infix  5 λ-begin
infixr 6 step→λ
infix  7 _λ-end

λ-begin : Λ → Λ
λ-begin = id

step→λ : ∀ (x {y z} : Λ) → 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-end : ∀ (e : Λ) → e →λ e
e λ-end = λ-refl
```

Examples:

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

_ = λ-begin     ("x" ↦ "y" ↦ "z" ↦ ` "x" ‿ ` "z" ‿ ` "y") ‿ ("x" ↦ "y" ↦ ` "x")
    →⟨ β ⟩      "y" ↦ "z" ↦ ("x" ↦ "y" ↦ ` "x") ‿ ` "z" ‿ ` "y"
    →⟨ β ⟩      "y" ↦ "z" ↦ ("y" ↦ ` "z") ‿ ` "y"
    →⟨ β ⟩      "y" ↦ "z" ↦ ` "z"
    →⟨ α {v = "x"} ⟩
                "x" ↦ "z" ↦ ` "z"
    →⟨ λ-body α ⟩
                "x" ↦ "y" ↦ ` "y"
    λ-end

_ = λ-begin     ("x" ↦ "y" ↦ "z" ↦ ` "x" ‿ ` "z" ‿ ` "y") ‿ ("x" ↦ "y" ↦ ` "z")
    →⟨ λ-left $ λ-body $ λ-body α ⟩
                ("x" ↦ "y" ↦ "w" ↦ ` "x" ‿ ` "w" ‿ ` "y") ‿ ("x" ↦ "y" ↦ ` "z")
    →⟨ β ⟩      "y" ↦ "w" ↦ ("x" ↦ "y" ↦ ` "z") ‿ ` "w" ‿ ` "y"
    →⟨ β ⟩      "y" ↦ "w" ↦ ("y" ↦ ` "z") ‿ ` "y"
    →⟨ β ⟩      "y" ↦ "w" ↦ ` "z"
    →⟨ α {v = "x"} ⟩
              "x" ↦ "w" ↦ ` "z"
    →⟨ λ-body α ⟩
              "x" ↦ "y" ↦ ` "z"
    λ-end

_ = λ-begin ("x" ↦ ` "y" ‿ ` "x") →⟨ η "x" ⟩ ` "y" λ-end
```

2 Syntax

2.1 Parser Combinators

2.2 Core Syntax

2.3 Extended Syntax

3 Combinatory Logic

3.1 Identity

3.2 Flipping with the Cardinal

3.3 Selection with the Kite and the Kestrel

3.4 Composition with the Blue- and Blackbirds

3.5 Storage with the Thrush and Vireo

3.6 SKI Combinator Calculus

4 Church Encodings

4.1 Church Booleans

4.1.1 Branching with True and False

4.1.2 Negation

4.1.3 Boolean And

4.1.4 Boolean Or

4.1.5 Boolean Xor

4.1.6 Boolean Equality

4.2 Church Numerals (1/2)

4.2.1 Definition

4.2.2 Successor

4.2.3 Addition

4.2.4 Multiplication

4.2.5 Exponentiation

4.3 Church Pairs and Lists

4.4 Church Numerals (2/2)