The Simply Typed λ-Calculus

(In Agda)

Jonathan Prieto-Cubides

Master in Applied Mathematics
Logic and Computation Group
Universidad EAFIT
Medellín, Colombia

1th June 2017

Contents

Definition

The set of λ-terms denoted by Λ is built up from a set of variables V using application and (function) abstraction.

$$
\begin{aligned}
x \in V &\Rightarrow x \in \Lambda, \\
M \in \Lambda, x \in V &\Rightarrow (\lambda x.M) \in \Lambda, \\
M, N \in \Lambda &\Rightarrow (M\,N) \in \Lambda, \\
M, N \in \Lambda, x \in V &\Rightarrow ((\lambda x.M)\,N) \in \Lambda.
\end{aligned}
$$

Definition

The set of λ-terms denoted by Λ is built up from a set of variables V using application and (function) abstraction.

$$
\begin{aligned}
x \in V &\Rightarrow x \in \Lambda, \\
M \in \Lambda, x \in V &\Rightarrow (\lambda x.M) \in \Lambda, \\
M, N \in \Lambda &\Rightarrow (M\,N) \in \Lambda, \\
M, N \in \Lambda, x \in V &\Rightarrow ((\lambda x.M)\,N) \in \Lambda.
\end{aligned}
$$

```
Name : Set
Name = String

data Expr : Set where
  var : Name → Expr
  lam : Name → Expr → Expr
  _•_ : Expr → Expr → Expr
```

Definition

- The set of types is noted with $\mathbb{T} = \text{Type}(\lambda \rightarrow)$.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T} \rightarrow \mathbb{T},$$

  where $\mathbb{V} = \{\alpha_1, \alpha_2, \cdots\}$ be a set of type variables, $\mathbb{B}$ stands for a collection of type constants for basic types like `Nat` or `Bool`

- A *statement* is of the form $M : \sigma$ with $M \in \Lambda$ and $\sigma \in \mathbb{T}$
- *Derivation* inference rules

$$\frac{M : \sigma \longrightarrow \tau \qquad N : \sigma}{M\,N : \tau} \qquad\qquad \frac{\dfrac{\dfrac{[x : \sigma]^{(1)}}{\vdots}}{M : \tau}}{\lambda x.M : \sigma \rightarrow \tau}\,(1)$$

- A statement $M : \sigma$ is derivable form a *basis* Γ denoted by $\Gamma \vdash M : \sigma$ where basis stands for be a set of statements with only distinct (term) variables as subjects

Syntax defintion based on (Érdi, 2013), and (Danielsson, n.d.)

- Typing syntax: $\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T} \to \mathbb{T}$,

Syntax defintion based on (Érdi, 2013), and (Danielsson, n.d.)

- Typing syntax: $\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T} \to \mathbb{T}$,

```
module Typing (U : Set) where

data Type : Set where
  base : U    → Type
  _→_  : Type → Type → Type
```

Syntax defintion based on (Érdi, 2013), and (Danielsson, n.d.)

- Typing syntax: $\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T} \to \mathbb{T}$,

```
module Typing (U : Set) where

data Type : Set where
  base : U    → Type
  _⇒_  : Type → Type → Type
```

- $\lambda \to$-Curry Syntax

```
module Syntax (Type : Set) where

open import Data.String

Name : Set
Name = String

-- Statements.
data Formal : Set where
  _∶_ : Name → Type → Formal

data Expr : Set where
  var : Name   → Expr
  lam : Formal → Expr → Expr
  _∙_ : Expr   → Expr → Expr
```

module Examples (Type : Set) where

```
open import Syntax Type

postulate
  A : Type

x = var "x"
y = var "y"
z = var "z"

-- I, K, S : Expr

I = lam ("x" : A) x                        -- λx.x, x ∈ A
K = lam ("x" : A) (lam ("y" : A) x)        -- λxy.x, x,y ∈ A
S =
  lam ("x" : A)
    (lam ("y" : A)
      (lam ("z" : A)
        ((x • z) • (y • z))))              -- λxyz.xz(yz), x,y,z ∈ A
```

Decibility of Type Assignment (Barendregt, Dekkers, and Statman, 2013)

Problem	**Question**
Typability	Given M does exists a $\sigma, \Gamma \vdash M : \sigma$?
Type-checking	Given M and $\tau, \Gamma \vdash M : \tau$?
Inhabitation	Given τ, does exists an M such that $\Gamma \vdash M : \sigma$?

Problem	**Question**
Typability	Given M does exists a $\sigma, \Gamma \vdash M : \sigma$?
Type-checking	Given M and $\tau, \Gamma \vdash M : \tau$?
Inhabitation	Given τ, does exists an M such that $\Gamma \vdash M : \sigma$?

Theorem

- *It is decidable whether a term is typable in $\lambda \rightarrow$.*
- *If a term M is typable in $\lambda \rightarrow$, then M has a principal type scheme, i.e. a type σ such that every possible type for M is a subsitution instance of σ. Moreover σ is computable from M.*

Theorem

Type checking for $\lambda \rightarrow$ is decidable.

- The indexes are natural numbers that represent the occurrences of the variable in a λ-term

$$\lambda x.\lambda y.x \rightsquigarrow \lambda\lambda 2$$

- The natural number denotes the number of binders that are in scope between that occurrence and its corresponding binder

$$\lambda x.\lambda y.\lambda z.xz(yz) \rightsquigarrow \lambda\lambda\lambda 31(21)$$

- Check for α-equivalence is the same as that for syntactic equality

module Bound (Type : Set) where

```
data Expr (n : ℕ) : Set where
  var : Fin n  → Expr n
  lam : Type   → Expr (suc n) → Expr n
  _•_ : Expr n → Expr n       → Expr n

Binder : ℕ → Set
Binder = Vec Name

data _⊢_⇝_ : ∀ {n} → Binder n → S.Expr → Expr n → Set where
  var-zero : ∀ {n x} {Γ : Binder n}
           → Γ , x ⊢ var x ⇝ var (# 0)

  var-suc  : ∀ {n x y k} {Γ : Binder n} {p : False (x ≟ y)}
           → Γ ⊢ var x ⇝ var k
           → Γ , y ⊢ var x ⇝ var (suc k)

  lam      : ∀ {n x τ t t′} {Γ : Binder n}
           → Γ , x ⊢ t ⇝ t′
           → Γ ⊢ lam (x : τ) t ⇝ lam τ t′

  _•_      : ∀ {n t₁ t₁′ t₂ t₂′} {Γ : Binder n}
           → Γ ⊢ t₁ ⇝ t₁′
           → Γ ⊢ t₂ ⇝ t₂′
           → Γ ⊢ t₁ • t₂ ⇝ t₁′ • t₂′
```

```
∅ : Binder 0
∅ = []

Γ : Binder 2
Γ = "x" ∷ "y" ∷ []

e1 : "x" ∷ "y" ∷ [] ⊢ var "x" ⇝ var (# 0)
e1 = var-zero

I : [] ⊢ lam ("x" : A) (var "x")
       ⇝ lam A (var (# 0))
I = lam var-zero

K : [] ⊢ lam ("x" : A) (lam ("y" : A) (var "x"))
       ⇝ lam A (lam A (var (# 1)))
K = lam (lam (var-suc var-zero))

K₂ : [] ⊢ lam ("x" : A) (lam ("y" : A) (var "y"))
        ⇝ lam A (lam A (var (# 0)))
K₂ = lam (lam var-zero)

P : Γ ⊢ lam ("x" : A) (lam ("y" : A) (lam ("z" : A) (var "x")))
      ⇝ lam A (lam A (lam A (var (# 2))))
P = {!!}
```

Scope-checking

```
name-dec : ∀ {n} {Γ : Binder n} {x y : Name} {t : Expr (suc n)}
         → Γ , y ⊢ var x ⇝ t
         → x ≡ y ⊎ ∃[ t′ ] (Γ ⊢ var x ⇝ t′)

⊢subst : ∀ {n} {x y} {Γ : Binder n} {t}
       → x ≡ y
       → Γ , x ⊢ var x ⇝ t
       → Γ , y ⊢ var x ⇝ t

find-name : ∀ {n}
          → (Γ : Binder n)
          → (x : Name)
          → Dec (∃[ t ] (Γ ⊢ var x ⇝ t))

check : ∀ {n}
      → (Γ : Binder n)
      → (t : S.Expr)
      → Dec (∃[ t′ ] (Γ ⊢ t ⇝ t′))

scope : (t : S.Expr) → {p : True (check [] t)} → Expr 0
scope t {p} = proj₁ (toWitness p)
```

```
postulate A : Type

I₁ : S.Expr
I₁ = S.lam ("x" : A) (S.var "x")

open import Data.Unit

I : Expr 0
I = scope I₁ {p = ⊤.tt}  -- Use C-C-C-n.

x, y, z : S.Expr

x = var "x"
y = var "y"
z = var "z"

S₁ : S.Expr
S₁ =
  lam ("x" : A)
    (lam ("y" : A)
    (lam ("z" : A)
    ((x • z) • (y • z))))

S : Expr 0
S = scope S₁ {p = ⊤.tt}
```

- Introduction

$$\frac{\Gamma(t) = \tau}{\Gamma \vdash t : \tau}$$

- Abstraction

$$\frac{\Gamma, \tau \vdash t : \tau'}{\Gamma \vdash \lambda(x : \tau).t : \tau \longrightarrow \tau'}$$

- Application

$$\frac{\Gamma \vdash t_1 : \tau \longrightarrow \tau' \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \bullet t_2 : \tau'}$$

module Typing (U : Set) where

```
open import Bound Type hiding (_,_)

Ctxt : ℕ → Set
Ctxt = Vec Type

_,_ : ∀ {n} → Ctxt n → Type → Ctxt (suc n)
Γ , x = x ∷ Γ

data _⊢_:_ : ∀ {n} → Ctxt n → Expr n → Type → Set where
  tVar : ∀ {n Γ} {x : Fin n}
       → Γ ⊢ var x : lookup x Γ

  tLam : ∀ {n} {Γ : Ctxt n} {t} {τ τ′}
       → Γ , τ ⊢ t : τ′
       → Γ ⊢ lam τ t : τ ⟶ τ′

  _•_  : ∀ {n} {Γ : Ctxt n} {t₁ t₂} {τ τ′}
       → Γ ⊢ t₁ : τ ⟶ τ′
       → Γ ⊢ t₂ : τ
       → Γ ⊢ t₁ • t₂ : τ′
```