The Simply Typed Lambda Calculus

(In Agda)

Jonathan Prieto-Cubides

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

1th June 2017

Contents

Lambda Calculus

Typed Lambda Calculus

Syntax Definitions

Decibility of Type Assignment

Well-Scoped Lambda Expressions

Typability and Type-checking

About

- The Agda source code of this talk is available in the repository

  https://github.com/jonaprieto/stlctalk

  and it was mostly based on the implementation by (Érdi, 2013) and (Danielsson, n.d.) of the simple lambda calculus
- Tested with Agda v$2.5.2$ and Agda Standard Library v0.13

Definition

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

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

- A simple syntax definition for lambda terms

```
Name : Set
Name = String

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

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

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

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

$$\frac{M : \sigma \rightarrowtail \tau \qquad N : \sigma}{M N : \tau}$$

$$\dfrac{\dfrac{\dfrac{[x : \sigma]^{(1)}}{\vdots}}{M : \tau}}{\lambda x.M : \sigma \rightarrowtail \tau}\,(1)$$

- 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

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

```
module Typing (U : Set) where

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

- A syntax definition including type annotations

```
module Syntax (Type : Set) where

open import Data.String

Name : Set
Name = String

data Formal : Set where
  _∶_ : Name → Type → Formal

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

Examples

```
open import Syntax Type

postulate A : Type

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

-- Combinators.
-- 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
```

Problem	**Question**
Typability	Given M does exists a σ such that $\Gamma \vdash M : \sigma$?
Type-checking	Given M and τ, can we have $\Gamma \vdash M : \tau$?
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
- A syntax definition using De Bruijn indexes

```
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′
```

Examples

```
∅ : 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 = {!!}   -- complete!!
```

module Scopecheck (Type : Set) where

```
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)
```

Examples

```
postulate A : Type

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

open import Data.Unit

I = scope I₁ {p = ⊤.tt}  -- Use C-C-C-n and check for I.

x, y, z : S.Expr
x = var "x"
y = var "y"
z = var "z"

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

S : Expr 0
S = scope S₁ {p = ⊤.tt}  -- Use C-C-C-n and check for S.
```

- Introduction

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

- Abstraction

$$\frac{\Gamma, \tau \vdash t : \sigma}{\Gamma \vdash \lambda\, \tau\, t : \tau \rightarrowtail \sigma}$$

- Application

$$\frac{\Gamma \vdash t_1 : \tau \rightarrowtail \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \bullet t_2 : \sigma}$$

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 σ}
       → Γ , τ ⊢ t : σ
       → Γ ⊢ lam τ t : τ ↣ σ

  _•_  : ∀ {n} {Γ : Ctxt n} {t₁ t₂} {τ σ}
       → Γ ⊢ t₁ : τ ↣ σ
       → Γ ⊢ t₂ : τ
       → Γ ⊢ t₁ • t₂ : σ
```

Examples

```
postulate
  Bool : Type

ex : [] , Bool ⊢ var (# 0) : Bool
ex = tVar

ex2 : [] ⊢ lam Bool (var (# 0)) : Bool ↣ Bool
ex2 = tLam tVar

postulate
  Word : Type
  Num  : Type

K : [] ⊢ lam Word (lam Num (var (# 1))) : Word ↣ Num ↣ Word
K = tLam (tLam tVar)
```

Equality Between Types

```
_T≟_ : (τ τ′ : Type) → Dec (τ ≡ τ′)
base A T≟ base B with A ≟ B
... | yes A≡B = yes (cong base A≡B)
... | no  A≢B = no (A≢B ∘ helper)
  where
    helper : base A ≡ base B → A ≡ B
    helper refl = refl
base A T≟ (_ ↣ _) = no (λ ())

(τ₁ ↣ τ₂) T≟ base B = no (λ ())
(τ₁ ↣ τ₂) T≟ (τ₁′ ↣ τ₂′) with τ₁ T≟ τ₁′
... | no  τ₁≢τ₁′ = no (τ₁≢τ₁′ ∘ helper)
  where
    helper :  τ₁ ↣ τ₂ ≡ τ₁′ ↣ τ₂′ → τ₁ ≡ τ₁′
    helper refl = refl
... | yes τ₁≡τ₁′
  with τ₂ T≟ τ₂′
...  | yes τ₂≡τ₂′ = yes (cong₂ _↣_ τ₁≡τ₁′ τ₂≡τ₂′)
...  | no  τ₂≢τ₂′ = no (τ₂≢τ₂′ ∘ helper)
  where
    helper : τ₁ ↣ τ₂ ≡ τ₁′ ↣ τ₂′ → τ₂ ≡ τ₂′
    helper refl = refl
```

```
-- Auxiliar Helper.
⊢-inj : ∀ {n Γ} {t : Expr n} → ∀ {τ σ}
      → Γ ⊢ t : τ
      → Γ ⊢ t : σ
      → τ ≡ σ

-- Var case.
⊢-inj tVar tVar = refl

-- Abstraction case.
⊢-inj {t = lam τ t} (tLam Γ,τ⊢t:τ′) (tLam Γ,τ⊢t:τ″)
  = cong (_↣_ τ) (⊢-inj Γ,τ⊢t:τ′ Γ,τ⊢t:τ″)

-- Application case.
⊢-inj (Γ⊢t₁:τ↣τ₂ • Γ⊢t₂:τ) (Γ⊢t₁:τ₁↣σ • Γ⊢t₂:τ₁)
  = helper (⊢-inj Γ⊢t₁:τ↣τ₂ Γ⊢t₁:τ₁↣σ)
  where
    helper : ∀ {τ τ₂ τ₁ σ} → (τ ↣ τ₂ ≡ τ₁ ↣ σ) → τ₂ ≡ σ
    helper refl = refl
```

```
infer : ∀ {n} Γ (t : Expr n) → Dec (∃[ τ ] (Γ ⊢ t : τ))

-- Var case.
infer Γ (var x) = yes (lookup x Γ -and- tVar)

-- Abstraction case.
infer Γ (lam τ t) with infer (τ ∷ Γ) t
... | yes (σ -and- Γ,τ⊢t:σ) = yes (τ ↣ σ -and- tLam Γ,τ⊢t:σ)
... | no  Γ,τ⊬t:σ = no helper
  where
    helper : ∄[ τ′ ] (Γ ⊢ lam τ t : τ′)
    helper (base A -and- ())
    helper (.τ ↣ σ -and- tLam Γ,τ⊢t:σ)
      = Γ,τ⊬t:σ (σ -and- Γ,τ⊢t:σ)
```

Typability II

```
-- Application case part I.
infer Γ (t₁ • t₂) with infer Γ t₁ | infer Γ t₂
... | no  ∄τ⟨Γ⊢t₁:τ⟩ | _ = no helper
    where
      helper : ∄[ σ ] (Γ ⊢ t₁ • t₂ : σ)
      helper (τ -and- Γ⊢t₁:τ • _)
        = ∄τ⟨Γ⊢t₁:τ⟩ (_ ↣ τ -and- Γ⊢t₁:τ)

... | yes (base x -and- Γ⊢t₁:base) | _ = no helper
    where
      helper : ∄[ σ ] (Γ ⊢ t₁ • t₂ : σ)
      helper (τ -and- Γ⊢t₁:_↣_ • _)
        with ⊢-inj Γ⊢t₁:_↣_ Γ⊢t₁:base
      ...  | ()
```

```
-- Application case part II.
... | yes (τ₁ ↣ τ₂ -and- Γ⊢t₁:τ₁↣τ₂) | no ∄τ⟨Γ⊢t₂:τ⟩ = no helper
    where
      helper : ∄[ σ ] (Γ ⊢ t₁ • t₂ : σ)
      helper (τ -and- Γ⊢t₁:τ₁′↣τ₂′ • Γ⊢t₂:τ)
        with ⊢-inj Γ⊢t₁:τ₁↣τ₂ Γ⊢t₁:τ₁′↣τ₂′
      ...   | refl = ∄τ⟨Γ⊢t₂:τ⟩ (τ₁ -and- Γ⊢t₂:τ)

... | yes (τ₁ ↣ τ₂ -and- Γ⊢t₁:τ₁↣τ₂) | yes (τ₁′ -and- Γ⊢t₂:τ₁′)
    with τ₁ T≟ τ₁′
...  | yes τ₁≡τ₁′ = yes (τ₂ -and- Γ⊢t₁:τ₁↣τ₂ • helper)
     where
       helper : Γ ⊢ t₂  : τ₁
       helper = subst (_⊢_:_ Γ t₂) (sym τ₁≡τ₁′) Γ⊢t₂:τ₁′
...  | no  τ₁≢τ₁′ = no helper
     where
       helper : ∄[ σ ] (Γ ⊢ t₁ • t₂ : σ)
       helper (_ -and- Γ⊢t₁:τ↣τ₂ • Γ⊢t₂:τ₁)
         with ⊢-inj  Γ⊢t₁:τ↣τ₂ Γ⊢t₁:τ₁↣τ₂
       ...   | refl = τ₁≢τ₁′ (⊢-inj Γ⊢t₂:τ₁ Γ⊢t₂:τ₁′)
```

```
check : ∀ {n} Γ (t : Expr n) → ∀ τ → Dec (Γ ⊢ t : τ)

-- Var case.
check Γ (var x) τ with lookup x Γ T≟ τ
... | yes refl = yes tVar
... | no ¬p    = no (¬p ∘ ⊢-inj tVar)

-- Abstraction case.
check Γ (lam τ t) (base A) = no (λ ())
check Γ (lam τ t) (τ₁ ⇾ τ₂) with τ₁ T≟ τ
... | no τ₁≢τ = no (τ₁≢τ ∘ helper)
    where
      helper : Γ ⊢ lam τ t : (τ₁ ⇾ τ₂) → τ₁ ≡ τ
      helper (tLam t) = refl

... | yes refl with check (τ ∷ Γ) t τ₂
...                | yes Γ,τ⊢t:τ₂ = yes (tLam Γ,τ⊢t:τ₂)
...                | no  Γ,τ⊬t:τ₂ = no helper
  where
    helper : ¬ Γ ⊢ lam τ t : τ ⇾ τ₂
    helper (tLam Γ,τ⊢t:_) = Γ,τ⊬t:τ₂ Γ,τ⊢t:_
```

Type-checking II

```
-- Application case.
check Γ (t₁ • t₂) σ with infer Γ t₂
... | yes (τ -and- Γ⊢t₂:τ)
    with check Γ t₁ (τ ⇒ σ)
...     | yes Γ⊢t₁:τ⇒σ = yes (Γ⊢t₁:τ⇒σ • Γ⊢t₂:τ)
...     | no  Γ⊬t₁:τ⇒σ = no helper
  where
    helper : ¬ Γ ⊢ t₁ • t₂ : σ
    helper (Γ⊢t₁:_⇒_ • Γ⊢t₂:τ′)
      with ⊢-inj Γ⊢t₂:τ Γ⊢t₂:τ′
    ...   | refl = Γ⊬t₁:τ⇒σ Γ⊢t₁:_⇒_

check Γ (t₁ • t₂) σ | no Γ⊬t₂:_ = no helper
  where
    helper : ¬ Γ ⊢ t₁ • t₂ : σ
    helper (_•_ {τ = σ} t Γ⊢t₂:τ′) = Γ⊬t₂:_ (σ -and- Γ⊢t₂:τ′)
```

References

- Barendregt, Henk, Wil Dekkers, and Richard Statman (2013). *Lambda calculus with types*. Cambridge University Press.
- Danielsson, Nils Anders. *Normalisation for the simply typed lambda calculus*.
- Érdi, Gergő (2013). *Simply Typed Lambda Calculus in Agda, Without Shortcuts*.