Bidirectional typing (WIP)

Typing for Simple type lambda calculus with implementation in Agda and Haskell

Jonathan Prieto-Cubides

heliaxdev/plt-seminar

September 30, 2021

Contents

Origins of lambda calculus

- Haskell Curry: untyped lambda-calculus as logical foundations (inconsistent).
- Alonzo Church: *Simple Theory of Types* (1936).

Definition

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

$$x \in V \Rightarrow x \in \Lambda,$$
$$M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda,$$
$$M, N \in \Lambda \Rightarrow (MN) \in \Lambda.$$

Lambda Curry system: STT

- *Type* formers of simple type theory $\lambda \rightarrow$:

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

  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 (e.g. `Nat` or `Bool`).
- A *statement* is of the form $M : \sigma$ with $M \in \Lambda$ and $\sigma \in \mathbb{T}$.
- A *derivation* rule is as in Natural deduction.

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

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

Cont.

- A *context*, often denoted by Γ, is a set of statements with only distinct (term) variables as subjects. A context can be empty or can be extended by adding new statements.
- A *type judgment* $\Gamma \vdash M : \sigma$ tells us that the statement $M : \sigma$ is *derivable* from a *context/basis* Γ.
- Derivation rules can be also presented using Church style (i.e. via type judgments).

Languages

$$
\begin{tikzcd}
\text{Concrete} \arrow[r, "\lambda(M{:}S).N"] \arrow[dr, dashed] & \text{Formal} \arrow[d, "\text{DeBrujin translation,scope checking}"] \\
 & \text{Internal} \arrow[d, "\text{typechecking,typeinference}"] \\
 & \mathbb{B}
\end{tikzcd}
$$

Figure: languages

Languages

$$
\begin{tikzcd}
\text{Concrete} \arrow[r, "\lambda(M{:}S).N"] \arrow[dr, dashed] & \text{Formal} \arrow[d, "\text{DeBrujin translation,scope checking}"] \\
& \text{Internal} \arrow[d, "\text{typechecking,typeinference}"] \\
& \mathbb{B}
\end{tikzcd}
$$

Figure: languages

```
Name : Set
Name = String
data Expr : Set where
  var : Name → Expr
  lam : Name → Expr → Expr
  _·_ : Expr → Expr → Expr
```

A modified syntax definition for STT, see (Érdi, 2013), and (Danielsson, n.d.)

- Typing syntax:

```
module Typing (U : Set) where

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

- This syntax definition includes a type annotation for bounded variables.

```
module Syntax (Type : Set) where

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

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

Quick 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
```

De Bruijn Index

- The indexes are natural numbers that represent the occurrences of the variable in its λ-term, e.g.

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

- The natural number denotes the number of binders that are in scope between the variable occurrence and its corresponding binder, e.g.

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

- α-equivalence is syntactic equality.
- Internal syntax:

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

Well-scoped Expressions with respect to a variable Binder.

Towards getting variable scope checking:

```
Binder : ℕ → Set
Binder = Vec Name

data _⊢_↠_ : forall {n} → Binder n → S.Expr → Expr n → Set where

  var-zero : forall {n x} {Γ : Binder n}
           → Γ , x ⊢ var x ↠ var (# 0)

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

  lam      : forall {n x τ t t′} {Γ : Binder n}
           → Γ , x ⊢ t ↠ t′
           → Γ ⊢ lam (x ∶ τ) t ↠ lam τ t′

  _•_      : forall {n t₁ t₁′ t₂ t₂′} {Γ : Binder n}
           → Γ ⊢ t₁ ↠ t₁′
           → Γ ⊢ t₂ ↠ t₂′
           → Γ ⊢ t₁ • t₂ ↠ t₁′ • t₂′
```

Examples

```
ϕ : Binder 0
ϕ = []

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)
```

Scope checking

Scope checking is meant to be a systematic way to determine if a given identifier is accessible at certain point in the program.
If we try to access a local variable declared in one function in another function, we should get an error message. This is because only variables declared in the current scope and in the open scopes containing the current scope are accessible.

```
check : forall {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)
```

Example

```
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.
```

Modes for bidirectional type checking

$$\boxed{\Gamma} \vdash \boxed{M} : \boxed{\tau}$$

$$\begin{tikzcd}
\text{Concrete} \arrow[r, "\lambda(M{:}S).N"] \arrow[dr, dashed] & \text{Formal} \arrow[d, "\text{DeBrujin translation,scope checking}"] \\
 & \text{Internal} \arrow[d, "\text{bidirectional typechecking}"] \\
 & \mathbb{B}
\end{tikzcd}$$

Type assignment decidability, see (H. Barendregt, Dekkers, and Statman, 2013)

Consider the following questions given meta variables Γ, M, and τ for a context, a term, and a type, respectively.

Problem	**Question**
Type checking	Given Γ, M, and τ, can we verify that $\Gamma \vdash M : \tau$?
Type inference	Given Γ, M, can we find τ such that $\Gamma \vdash M : \boxed{\tau}$?
Typing inference	Given M, can we find any scenario such that $\boxed{\Gamma} \vdash M : \boxed{\tau}$?
Program synthesis	Given Γ, τ, does it exist any M such that $\Gamma \vdash \boxed{M} : \sigma$?

In all these problems, the meta variables play a specific role. A variable can be either an input or an output. The status of each meta-variable is called its mode.

Theorem

- *It is decidable whether a term is typable in* $\lambda\to$ *or not.*
- *If a term* M *is typable in* $\lambda\to$*, then* M *has a principal type scheme, i.e. a type* σ *such that every possible type for* M *is a substitution instance of* σ*. Moreover* σ *is computable from* M*.*

Theorem

Type checking for $\lambda\to$ *is decidable.*

Known fact: Typability and type checking in System F are equivalent and undecidable, see Wells, 1999.

Typing Rules

- 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

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

Typing rules for Internal

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

Ctxt : ℕ → Set
Ctxt = Vec Type

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

data _⊢_:_ : forall {n} → Ctxt n → Expr n → Type → Set where

  tVar : forall {n Γ} {x : Fin n}
       → Γ ⊢ var x : lookup x Γ

  tLam : forall {n} {Γ : Ctxt n} {t} {τ σ}
       → Γ , τ ⊢ t : σ
       → Γ ⊢ lam τ t : τ ↠ σ

  _•_  : forall {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)
```

Type inference I

```
infer : forall {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)
```

Type inference 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
      ...  | ()
```

Type inference III

```
-- 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′)
```

Equality for 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
```

Type checking I

```
check : forall {n} Γ (t : Expr n) → forall 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′)
```

Designing bidirectional typing algorithms, see Dunfield and Krishnaswami, 2020

Bidirectional typing splits the typing judgment $\Gamma \vdash M : S$ into two judgments:

$$\begin{array}{ll} \textit{type checking} & \Gamma \vdash M \Longleftarrow S \\ \textit{type synthesis} & \Gamma \vdash M \Longrightarrow S \end{array}$$

Recall, type synthesis is also called *type inference* and the term M is the *subject* of these judgments.

Whether a given term is type synthesising or only type checkable is determined by the role it plays in the bidirectional typing rules.

- *Pfenning principle*:

  > *If the rule is an introduction rule, make the principal judgment checking and if the rule is an elimination rule, make the principal judgment synthesising.*

A *principal judgment* is the judgment containing the logical connective that is being introduced or eliminated.

- For introduction rules, the principal judgment is the conclusion.
- For the elimination rules, the principal judgment is the first premise.

- (not today) Mode-correctness, Completeness for annotatability, Size, and Annotation character.

More details of these criteria along with case studies can be found in Dunfield and Krishnaswami, 2020.

Bidirectional typing for STT

- Introduction

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

- Abstraction

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

- Application

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

Barendregt, Henk, Wil Dekkers, and Richard Statman (2013). *Lambda calculus with types*. Cambridge University Press.

Barendregt, Henk P. (1993). "Lambda Calculi with Types". In: *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*. USA: Oxford University Press, Inc., pp. 117–309.

Danielsson, Nils Anders (n.d.). *Normalisation for the simply typed lambda calculus*. URL: `http://www.cse.chalmers.se/~nad/listings/simply-typed/SimplyTyped.TypeSystem.html`.

Dunfield, Jana and Neel Krishnaswami (2020). "Bidirectional Typing". In: *CoRR* abs/1908.05839. URL: `http://arxiv.org/abs/1908.05839`.

Érdi, Gergő (May 2013). *Simply Typed Lambda Calculus in Agda, Without Shortcuts*. URL: `https://gergo.erdi.hu/blog/2013-05-01-simply_typed_lambda_calculus_in_agda,_without_shortcuts/`.

Wells, Joe B. (1999). "Typability and type checking in System F are equivalent and undecidable". In: *Annals of Pure and Applied Logic* 98.1-3, pp. 111–156.

To be continued…
Next time, a more detailed accounting for bidirectional typing recipe.
Next Next time, a case study: QTT.
Next Next time, a review of the Juvix bidirectional typechecker.