

λ -CALCULUS

SIMPLE TYPES AND THEIR EXTENSIONS

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2024

Institute of Information Science
Academia Sinica

SIMPLY TYPED λ -CALCULUS:

INTRODUCTION

While λ -calculus is expressive and computationally powerful, it is rather painful to write programs inside λ -calculus.

Function can be applied to an arbitrary term which can represent a Boolean value, a number, or even a function, so as a programming language it is not easy to see the **intention** of a program.

Therefore, we will consider a formal definition of a **typing judgement**

$$\Gamma \vdash t : A$$

which specifies the type A of a term t under a list of free (typed) variables, allowing us to *restrict the formation* of a valid term by typing.

SIMPLY TYPED λ -CALCULUS: STATICS

HIGHER-ORDER FUNCTION TYPE

Assume \mathbb{V} is a set of type variables different from variables in untyped λ -terms. (And suppress its existence from now on.)

Definition 1

The judgement $A : \text{Type}$ is defined inductively as follows.

$$\frac{}{X : \text{Type}} \text{ if } X \in \mathbb{V} \qquad \frac{A : \text{Type} \quad B : \text{Type}}{A \rightarrow B : \text{Type}}$$

where $A \rightarrow B$ represents a function type from A to B .

We say that A is a type if $A : \text{Type}$ is derivable.

The function type is **higher-order**, because

1. functions can be arguments of another function;
2. functions can be the result of a computation.

For example,

$(A_1 \rightarrow A_2) \rightarrow B$ a function type whose argument is of type $A_1 \rightarrow A_2$;

$A_1 \rightarrow (A_2 \rightarrow B)$ a function whose return type is $A_2 \rightarrow B$.

Following the convention of function application, we introduce the convention for the function type:

Convention

$$A_1 \rightarrow A_2 \rightarrow \dots A_n \quad := \quad A_1 \rightarrow (A_2 \rightarrow (\dots \rightarrow (A_{n-1} \rightarrow A_n) \dots))$$

Definition 2

A *typing context* Γ is a sequence

$$\Gamma \equiv x_1 : A_1, x_2 : A_2, \dots, x_n : A_n$$

of *distinct variables* x_i of type A_i .

Definition 3

The membership judgement $\Gamma \ni (x : A)$ is defined inductively:

$$\frac{}{\Gamma, x : A \ni x : A} \qquad \frac{\Gamma \ni x : A}{\Gamma, y : B \ni x : A}$$

We say that x of type A occurs in Γ if $\Gamma \ni (x : A)$ if derivable.

TYPING RULE – CURRY-STYLE TYPING SYSTEM

The implicit typing system for simply typed λ -calculus is defined by the following typing rules, i.e. inference rules with its conclusion a typing judgement:

$$\frac{}{\Gamma \vdash_i x : A} \text{ (var) } \quad \text{if } \Gamma \ni (x : A)$$

$$\frac{\Gamma, x : A \vdash_i t : B}{\Gamma \vdash_i \lambda x. t : A \rightarrow B} \text{ (abs)}$$

$$\frac{\Gamma \vdash_i t : A \rightarrow B \quad \Gamma \vdash_i u : A}{\Gamma \vdash_i t u : B} \text{ (app)}$$

We say that t is a **closed term** if $\vdash t : A$ is derivable.

N.B. Whether a term t has a typing derivation is a *property* of t .

A typing system is *syntax-directed* if it has *exactly* one typing rule for each term construct.

By being syntax-directed, every typing derivation can be inverted:

Lemma 4 (Typing inversion)

Suppose that $\Gamma \vdash_i t : A$ is derivable. Then,

$t \equiv x$ implies $x : A$ occurs in Γ .

$t \equiv \lambda x. t'$ implies $A = B \rightarrow C$ and $\Gamma, x : B \vdash_i u' : C$.

$t \equiv u v$ implies there is some B such that $\Gamma \vdash_i u : B \rightarrow A$ and $\Gamma \vdash_i v : B$.

This lemma is particularly useful when constructing a typing derivation by hand.

For any types A and B , the judgement $\vdash_i \lambda x y. x : A \rightarrow B \rightarrow A$ has a derivation

$$\frac{\frac{\frac{}{x : A, y : B \vdash_i x : A} \text{(var)}}{x : A \vdash_i \lambda y. x : B \rightarrow A} \text{(abs)}}{\vdash_i \lambda x y. x : A \rightarrow B \rightarrow A} \text{(abs)}$$

Therefore, $\lambda x y. x$ is a program of type $A \rightarrow B \rightarrow A$.

Derive the typing judgement

$$\vdash_i \lambda f g x. f x (g x) : (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$$

for every types A, B and C .

Can we answer the following questions algorithmically?

Type inference Given a context Γ and a term t , is there a type $?$ such that the typing judgement $\Gamma \vdash t : ?$ is derivable?

Type checking Given a context Γ , a type A , and a term t , is the typing judgement $\Gamma \vdash t : A$ derivable?

Typability is reducible to type checking problem of

$$x_0 : A \vdash \text{fst } x_0 \ t : A$$

Theorem 5

Type checking is decidable in simply typed λ -calculus.

PROGRAMMING IN SIMPLY TYPED λ -CALCULUS

The type of natural numbers is of the form

$$\text{nat}_A := (A \rightarrow A) \rightarrow A \rightarrow A$$

for every type A .

Church numerals

$$\mathbf{c}_n := \lambda f x. f^n x$$

$$\vdash \mathbf{c}_n : \text{nat}_A$$

Successor

$$\text{suc} := \lambda n f x. f (n f x)$$

$$\vdash \text{suc} : \text{nat}_A \rightarrow \text{nat}_A$$

Addition

$$\text{add} := \lambda n m f x. (m f) (n f x)$$
$$\vdash \text{add} : \text{nat}_A \rightarrow \text{nat}_A \rightarrow \text{nat}_A$$

Multiplication

$$\text{mul} := \lambda n m f x. (m (n f)) x$$
$$\vdash \text{mul} : \text{nat}_A \rightarrow \text{nat}_A \rightarrow \text{nat}_A$$

Conditional

$$\text{ifz} := \lambda n x y. n (\lambda z. x) y$$
$$\vdash \text{ifz} : ?$$

We can also define the type of Boolean values for each type variable as

$$\text{bool}_A := A \rightarrow A \rightarrow A$$

Boolean values

$$\text{true} := \lambda x y. x \quad \text{and} \quad \text{false} := \lambda x y. y$$

Conditional

$$\begin{aligned} \text{cond} &:= \lambda b x y. b x y \\ \vdash \text{cond} &: \text{bool}_A \rightarrow A \rightarrow A \rightarrow A \end{aligned}$$

EXERCISE

1. Define conjunction `and`, disjunction `or`, and negation `not` in simply typed λ -calculus.
2. Prove that `and`, `or`, and `not` are well-typed.

PROPERTIES OF SIMPLY TYPED λ -CALCULUS

“Well-typed programs cannot ‘go wrong’.”

—(Milner, 1978)

Preservation If $\Gamma \vdash t : A$ is derivable and $t \longrightarrow_{\beta} u$, then $\Gamma \vdash u : A$.

Progress If $\Gamma \vdash t : A$ is derivable, then either t is in *normal form* or there is u with $t \longrightarrow_{\beta} u$.

By combining the above two properties, we can extend the progress theorem to $\longrightarrow_{\beta}^*$: if $\Gamma \vdash t : A$ then $t \longrightarrow_{\beta}^* u$ for some $\Gamma \vdash u : A$ which is either reducible or in normal form.

The converse of preservation might not hold.

Lemma 6 (Typability of subterms)

Let t be a term with $\Gamma \vdash t : A$ derivable. Then, for every subterm t' of t there exists Γ' such that

$$\Gamma' \vdash t' : A'.$$

Recall that

1. $\mathbf{K}_1 = \lambda x y. x$

2. $\Omega = (\lambda x. x x) (\lambda x. x x)$

and $\mathbf{K}_1 (\lambda x. x) \Omega \longrightarrow_{\beta} \mathbf{I}$.

Ω is not typable, so $\mathbf{K}_1 \mathbf{I} \Omega$ is not typable.

PRESERVATION THEOREM

Weakening **If $\Gamma \vdash t : A$ and $x \notin \Gamma$, then $\Gamma, x : B \vdash t : A$.**

Substitution **If $\Gamma, x : A \vdash t : B$ and $\Gamma \vdash u : A$ then $\Gamma \vdash t[u/x] : B$.**

Corollary 7 (Variable renaming)

If $\Gamma, x : A \vdash t : B$ and $y \notin \text{dom}(\Gamma)$, then $\Gamma, y : A \vdash t[y/x] : B$ where $\text{dom}(\Gamma)$ denotes the set of variables which occur in Γ .

Theorem 8

For any t and u if $\Gamma \vdash t : A$ is derivable and $t \rightarrow_{\beta} u$, then $\Gamma \vdash u : A$.

Proof sketch.

By induction on both the derivation of $\Gamma \vdash t : A$ and $t \rightarrow_{\beta} u$. \square

N.B. The only non-trivial case is $\Gamma \vdash (\lambda x. t) u : B$ which needs the above results.

PROOF OF PRESERVATION THEOREM

Proof.

By induction on both the derivation of $\Gamma \vdash t : A$ and $t \rightarrow_{\beta} u$.

1. Suppose $\Gamma \vdash x : A$. However, $x \not\rightarrow_{\beta} u$ for any u . Therefore, it is vacuously true that $\Gamma \vdash u : A$.
2. Suppose $\Gamma \vdash \lambda x. t : A \rightarrow B$ and $\lambda x. t \rightarrow_{\beta} u$. Then, u must be $\lambda x. u'$ for some u' ; $\Gamma, x : A \vdash t : B$ and $t \rightarrow_{\beta} u'$ must be derivable. By induction hypothesis, $\Gamma, x : A \vdash u'$ is derivable, so is $\Gamma \vdash \lambda x. u' : A \rightarrow B$.
3. Suppose $\Gamma \vdash t u$. Then ...
4. ...



Theorem 9

If $\Gamma \vdash t : A$ is derivable, then t is in normal form or there is u with $t \rightarrow_{\beta} u$.

To prove the theorem, we would like to use induction on $\Gamma \vdash t : A$ again.

However, the fact that t is in normal form does not tell us much what t is. Can we characterise t syntactically?

The notion of normal form can be characterised syntactically:

Definition 10

Define judgements $\text{Neutral } t$ and $\text{Normal } u$ mutually by

$$\begin{array}{c}
 \frac{}{\text{Neutral } x} \\
 \\
 \frac{\text{Neutral } t \quad \text{Normal } u}{\text{Neutral } t u} \\
 \\
 \frac{\text{Normal } t}{\text{Normal } t} \\
 \\
 \frac{\text{Normal } u}{\text{Normal } \lambda x. u}
 \end{array}$$

Idea. $\text{Neutral } u$ and $\text{Normal } t$ are derivable iff

$$t \equiv x u_1 \cdots u_n \quad \text{and} \quad u \equiv \lambda x_1 \cdots x_n. x u_1 \cdots u_m.$$

That is, β -redex cannot exist in u if u is normal.

A term t has no β -reduction if and only if t is normal:

Lemma 11

Soundness *If $\text{Normal } t$ (resp. $\text{Neutral } t$) is derivable, then t is in normal form.*

Completeness *If t is in normal form, then $\text{Normal } t$ is derivable.*

Proof sketch.

Soundness By mutual induction on the derivation of $\text{Normal } t$ and $\text{Neutral } t$.

Completeness By induction on the formation of t .



Theorem 12

If $\Gamma \vdash t : A$ is derivable, then $\text{Normal } t$ or there is u with $t \longrightarrow_{\beta} u$.

Proof sketch.

By induction on the derivation of $\Gamma \vdash t : A$. □

The statement is trivial in classical logic, as a direct consequence of the Law of Excluded Middle.

Yet, the progress theorem can be proved constructively without LEM. What is the *computational meaning* of this theorem?

Definition 13

t is *weakly normalising* denoted by $t \Downarrow$ if

$$\frac{\text{Normal } t}{t \Downarrow}$$

$$\frac{t \longrightarrow_{\beta} u \quad u \Downarrow}{t \Downarrow}$$

That is, t is weakly normalising if there is a sequence

$$t \longrightarrow_{\beta} t_1 \longrightarrow_{\beta} t_2 \longrightarrow_{\beta} \dots u \not\longrightarrow_{\beta}$$

Theorem 14 (Weak normalisation)

Every term t with $\Gamma \vdash t : A$ is weakly normalising.

Definition 15

t is *strongly normalising* denoted by $t \Downarrow$ if

$$\frac{\forall u. (t \longrightarrow_{\beta} u \implies u \Downarrow)}{t \Downarrow}$$

Intuitively, *strong normalisation* says every sequence

$$t \longrightarrow_{\beta} t_1 \longrightarrow_{\beta} t_2 \cdots$$

terminates, but the definition builds the sequence backwards.

Theorem 16

Every term t with $\Gamma \vdash t : A$ is strongly normalising.

EXTENSIONS TO SIMPLY TYPED λ -CALCULUS

Self-applicative term cannot be typed in simply typed λ -calculus.

E.g.,

$$\lambda x. x x$$

cannot be typed, since $A \rightarrow A$ is not equal to A . Hence, the Y -combinator in untyped λ -calculus cannot be typed.

A construct is introduced explicitly for general recursion:

Let $\Lambda_{\text{fix}}(V)$ be the set of terms defined with an additional construct:

fixpoint $\text{fix } f.t$ is a term in $\Lambda_{\text{fix}}(V)$, if $t \in \Lambda_{\text{fix}}(V)$ and $f \in V$

An additional typing rule is added to simply typed λ -calculus:

$$\frac{\Gamma, f : A \vdash_i t : A}{\Gamma \vdash_i \text{fix } f.t : A}$$

β -reduction for the general recursion `fix` is extended with the relation

$$\text{fix } x.t \longrightarrow_{\beta} t[\text{fix } x.t/x]$$

A term which never terminates can be defined easily.

$$\begin{aligned} \text{fix } x.x & \longrightarrow_{\beta} x[\text{fix } x.x/x] \\ \equiv \text{fix } x.x & \longrightarrow_{\beta} x[\text{fix } x.x/x] \\ \equiv \text{fix } x.x & \longrightarrow_{\beta} x[\text{fix } x.x/x] \\ \equiv \dots \end{aligned}$$

Other notions such as $=_{\alpha}$, \longrightarrow_{β} , and **FV** are extended similarly.

While Church numerals can have multiple types nat_A , for any A , we extend the calculus with a single type of natural numbers instead.

Let $\Lambda_{\text{fix},\mathbb{N}}(V)$ be the set of terms defined with additional constructs:

- **zero** is a term in $\Lambda_{\text{fix},\mathbb{N}}(V)$
- **suc**(t) is a term in $\Lambda_{\text{fix},\mathbb{N}}(V)$ if t is
- **ifz**($t; x. u; v$) is a term in $\Lambda_{\text{fix},\mathbb{N}}(V)$ if $t, u, v \in \Lambda_{\text{fix},\mathbb{N}}(V)$ and $x \in V$

with additional typing rules

$$\begin{array}{c}
 \hline
 \Gamma \vdash \text{zero} : \mathbb{N} \\
 \hline
 \end{array}
 \qquad
 \begin{array}{c}
 \Gamma \vdash t : \mathbb{N} \\
 \hline
 \Gamma \vdash \text{suc}(t) : \mathbb{N}
 \end{array}$$

$$\frac{\Gamma \vdash v : \mathbb{N} \quad \Gamma \vdash t : A \quad \Gamma, x : \mathbb{N} \vdash u : A}{\Gamma \vdash \text{ifz}(t; x. u; v) : A}$$

The third rule is akin to pattern matching on natural numbers.

β -reduction for natural numbers is extended with two rules:

$$\begin{aligned}\text{ifz}(t; x. u; \text{zero}) &\longrightarrow_{\beta} t \\ \text{ifz}(t; x. u; \text{suc}(n)) &\longrightarrow_{\beta} u[n/x]\end{aligned}$$

Define the predecessor of natural numbers as a program

$$\text{pred} : \mathbb{N} \rightarrow \mathbb{N}.$$

Evaluate the following terms to their normal forms.

1. pred zero
2. $\text{pred (suc (suc (suc zero)))}$

Extend simply typed λ -calculus $\Lambda_{\text{fix},\mathbf{N}}(V)$ further with a type of Boolean values.

1. What term constructs are needed?
2. What typing rules should be added?
3. How β -reduction should be updated?
4. Define Boolean operations, i.e. conjunction, disjunction, and negation, in this extension.

HOMEWORK

1. (2.5%) Complete the proof of the Preservation Theorem.
2. (5%) Show the Progress Theorem.
3. (2.5%) Show that if t is in normal form then `Normal` t is derivable.
4. (5%) Extend $\Lambda_{\text{fix},N}(V)$ further with product types $A \times B$, for any A and B where additional constructs should include pairs (t, u) and a construct to pattern match on a pair.