Lambda Calculus and Types

Higher-Order Functions

陳亮廷 Chen, Liang-Ting

2018 邏輯、語言與計算暑期研習營
Formosan Summer School on
Logic, Language, and Computation

Swansea University, UK

Type-driven Development

1. Untyped lambda calculus is as powerful as Turing machine—every computable function on natural numbers can be expressed.
2. It is difficult to see the intention of λ-terms.
3. Let types guide you how to design programs.

Implicit typing types are reconstructed from λ-terms

Explicit typing (preferable) λ-terms are annotated with types.

Simply Typed Lambda Calculus

Implicit Typing i

For an implicit typing system, types are introduced separately.

Definition 1

Given type variables $\mathbb{V}$, the set $\mathbb{T}$ of *types* is defined by

$$\frac{B \in \mathbb{V}}{B \in \mathbb{T}} \qquad \frac{\sigma \in \mathbb{T} \qquad \tau \in \mathbb{T}}{\sigma \to \tau \in \mathbb{T}}\ (\text{fun})$$

where $\sigma \to \tau$ is the type of functions from σ to τ.

N.B. If $\mathbb{V}$ is empty, then $\mathbb{T}$ is also empty. Why?

In line with the convention for application,

$$\sigma_1 \to \sigma_2 \to \ldots \sigma_n \quad := \quad \sigma_1 \to (\sigma_2 \to (\cdots \to (\sigma_{n-1} \to \sigma_n) \ldots))$$

Implicit Typing ii

Definition 2

That *M has type* σ is denoted by

$$M : \sigma$$

Given distinct variables x_i's, a *typing context* Γ is a set

$$\Gamma = \{x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n\}$$

Convention

$$\Gamma, x : \sigma \quad \text{stands for} \quad \Gamma \cup \{x : \sigma\}.$$

Implicit Typing iii

Definition 3 (Typing judgement)

A *typing judgement* is a context Γ with $M : \sigma$ denoted by

$$\Gamma \vdash M : \sigma$$

Definition 4 (Typing rules)

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (\text{var}) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.\ M : (\sigma \to \tau)}\ (\text{abs})$$

$$\frac{\Gamma \vdash M : (\sigma \to \tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}\ (\text{app})$$

Typing Derivation i

- A *typing derivation* of a typing judgement $\Gamma \vdash M : \sigma$ is a tree of instances of typing rules whose root is $\Gamma \vdash M : \sigma$.
- A typing judgement is *derivable* if there is a typing derivation of that judgement.

Example 5

The judgement $\vdash \lambda x.\, x : \sigma \to \sigma$, for all $\sigma \in \mathbb{T}$, can be derived by

$$\dfrac{\dfrac{}{x : \sigma \vdash x : \sigma}\,(\text{var})}{\vdash \lambda x.\, x : (\sigma \to \sigma)}\,(\text{abs})$$

Typing Derivation ii

Example 6

The judgement $\vdash \lambda x\, y.\, x : \sigma \to \tau \to \sigma$ for all $\sigma, \tau \in \mathbb{T}$, can be derived by

$$\dfrac{\dfrac{\dfrac{}{x : \sigma, y : \tau \vdash x : \sigma}\,(\text{var})}{x : \sigma \vdash \lambda y.\, x : \tau \to \sigma}\,(\text{abs})}{\vdash \lambda x\, y.\, x : \sigma \to \tau \to \sigma}\,(\text{app})$$

Exercise

Derive $\vdash \lambda f\, g\, x.\, f\, x\, (g\, x) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$ for all $\sigma, \tau, \rho \in \mathbb{T}$.

Typing Derivation iii

Example 7

Not every λ-term has a type. For example,

$$\lambda x.\, x\, x$$

does not have a type, since $\sigma \to \sigma$ is not equal to σ.

Exercise

Describe all possible types for Church numeral $\mathbf{c}_n$. Hint. You may find $\mathbf{c}_0$ and $\mathbf{c}_1$ confusing.

Explicit typing i

Definition 8 (Typed terms)

The set $\Lambda_{\mathbb{T}}$ of typed λ-terms is defined by

$$\frac{x \in V}{x \in \Lambda_{\mathbb{T}}} \qquad \frac{M \in \Lambda_{\mathbb{T}} \qquad x \in V \qquad \tau \in \mathbb{T}}{\lambda x : \tau.\, M \in \Lambda_{\mathbb{T}}}$$

$$\frac{M \in \Lambda_{\mathbb{T}} \qquad N \in \Lambda_{\mathbb{T}}}{(M\,N) \in \Lambda_{\mathbb{T}}}$$

Substitution and $\alpha\beta\eta$-conversions are defined in a similar way.

Explicit typing ii

Definition 9 (Typing Rules)

A typing derivation on *typed terms* is defined by

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (\text{var}) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\, M : \sigma \to \tau}\ (\text{abs})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}\ (\text{app})$$

A typed λ-terms M with a derivable judgement $\Gamma \vdash M : \sigma$ is well-typed.

Explicit typing iii

Proposition 10

For every typed term M, context Γ, and types σ, τ,

$$\Gamma \vdash M : \sigma \quad \textit{and} \quad \Gamma \vdash M : \tau \implies \sigma = \tau$$

Exercise

Show that $\lambda(x : \sigma).\, x\, x$ is a typed λ-term but there exists no type τ with

$$\vdash \lambda(x : \sigma).\, x\, x : \tau$$

From Typed Terms to Untyped Terms

An *erasing map* $|-|\colon \Lambda_{\mathbb{T}} \to \Lambda$ is defined by

$$\begin{aligned} |x| &= x \\ |(M\ N)| &= (|M|\ |N|) \\ |\lambda(x:\sigma).\, M| &= \lambda x.\, |M| \end{aligned}$$

Proposition 11

Let M and N be typed λ-terms in $\Lambda_{\mathbb{T}}$. Then,

$$\Gamma \vdash M : \sigma \textit{ implies } \Gamma \vdash |M| : \sigma$$

$$M \longrightarrow_{\beta *} N \textit{ implies } |M| \longrightarrow_{\beta *} |N|$$

From Untyped Terms to Typed Terms

Proposition 12

Let M and N be λ-terms in Λ.

1. *If $\Gamma \vdash M : \sigma$, then there is a typed term M' with*

$$|M'| = M \quad \textit{and} \quad \Gamma \vdash M' : \sigma$$

2. *If $M \longrightarrow_{\beta *} N$ and $M = |M'|$ for some typed λ-term M', then there exists N' with $|N'| = N$ and $M' \longrightarrow_{\beta *} N'$.*

Homework

Prove Propositions 11 and 12 by induction.

Type Inference

Typability: Is there a type σ with $\vdash M : \sigma$?

Type checking: Does $\vdash M : \sigma$ hold for a given type σ?

Both questions are positive.

Theorem 13

Typability and type checking are both decidable *in simply typed λ-calculus.*

Properties of Simply Typed Lambda Calculus

Type Safety

"Well-typed programs do not go wrong!"

Theorem 14 (Preservation Theorem)

Let M and N be typed λ-terms. If $M \longrightarrow_{\beta 1} N$, then

$$\Gamma \vdash M : \sigma \text{ implies } \Gamma \vdash N : \sigma$$

"Well-typed programs never get stuck!"

Theorem 15 (Progress Theorem)

If $\vdash M : \sigma$ for $M \in \Lambda_{\mathbb{T}}$, then either M is a value or there is N with $M \longrightarrow_{\beta 1} N$.

The evaluation of well-typed programs always produces a value of the same type if terminates.

Converse of Preservation i

The converse of Preservation Theorem does not hold.

Example 16

Recall that

1. $\mathsf{I} = \lambda x.\, x$
2. $\mathsf{K}_1 = \lambda x\, y.\, x$
3. $\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$

and $\mathsf{K}_1\, \mathsf{I}\, \Omega \longrightarrow_{\beta *} \mathsf{I}$. However,

$$\vdash \mathsf{I} : \sigma \to \sigma \not\Longleftrightarrow \vdash \mathsf{K}_1\, \mathsf{I}\, \Omega : \sigma \to \sigma.$$

Converse of Preservation ii

To see why the latter judgement fails, we need the following:

Lemma 17 (Typability of subterms)

Let M be a well-typed λ-term and σ a type under the context Γ. Then, for every subterm M' of M there exists Γ' such that

$$\Gamma' \vdash M' : \sigma'.$$

Proof.

By induction on $\Gamma \vdash M : \sigma$. □

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

Proving Preservation Theorem i

To prove type preservation, we need a few lemmas.

Let $\mathrm{dom}(\Gamma)$ denote the set of variables which occur in Γ.

Lemma 18 (Weakening)

If $\Gamma \vdash M : \tau$ *and* $x \notin \Gamma$, *then* $\Gamma, x : \sigma \vdash M : \tau$.

Lemma 19 (Substitution)

If $\Gamma, x : \tau \vdash M : \sigma$ *and* $\Gamma \vdash N : \tau$ *then* $\Gamma \vdash M[x \mapsto N] : \sigma$.

Corollary 20

If $\Gamma, x : \tau \vdash M : \sigma$ *and* x' *is not in* $\mathrm{dom}(\Gamma)$, *then*
$\Gamma, x' : \tau \vdash M[x \mapsto x'] : \sigma$.

Proving Preservation Theorem ii

Proof.

Since x' is not in Γ, it follows that

$$\Gamma, x' : \tau, x : \tau \vdash M$$

by Weakening Lemma and also by definition $\Gamma, x' : \tau \vdash x' : \tau$. Therefore, by Substitution Lemma, it follows that

$$\Gamma, x' : \tau \vdash M[x \mapsto x'] : \sigma$$

□

Proving Preservation Theorem iii

Homework

Prove Preservation Theorem by induction. Use Substitution Lemma if applicable.

Progress Theorem

A *value* is a λ-abstraction.

Theorem 21 (Progress Theorem)

If $\vdash M : \sigma$ *for* $M \in \Lambda_{\mathbb{T}}$, *then either* M *is a value or there is* N *with* $M \longrightarrow_{\beta 1} N$.

Proof.

By induction on typing derivations. $\square$

Strong Normalisation

Another property of type system is called *strong normalisation*. It means that every well-typed terms terminates eventually.

Theorem 22

Suppose that $\vdash M : \sigma$. *Then, there is no infinite sequence of reductions*

$$M \longrightarrow_{\beta 1} M_1 \longrightarrow_{\beta 1} \ldots \longrightarrow_{\beta 1} \ldots.$$

Corollary 23

Every well-typed λ*-term* $\vdash M : \sigma$ *has a normal form.*

Programming in Simply Typed Lambda Calculus

Church Encodings of Natural Numbers i

The type of natural numbers is of the form

$$\mathtt{nat}_\tau := (\tau \to \tau) \to \tau \to \tau$$

for every type $\tau \in \mathbb{T}$.

Church numerals

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

$$\vdash \mathsf{c}_n : \mathtt{nat}_\tau$$

Church Encodings of Natural Numbers ii

Successor

$$\begin{aligned} &\mathtt{suc} := \lambda n\, f\, x\,.\, f\,(n\, f\, x) \\ \vdash\ &\mathtt{suc} : \mathtt{nat}_\tau \to \mathtt{nat}_\tau \end{aligned}$$

Addition

$$\begin{aligned} &\mathtt{add} := \lambda n\, m\, f\, x.\ (m\ f)\ (n\ f\ x) \\ \vdash\ &\mathtt{add} : \mathtt{nat}_\tau \to \mathtt{nat}_\tau \to \mathtt{nat}_\tau \end{aligned}$$

Muliplication

$$\begin{aligned} &\mathtt{mul} := \lambda n\, m\, f\, x.\ (m\ (n\ f))\ x \\ \vdash\ &\mathtt{mul} : \mathtt{nat}_\tau \to \mathtt{nat}_\tau \to \mathtt{nat}_\tau \end{aligned}$$

Church Encodings of Natural Numbers iii

Conditional

$$\texttt{ifz} := \lambda n\, x\, y.\, n\ (\lambda z.\, x)\ y$$

$$\vdash \texttt{ifz} : ?$$

The type of `ifz` may not be as obvious as you may expect. Try to find one as general as possible and justify your guess.

Church Encodings of Boolean values

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

$$\texttt{bool}_\tau := \tau \to \tau \to \tau$$

Boolean values

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

Conditional

$$\texttt{cond} := \lambda b\, x\, y.\, b\, x\, y$$

$$\vdash \texttt{cond} : \texttt{bool}_\tau \to \tau \to \tau \to \tau$$

Exercise

Define conjunction `and`, disjunction `or`, and negation `not` in simply typed lambda calculus.

Definability

A function $f\colon \mathbb{N}^k \to \mathbb{N}$ is called *$\lambda_\to$-definable* if there is a λ-term F of type $\mathtt{nat} \to \mathtt{nat} \to \ldots \mathtt{nat} \to \mathtt{nat}$ such that

$$F\,\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k} \longrightarrow_{\beta *} \mathbf{c}_{f(n_1,\ldots,n_k)}$$

for every sequence $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$.

Theorem 24 (Schwichtenberg [Sch76])

The $\lambda_\to$-definable functions are the class of functions of the form $f\colon \mathbb{N}^k \to \mathbb{N}$ closed under compositions and containing the constant functions, projections, additions, multiplications, and the conditional

$$\mathrm{ifz}(n_0, n_1, n_2) = \begin{cases} n_1 & \text{if } n_0 = 0 \\ n_2 & \text{otherwise.} \end{cases}$$

Remark

With the decidability of type checking, Preservation Theorem, Progress Theorem (a well-typed term is either a “value” or a reducible term), and the strong normalisation, we actually have exhibited a decidable evaluator of simply typed lambda calculus that always reduce a well-typed term of type σ to a value of type σ.

References i

H. Schwichtenberg, *Definierbare funktionen im lambda-kalkul mit typen*, Arch. Logik Grundlagenforsch. **17** (1976), 113–114.