

Lambda Calculus and Types

Parametric Polymorphism

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

Polymorphism

Abstraction principle [Pie02]:

Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

In Haskell

```
head :: [a] -> a  
head (x:_) = x
```

System F—Polymorphic Lambda Calculus

Types

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

$$\frac{t \in \mathbb{V}}{t \in \mathbb{T}}$$

$$\frac{\sigma \in \mathbb{T} \quad t \in \mathbb{V}}{\forall t. \sigma \in \mathbb{T}} \text{ (poly)}$$

$$\frac{\sigma \in \mathbb{T} \quad \tau \in \mathbb{T}}{\sigma \rightarrow \tau \in \mathbb{T}}$$

For example, the type $\forall t. t \rightarrow t$ corresponds to a type variable universally quantified in GHC with extension **RankNTType**

`forall a. a -> a`

or simply

`a -> a`

Free and Bound Variables, again

Definition 1

The *free variable* $\mathbf{FV}: \mathbb{T} \rightarrow \mathcal{P}\mathbb{V}$ of a type is defined by

$$\mathbf{FV}(t) = t$$

$$\mathbf{FV}(\sigma \rightarrow \tau) = \mathbf{FV}(\sigma) \cup \mathbf{FV}(\tau)$$

$$\mathbf{FV}(\forall t. \sigma) = \mathbf{FV}(\sigma) - \{t\}$$

For convenience, the function extends to contexts:

$$\mathbf{FV}(\Gamma) = \{ t \in \mathbb{V} \mid \exists (x : \sigma) \in \Gamma \wedge t \in \mathbf{FV}(\sigma) \}.$$

1. $\mathbf{FV}(t_1) = \{t_1\}$.
2. $\mathbf{FV}(\forall t. (t \rightarrow t) \rightarrow t \rightarrow t) = \emptyset$.
3. $\mathbf{FV}(x : t_1, y : t_2, z : \forall t. t) = \{t_1, t_2\}$.

Capture-Avoiding Substitution for Type

Definition 2

The (*capture-avoidance*) *substitution* of a type ρ for the free occurrence of a type variable t is defined by

$$t[t \mapsto \rho] = \rho$$

$$u[t \mapsto \rho] = u \quad \text{if } u \neq t$$

$$(\sigma \rightarrow \tau)[t \mapsto \rho] = \sigma[t \mapsto \rho] \rightarrow \tau[t \mapsto \rho]$$

$$(\forall t. \sigma)[t \mapsto \rho] = \forall t. \sigma$$

$$(\forall u. \sigma)[t \mapsto \rho] = \forall u. \sigma[t \mapsto \rho] \quad \text{if } u \neq t, u \notin \mathbf{FV}(\rho)$$

Recall that $u \notin \mathbf{FV}(\rho)$ means that u is *fresh* for ρ .

Definition 3

Given a set V of variables, a set \mathbb{V} of type variables, the set terms for System F is defined by

$$\frac{x \in V}{x \in \mathbf{Term}}$$

$$\frac{M \in \mathbf{Term} \quad t \in \mathbb{V}}{\lambda t. M \in \mathbf{Term}}$$

$$\frac{M \in \mathbf{Term} \quad N \in \mathbf{Term}}{(M N) \in \mathbf{Term}}$$

$$\frac{M \in \mathbf{Term} \quad x \in V \quad \tau \in \mathbb{T}}{\lambda(x : \tau). M \in \mathbf{Term}}$$

$$\frac{M \in \mathbf{Term} \quad \tau \in \mathbb{T}}{(M \tau) \in \mathbf{Term}}$$

Typed λ -terms ii

GHC with the extension `ScopedTypeVariables` allows you to use type variables.

```
f :: forall a. [a] -> [a]
f = \(xs :: [a]) -> reverse xs
```

GHC with the extension `TypeApplications` allows you to apply a type argument.

```
Prelude> :t id
id :: a -> a
Prelude> :t id @Int
id @Int :: Int -> Int
```


Typed λ -terms iii

Can you see the corresponding untyped/simply typed λ -terms of the following λ -terms in System F?

1. $\Lambda t. \lambda(x : t). x$
2. $\Lambda t. \lambda(x : t)(y : t). x$
3. $(\Lambda t. \lambda(x : t)(y : t). x) \sigma M N$ where M, N are terms and σ a type.
4. $\Lambda t. \lambda(f : t \rightarrow t)(x : t). f (f x)$

System F is more expressive than simply typed lambda calculus.

Typing Rules

The typing rule for simply typed lambda calculus are extended to accommodate polymorphism:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, t \vdash M : \sigma}{\Gamma \vdash \Lambda t. M : \forall t. \sigma} \text{ (\forall-intro)}$$

$$\frac{\Gamma \vdash M : (\sigma \rightarrow \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M N) : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda(x : \sigma). M : (\sigma \rightarrow \tau)}$$

$$\frac{\Gamma \vdash M : \forall v. \sigma}{\Gamma \vdash (M \tau) : \sigma[v \mapsto \tau]} \text{ (\forall-elim)}$$

Typing Derivation

The typing judgement $\vdash \Lambda t. \lambda(x : t)(y : t). x : \forall t. t \rightarrow t \rightarrow t$ is derivable from the following derivation:

$$\frac{\frac{\frac{\overline{x : t, y : t \vdash x : t} \text{ (var)}}{x : t \vdash \lambda(y : t). x : t \rightarrow t}}{\vdash \lambda(x : t)(y : t). x : t \rightarrow t \rightarrow t}}{\vdash \Lambda t. \lambda(x : t)(y : t). x : \forall t. t \rightarrow t \rightarrow t}$$

Exercise

Derive the following judgements:

1. $\vdash \Lambda t. \lambda(x : t). x : \forall t. t \rightarrow t$
2. $a : \sigma, b : \sigma \vdash (\Lambda t. \lambda(x : t)(y : t). x) \sigma a b : \sigma$
3. $\vdash \Lambda t. \lambda(f : t \rightarrow t)(x : t). f(f x) : \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$

Substitution for Term

In line with the new syntax, we need to extend *substitution over terms*. Pretty obvious and routine.

Homework

Define *capture-avoiding substitution* for terms of System F.

Evaluation

The β -conversion has two rules

$$(\lambda(x : \tau). M) N \longrightarrow_{\beta} M[x \mapsto N] \quad \text{and} \quad (\Lambda t. M) \tau \longrightarrow_{\beta} M[t \mapsto \tau]$$

The full β -reduction is a relation on λ -terms defined by

$$\frac{M_1 \longrightarrow_{\beta} M_2}{M_1 \longrightarrow_{\beta 1} M_2}$$

$$\frac{M_1 \longrightarrow_{\beta 1} M_2}{M_1 N \longrightarrow_{\beta 1} M_2 N}$$

$$\frac{M_1 \longrightarrow_{\beta 1} M_2}{\lambda(x : \tau). M_1 \longrightarrow_{\beta 1} \lambda(x : \tau). M_2}$$

$$\frac{N_1 \longrightarrow_{\beta 1} N_2}{M N_1 \longrightarrow_{\beta 1} M N_2}$$

If $M \longrightarrow_{\beta 1} N$, then M and N **denotes** the same value. So,

$$M =_{\beta} N$$

where $=_{\beta}$ is the congruence and equivalence closure of \longrightarrow_{β} .

Programming in System F

Self Application

Self-application is not typable in simply typed lambda calculus.

$$\lambda(x : t). x x$$

However, self-application is possible in System F.

$$\lambda(x : \forall t. t \rightarrow t). x (\forall t. t \rightarrow t) x$$

Church Encoding of Bottom Type i

Definition 4

The bottom type is a type defined by

$$\perp := \forall t. t$$

Exercise

Does the bottom type contain any inhabitant? Try to define one in System F and one in Haskell

```
contradiction :: forall a. a  
contradiction = ?
```


Definition 5

The *sum type* (i.e. **Either** in Haskell) is defined by

$$\sigma + \tau := \forall t. (\sigma \rightarrow t) \rightarrow (\tau \rightarrow t) \rightarrow t$$

It has two injection functions: the first injection is defined by

$$\begin{aligned} \text{left} &:= \lambda(x : \sigma). \Lambda t. \lambda(f : \sigma \rightarrow t)(g : \tau \rightarrow t). f x \\ \text{right} &:= ? \end{aligned}$$

Church Encoding of Sum Type ii

With the sum type, we can implement the usual construct **either**:

$$\mathbf{either} : \forall t. (\sigma \rightarrow t) \rightarrow (\tau \rightarrow t) \rightarrow (\sigma + \tau) \rightarrow t$$

which corresponds to the Haskell function

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left a) = f a
either f g (Right b) = g b
```

Exercise

Define the other injection function **right**. Define **either** in System F.

Church Encoding of Product Type

Definition 6 (Product Type)

The product type is defined by

$$\sigma \times \tau := \forall t. (\sigma \rightarrow \tau \rightarrow t) \rightarrow t$$

The pairing function is defined by

$$\langle _, _ \rangle := \lambda(x : \sigma)(y : \tau). \Lambda t. \lambda(f : \sigma \rightarrow \tau \rightarrow t). f \ x \ y$$

Exercise

Define projections

$$\mathbf{proj}_1 : \sigma \times \tau \rightarrow \sigma \quad \text{and} \quad \mathbf{proj}_2 : \sigma \times \tau \rightarrow \tau$$

Church Encoding of Natural Numbers i

The type of Church numerals is defined by

$$\mathbf{nat} := \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$$

Church numerals

$$\mathbf{c}_n : \mathbf{nat}$$

$$\mathbf{c}_n := \Lambda t. \lambda(f : t \rightarrow t) (x : t). f^n x$$

Church Encoding of Natural Numbers ii

Successor

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

$\text{suc} := \lambda(n : \text{nat}). \Lambda t. \lambda(f : t \rightarrow t) (x : t). f (n \ t \ f \ x)$

Addition

$\text{add} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$

$\text{add} := \lambda(n : \text{nat}) (m : \text{nat}) \quad \Lambda t. \lambda(f : t \rightarrow t) (x : t). \\ (m \ t \ f) (n \ t \ f \ x)$

Church Encoding of Natural Numbers iii

Multiplication

$\text{mul} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$

$\text{mul} := ?$

Conditional

$\text{ifz} : \forall t. \text{nat} \rightarrow t \rightarrow t \rightarrow t$

$\text{ifz} := ?$

Church Encoding of Natural Numbers iv

However, polymorphic lambda calculus allows us to define *iterator* over natural numbers or the **fold** function in Haskell.

$$\mathbf{fold} : \forall t. (t \rightarrow t) \rightarrow t \rightarrow \mathbf{nat} \rightarrow t$$
$$\mathbf{fold} := \Lambda t. \lambda(f : t \rightarrow t)(e_0 : t)(n : \mathbf{nat}). n \ t \ f \ e_0$$

In Haskell, the above λ -term can be given as

```
fold :: (a -> a) -> a -> [()] -> a
```

```
fold f e [] = e
```

```
fold f e (():xs) = f () (fold f e xs)
```

Church Encoding of Natural Numbers v

Exercise

Define `add` and `mul` using `fold` and justify your answer.

1. $\text{add}' := ? : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$
2. $\text{mul}' := ? : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$

Homework

Show that

$$\begin{aligned} \text{fold } \tau f e_0 c_0 &=_{\beta} e_0 \\ \text{fold } \tau f e_0 c_{n+1} &=_{\beta} f(\text{fold } \tau f e_0 c_n) \end{aligned}$$

Church Encoding of Lists i

Definition 7

For any type σ , the type of lists over σ is

$$\mathbf{list} \sigma := \forall t. t \rightarrow (\sigma \rightarrow t \rightarrow t) \rightarrow t$$

with “list constructors”:

$$\mathbf{nil}_\sigma := \Lambda t. \lambda(h : t)(f : \sigma \rightarrow t \rightarrow t). h$$

and

$$\mathbf{cons}_\sigma := \lambda(x : \sigma)(xs : \mathbf{list} \sigma). \Lambda t. \lambda(h : t)(f : \sigma \rightarrow t \rightarrow t). f x (xs t h f)$$

Church Encoding of Lists ii

Homework

1. Define **fold** over the list type over σ .
2. Define $\text{length}_\sigma : \text{list } \sigma \rightarrow \text{nat}$ calculating the length of a given list so that

$$\text{length nil}_\sigma =_\beta \text{c}_0$$

and

$$\text{length (cons}_\sigma x xs) =_\beta \text{suc (length } xs)$$

3. The above type of lists depends on a specific type σ . Generalise it further.

Abstract Data Type and Existential Type i

Type does not only prevent simple type-mismatch but also help program abstraction, e.g., abstract data types where the implementation detail is invisible to outside.

Example 8

Roughly speaking, a *stack* is a data structure with two operations:

- push
- pop

Any implementation satisfying $\text{pop} \circ (\text{push } n \text{ st}) = (n, \text{st})$ can be seen as a stack.

Abstract Data Type and Existential Type ii

Such abstraction can be described by an *existential type*.

$$\exists t. \tau(t) \quad \text{or simply} \quad \exists t. \tau$$

where t is a type variable and τ is a type (potentially containing t as a free variable).

A *witness* of type $\exists t. \tau$ is a pair (σ, M) of a type σ and an element M of type $\tau[t \mapsto \sigma]$.

(The typing rule and β -conversion are omitted here.)

Example 9

The type of stacks over natural numbers can be defined by

$$\exists t. (t \times \mathbf{nat} \rightarrow t) \times (t \rightarrow 1 + \mathbf{nat} \times t)$$

1. The first component is the push function.
2. The second component is the pop function.

An instance of a stack is an element (σ, M) of the above type

1. σ is a type
2. $\mathbf{proj}_1 M: \sigma \times \mathbf{nat} \rightarrow \sigma$
3. $\mathbf{proj}_2 M: \sigma \rightarrow 1 + \mathbf{nat} \times \sigma$.

Encoding of Existential Type in Haskell

A limited form of existential type can be defined by

```
data Exists t where  
  Ex :: t a -> Exists t
```

and another version for typeclass

```
data Exists' c where  
  Ex' :: c a => a -> Exists' c
```

in GHC with extensions **GADTs** and **ConstraintKinds**.

Exercise

Define the top type **T** which contains every typable term.

Encoding of Existential Type in System F

The existential type can also be encoded in System F as

$$\exists t. \tau := \forall u. (\forall t. \tau \rightarrow u) \rightarrow u$$

using continuation passing style.¹

A witness (σ, M) of $\exists t. \tau$ is encoded as

$$\Lambda u. \lambda(f : \forall t. \tau \rightarrow u). f \sigma M$$

Exercise

Check that the above terms do make sense. How to use a term of some existential type?

¹Recall that $\exists x. \varphi = (\forall x. (\varphi \rightarrow \perp)) \rightarrow \perp$ classically.

Properties of System F

Type Safety and Normalisation

Theorem 10 (Preservation)

If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_{\beta_1} N$, then $\Gamma \vdash N : \sigma$.

Theorem 11 (Progress)

If $\vdash M : \sigma$, then either M is a value or there is N such that $M \longrightarrow_{\beta_1} N$.

The above two properties are proved by induction.

Theorem 12 (Normalisation)

If $\vdash M : \sigma$, then M terminates.

Definition 13

The *erasing map* is a function defined by

$$|x| = x$$

$$|\lambda(x : \tau). M| = \lambda x. |M|$$

$$|M N| = (|M| |N|)$$

$$|\Lambda t. M| = |M|$$

$$|M \tau| = |M|$$

Proposition 14

Within System F , if $\vdash M : \sigma$ and $|M| \longrightarrow_{\beta_1} N'$, then there exists a well-typed term N with $\vdash N : \sigma$ and $|N| = N'$.

Undecidability of Type Inference

Theorem 15

It is undecidable whether, given a closed term M of the untyped lambda-calculus, there is a well-typed term M' in System F such that $|M'| = M$.

Arbitrary Rank Polymorphism \forall can appear anywhere (GHC with `-XRankNTType`).

Rank-1 Polymorphism \forall only appear in the outermost position.

Hindley-Milner type system (adopted by Haskell, Standard ML, etc.) only supports rank-1 polymorphism and type inference becomes decidable.²

²It is decidable up to rank-2 polymorphism.



What functions can you write for the following type?

$$\Lambda t. t \rightarrow t$$

Homework

Read [Wad89] if interested.

References

-  Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.
-  Philip Wadler, *Theorems for free!*, Proc. 4th Int. Conf. Funct. Program. Lang. Comput. Archit. (New York, NY, USA), ACM Press, 1989, pp. 347–359.