

Honors Programming Languages

Types and the λ -calculus

Types

A *type* denotes a set of values.

- `bool` denotes the set `{ False, True }`.
- `int` denotes the set of integers.
- $\tau_1 \rightarrow \tau_2$ denotes the set of functions from the denotation of τ_1 to the denotation of τ_2 .
(Actually, not all functions in the set-theoretic sense, just the computable ones.)

Syntax of types

For our simply-typed λ -calculus with constants:

$$\begin{array}{ll} \tau ::= b & \text{type constants (int, bool, etc.)} \\ | \tau \rightarrow \tau & \text{function types} \end{array}$$

Note: \rightarrow associates to the right, so

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \equiv \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$$

Components of a type system: judgements

Judgements

$$\Gamma \vdash M : \tau$$

Γ is a *type assignment* of types to identifiers.

Also known as:

- *environment*
- *type context*
- *context*

Typically written:

$$x_1 : \tau_1, \dots, x_n : \tau_n$$

where all the x 's are distinct

Components of a type system: rules

- Inference rules:

$$\text{(name)} \quad \frac{J_1 \quad \dots \quad J_n}{J}$$

The J 's are judgements. The rule says,
“if you can prove J_1 through J_n , you can prove J ”.

- Axioms:

$$\text{(name)} \quad \frac{}{J} \quad \text{or just} \quad \text{(name)} \quad J$$

Inference rules without premises

Predefined symbols

The terms depend on our choice of type and term constants.

A λ^{\rightarrow} signature $\Sigma = (B, C)$ where

- B : a set of *base types*
- C : a collection of pairs of an identifier c and its type τ
written: $c : \tau$

Terms and derivations

M is a λ^{\rightarrow} term over signature Σ with type τ in Γ if the judgement

$$\Gamma \vdash M : \tau$$

follows from the inference rules.

The proof of a typing judgement is a *typing derivation*.

Typing derivations are drawn as trees:

- root : the judgement to be proved
- leaves : instances of axioms
- nodes : instances of rules

The rules for λ^{\rightarrow}

(constant) $\emptyset \vdash c : \tau$ where $c : \tau \in C$

(var)
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

(\rightarrow -intro)
$$\frac{\Gamma | x, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2}$$

(\rightarrow -elim)
$$\frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

Examples:

- $x : \tau, y : \tau \rightarrow \tau' \vdash yx : \tau'$
- $x : \tau, y : \tau \vdash \lambda y : \tau \rightarrow \tau'. yx : (\tau \rightarrow \tau') \rightarrow \tau'$
- $\emptyset \vdash (\lambda x : \tau \rightarrow \tau. \lambda y : \tau'. x)(\lambda x : \tau. x) : \tau' \rightarrow \tau \rightarrow \tau$

Enhancements: Cartesian products

$\lambda^{\rightarrow} + \text{Cartesian products} = \lambda^{\rightarrow, \times}$

$$(\times\text{-intro}) \quad \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2}$$

$$(\times\text{-elim-1}) \quad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \text{Proj}_1^{\tau_1, \tau_2} M : \tau_1}$$

$$(\times\text{-elim-2}) \quad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \text{Proj}_2^{\tau_1, \tau_2} M : \tau_2}$$

Enhancements: Disjoint unions

$$\lambda^{\rightarrow} + \text{Sums} = \lambda^{\rightarrow,+}$$

$$\text{(+intro-1)} \quad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \text{Inleft}^{\tau,\tau'} M : \tau + \tau'}$$

$$\text{(+intro-1)} \quad \frac{\Gamma \vdash M : \tau'}{\Gamma \vdash \text{Inright}^{\tau,\tau'} M : \tau + \tau'}$$

$$\text{(+elim)} \quad \frac{\Gamma \vdash M : \tau_1 + \tau_2 \quad \begin{array}{l} \Gamma \vdash N : \tau_1 \rightarrow \tau \\ \Gamma \vdash P : \tau_2 \rightarrow \tau \end{array}}{\Gamma \vdash \text{Case}^{\tau_1,\tau_2,\tau} MNP : \tau}$$

Flavors of polymorphism

- *parametric* polymorphism:
Uses type variables that may be instantiated by any type.
Programs work uniformly over all instantiations
- *subtype* polymorphism:
Type systems where one type can be “contained” in another.
Functions that take a value of a type can also take a value of any subtype.
- *ad-hoc* polymorphism:
More properly known as overloading. Programs have different run-time behaviors when used on different types.

Polymorphism

Polymorphism allows a single term to be used with many different types.

Example: function composition.

Have one `compose` function instead of one for each type:

```
composeint,int,int  
=  $\lambda f : \text{int} \rightarrow \text{int} . (\lambda g : \text{int} \rightarrow \text{int} . (\lambda x : \text{int} . f(gx)))$ 
```

```
composebool,int,int  
=  $\lambda f : \text{bool} \rightarrow \text{int} . (\lambda g : \text{int} \rightarrow \text{int} . (\lambda x : \text{bool} . f(gx)))$ 
```

compose, polymorphic style

$$\text{compose}_{r,s,t} = \lambda f : s \rightarrow t. (\lambda g : r \rightarrow s. (\lambda x : r. f(gx)))$$

r , s , and t can be instantiated by any type.

If T is a collection of types (a *universe*), we can abstract over the types in `compose`:

$$\text{compose} = \Lambda r : T. \Lambda s : T. \Lambda t : T. \text{compose}_{r,s,t}$$

apply `compose` to types from T

```
compose int int bool
```

and reduce using β -reduction, to:

```
composeint,int,bool
```

Typing the polymorphic compose

What is the type of the polymorphic compose?

$$\prod r : T. \prod s : T. \prod t : T. (r \rightarrow s) \rightarrow (s \rightarrow t) \rightarrow (r \rightarrow t)$$

Universes

What can T be?

Three possibilities, leading to different type systems:

- T contains only “simple” types. This gives us *predicative* polymorphism.
- T contains simple types and polymorphic types. This gives us *impredicative* polymorphism.
- T contains simple types, polymorphic types, and T itself. Also known as `Type : Type`.

Predicate polymorphism

- type variables always denote simple types
- close to the ML type system (except that types of variables are explicit)
- types are statically checkable

Impredicative polymorphism

Impredicative polymorphism:

- type variables may denote polymorphic types
- discovered independently by Girard and Reynolds in the 1970's
- also called System F, or 2nd-order λ -calculus
- types cannot be considered sets any more!
- types are statically checkable

“Type : Type”

- permits writing functions from types to types
- loose strong normalization
- typechecking is undecidable

Predicative polymorphic calculus

Judgements are $\Gamma \vdash A : B$, where

- A is a term or type
- B is a type or universe

Two universes for types:

- U_1 : simple types (from λ^{\rightarrow})
- U_2 : simple or polymorphic types

Two sorts of type variables:

types in U_1 : $\tau ::= t \mid b \mid \tau \rightarrow \tau$

types in U_2 : $\sigma ::= \tau \mid \Pi t.\sigma$

Formation rules for contexts

(empty context)

\emptyset context

(U_1 context)

$$\frac{\Gamma \text{ context}}{\Gamma, t : U_1 \text{ context}} \quad (t \text{ not in } \Gamma)$$

(U_i context)

$$\frac{\Gamma \vdash \sigma : U_i}{\Gamma, x : \sigma \text{ context}}$$

Typing rules for variables

$$\text{(var)} \quad \frac{\Gamma, x : A \text{ context}}{\Gamma, x : A \vdash x : A}$$

$$\text{(add var)} \quad \frac{\Gamma \vdash a : B \quad \Gamma, x : C \text{ context}}{\Gamma, x : C \vdash a : B}$$

Type formation rules

(const U_1)

$$\emptyset \vdash b : U_1$$

($\rightarrow U_1$)

$$\frac{\Gamma \vdash \tau : U_1 \quad \Gamma \vdash \tau' : U_1}{\Gamma \vdash \tau \rightarrow \tau' : U_1}$$

($U_1 \subseteq U_2$)

$$\frac{\Gamma \vdash \tau : U_1}{\Gamma \vdash \tau : U_2}$$

(ΠU_2)

$$\frac{\Gamma, t : U_1 \vdash \sigma : U_2}{\Gamma \vdash \Pi t : U_1. \sigma : U_2}$$

Example: $\Pi s : U_1. (\Pi t : U_1. (s \rightarrow t))$

Pre-terms

The syntax of unchecked *pre-terms*:

$$M ::= x \mid \lambda x : \tau. M \mid MM \mid \Lambda t : U_1. M \mid M\tau$$

Terms are the subset of pre-terms which are type checked.

Type rules

(const) $\emptyset \vdash c : \sigma$

(\rightarrow -intro)
$$\frac{\Gamma | x, x : \tau_1 \vdash M : \tau_2 \quad \Gamma \vdash \tau_1 : U_1 \quad \Gamma \vdash \tau' : U_1}{\Gamma \vdash (\lambda x : \tau_1. M) : \tau_1 \rightarrow \tau_2}$$

(\rightarrow -elim)
$$\frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

(Π intro)
$$\frac{\Gamma, t : U_1 \vdash M : \sigma}{\Gamma \vdash (\Lambda t : U_1. M) : \Pi t : U_1. \sigma}$$

(Π elim)
$$\frac{\Gamma \vdash M : (\Pi t : U_1. \sigma) \quad \Gamma \vdash \tau : U_1}{\Gamma \vdash M\tau : [r/t]\sigma}$$

Type rules, part II

$$\text{(ty eq)} \quad \frac{\Gamma \vdash M : \sigma_1 \quad \Gamma \vdash \sigma_1 = \sigma_2 : U_1}{\Gamma \vdash M : \sigma_2}$$

In the predicative λ -calculus, the only type equalities are due to α -conversion. In System F and beyond, things get more complex.

Results

Most basic theorems about reduction in λ^{\rightarrow} generalize to $\lambda^{\rightarrow, \Pi}$:

- confluence
- strong normalization
- subject reduction

Example

$$\emptyset \vdash ((\Lambda t : U_1. (\lambda x : t.x)) \text{int}3) : \text{int}$$

See board for derivation

Drawbacks

- cannot abstract over polymorphic term variables
- cannot use the same polymorphic function twice at different types
- need an additional construct to get to ML

Explicit ML

ML polymorphism uses the following construct:

```
let x = N in M
```

which declares x to be bound to the value of N during evaluation of M .

This has the same operational behavior as $[N/x]M$, or $(\lambda x.M)N$, but x is allowed to be polymorphic.

Let type rule

$$\text{(let)} \quad \frac{\Gamma \vdash N : \sigma \quad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\text{let } x : \sigma = N \text{ in } M) : \tau}$$

Example

```
let id : ( $\Pi t : U_1. t \rightarrow t$ ) = ( $\Lambda t : U_1. \lambda x : t. x$ )  
in (id(int  $\rightarrow$  int))(idint)3
```