Honors Programming Languages

Types and the $\lambda$-calculus
Types

A *type* denotes a set of values.

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

For our simply-typed $\lambda$-calculus with constants:

$$\tau ::= b \quad \text{type constants (int, bool, etc.)}$$

$$\mid \tau \to \tau \quad \text{function types}$$

Note: $\to$ associates to the right, so

$$\tau_1 \to \tau_2 \to \tau_3 \equiv \tau_1 \to (\tau_2 \to \tau_3)$$
Components of a type system: judgements

Judgements

\[ \Gamma \vdash M : \tau \]

\( \Gamma \) is a type assignment of types to identifiers.
Also known as:
- environment
- type context
- context

Typically written:

\[ x_1 : \tau_1, \ldots, x_n : \tau_n \]

where all the \( x \)'s are distinct
Components of a type system: rules

■ Inference rules:

\[
\begin{array}{c}
\text{(name)} \ \ \ \ J_1 \ \ \ \ldots \ \ J_n \\
\hline
J
\end{array}
\]

The \( J \)’s are judgements. The rule says, “if you can prove \( J_1 \) through \( J_n \), you can prove \( J \)”.

■ Axioms:

\[
\begin{array}{c}
\text{(name)} \\
\hline
J
\end{array}
\]

or just \( \text{(name)} \ J \)

Inference rules without premises
The terms depend on our choice of type and term constants.

A $\lambda \rightarrow$ signature $\Sigma = (B, C)$ where

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

written: $c : \tau$
Terms and derivations

$M$ is a $\lambda \rightarrow$ term over signature $\Sigma$ with type $\tau$ in $\Gamma$ 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 $\lambda \to$

(constant) \hspace{1cm} \emptyset \vdash c : \tau \quad \text{where } c : \tau \in C

(var) \hspace{1cm} \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}

($\to$-intro) \hspace{1cm} \frac{\Gamma|_{x, x : \tau_1} \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1.M : \tau_1 \to \tau_2}

($\to$-elim) \hspace{1cm} \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}

Examples:

\- $x : \tau, y : \tau \to \tau' \vdash y x : \tau'$

\- $x : \tau, y : \tau \vdash \lambda y : \tau \to \tau'.y x : (\tau \to \tau') \to \tau'$

\- $\emptyset \vdash (\lambda x : \tau \to \tau.\lambda y : \tau'.x)(\lambda x : \tau.x) : \tau' \to \tau \to \tau$
Enhancements: Cartesian products

\[ \lambda \to + \text{ Cartesian products} = \lambda \to, \times \]

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

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

\[(\times\text{-elim-2}) \quad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \text{Proj}^{\tau_1, \tau_2} M : \tau_2}\]
Enhancements: Disjoint unions

\[ \lambda \to + \text{Sums} = \lambda \to^{+} \]

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

2. \((+-\text{intro-1})\)
   \[
   \frac{
   \Gamma \vdash M : \tau' \\
   }{
   \Gamma \vdash \text{Inright}^{\tau,\tau'} M : \tau + \tau'
   }
   \]

3. \((+-\text{elim})\)
   \[
   \frac{
   \Gamma \vdash M : \tau_1 + \tau_2 \\
   \Gamma \vdash N : \tau_1 \to \tau \\
   \Gamma \vdash P : \tau_2 \to \tau \\
   }{
   \Gamma \vdash \text{Case}^{\tau_1,\tau_2,\tau} M N P : \tau
   }
   \]
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 \texttt{compose} function instead of one for each type:

\[
\text{compose}_{\text{int}, \text{int}, \text{int}} = \lambda f : \text{int} \to \text{int}. (\lambda g : \text{int} \to \text{int}. (\lambda x : \text{int}. f(gx)))
\]

\[
\text{compose}_{\text{bool}, \text{int}, \text{int}} = \lambda f : \text{bool} \to \text{int}. (\lambda g : \text{int} \to \text{int}. (\lambda x : \text{bool}. f(gx)))
\]
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 \text{compose} to types from \(T\)

\[
\text{compose} \text{int int bool}
\]

and reduce using \(\beta\)-reduction, to:

\[
\text{compose}_{\text{int, int, bool}}
\]
Typing the polymorphic compose

What is the type of the polymorphic compose?

$$\Pi r : T. \Pi s : T. \Pi t : T. (r \to s) \to (s \to t) \to (r \to t)$$
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 \texttt{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:

- type variables may denote polymorphic types
- discovered independently by Girard and Reynolds in the 1970’s
- also called System F, or 2nd-order \( \lambda \)-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 undecideable
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 $\lambda \to$)
- $U_2$: simple or polymorphic types

Two sorts of type variables:

- types in $U_1$: $\tau ::= t \mid b \mid \tau \to \tau$
- types in $U_2$: $\sigma ::= \tau \mid \Pi t.\sigma$
Formation rules for contexts

(empty context) \[ \emptyset \text{ context} \]

\[ \Gamma \text{ context} \]

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

(\(U_i\) context) \[ \frac{\Gamma \vdash \sigma : U_i}{\Gamma, x : \sigma \text{ context}} \]
Typing rules for variables

(var) \[ \begin{array}{c} \Gamma, x : A \text{ context} \\ \hline \\ \Gamma, x : A \vdash x : A \end{array} \]

(add var) \[ \begin{array}{c} \Gamma \vdash a : B \\ \Gamma, x : C \text{ context} \\ \hline \\ \Gamma, x : C \vdash a : B \end{array} \]
Type formation rules

(const $U_1$) \hspace{1cm} \emptyset \vdash b : U_1

($\rightarrow U_1$) \hspace{1cm} \frac{\Gamma \vdash \tau : U_1 \quad \Gamma \vdash \tau' : U_1}{\Gamma \vdash \tau \rightarrow \tau' : U_1}

($U_1 \subseteq U_2$) \hspace{1cm} \frac{\Gamma \vdash \tau : U_1}{\Gamma \vdash \tau : U_2}

($\Pi U_2$) \hspace{1cm} \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) \[\emptyset \vdash c : \sigma\]

(\to\text{-intro}) \[
\frac{\Gamma|_{x, x : \tau_1} \vdash M : \tau_2 \quad \Gamma \vdash \tau_1 : U_1 \quad \Gamma \vdash \tau' : U_1}{\Gamma \vdash (\lambda x : \tau_1.M) : \tau_1 \to \tau_2}
\]

(\to\text{-elim}) \[
\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}
\]

(\Pi\text{ intro}) \[
\frac{\Gamma, t : U_1 \vdash M : \sigma}{\Gamma \vdash (\Lambda t : U_1.M) : \Pi t : U_1.\sigma}
\]

(\Pi\text{ 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

\[
\frac{
\Gamma ⊢ M : \sigma_1 \quad \Gamma ⊢ \sigma_1 = \sigma_2 : U_1
}{
\Gamma ⊢ M : \sigma_2}
\]

(ty eq)

In the predicative \(\lambda\)-calculus, the only type equalities are due to \(\alpha\)-conversion. In System F and beyond, things get more complex.
Most basic theorems about reduction in $\lambda \rightarrow$ 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
ML polymorphism uses the following construct:

\[
\text{let } x = N \text{ in } M
\]

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

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

\[
\begin{align*}
\text{let } & \text{id : } (\prod t : U_1.t \to t) = (\Lambda t : U_1.\lambda x : t.x) \\
\text{in } (\text{id(int \to int)})(\text{idint}3)
\end{align*}
\]