Types

CS 131: Programming Languages
October 16, 2001

One-step $\beta$-Reduction

- The relation $\rightarrow_\beta$ is defined by:

$$\frac{(\lambda x : t . M) N \rightarrow_\beta M[x \rightarrow N]}{M \rightarrow_\beta M'}$$

$$\frac{M N \rightarrow_\beta M' N'}{M N \rightarrow_\beta M' N'}$$

$$\frac{M \rightarrow_\beta M'}{\lambda x : t . M \rightarrow_\beta \lambda x : t . M'}$$

Pure Simply-Typed $\lambda$-Calculus

- Syntax

$$M, N ::= x \quad \text{variables}$$
$$| \lambda x : t . M \quad \text{functions}$$
$$| M N \quad \text{applications}$$

$$t, u ::= \alpha | \beta | \ldots \quad \text{type variables}$$
$$| t \rightarrow u \quad \text{function types}$$

Typechecking

- Once we have a typed language, we can ask questions about pieces of code
  - Is the code make sense? Does it typecheck?
  - What is the type of the expression (i.e., what sort of value does it evaluate to?)

$$\frac{(\lambda f : \alpha \rightarrow \alpha . f) (\lambda y : \alpha . y) \rightarrow \lambda f : \beta . (f \ f)}{\lambda f : \alpha \rightarrow \alpha . f}$$
Conditional Judgments

- Well-typedness is context-sensitive
  - Is \((\lambda f : \alpha . f)(x)\) well-typed?
- So, statements about typing are conditional
  - "If \(x : \alpha\) then \((\lambda f : \alpha . f)(x) : \alpha\)"
  - "If \(x : \beta\) then \((\lambda f : \alpha . f)(x)\) is ill-typed"
- Given assumptions about the types of variables, we can conclude code is well-typed

\[ x : \alpha \vdash (\lambda f : \alpha . f)(x) : \alpha \]

Typing Environments

- Recall: an environment is a lookup table
  - A type environment associates variables with types
- Notation
  - Usual to use \(\Gamma\) to denote a type environment.
  - Specific type environments can be written as a list
    - e.g., \(x : \alpha, y : \beta, z : \alpha \rightarrow \beta\)
  - By the notation \(\Gamma (x)\) we mean the type that \(\Gamma\) says we've assumed for \(x\).
  - The notation \(\Gamma, x : t\) means "all the assumptions about variables made in \(\Gamma\), plus the additional assumption that \(x\) has type \(t\)".

Typing Rules

\[ \Gamma \vdash x : \Gamma (x) \]

\[ \Gamma \vdash M : t \rightarrow u \quad \Gamma \vdash N : t \]
\[ \Gamma \vdash M \ N : u \]

\[ \Gamma, x : t_1 \vdash M : t_2 \]
\[ \Gamma \vdash \lambda x : t_1 . M : t_1 \rightarrow t_2 \]

We say that \(\Gamma \vdash M : t\) holds (or is true, or is provable) if and only if there is a proof of this fact using these rules!

Example Proofs

\[ x : \alpha, y : \beta, z : \alpha \rightarrow \alpha \vdash y : \beta \]

\[ x : \alpha \vdash x : \alpha \]
\[ \vdash \lambda x : \alpha . x : \alpha \rightarrow \alpha \]

\[ z : \alpha \vdash \lambda x : \alpha . x : \alpha \rightarrow \alpha \]
\[ z : \alpha \vdash z : \alpha \]
\[ z : \alpha \vdash (\lambda x : \alpha . x) z : \alpha \]
Extension: Pairs

\[ M, N ::= x \mid \lambda x : t . M \mid M \, N \mid <M, N> \mid \text{projections} \]

\[ t, u ::= \alpha \mid \beta \mid \ldots \mid t \to u \mid t \times u \]

\[ \text{Typing Rules} \]

\[ \ldots, x : t, \ldots \vdash x : t \]

\[ \Gamma, x : t \vdash M : u \quad \Gamma \vdash N : t \]

\[ \Gamma \vdash (\lambda x : t . M) : t \to u \quad \Gamma \vdash M \, N : u \]

\[ \Gamma \vdash M : \quad \Gamma \vdash N : \]

\[ \Gamma \vdash <M, N> : \]

\[ \Gamma \vdash M : \quad \Gamma \vdash M : \]

\[ \Gamma \vdash \text{fst} \, M : \quad \Gamma \vdash \text{snd} \, M : \]

\[ \text{A Sample Derivation} \]

\[ x : \alpha, y : \beta \vdash x : \alpha \]

\[ x : \alpha, y : \beta \vdash y : \beta \]

\[ x : \alpha, y : \beta \vdash <x, y> : \alpha \times \beta \]

\[ x : \alpha, y : \beta \vdash \text{fst} <x, y> : \alpha \]

\[ x : \alpha \vdash \lambda y : \beta . (\text{fst} <x, y>) : \beta \to \alpha \]

\[ \vdash \lambda x : \alpha . (\lambda y : \beta . (\text{fst} <x, y>)) : \alpha \to (\beta \to \alpha) \]

\[ \text{Erasing All but the Types} \]

\[ \ldots, t, \ldots \vdash t \]

\[ \Gamma, t \vdash u \quad \Gamma \vdash t \to u \quad \Gamma \vdash t \]

\[ \Gamma \vdash t \to u \quad \Gamma \vdash t \]

\[ \Gamma \vdash u \]

\[ \Gamma \vdash t \times u \quad \Gamma \vdash t \times u \]

\[ \Gamma \vdash t \quad \Gamma \vdash t \times u \]

\[ \Gamma \vdash t \quad \Gamma \vdash t \times u \]
Rules for Propositional Logic

\[ \begin{align*}
\Gamma, p &\vdash q \\
\Gamma &\vdash p \Rightarrow q \\
\Gamma &\vdash p \Rightarrow q \\
\Gamma &\vdash q \\
\Gamma &\vdash p \wedge q \\
\Gamma &\vdash p \\
\Gamma &\vdash q
\end{align*} \]

Curry-Howard Isomorphism

- a.k.a. "Proofs as programs", "Propositions as types"
- Types correspond to a logical proposition
  - Type variables correspond to propositional variables
  - Function types correspond to implications
  - Pair types correspond to conjunctions
- A proposition is provable if and only if there is a closed term of the corresponding type
  - Such types are said to be inhabited.
  - Typed \( \lambda \)-terms are encodings of logical proofs.
  - Proof-checking = typechecking

Examples

1. Show that
   \[ \Gamma \vdash p \Rightarrow (p \wedge p) \]
   by finding a term of type
   \[ \alpha \rightarrow (\alpha \times \alpha) \]

2. Show that
   \[ \Gamma \vdash (p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge q) \Rightarrow r) \]
   by finding a term of type
   \[ (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \times \beta) \rightarrow \gamma) \]

Extensions

- The true proposition corresponds to any non-empty type
  - E.g., \texttt{unit}.
- The false proposition corresponds to an empty type.
  - Usually called \texttt{void}.
  - Encode \( \neg p \) as \( (p \Rightarrow \text{false}) \).
- Second-order predicate calculus: polymorphic types
  - Second-order = quantifying over propositions.
  - E.g., \( \forall \alpha. \alpha \rightarrow (\alpha \times \alpha) \) vs. \( \forall p. p \Rightarrow (p \wedge p) \)
- Disjunctions: sum types (simple non-recursive datatype)
- Propositional logic: dependent types
- Modal logic: types for run-time code generation
- Linear logic: linear types
Intuitionism

- Generally, $\lambda$-calculi correspond to constructive or intuitionistic logics.
  - E.g., no terms of type
    $\forall \alpha. (\alpha \rightarrow \text{void}) \rightarrow \alpha$
    $\forall \alpha. (\forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha$ (Pierce's Law)

Proof Normalization

- Reductions as proof "simplifications".

Hilbert System for Implication

- Axiom Schema
  $\vdash p \rightarrow (q \rightarrow p)$
  $\vdash (p \rightarrow (q \rightarrow r)) \Rightarrow ((p \rightarrow q) \Rightarrow (p \rightarrow r))$

- Modus Ponens
  $\vdash p \rightarrow q, p \vdash q$

Summary

$\lambda$-Calculus

Type

Term (program)

Reduction

Logic

Proposition

Proof

Proof Normalization
A Typed Functional Language:
Base Types

\[
\begin{align*}
\Gamma &\vdash n : \text{int} & \Gamma &\vdash tt : \text{bool} & \Gamma &\vdash ff : \text{bool} \\
\Gamma &\vdash M_1 : & \Gamma &\vdash M_2 : & \Gamma &\vdash M_1 + M_2 : \\
\Gamma &\vdash M_1 : & \Gamma &\vdash M_2 : & \Gamma &\vdash M_1 == M_2 : \\
\Gamma &\vdash M_1 : & \Gamma &\vdash M_2 : & \Gamma &\vdash M_3 : & \Gamma &\vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 :
\end{align*}
\]

A Typed Functional Language:
Local Variables

\[
\begin{align*}
\Gamma &\vdash M : & \Gamma, x : &\vdash N : & \Gamma &\vdash \text{let } x = M \text{ in } N :
\end{align*}
\]

Are These Typing Rules Right?

- What does it even mean to be right?

But...

- Some programs wouldn't get stuck but still don't typecheck

\[(\text{if } ff \text{ then } tt \text{ else } 4) + 1\]

- For any interesting language, a type system preventing all bad programs also rejects programs that would run without problems.

- Research topic: type systems that catch as many errors as possible, but don't reject useful programs
Strong Normalization

- Consider the language we have so far:

\[
M, N ::= x \mid n \mid tt \mid if \ M+N \mid M==N \\
\lambda x : t. M \mid M \cdot N \\
\langle M, N \rangle \mid \text{fst } M \mid \text{snd } M \\
\text{if } M \text{ then } N_1 \text{ else } N_2 \\
\text{let } x = M \text{ in } N
\]

- Theorem:
  Every well-typed program terminates.
  (even if you add in unit, sum types, more arith., ...)
  Corollary: as it stands, this language is weaker than TM

Increasing Expressive Power

- Simplest solution: add recursion as a new primitive feature.
  - E.g., by throwing in \( \text{Y} \) as a primitive, with a rule like \( \text{Y} \ M \to M(\text{Y} \ M) \)
  - Or, by taking recursive functions as primitive.

Recursive Function Values

- The function value
  \[
  \text{fix } f(x) \text{ is } M
  \]
  corresponds roughly to the SML code
  ```sml
  let
  fun f(x) = M
  in
  f
  end
  ```
- In particular, note that the scope of \( f \) is \( e \) and the scope of \( x \) is \( e \), and that's it.
  - This does not permit other code to refer to this function as \( f \)!