Types

CS 131: Programming Languages

October 16, 2001

Pure Simply-Typed λ-Calculus

- Syntax

```
M,N ::=  x              variables
      |  λx:t.M         functions
      |  M N            applications

t,u ::=  α | β | ...    type variables
      |  t → u          function types
```

One-step β-Reduction

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

$$\frac{}{(\lambda x{:}t.M)N \;\rightarrow_\beta\; M[x \rightarrow N]}$$

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

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

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?)

```
(λf:α->α.f)(λy:α.y)
(λf:α.f)(λy:α.y)
λf:β.(f f)
```

Conditional Judgments

- Well-typedness is *context-sensitive*
  - Is `(λf:α.f)(x)` well-typed?
- So, statements about typing are *conditional*
  - "If `x : α` then `(λf:α.f)(x) : a`"
  - "If `x : β` then `(λf:α.f)(x)` is ill-typed"
- Given *assumptions* about the types of variables, we can conclude code is well-typed

$$\boxed{x{:}\alpha \vdash (\lambda f{:}\alpha.f)(x) : \alpha}$$

Typing Environments

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

Typing Rules

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash M : t\text{->}u \qquad \Gamma \vdash N : t}{\Gamma \vdash M\ N : u}$$

$$\frac{\Gamma, x{:}t_1 \vdash M : t_2}{\Gamma \vdash \lambda x{:}t_1.M : t_1\text{->}t_2}$$

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

$$\frac{}{x{:}\alpha, y{:}\beta, z{:}\alpha\text{->}\alpha \vdash y : \beta}$$

$$\frac{\frac{}{x{:}\alpha \vdash x : \alpha}}{\vdash \lambda x{:}\alpha.x : \alpha\text{->}\alpha}$$

$$\frac{\frac{\frac{}{z{:}\alpha, x{:}\alpha \vdash x : \alpha}}{z{:}\alpha \vdash \lambda x{:}\alpha.x : \alpha\text{->}\alpha} \qquad \frac{}{z{:}\alpha \vdash z : \alpha}}{z{:}\alpha \vdash (\lambda x{:}\alpha.x)z : \alpha}$$

Extension: Pairs

`M,N ::=`	`x`	*variables*
`|`	`λx:t.M`	*functions*
`|`	`M N`	*applications*
`|`	`<M,N>`	*pairs*
`	`	`fst M	snd M`	*projections*
`t,u ::=`	`α	β	...`	*type variables*
`|`	`t -> u`	*function types*
`|`	`t * u`	*pair types*

$$\frac{}{\texttt{fst <M,N>} \longrightarrow \texttt{M}} \qquad \frac{}{\texttt{snd <M,N>} \longrightarrow \texttt{N}}$$

Typing Rules

$$\frac{}{\ldots, \texttt{x:t}, \ldots \vdash \texttt{x : t}}$$

$$\frac{\Gamma, \texttt{x:t} \vdash \texttt{M : u}}{\Gamma \vdash \texttt{(}\lambda\texttt{x:t.M) : t->u}} \qquad \frac{\Gamma \vdash \texttt{M : t->u} \qquad \Gamma \vdash \texttt{N : t}}{\Gamma \vdash \texttt{M N : u}}$$

$$\frac{\Gamma \vdash \texttt{M :} \qquad \Gamma \vdash \texttt{N :}}{\Gamma \vdash \texttt{<M,N> :}}$$

$$\frac{\Gamma \vdash \texttt{M :}}{\Gamma \vdash \texttt{fst M :}} \qquad \frac{\Gamma \vdash \texttt{M :}}{\Gamma \vdash \texttt{snd M :}}$$

A Sample Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{x:}\alpha\texttt{,y:}\beta \vdash \texttt{x : }\alpha} \qquad \dfrac{}{\texttt{x:}\alpha\texttt{,y:}\beta \vdash \texttt{y : }\beta}}{\texttt{x:}\alpha\texttt{,y:}\beta \vdash \texttt{<x,y> : }\alpha*\beta}}{\texttt{x:}\alpha\texttt{,y:}\beta \vdash \texttt{fst <x,y> : }\alpha}}{\texttt{x:}\alpha \vdash \lambda\texttt{y:}\beta\texttt{.(fst <x,y>) : }\beta\texttt{->}\alpha}}{\vdash \lambda\texttt{x:}\alpha\texttt{.(}\lambda\texttt{y:}\beta\texttt{.(fst <x,y>)) : }\alpha\texttt{->(}\beta\texttt{->}\alpha\texttt{)}}$$

Erasing All but the Types

$$\frac{}{\ldots, \texttt{t}, \ldots \vdash \texttt{t}}$$

$$\frac{\Gamma, \texttt{t} \vdash \texttt{u}}{\Gamma \vdash \texttt{t->u}} \qquad \frac{\Gamma \vdash \texttt{t->u} \qquad \Gamma \vdash \texttt{t}}{\Gamma \vdash \texttt{u}}$$

$$\frac{\Gamma \vdash \texttt{t} \qquad \Gamma \vdash \texttt{u}}{\Gamma \vdash \texttt{t*u}}$$

$$\frac{\Gamma \vdash \texttt{t*u}}{\Gamma \vdash \texttt{t}} \qquad \frac{\Gamma \vdash \texttt{t*u}}{\Gamma \vdash \texttt{u}}$$

Rules for Propositional Logic

$$\frac{}{\ldots,p,\ldots \vdash p}$$

$$\frac{\Gamma, p \vdash q}{\Gamma \vdash p \Rightarrow q} \qquad \frac{\Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p}{\Gamma \vdash q}$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q}$$

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \qquad \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q}$$

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 **λ**-terms are encodings of logical proofs.
  - Proof-checking = typechecking

Examples

1. Show that

   $$\vdash p \Rightarrow (p \wedge p)$$

   by finding a term of type

   $$\alpha \texttt{ -> } (\alpha * \alpha)$$

2. Show that

   $$\vdash (p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge q) \Rightarrow r))$$

   by finding a term of type

   $$(\alpha \texttt{->} (\beta \texttt{->} \gamma)) \texttt{ -> } ((\alpha * \beta) \texttt{->} \gamma)$$

Extensions

- The `true` proposition corresponds to any non-empty type
  - E.g., `unit`.
- The `false` proposition corresponds to an empty type.
  - Usually called `void`.
  - Encode $\neg p$ as $(p \Rightarrow$ `false`).
- Second-order predicate calculus: polymorphic types
  - Second-order = quantifying over *propositions*.
  - E.g., $\forall\alpha.\ \alpha \to (\alpha * \alpha)$ vs. $\forall p.\ p \to (p \wedge p)$
- 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, λ-calculi correspond to *constructive* or *intuitionistic* logics.
  - E.g., no terms of type

∀α. α + (α->void)	*(∀p. p or ¬p)*
∀α. ((α->void)->void) -> α	*(∀p. ¬¬p ⇒p)*
∀α.∀β.((α->β)->α) -> α	*(Pierce's Law)*

Proof Normalization

- Reductions as proof "simplifications".

$$\dfrac{\dfrac{\Gamma \vdash M : t \quad \Gamma \vdash N : u}{\Gamma \vdash \langle M,N\rangle : t*u}}{\Gamma \vdash \mathtt{fst}\ \langle M,N\rangle : t} \longrightarrow \Gamma \vdash M : t$$

$$\dfrac{\dfrac{\Gamma \vdash t \quad \Gamma \vdash u}{\Gamma \vdash t \wedge u}}{\Gamma \vdash t} \qquad \Gamma \vdash t$$

Hilbert System for Implication

Axiom Schema

$$\dfrac{}{p \Rightarrow (q \Rightarrow p)}$$

$$\dfrac{}{(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))}$$

Modus Ponens

$$\dfrac{p \Rightarrow q \quad p}{q}$$

Summary

λ-Calculus		Logic
Type	⟷	Proposition
Term (program)	⟷	Proof
Reduction	⟷	Proof Normalization

A Typed Functional Language: Base Types

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{tt} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{ff} : \texttt{bool}}$$

$$\frac{\Gamma \vdash M_1 : \qquad\qquad \Gamma \vdash M_2 :}{\Gamma \vdash M_1 + M_2 :}$$

$$\frac{\Gamma \vdash M_1 : \qquad\qquad \Gamma \vdash M_2 :}{\Gamma \vdash M_1 == M_2 :}$$

$$\frac{\Gamma \vdash M_1 : \qquad\qquad \Gamma \vdash M_2 : \qquad\qquad \Gamma \vdash M_3 :}{\Gamma \vdash \texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 :}$$

A Typed Functional Language: Local Variables

$$\frac{\Gamma \vdash M : \qquad\qquad \Gamma, \texttt{x}: \quad \vdash N :}{\Gamma \vdash \texttt{let } x{=}M \texttt{ in } N :}$$

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

```
(if ff then tt 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 | n | tt | ff
      |  M+N | M==N
      |  λx:t.M |  M N
      |  <M,N> | fst M | snd M
      |  if M then N1 else N2
      |  let x=M in N

t,u ::=  int | bool
      |  t -> u
      |  t * u
```

- 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 Y as a primitive, with a rule like Y $M \rightarrow M($Y $M)$
  - Or, by taking recursive functions as primitive.

Recursive Function Values

- The function value

  `fix f(x) is M`

  corresponds roughly to the SML code

```
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`!

Recursive Function Values

M,N ::= ... | fix $f(x)$ is M

$$\frac{}{(\texttt{fix } f(x) \texttt{ is } M)\ N \rightarrow M[x \rightarrow N][f \rightarrow \texttt{fix } f(x) \texttt{ is } M]}$$

$$\frac{\Gamma, x{:}t_1, f{:}t_1\text{->}t_2 \vdash M : t_2}{\Gamma \vdash \texttt{fix } f(x) \texttt{ is } M : t_1\text{->}t_2}$$