Objectives

Polymorphic Typing Semantics

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

The Language

- We are going to type λ-calculus extended with let, if, arithmetic, and comparisons.

\[
L ::= \lambda x.L \mid LL \mid \text{let } x = L \text{ in } L \mid \text{if } L \text{ then } L \text{ else } L \text{ fi} \mid E
\]

\[
E ::= x \mid n \mid b \mid E \oplus E \mid E \sim E \mid E \&\& E \mid E || E
\]

Remember the Let Rule?

- Remember this rule for let:

\[
\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \cup [x : \sigma] \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \quad \text{LET}
\]

- We cannot type check things like this:

\[
\text{let } f = \lambda x \rightarrow x \text{ in } (f \ "hi", f \ 30)
\]

- What is the type of id here?

\[
id x = x
\]
Type Variables in Rules

A monotype \( \tau \) can be a

- type constant (e.g., \( \text{Int} \), \( \text{Bool} \), etc.)
- instantiated type constructor (e.g., \([\text{Int}]\), \(\text{Int} \to \text{Int}\))
- a type variable \( \alpha \)

A polytype \( \sigma \) can be a

- monotype \( \tau \)
- qualified type \( \forall \alpha. \sigma \)

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
id :: forall a . a -> a
id x = x
```

- The UnicodeSyntax extension allows us to put \( \forall \) directly in the source code.

```
id :: \forall a . a -> a
```

Monotypes and Polytypes

```haskell
-- Some Haskell polytype functions
head :: forall a . [a] -> a
length :: forall a . [a] -> Int -- sortof
id :: forall a . a -> a
map :: forall a b . (a -> b) -> [a] -> [b] -- sortof
```

- In Haskell, the forall part is implicit at the top level!

Some Rules

- Monomorphic variable rule:
  \[ \Gamma \vdash x : \tau \quad \text{VAR, if } x : \tau \in \Gamma \]

- Polymorphic variable rule:
  \[ \Gamma \vdash x : \sigma \quad \text{VAR, if } x : \sigma \in \Gamma \]

- The function and application rules are the same as before.

```
Γ ⊢ e1 : α2 → α
Γ ⊢ e2 : α2
Γ ⊢ e1 e2 : α
```

```
Γ ⊢ λx.e : α1 → α2
```

Leveling up Let

- Here is the old let rule again.

```
Γ ∪ [x : τ1] ⊢ e2 : τ2
Γ ⊢ e1 : τ1
Γ ⊢ let x = e1 in e2 : τ2
```

- Here is our new one.

```
Γ ∪ [x : σ1] ⊢ e2 : τ2
Γ ⊢ e1 : σ1
Γ ⊢ let x = e1 in e2 : τ2
```

Gen and Inst

Gen

\[
\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\sigma}, \text{ where } \alpha \text{ is not free in } \Gamma
\]

Example:

\[
\frac{\Gamma \vdash \lambda x.x : \alpha \to \alpha}{\Gamma \vdash \lambda x.x : \forall \alpha.\alpha \to \alpha}
\]

Inst

\[
\frac{\Gamma \vdash e : \sigma'}{\Gamma \vdash e : \sigma}, \text{ when } \sigma' \geq \sigma
\]

Example:

\[
\frac{\Gamma \vdash id : \forall \alpha.\alpha \to \alpha}{\Gamma \vdash id : \text{Int} \to \text{Int}}
\]

Type Hierarchy

- What is $\sigma \geq \sigma'$
  - We can get $\sigma'$ from $\forall \alpha.\sigma$ by consistently replacing a particular $\alpha$ with a monotype $\tau$ and removing the quantifier.
  - Type variables in the result that are free can be quantified.
  - Examples:
    - $\forall \alpha.\alpha \to \alpha \geq \text{Int} \to \text{Int}$
    - $\forall \alpha.\alpha \to \alpha \geq \text{Bool} \to \text{Bool}$
    - $\forall \alpha.\alpha \to \alpha \geq \forall \beta.\beta \to \beta$
  - Nonexamples:
    - $\forall \alpha.\alpha \to \alpha \geq \text{Int} \to \text{Bool}$
    - $\forall \alpha.\alpha \to \alpha \geq \alpha \to \text{Bool}$
    - $\forall \alpha.\alpha \to \alpha \geq \forall \beta.\beta \to \text{Int}$

Example 1

To Prove:

\[
\Gamma \equiv \{ \text{id} : \forall \alpha.\alpha \to \alpha, n : \text{Int} \} \vdash \text{id} \ n : \text{Int}
\]
Example 1

\[ \Gamma \vdash \text{id} : \forall \alpha. \alpha \to \alpha \quad \forall \alpha. \alpha \to \alpha \geq \text{Int} \to \text{Int} \]

\[ \Gamma \vdash \text{id} : \text{Int} \to \text{Int} \]

\[ \text{Inst} \]

\[ \Gamma \equiv \{ \text{id} \in \forall \alpha. \alpha \to \alpha, n \in \text{Int} \} \vdash \text{id} \; n : \text{Int} \]

Example 2

To Prove:

\[ \Gamma \equiv \{ \} \vdash \text{let} \; f = \lambda x. x \; \text{in} \; f : \forall \alpha. \alpha \to \alpha \]

\[ \text{LET} \]

A weird thing about let and functions.

To Prove:

\[ \{ x : \alpha \} \vdash x : \alpha \]

\[ \text{VAR} \]

\[ \{ \} \vdash \lambda x. x : \alpha \to \alpha \]

\[ \text{ABS} \]

\[ \{ \} \vdash \lambda x. \forall \alpha. \alpha \to \alpha \]

\[ \text{GEN} \]

\[ \{ f : \forall \alpha. \alpha \to \alpha \} \vdash f : \forall \alpha. \alpha \to \alpha \]

\[ \text{VAR} \]

\[ \Gamma \equiv \{ \} \vdash \text{let} \; f = \lambda x. x \; \text{in} \; f : \forall \alpha. \alpha \to \alpha \]

\[ \text{LET} \]

▶ The two following expressions would seem to be equivalent, yes?

▶ Expression 1:

\[ \text{let} \; f = \lambda x. x \; \text{in} \; (f \; \text{"hi"}, f \; 10) \]

▶ Expression 2:

\[ (\lambda x. f \; \text{"hi"}, f \; 10) \; (\lambda x. x) \]

▶ Try this at home and see what happens!
What happens...

- What’s going on here?

1. `Main> let f = \x -> x in (f "hi", f 10)`
2. `("hi",10)`
3. `Main> (\f -> (f "hi", f 10)) (\x -> x)`
4. No instance for (Num [Char]) arising from the literal '10'
5. In the first argument of 'f', namely '10'
6. In the expression: `f 10`
7. In the expression: `(f "hi", f 10)`

Type checking the trouble maker

- Add pairs to our list of type constructors.
- Type check this:

```
App{} ⊢ (\f .(f "hi",f 10))(\x .x):(String,Int)
```

- And then type check this:

```
App{} ⊢ let f = (\x .x) in (f "hi", f 3):(String,Int)
```