Polytype Semantics

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Use the `Gen` and `Inst` rules to introduce polymorphic types.
- Explain the $\forall$ syntax in type signatures.
- Explain the type difference between `let` and function application.
- Draw some proof trees for polymorphically typed programs.

The Language

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

$$
\begin{array}{rcll}
L & ::= & \lambda x.L & \text{abstractions} \\
 & | & L\ L & \text{applications} \\
 & | & \texttt{let}\ x = L\ \texttt{in}\ L & \text{let expressions} \\
 & | & \texttt{if}\ L\ \texttt{then}\ L\ \texttt{else}\ L\ \texttt{fi} & \text{if expressions} \\
 & | & E & \text{expressions} \\
E & ::= & x & \text{variables} \\
 & | & n & \text{integers} \\
 & | & b & \text{booleans} \\
 & | & E \oplus E & \text{integer operations} \\
 & | & E \sim E & \text{integer comparisons} \\
 & | & E\ \&\&\ E & \text{boolean and} \\
 & | & E\ ||\ E & \text{boolean or}
\end{array}
$$

Remember the Let Rule?

- Remember this rule for `let` :

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \cup [x : \sigma] \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\, \texttt{x} = e_1 \,\texttt{in}\, e_2 : \tau}\ \textsc{Let}$$

- We cannot type check things like this:

```
let f = \x -> x in (f "hi", f 30)
```

- What is the type of `id` here?

```
id x = x
```

Type Variables in Rules

A *monotype* τ can be a

- Type constant (e.g., `Int` , `Bool` , etc.)
- Instantiated type constructor (e.g., `[Int]`, `Int` $\to$ `Int`)
- A type variable α

A *polytype* σ can be a

- Monotype τ
- Qualified type $\forall \alpha . \sigma$

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

- The `UniodeSyntax` extension allows us to put $\forall$ directly in the source code.
  `id :: ∀ 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:

$$\frac{}{\Gamma \vdash x : \tau}\ \textsc{Var}, \text{ if } x : \tau \in \Gamma$$

- Polymorphic variable rule:

$$\frac{}{\Gamma \vdash x : \sigma}\ \textsc{Var}, \text{ if } x : \sigma \in \Gamma$$

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

$$\frac{\Gamma \vdash e_1 : \alpha_2 \to \alpha \qquad \Gamma \vdash e_2 : \alpha_2}{\Gamma \vdash e_1\, e_2 \ : \alpha}\ \textsc{App}$$

$$\frac{\Gamma \cup \{x : \alpha_1\} \vdash e : \alpha_2}{\Gamma \vdash \lambda x.e : \alpha_1 \to \alpha_2}\ \textsc{Abs}$$

Leveling Up Let

- Here is the old `let` rule again.

$$\frac{\Gamma \cup [x : \tau_1] \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \texttt{let x} = e_1 \texttt{ in } e_2 : \tau_2}\ \textsc{Let}$$

- Here is our new one.

$$\frac{\Gamma \cup [x : \sigma_1] \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash \texttt{let x} = e_1 \texttt{ in } e_2 : \tau_2}\ \textsc{Let}$$

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:

$$\frac{\Gamma \vdash \lambda x . x : \alpha \to \alpha}{\Gamma \vdash \lambda x . x : \forall \alpha . \alpha \to \alpha}\ \textsc{Gen}$$

Inst

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

Example:

$$\frac{\Gamma \vdash id : \forall \alpha . \alpha \to \alpha}{\Gamma \vdash id : \mathtt{Int} \to \mathtt{Int}}\ \textsc{Inst}$$

Type Hierarchy

- What is $\sigma \geq \sigma'$?
- We can get σ' from $\forall \alpha . \sigma$ by consistently replacing a particular α with a monotype τ and removing the quantifier.
- Type variables in the result that are free can be quantified.
- Examples:

$$\forall \alpha . \alpha \to \alpha \geq \texttt{Int} \to \texttt{Int}$$
$$\forall \alpha . \alpha \to \alpha \geq \texttt{Bool} \to \texttt{Bool}$$
$$\forall \alpha . \alpha \to \alpha \geq \forall \beta . \beta \to \beta$$

t

- Nonexamples:

$$\forall \alpha . \alpha \to \alpha \geq \texttt{Int} \to \texttt{Bool}$$
$$\forall \alpha . \alpha \to \alpha \geq \alpha \to \texttt{Bool}$$
$$\forall \alpha . \alpha \to \alpha \geq \forall \beta . \beta \to \texttt{Int}$$

Example 1

To prove:

$$\frac{}{\Gamma \equiv \{\texttt{id} : \forall\alpha.\alpha \to \alpha, \texttt{n} : \texttt{Int}\} \vdash \texttt{id n} : \texttt{Int}}$$

Example 1

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{id} : \texttt{Int} \rightarrow \texttt{Int}}\textsc{Inst} \quad \dfrac{}{\Gamma \vdash \texttt{n} : \texttt{Int}}\textsc{Var}}{\Gamma \equiv \{\texttt{id} : \forall\alpha.\alpha \rightarrow \alpha, \texttt{n} : \texttt{Int}\} \vdash \texttt{id n} : \texttt{Int}}\textsc{App}$$

Example 1

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{id} : \forall\alpha.\alpha \to \alpha}\ \textsc{Var} \qquad \forall\alpha.\alpha \to \alpha \geq \texttt{Int} \to \texttt{Int}}{\Gamma \vdash \texttt{id} : \texttt{Int} \to \texttt{Int}}\ \textsc{Inst} \qquad \dfrac{}{\Gamma \vdash \texttt{n} : \texttt{Int}}\ \textsc{Var}}{\Gamma \equiv \{\texttt{id} : \forall\alpha.\alpha \to \alpha, \texttt{n} : \texttt{Int}\} \vdash \texttt{id}\ \texttt{n} : \texttt{Int}}\ \textsc{App}$$

Example 2

To prove:

$$\frac{}{\Gamma \equiv \{\} \vdash \texttt{let}\ f\ =\ \lambda x.x\ \texttt{in}\ f : \forall\alpha.\alpha \to \alpha}\ \textsc{Let}$$

Example 2

To prove:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\{x : \alpha\} \vdash x : \alpha}\ \textsc{Var}}{\{\} \vdash \lambda x.x : \alpha \rightarrow \alpha}\ \textsc{Abs}}{\{\} \vdash \lambda x.x : \forall \alpha.\alpha \rightarrow \alpha}\ \textsc{Gen} \qquad \dfrac{}{\{f : \forall \alpha.\alpha \rightarrow \alpha\} \vdash f : \forall \alpha.\alpha \rightarrow \alpha}\ \textsc{Var}}{\Gamma \equiv \{\} \vdash \texttt{let}\ f\ =\ \lambda x.x\ \texttt{in}\ f : \forall \alpha.\alpha \rightarrow \alpha}\ \textsc{Let}
$$

A Weird Thing about Let and Functions

- The two following expressions would seem to be equivalent, yes?
  - Expression 1:

```
let f = \ x -> x in (f "hi", f 10)
```

  - Expression 2:

```
(\f -> (f "hi", f 10)) (\x -> x)
```

- Try this at home and see what happens!

What Happens ...

- What's going on here?

```
Main> let f = \x -> x in (f "hi", f 10)
("hi",10)
Main> (\f -> (f "hi", f 10)) (\x -> x)

    No instance for (Num [Char]) arising from the literal ‘10’
    In the first argument of ‘f’, namely ‘10’
    In the expression: f 10
    In the expression: (f "hi", f 10)
```

Type Checking the Troublemaker

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

$$\dfrac{}{\{\} \vdash (\lambda \mathtt{f}\ .(\mathtt{f}\ \text{"}hi\text{"}, \mathtt{f}\ 10))\ (\lambda \mathtt{x}\ .\mathtt{x}\ ) : (\mathtt{String}, \mathtt{Int})}\ \text{App}$$

- And then type check this:

$$\dfrac{}{\{\} \vdash \mathtt{let}\ \mathtt{f}\ = (\lambda \mathtt{x}\ .\mathtt{x}\ )\ \mathtt{in}\ (\mathtt{f}\ \text{"}hi\text{"}, \mathtt{f}\ 3) : (\mathtt{String}, \mathtt{Int}\ )}\ \text{Let}$$