Monotype Semantics

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Explain the parts of a type judgment.
- Build proof trees to indicate the derivation of a type for a program.
- Explain the circumstances under which a type environment can be modified.

The Language

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

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

Format of a Type Judgment

A *type judgment* has the following form:

$$\Gamma \vdash e : \alpha$$

where Γ is a *type environment*, e is some expression, and α is a *type*.

- $\Gamma \vdash \texttt{if true then}\ 4\ \texttt{else}\ 38 : \texttt{Int}$
- $\Gamma \vdash \texttt{true \&\& false} : \texttt{Bool}$

Note: the $\vdash$ is pronounced "turnstile" or "entails."

The Parts of a Rule

Assumptions on top

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{Int}} \textsc{BinOp}$$

Conclusion on the bottom

- If a rule has no assumptions, then it is called an *axiom*.
- Γ is a set of the form $\{x : \alpha; \ldots\}$.
- Γ may be left out if we don't need a type environment.
- **Basic idea**: the meaning of an expression can be determined by combining the meaning of its parts.

Axioms

Constants

$$\frac{}{\Gamma \vdash n : \texttt{Int}} \textsc{Const}, \text{ when } n \text{ is an integer.}$$

Similarly for `True` and `False`.

Variables

$$\frac{}{\Gamma \vdash x : \alpha} \textsc{Var}, \text{ when } x : \alpha \in \Gamma$$

- Here, α is a *type variable*; it stands for another type.
- These are rules that are true no matter what the context is.

Simple Rules

Binary Arithmetic

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{Int}} \textsc{BinOp}$$

Integer Relations

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 \sim e_2 : \texttt{Bool}} \textsc{RelOp}$$

Booleans Ops

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \texttt{Bool}}{\Gamma \vdash e_1 \&\& \, e_2 : \texttt{Bool}} \textsc{BoolOp}$$

You can actually conflate these rules by using signatures.

Example 0

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7) \&\& \, y : \texttt{Bool}$.
Assume that $\Gamma = \{x : \texttt{Int} \,; y : \texttt{Bool}\}$

First thing: Write down the thing you are trying to prove, and put a bar over it.

$$\frac{}{\Gamma \vdash (x * 5 > 7) \&\& \, y : \texttt{Bool}}$$

Look at the *outermost* expression. What rule applies here?

Example 0

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}$.
Assume that $\Gamma = \{x : \mathtt{Int} \, ; y : \mathtt{Bool}\}$

First thing: Write down the thing you are trying to prove, and put a bar over it.

$$\frac{}{\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}}$$

Look at the *outermost* expression. What rule applies here?

$$\frac{\Gamma \vdash e_1 : \mathtt{Bool} \qquad \Gamma \vdash e_2 : \mathtt{Bool}}{\Gamma \vdash e_1 \&\& \, e_2 : \mathtt{Bool}}\ \textsc{BoolOp}$$

Example 0

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}$.
Assume that $\Gamma = \{x : \mathtt{Int} \, ; y : \mathtt{Bool}\}$

Write parts on top and put a bar over them as well.

$$\frac{\dfrac{}{\Gamma \vdash x * 5 > 7 : \mathtt{Bool}} \qquad \dfrac{}{\Gamma \vdash y : \mathtt{Bool}}}{\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}}\ \textsc{BoolOp}$$

What to do next? Let's work left to right. The expression we want next is a "greater" expression. (Besides, the y expression is already an axiom.)

Example 0

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}$.
Assume that $\Gamma = \{x : \mathtt{Int} \, ; y : \mathtt{Bool}\}$

Following the "greater" rule, we break the `x * 5 > 7` into two parts.

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash x * 5 : \mathtt{Int}} \qquad \dfrac{}{\Gamma \vdash 7 : \mathtt{Int}}}{\Gamma \vdash x * 5 > 7 : \mathtt{Bool}}\ \textsc{RelOp} \qquad \dfrac{}{\Gamma \vdash y : \mathtt{Bool}}}{\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}}\ \textsc{BoolOp}$$

We will turn our attention to the multiplication now.

Example 0

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}$.
Assume that $\Gamma = \{x : \mathtt{Int} \, ; y : \mathtt{Bool}\}$

$$\frac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \mathtt{Int}}\ \textsc{Var} \qquad \dfrac{}{\Gamma \vdash 5 : \mathtt{Int}}\ \textsc{Const}}{\Gamma \vdash x * 5 : \mathtt{Int}}\ \textsc{BinOp} \qquad \dfrac{}{\Gamma \vdash 7 : \mathtt{Int}}\ \textsc{Const}}{\Gamma \vdash x * 5 > 7 : \mathtt{Bool}}\ \textsc{RelOp} \qquad \dfrac{}{\Gamma \vdash y : \mathtt{Bool}}\ \textsc{Var}}{\Gamma \vdash (x * 5 > 7) \&\& \, y : \mathtt{Bool}}\ \textsc{BoolOp}$$

At this point, there are no more subtrees to expand out. We are done.

Type Variables in Rules

A *monotype* τ can be a

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

If Rule

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \alpha \qquad \Gamma \vdash e_3 : \alpha}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \alpha}\ \textsc{If}$$

- Here, α is a meta-variable.
- This rule says that `if` can result in any type, as long as the `then` and `else` branches have the same type. This could even include functions.

Function Application

$$\frac{\Gamma \vdash e_1 : \alpha_2 \rightarrow \alpha \qquad \Gamma \vdash e_2 : \alpha_2}{\Gamma \vdash e_1\ e_2\ : \alpha}\ \textsc{Fun}$$

- If you have a function of type $\alpha_2 \rightarrow \alpha$ and an argument e_2 of type α_2, then applying e_1 to e_2 will produce an expression of type α.
- You can generalize this rule to multiple arguments.

$$\frac{\Gamma \vdash \texttt{incList} : [\texttt{Int}] \rightarrow [\texttt{Int}] \qquad \Gamma \vdash \texttt{xx} : [\texttt{Int}]}{\Gamma \vdash \texttt{incList xx} : [\texttt{Int}]}\ \textsc{Fun}$$

Function Rule

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

- Important point: this rule describes types and also describes when you may change Γ.
- You may **NOT** change Γ except as described!

Example: show that $\{\} \vdash \lambda x.x + 1 : \texttt{Int} \rightarrow \texttt{Int}$.

Function Rule

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

- Important point: this rule describes types and also describes when you may change Γ.
- You may **NOT** change Γ except as described!

$$\frac{}{\{\} \vdash \lambda x.x + 1 : \texttt{Int} \rightarrow \texttt{Int}}\ \textsc{Abs}$$

Function Rule

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

- Important point: this rule describes types and also describes when you may change Γ.
- You may **NOT** change Γ except as described!

$$\frac{\frac{}{\{x : \texttt{Int}\} \vdash x + 1 : \texttt{Int}}}{\{\} \vdash \lambda x.x + 1 : \texttt{Int} \to \texttt{Int}}\ \textsc{Abs}$$

Introduction Typing Rules Monotypes

Function Rule

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

- Important point: this rule describes types and also describes when you may change Γ.
- You may **NOT** change Γ except as described!

$$\frac{\dfrac{\dfrac{}{\{x : \texttt{Int}\} \vdash x : \texttt{Int}}\ \textsc{Var} \qquad \dfrac{}{\{x : \texttt{Int}\} \vdash 1 : \texttt{Int}}\ \textsc{Const}}{\{x : \texttt{Int}\} \vdash x + 1 : \texttt{Int}}\ \textsc{BinOp}}{\{\} \vdash \lambda x.x + 1 : \texttt{Int} \to \texttt{Int}}\ \textsc{Abs}$$

Let Rule

- Here is `let`. Note that Haskell uses the recursive rule, and it is polymorphic.

Let

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

Letrec

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