Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Format of Type Judgments

- A *type judgement* has the form

  Γ |- exp : τ
- Γ is a typing environment
  - Supplies the types of variables and functions
  - Γ is a list of the form [x : σ , . . .]
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies")

Axioms - Constants

$$\frac{}{\vdash n : \text{int}} \quad \text{(assuming } n \text{ is an integer constant)}$$

$$\frac{}{\vdash \text{true} : \text{bool}} \qquad \frac{}{\vdash \text{false} : \text{bool}}$$

- These rules are true with any typing environment
- *n* is a meta-variable

Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$ and there is no $x : \tau$ to the left of $x : \sigma$ in Γ

Variable axiom:

$$\frac{}{\Gamma \vdash x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Simple Rules - Arithmetic

Primitive operators ($\oplus \in \{ +, -, *, \ldots\}$):

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad (\oplus) : \tau \rightarrow \tau \rightarrow \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

Relations ($\sim\ \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \text{bool}}$$

Type Variables in Rules

- If_then_else rule:

$$\frac{\Gamma \mid\!- e_1 : \text{bool} \quad \Gamma \mid\!- e_2 : \tau \quad \Gamma \mid\!- e_3 : \tau}{\Gamma \mid\!- (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- τ is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if_then_else must all have same type

Function Application

- Application rule:

$$\frac{\Gamma \mid\!- e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \mid\!- e_2 : \tau_1}{\Gamma \mid\!- (e_1\ e_2) : \tau_2}$$

- If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument of type τ_1, the resulting expression has type τ_2

Fun Rule

- Rules describe types, but also how the environment Γ may change
- Can only do what rule allows!
- fun rule:

$$\frac{[x : \tau_1] + \Gamma \mid\!- e : \tau_2}{\Gamma \mid\!- \text{fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

Fun Examples

$$\frac{[y : \text{int}] + \Gamma \mid\!- y + 3 : \text{int}}{\Gamma \mid\!- \text{fun } y \text{ -> } y + 3 : \text{int} \rightarrow \text{int}}$$

$$\frac{[f : \text{int} \rightarrow \text{bool}] + \Gamma \mid\!- f\ 2 :: [\text{true}] : \text{bool list}}{\Gamma \mid\!- (\text{fun } f \text{ -> } f\ 2 :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}}$$

(Monomorphic) Let and Let Rec

- let rule:

$$\frac{\Gamma \mid\!- e_1 : \tau_1 \qquad [x : \tau_1] + \Gamma \mid\!- e_2 : \tau_2}{\Gamma \mid\!- (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

$$\frac{[x : \tau_1] + \Gamma \mid\!- e_1 : \tau_1 \quad [x : \tau_1] + \Gamma \mid\!- e_2 : \tau_2}{\Gamma \mid\!- (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$

Example

- Which rule do we apply?

$$\frac{?}{\mid\!- (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \text{ -> } (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}}$$

Example

- Let rec rule: (2) [one : int list] |- (let x = 2 in fun y -> (x :: y :: one)) : int → int list

(1) [one : int list] |- (1 :: one) : int list

|- (let rec one = 1 :: one in
let x = 2 in
fun y -> (x :: y :: one)) : int → int list

Proof of 1

- Which rule?

[one : int list] |- (1 :: one) : int list

Proof of 1

- Application

(3) [one : int list] |- ((::) 1): int list→ int list

(4) [one : int list] |- one : int list

[one : int list] |- (1 :: one) : int list

Proof of 3

Constants Rule Constants Rule

[one : int list] |- (::) : int → int list→ int list

[one : int list] |- 1 : int

[one : int list] |- ((::) 1) : int list → int list

Proof of 4

- Rule for variables

[one : int list] |- one:int list

Proof of 2

- Constant

[one : int list] |- 2:int

(5) [x:int; one : int list] |- fun y -> (x :: y :: one)) : int → int list

[one : int list] |- (let x = 2 in fun y -> (x :: y :: one)) : int → int list

Proof of 5

?

[x:int; one : int list] |- fun y -> (x :: y :: one))
: int → int list

Proof of 5

?

[y:int; x:int; one : int list] |- (x :: y :: one) : int list

[x:int; one : int list] |- fun y -> (x :: y :: one))
: int → int list

Proof of 5

6
[y:int; x:int; one : int list] |- ((::) x):int list→ int list

7
[y:int; x:int; one : int list] |- (y :: one) : int list

[y:int; x:int; one : int list] |- (x :: y :: one) : int list

[x:int; one : int list] |- fun y -> (x :: y :: one))
: int → int list

Proof of 6

Constant

[...] |- (::)
: int→ int list→ int list

Variable

[...; x:int;...] |- x:int

[y:int; x:int; one : int list] |- ((::) x)
:int list→ int list

Proof of 7

Pf of 6 [y/x]

⋮

[y:int; ...] |- ((::) y)
:int list→ int list

Variable

[...; one: int list] |-
one: int list

[y:int; x:int; one : int list] |- (y :: one) : int list

Curry - Howard Isomorphism

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms
- Functions space arrow corresponds to implication; application corresponds to modus ponens

Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

- Application

$$\frac{\Gamma \;|\!\!-\; e_1 : \alpha \to \beta \quad \Gamma \;|\!\!-\; e_2 : \alpha}{\Gamma \;|\!\!-\; (e_1\ e_2) : \beta}$$

Mia Copa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only meta-variable in the logic)
- Would need:
  - Object level type variables and some kind of type quantification
  - **let** and **let rec** rules to introduce polymorphism
  - Explicit rule to eliminate (instantiate) polymorphism