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

Type Inference

- *Type inference*: A program analysis to assign a type to an expression from the program context of the expression
  - Fully static type inference first introduced by Robin Miller in ML
  - Haskle, OCAML, SML all use type inference
    - Records are a problem for type inference

Format of Type Judgments

- A *type judgement* has the form

  Γ |- exp : τ
- Γ is a typing environment
  - Supplies the types of variables (and function names when function names are not variables)
  - Γ is a set of the form $\{ x:\sigma , \ldots \}$
  - For any x at most one σ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies" or, informally, "shows")

Axioms - Constants

$$\frac{}{\Gamma \text{ |- } n : \text{int}}$$ (assuming n is an integer constant)

$$\frac{}{\Gamma \text{ |- true : bool}} \qquad \frac{}{\Gamma \text{ |- false : bool}}$$

- These rules are true with any typing environment
- Γ, n are meta-variables

Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$

Note: if such σ exits, its unique

Variable axiom:

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

Simple Rules - Arithmetic

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

$$\frac{\Gamma \text{ |- } e_1 : \tau_1 \quad \Gamma \text{ |- } e_2 : \tau_2 \quad (\oplus) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3}{\Gamma \text{ |- } e_1 \oplus e_2 : \tau_3}$$

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

$$\frac{\Gamma \text{ |- } e_1 : \tau \quad \Gamma \text{ |- } e_2 : \tau \quad (\sim) : \tau \rightarrow \tau \rightarrow \text{bool}}{\Gamma \text{ |- } e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

Example: {x:int} |- x + 2 = 3 :bool

What do we need to show first?

{x:int} |- x + 2 = 3 : bool

Example: {x:int} |- x + 2 = 3 :bool

What do we need for the left side?

$$\frac{\{x : int\} \mid\!- x + 2 : int \qquad \{x:int\} \mid\!- 3 : int}{\{x:int\} \mid\!- x + 2 = 3 : bool}\ \text{Bin}$$

Example: {x:int} |- x + 2 = 3 :bool

How to finish?

$$\frac{\dfrac{\{x:int\} \mid\!- x:int \quad \{x:int\} \mid\!- 2:int}{\{x : int\} \mid\!- x + 2 : int}\ \text{Bin} \qquad \{x:int\} \mid\!- 3 : int}{\{x:int\} \mid\!- x + 2 = 3 : bool}\ \text{Bin}$$

Example: {x:int} |- x + 2 = 3 :bool

Complete Proof (type derivation)

$$\frac{\dfrac{\dfrac{}{\{x:int\} \mid\!- x:int}\ \text{Var} \quad \dfrac{}{\{x:int\} \mid\!- 2:int}\ \text{Const}}{\{x : int\} \mid\!- x + 2 : int}\ \text{Bin} \qquad \dfrac{}{\{x:int\} \mid\!- 3 : int}\ \text{Const}}{\{x:int\} \mid\!- x + 2 = 3 : bool}\ \text{Bin}$$

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid\!- e_1 : bool \qquad \Gamma \mid\!- e_2 : bool}{\Gamma \mid\!- e_1 \mathbin{\&\&} e_2 : bool}$$

$$\frac{\Gamma \mid\!- e_1 : bool \qquad \Gamma \mid\!- e_2 : bool}{\Gamma \mid\!- e_1 \mathbin{||} e_2 : bool}$$

Type Variables in Rules

- If_then_else rule:

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

- 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 e_2 of type τ_1, the resulting expression $e_1 e_2$ 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

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

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

(Monomorphic) Let and Let Rec

- let rule:

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

- 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?

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

Example

- Let rec rule:

(1) {one : int list} |- (1 :: one) : int list

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

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

Proof of 1

- Which rule?

{one : int list} |- (1 :: one) : int list

Proof of 1

- Binary Operator

③ {one : int list} |- 1: int

④ {one : int list} |- one : int list

{one : int list} |- (1 :: one) : int list

where (::) : int → int list → int list

Proof of 1

③ Constant Rule

{one : int list} |- 1: int

④ Variable Rule

{one : int list} |- one : int list

{one : int list} |- (1 :: one) : int list

Proof of 2

- Let Rule

{one : int list} |- 2:int

{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 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

By the Fun Rule

Proof of 5

⑥ ⑦

{y:int; x:int; one:int list} |- x:int   {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

By BinOp where (::) : int → int list → int list

Proof of 6

⑥ ⑦

Variable Rule

{y:int; x:int; one:int list} |- x:int   {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 7

- Binary Operation Rule

{y:int; …} |- y:int   {…; one:int list;…} |- one : int list

{y:int; x:int; one : int list}|- (y :: one) : int list

By BinOp where (::) : int → int list → int list

Proof of 7

Variable Rule   Variable Rule

{y:int; …} |- y:int   {…; 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

- Function space arrow corresponds to implication; application corresponds to modus ponens

Curry - Howard Isomorphism

- Modus Ponens

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

- Application

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

Mea Culpa

- 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

Support for Polymorphic Types

- Monomorpic Types (τ):
  - Basic Types: int, bool, float, string, unit, …
  - Type Variables: α, β, γ, δ, ε
  - Compound Types: $\alpha \rightarrow \beta$, int * string, bool list, …
- Polymorphic Types:
  - Monomorphic types τ
  - Universally quantified monomorphic types
  - $\forall \alpha_1, \ldots, \alpha_n . \tau$
  - Can think of τ as same as $\forall . \tau$

Example FreeVars Calculations

- Vars('a -> (int -> 'b) -> 'a) ={'a , 'b}
- FreeVars (All 'b. 'a -> (int -> 'b) -> 'a) =
- {'a , 'b} – {'b}= {'a}
- FreeVars {x : All 'b. 'a -> (int -> 'b) -> 'a,
- id: All 'c. 'c -> 'c,
- y: All 'c. 'a -> 'b -> 'c} =
- {'a} U {} U {'a, 'b} = {'a, 'b}

Support for Polymorphic Types

- Typing Environment Γ supplies polymorphic types (which will often just be monomorphic) for variables
- Free variables of monomorphic type just type variables that occur in it
  - Write FreeVars(τ)
- Free variables of polymorphic type removes variables that are universally quantified
  - FreeVars($\forall \alpha_1, \ldots, \alpha_n . \tau$) = FreeVars($\tau$) – $\{\alpha_1, \ldots, \alpha_n\}$
- FreeVars(Γ) = all FreeVars of types in range of Γ

Monomorphic to Polymorphic

- Given:
  - type environment Γ
  - monomorphic type τ
  - τ shares type variables with Γ
- Want most polymorphic type for τ that doesn't break sharing type variables with Γ
- Gen(τ, Γ) = $\forall \alpha_1, \ldots, \alpha_n . \tau$ where $\{\alpha_1, \ldots, \alpha_n\}$ = freeVars(τ) – freeVars(Γ)

Polymorphic Typing Rules

- A *type judgement* has the form

  Γ |- exp : τ
  - Γ uses polymorphic types
  - τ still monomorphic
- Most rules stay same (except use more general typing environments)
- Rules that change:
  - Variables
  - Let and Let Rec
  - Allow polymorphic constants
- Worth noting functions again

Polymorphic Let and Let Rec

- let rule:

$$\frac{\Gamma \text{ |- } e_1 : \tau_1 \quad \{x : \text{Gen}(\tau_1,\Gamma)\} + \Gamma \text{ |- } e_2 : \tau_2}{\Gamma \text{ |- (let x = } e_1 \text{ in } e_2\text{) : } \tau_2}$$

- let rec rule:

$$\frac{\{x : \tau_1\} + \Gamma \text{ |- } e_1 : \tau_1 \quad \{x : \text{Gen}(\tau_1,\Gamma)\} + \Gamma \text{ |- } e_2 : \tau_2}{\Gamma \text{ |- (let rec x = } e_1 \text{ in } e_2\text{) : } \tau_2}$$

Polymorphic Variables (Identifiers)

Variable axiom:

$$\frac{}{\Gamma \text{ |- } x : \varphi(\tau)} \quad \text{if } \Gamma(x) = \forall \alpha_1, \ldots, \alpha_n . \tau$$

- Where φ replaces all occurrences of $\alpha_1, \ldots, \alpha_n$ by monotypes $\tau_1, \ldots, \tau_n$
- Note: Monomorphic rule special case:

$$\frac{}{\Gamma \text{ |- } x : \tau} \quad \text{if } \Gamma(x) = \tau$$

- Constants treated same way

Fun Rule Stays the Same

- fun rule:

$$\frac{\{x : \tau_1\} + \Gamma \text{ |- } e : \tau_2}{\Gamma \text{ |- fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

- Types τ_1, τ_2 monomorphic
- Function argument must always be used at same type in function body

Polymorphic Example

- Assume additional constants and primitive operators:
- hd : $\forall\alpha.\ \alpha$ list -> α
- tl: $\forall\alpha.\ \alpha$ list -> α list
- is_empty : $\forall\alpha.\ \alpha$ list -> bool
- (::) : $\forall\alpha.\ \alpha$ -> α list -> α list
- [] : $\forall\alpha.\ \alpha$ list

Polymorphic Example

- Show:

```
                                   ?
--------------------------------------------------------------
{} |- let rec length =
        fun l -> if is_empty l then 0
                 else 1 + length (tl l)
  in length (2 :: []) + length(true :: []) : int
```

Polymorphic Example: Let Rec Rule

- Show: (1) (2)

$$\frac{\{length:\alpha\ list \to int\} \mid\!- fun\ l \to \ldots : \alpha\ list \to int \qquad \{length:\forall\alpha.\ \alpha\ list \to int\} \mid\!- length\ (2 :: []) + length(true :: []) : int}{\{\} \mid\!- let\ rec\ length = fun\ l \to if\ is\_empty\ l\ then\ 0\ else\ 1 + length\ (tl\ l)\ in\ length\ (2 :: []) + length(true :: []) : int}$$

Polymorphic Example (1)

- Show:

$$\frac{?}{\{length:\alpha\ list \to int\} \mid\!- fun\ l \to if\ is\_empty\ l\ then\ 0\ else\ 1 + length\ (tl\ l) : \alpha\ list \to int}$$

Polymorphic Example (1): Fun Rule

- Show: (3)

$$\frac{\{length:\alpha\ list \to int,\ l: \alpha\ list\} \mid\!- if\ is\_empty\ l\ then\ 0\ else\ length\ (hd\ l) + length\ (tl\ l) : int}{\{length:\alpha\ list \to int\} \mid\!- fun\ l \to if\ is\_empty\ l\ then\ 0\ else\ 1 + length\ (tl\ l) : \alpha\ list \to int}$$

Polymorphic Example (3)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{?}{\Gamma \mid\!- if\ is\_empty\ l\ then\ 0\ else\ 1 + length\ (tl\ l) : int}$$

Polymorphic Example (3):IfThenElse

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{(4)\ \Gamma \mid\!- is\_empty\ l : bool \qquad (5)\ \Gamma \mid\!- 0:int \qquad (6)\ \Gamma \mid\!- 1 + length\ (tl\ l) : int}{\Gamma \mid\!- if\ is\_empty\ l\ then\ 0\ else\ 1 + length\ (tl\ l) : int}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{?}{\Gamma \mid\!- is\_empty\ l : bool}$$

Polymorphic Example (4):Application

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{?}{\Gamma|\text{- is\_empty} : \alpha \text{ list -> bool}} \quad \dfrac{?}{\Gamma|\text{- l} : \alpha \text{ list}}}{\Gamma|\text{- is\_empty l : bool}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{\text{By Const since } \alpha \text{ list -> bool is instance of } \forall\alpha.\ \alpha \text{ list -> bool}}{\Gamma|\text{- is\_empty} : \alpha \text{ list -> bool}} \quad \dfrac{?}{\Gamma|\text{- l} : \alpha \text{ list}}}{\Gamma|\text{- is\_empty l : bool}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{\text{By Const since } \alpha \text{ list -> bool is instance of } \forall\alpha.\ \alpha \text{ list -> bool}}{\Gamma|\text{- is\_empty} : \alpha \text{ list -> bool}} \quad \dfrac{\text{By Variable } \Gamma(\text{l}) = \alpha \text{ list}}{\Gamma|\text{- l} : \alpha \text{ list}}}{\Gamma|\text{- is\_empty l : bool}}$$

- This finishes (4)

Polymorphic Example (5):Const

- Let Γ ={length:α list -> int, l: α list }
- Show

By Const Rule

$$\frac{}{\Gamma|\text{- 0:int}}$$

Polymorphic Example (6):Arith Op

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{\text{By Const}}{\Gamma|\text{- 1:int}} \quad \dfrac{\dfrac{\text{By Variable}}{\Gamma|\text{- length} : \alpha \text{ list -> int}} \quad \dfrac{(7)}{\Gamma|\text{- (tl l)} : \alpha \text{ list}}}{\Gamma|\text{- length (tl l) : int}}}{\Gamma|\text{- 1 + length (tl l) : int}}$$

Polymorphic Example (7):App Rule

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{\text{By Const}}{\Gamma|\text{- tl} : \alpha \text{ list -> } \alpha \text{ list}} \quad \dfrac{\text{By Variable}}{\Gamma|\text{- l} : \alpha \text{ list}}}{\Gamma|\text{- (tl l)} : \alpha \text{ list}}$$

By Const since α list -> α list is instance of $\forall\alpha.$ α list -> α list

Polymorphic Example: (2) by ArithOp

- Let Γ' = {length:∀α. α list -> int}
- Show:

(8) (9)

$$\frac{\Gamma' \mid\!- \text{ length } (2 :: []) : \text{int} \qquad \Gamma' \mid\!- \text{ length(true} :: []) : \text{int}}{\{\text{length}:\forall\alpha.\ \alpha \text{ list} \to \text{int}\} \mid\!- \text{ length } (2 :: []) + \text{length(true} :: []) : \text{int}}$$

Polymorphic Example: (8)AppRule

- Let Γ' = {length:∀α. α list -> int}
- Show:

$$\frac{\Gamma' \mid\!- \text{ length} : \text{int list} \to \text{int} \qquad \Gamma' \mid\!- (2 :: []) : \text{int list}}{\Gamma' \mid\!- \text{ length } (2 :: []) : \text{int}}$$