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 Declarations

- *Type declarations*: explicit assignment of types to variables (signatures to functions) in the code of a program
  - Must be checked in a strongly typed language
  - Often not necessary for strong typing or even static typing (depends on the type system)

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

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

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

- 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:

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

Simple Rules - Arithmetic

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

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

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

$$\frac{\Gamma \mid\!- e_1 : \tau \quad \Gamma \mid\!- e_2 : \tau}{\Gamma \mid\!- e_1 \sim e_2 : \text{bool}}$$

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?

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

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

How to finish?

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

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

Complete Proof (type derivation)

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

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid\!- e_1 : \text{bool} \quad \Gamma \mid\!- e_2 : \text{bool}}{\Gamma \mid\!- e_1 \;\&\&\; e_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- e_1 : \text{bool} \quad \Gamma \mid\!- e_2 : \text{bool}}{\Gamma \mid\!- e_1 \;||\; e_2 : \text{bool}}$$

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 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

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

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

(Monomorphic) Let and Let Rec

- let rule:

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

- 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?

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

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) (7)

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

{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 Variable

{...} |- (::)
: int→ int list→ int list

{...; x:int;...} |- x:int

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

Proof of 7

Pf of 6 [y/x] Variable

⋮

{y:int; ...} |- ((::) y)
:int list→ int list

{...; 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 $\forall . \tau$

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

  $\Gamma \mid\!- \text{exp} : \tau$
  - 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:

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

- let rec rule:

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

Polymorphic Variables (Identifiers)

Variable axiom:

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

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

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

- Constants treated same way

Fun Rule Stays the Same

- fun rule:

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

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

Polymorphic Example

- Assume additional constants:
- hd :∀α. α list -> α
- tl: ∀α. α list -> α list
- is_empty : ∀α. α list -> bool
- :: : ∀α. α -> α list -> α list
- [] : ∀α. α list

Polymorphic Example: Let Rec Rule

-

?

{} |- 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)

{length:α list -> int} |- fun l -> … : α list -> int

{length:∀α. α list -> int} |- length (2 :: []) + length(true :: []) : int

---

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

Polymorphic Example (1)

- Show:

?

{length:α list -> int} |-
fun l -> if is_empty l then 0
else 1 + length (tl l)
: α list -> int

Polymorphic Example (1): Fun Rule

- Show: (3)

{length:α list -> int, l: α list } |-
if is_empty l then 0
else length (hd l) + length (tl l) : int

{length:α list -> int} |-
fun l -> if is_empty l then 0
else 1 + length (tl l)
: α list -> int

Polymorphic Example (3)

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

?

Γ|- 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

(4) (5) (6)
Γ|- is_empty l : bool Γ|- 0:int Γ|- 1 + length (tl l) : int

Γ|- 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

?

Γ|- is_empty l : bool

Polymorphic Example (4):Application

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

? ?

Γ|- is_empty : α list -> bool Γ|- l : α list

Γ|- is_empty l : bool

Polymorphic Example (4)

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

By Const since α list -> bool is instance of ∀α. α list -> bool ?

Γ|- is_empty : α list -> bool   Γ|- l : α list

Γ|- is_empty l : bool

Polymorphic Example (4)

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

By Const since α list -> bool is instance of ∀α. α list -> bool   By Variable Γ(l) = α list

Γ|- is_empty : α list -> bool   Γ|- l : α list

Γ|- is_empty l : bool

- This finishes (4)

Polymorphic Example (5):Const

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

By Const Rule

Γ|- 0:int

Polymorphic Example (6):Arith Op

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

By Variable

Γ|- length : α list -> int   (7) Γ|- (tl l) : α list

By Const Γ|- 1:int   Γ|- length (tl l) : int

Γ|- 1 + length (tl l) : int

Polymorphic Example (7):App Rule

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

By Const Γ|- tl : α list -> α list   By Variable Γ|- l : α list

Γ|- (tl l) : α list

By Const since α list -> α list is instance of ∀α. α list -> α list

Polymorphic Example: (2) by ArithOp

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

(8) Γ' |- length (2 :: []) :int   (9) Γ' |- length(true :: []) : int

{length: α. α list -> int}

|- length ((::) 2 []) + length((::) true []) : int

Polymorphic Example: (8)AppRule

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

Γ' |- length : int list ->int   Γ' |- (2 :: []):int list
Γ' |- length (2 ::[]) :int

Polymorphic Example: (8)AppRule

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

By Var since int list -> int is instance of ∀α. α list -> int

(10)

Γ' |- length : int list ->int   Γ' |- ((::)2 []):int list
Γ' |- length ((::) 2 []) :int

Polymorphic Example: (10)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:
- By Const since α list is instance of ∀α. α list

(11)

Γ' |-((::) 2) : int list -> int list   Γ' |- [] : int list
Γ' |- ((::) 2 []) :int list

Polymorphic Example: (11)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:
- By Const since α list is instance of ∀α. α list

By Const

Γ' |- (::) : int -> int list -> int list   Γ' |- 2 : int
Γ' |- ((::) 2) : int list -> int list

Polymorphic Example: (9)AppRule

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

Γ' |- length:bool list ->int   Γ' |- ((::) true []):bool list
Γ' |- length ((::) true []) :int

Polymorphic Example: (9)AppRule

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

By Var since bool list -> int is instance of ∀α. α list -> int

(12)

Γ' |- length:bool list ->int   Γ' |- ((::) true []):bool list
Γ' |- length ((::) true []) :int

Polymorphic Example: (12)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:
- By Const since α list is instance of ∀α. α list

(13)

Γ' |-((::)true):bool list ->bool list Γ' |- []:bool list

Γ' |- ((::) true []) :bool list

Polymorphic Example: (13)AppRule

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

By Const since bool list is instance of ∀α. α list — By Const

Γ' |- (::):bool ->bool list ->bool list   Γ' |- true : bool

Γ' |- ((::) true) : bool list -> bool list