Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based on slides by Elsa Gunter, which were inspired by earlier slides by Mattox Beckman, Vikram Adve, and Gul Agha

Terminology

- Type: A type *t* defines a set of possible data values
  - E.g. short in C is {x| 2^{15} - 1 ≥ x ≥ -2^{15}}
  - A value in this set is said to have type *t*
- Type system: rules of a language assigning types to expressions

Why Data Types?

- Data types play a key role in:
  - **Data abstraction** in the design of programs
  - **Type checking** in the analysis of programs
  - **Compile-time code generation** in the translation and execution of programs
    - Data layout (how many words; which are data and which are pointers) dictated by type

Types as Specifications

- Types describe **properties**
- Different type systems describe different properties:
  - Data is read-write versus read-only
  - Operation has authority to access data
  - Data came from "right" source
  - Operation might or could not raise an exception
- Common type systems focus on types describing same data layout and access methods

Sound Type System

- Type: A type *t* defines a set of possible data values
  - E.g. short in C is {x| 2^{15} - 1 ≥ x ≥ -2^{15}}
  - A value in this set is said to have type *t*
- Type system: rules of a language assigning types to expressions
- If an expression is assigned type *t*, and it evaluates to a value *v*, then *v* is in the set of values defined by *t*
- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not

Sound Type System

- But Java and Scala are also unsound

```
class Unsound {
  static class Constrain<A, B extends A> {}
  static class Bind<A> {
    <B extends A>
    A upcast(Constrain<A,B> constrain, B b) {
      return b;
    }
  }
  static <T,U> U coerce(T t) {
    Constrain<U,? super T> constrain = null;
    Bind<U> bind = new Bind<U>();
    return bind.upcast(constrain, t);
  }
  public static void main(String[] args) {
    String zero = Unsound.<Integer,String>coerce(0);
  }
}
```

Figure 1. Unsound valid Java program compiled by javac, version 1.8.0_25

- For details, see this paper: Java and Scala's Type Systems are Unsound * The Existential Crisis of Null Pointers. Amin and Tate (OOPSLA 2016)

Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is *strongly typed*
  - Eg: 1 + 2.3;;
- Depends on definition of "type error"

Strongly Typed Language

- C++ claimed to be "strongly typed", but
  - Union types allow creating a value at one type and using it at another
  - Type coercions may cause unexpected (undesirable) effects
  - No array bounds check (in fact, no runtime checks at all)
- SML, OCAML "strongly typed" but still must do dynamic array bounds checks, runtime type case analysis, and other checks

Static vs Dynamic Types

- **Static type:** type assigned to an expression at compile time
- **Dynamic type:** type assigned to a storage location at run time
- **Statically typed language:** static type assigned to every expression at compile time
- **Dynamically typed language:** type of an expression determined at run time

Type Checking

- When is op(arg1,…,argn) allowed?
- **Type checking** assures that operations are applied to **the right number of arguments** of **the right types**
  - Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- Used to resolve overloaded operations

Type Checking

- Type checking may be done **statically** at compile time or **dynamically** at run time
- Dynamically typed (aka untyped) languages (eg LISP, Prolog, JavaScript) do only dynamic type checking
- Statically typed languages can do most type checking statically

Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different types

Dynamic Type Checking

- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe years after the code was written)

Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time

Static Type Checking

- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
  - Eg: array bounds

Static Type Checking

- Typically places restrictions on languages
  - Garbage collection
  - References instead of pointers
  - All variables initialized when created
  - Variable only used at one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks

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

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

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

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

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?

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

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

How to finish?

$$\dfrac{\dfrac{\{x{:}int\} \text{ |- } x{:}int \quad \{x{:}int\} \text{ |- } 2{:}int}{\{x : int\} \text{ |- } x + 2 : int}\text{AO} \qquad \{x{:}int\} \text{ |- } 3 \, {:}int}{\{x{:}int\} \text{ |- } x + 2 = 3 : bool}\text{Rel}$$

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

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

Complete Proof (type derivation)

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

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \text{ |- } e_1 : \text{bool} \qquad \Gamma \text{ |- } e_2 : \text{bool}}{\Gamma \text{ |- } e_1 \; \&\& \; e_2 : \text{bool}}$$

$$\frac{\Gamma \text{ |- } e_1 : \text{bool} \qquad \Gamma \text{ |- } e_2 : \text{bool}}{\Gamma \text{ |- } e_1 \; || \; e_2 : \text{bool}}$$

Type Variables in Rules

- If_then_else rule:

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

- 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:

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

- 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:

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

Fun Examples

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

$$\frac{\{y : \text{int}\} + \Gamma \text{ |- } y + 3 : \text{int}}{\Gamma \text{ |- fun } y \text{ -> } y + 3 : \text{int} \to \text{int}}$$

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

10/9/2018 31

(Monomorphic) Let and Let Rec

- let rule:

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

- let rec rule:

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

10/9/2018 32

Example

- let rec rule:

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

- Which rule do we apply?

?

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

10/9/2018 33

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**

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

10/9/2018 34

Proof of 1

- Application rule:

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

- Which rule?

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

10/9/2018 35

Proof of 1

- Application rule:

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

- Application

(3) {one : int list} |- ((::) 1): **int list→ int list**

(4) {one : int list} |- one : **int list**

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

10/9/2018 36

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 1

- Application rule:

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

- 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 4

- Rule for variables

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

|- (let rec one = 1 :: one in
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

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

Like Pf of 6 [replace x w/ y]   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

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

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

Polymorphic Types

- Polymorphism is the ability of a type term to simultaneously admit several distinct types (Reynolds, 1974)
- id : 'a -> 'a ;
  - Can be instatiated to e.g.:
  - int -> int
  - bool list -> bool list
  - ('b -> 'b) -> ('b -> b')
- length : 'a list -> int
  - int list -> int
  - (float->bool) list -> int

Polymorphic Types

- let swap (x,y) = (y,x) ;;
- val swap : 'a * 'b -> 'b * 'a = <fun>

- Replaces all these spurious variants:
  - val swapInt : int * int -> int * int = <fun>
  - val swapFloat : float * float -> float * float = <fun>
  - val swapIntFloat : int * float -> float * int = <fun>
  - val swapFloatInt : float * int -> int * float = <fun>
  - ….
- The road to type abstraction: function swap knows nothing about the variables x and y except to treat them as ***generic*** "black boxes" (or generic objects)

Support for Polymorphic Types

- Monomorpic Types (τ):
  - Basic Types: int, bool, float, string, unit, …
  - Type Variables: $\alpha, \beta, \gamma, \delta, \varepsilon$
  - Compound Types: $\alpha \rightarrow \beta$, int * string, bool list, …
- Polymorphic Types:
  - Monomorphic types τ
  - Generic types: universally quantified monomorphic t
    $\forall \alpha_1, \dots, \alpha_n . \tau$
    - The variables $\alpha_1, \dots, \alpha_n$ are distinguished as being generic
    - Can think of τ as same as $\forall . \tau$

Support for Polymorphic Types

- We extend typing Environment Γ to supply polymorphic types for variables
- Free variables of monomorphic type just type variables that occur in it
  - Write FreeVars(τ)
  - They are not bound by the quantifier
- Free variables of polymorphic type remove variables that are universally quantified
  - FreeVars($\forall \alpha_1, \dots, \alpha_n . \tau$) = FreeVars($\tau$) – {$\alpha_1, \dots, \alpha_n$ }

- FreeVars(Γ) = all FreeVars of types in range of Γ

From 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) = $\forall \alpha_1, \dots, \alpha_n . \tau$ where
  $\{\alpha_1, \dots, \alpha_n\}$ = 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 Variables (Identifiers)

Variable axiom:

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

- Where φ replaces all occurrences of $\alpha_1, \dots, \alpha_n$ by monotypes $\tau_1, \dots, \tau_n$
- Examples:

$$\frac{}{\{ x : \forall \alpha . \alpha \} \text{ |- } x : \text{int}}$$

$$\frac{}{\{ f : \forall \alpha, \beta . \alpha \text{ -> } \beta \} \text{ |- } f : \text{int -> float}}$$

Polymorphic Variables (Identifiers)

Variable axiom:

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

- Where φ replaces all occurrences of $\alpha_1, \dots, \alpha_n$ by monotypes $\tau_1, \dots, \tau_n$
- Examples:

$$\frac{}{\{ g : \forall \alpha, \beta . \alpha \text{ -> } \beta \} \text{ |- } g : \alpha \text{ -> int}}$$

~~{ h : ∀ α . int -> α } |- h : float -> int~~

Polymorphic Variables (Identifiers)

Variable axiom:

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

- Where φ replaces all occurrences of $\alpha_1, \dots, \alpha_n$ by monotypes $\tau_1, \dots, \tau_n$
- 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 built-in **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 (1)

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

- Show:

?

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

Polymorphic Example: Fun Rule

- Show: (2)

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

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

Polymorphic Example (2)

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

?

Γ|- if is_empty lst then 0
else 1 + length (tl lst) **: int**

Polymorphic Example (3):IfThenElse

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

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

Γ|- if is_empty lst then 0
else 1 + length (tl lst) **: int**

Polymorphic Example (4)

- Let Γ ={length:α list -> int, lst: α list }
- Recall: is_empty : ∀ α . α list -> bool
- Show

?

Γ|- is_empty lst **: bool**

Polymorphic Example (4):Application

- Let Γ ={length:α list -> int, lst: α list }
- Recall: is_empty : ∀ α . α list -> bool

? ?

Γ|- is_empty : **α list -> bool** Γ|- lst **: α list**

Γ|- is_empty lst **: bool**

Polymorphic Example (4)

- Let Γ ={length:α list -> int, lst: α list }
- Recall: is_empty : ∀ α . α list -> bool

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

Γ|- is_empty : **α list -> bool** Γ|- lst : **α list**

Γ|- is_empty lst : **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(lst) = α list

Γ|- is_empty : **α list -> bool** Γ|- lst **: α list**

Γ|- is_empty lst : **bool**

- This finishes (4)

Polymorphic Example (3):IfThenElse (Repeat)

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

(4) ✓ (5) (6)

Γ|- is_empty lst **: bool** Γ|- 0:int Γ|- 1 + length (tl lst) **: int**

Γ|- if is_empty lst then 0 else 1 + length (tl lst) **: int**

Polymorphic Example (5):Const

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

By Const Rule

Γ|- 0:**int**

Polymorphic Example (6):Arith Op

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

By Variable (7)

By Const

Γ|- 1 : **int** Γ|- length **: α list -> int** Γ|- (tl lst) **: α list**

Γ |- length (tl lst) : **int**

Γ|- 1 + length (tl lst) : **int**

Polymorphic Example (7):App Rule

- Let Γ ={length:α list -> int, lst: α list }
- Recall const tl: ∀ α . α list -> α list

By Const By Variable

Γ|- (tl lst) : **α list -> α list** Γ|- lst : **α list**

Γ|- (tl lst) : **α list**

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

Polymorphic Example (3):IfThenElse (Repeat)

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

(4) ✓ Γ|- is_empty lst **: bool**

(5) ✓ Γ|- 0:int

(6) ✓ Γ|- l + length (tl lst) **: int**

Γ|- if is_empty lst then 0
else l + length (tl lst) **: int**

Proved by deriving the proof tree in the previous slides

Polymorphic Example: Fun Rule (Repeat-Done)

- Show: (3) ✓

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

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

Proved by deriving the proof tree in the previous slides

(Monomorphic) Let and Let Rec [Reminder]

- let rule:

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

- let rec rule:

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

Polymorphic Let and Let Rec

$\text{Gen}(\tau, \Gamma) = \forall\, \alpha_1, \ldots, \alpha_n .\, \tau$ where $\{\alpha_1, \ldots, \alpha_n\} = \text{freeVars}(\tau) - \text{freeVars}(\Gamma)$

- let rule:

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

- let rec rule:

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

Polymorphic Example

- let rec rule:

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

- Show:

?

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

Polymorphic Example:

- let rec rule:

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

Our previous function example

- Show: (1) ✓
{length:α list -> int}
|- fun lst -> …
: **α list -> int**

(2)
{length: $\forall$ α. α list -> int}
|- length ((::) 2 []) +
length((::) true []) **: int**

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

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:$\forall\alpha$. α list -> int}
- Show:

By Var since bool list -> int is instance of
$\forall\alpha$. α list -> int

(12)

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

Γ' |- length ((::) true []) :**int**

Polymorphic Example: (12)AppRule

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

(13)

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

Γ' |- ((::) true []) :**bool list**

Polymorphic Example: (13)AppRule

- Let Γ' = {length:$\forall\alpha$. α list -> int}
- Show:
By Const since bool list is instance of $\forall\alpha$. α list

By Const

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

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

Polymorphic Example: Let Rec Rule – Done!

- Show: (1) ✓ (2) ✓

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

{length: $\forall$ α. α list -> int} |- length ((::) 2 []) + length((::) true []) : **int**

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

Two Problems

- **Type checking**
  - Question: Does exp. e have type τ in env Γ?
  - Answer: Yes / No
  - Method: Type derivation
- **Typability**
  - Question Does exp. e have some type in env. Γ? If so, what is it?
  - Answer: Type τ / error
  - Method: Type inference

Type Inference - Outline

- Begin by assigning a **type variable** as the type of the **whole expression**
- **Decompose the expression** into component expressions
- **Use typing rules to generate constraints** on components and whole
- **Recursively find substitution** that solves typing judgment of first subcomponent
- **Apply substitution to next subcomponent** and find substitution solving it; compose with first, etc.
- **Apply composition of all substitutions** to original type variable to get answer

Type Inference - Example

- What type can we give to

(fun x -> fun f -> f (f x))

- Start with a type variable and then look at the way the term is constructed

Type Inference - Example

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

- First approximate: **Give type to full expr**

$\{\,\}$|- (fun x -> fun f -> f (f x)) : α

- Second approximate: **use fun rule**

$$\frac{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}{\{\,\} \mid\!- (\text{fun x -> fun f -> f (f x)}) : \alpha}$$

- Remember constraint $\alpha \equiv (\beta \to \gamma)$

Type Inference - Example

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

- Third approximate: **use fun rule**

$$\frac{\dfrac{\{f : \delta\,;\, x : \beta\} \mid\!- \text{f (f x)} : \varepsilon}{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}}{\{\,\} \mid\!- (\text{fun x -> fun f -> f(f x)}) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

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

- Fourth approximate: **use app rule**

$$\frac{\dfrac{\dfrac{\{f{:}\delta;\ x{:}\beta\} \mid\!- \text{f} : \varphi \to \varepsilon \quad \{f{:}\delta;\ x{:}\beta\} \mid\!- \text{f x} : \varphi}{\{f : \delta\,;\, x : \beta\} \mid\!- (\text{f (f x)}) : \varepsilon}}{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}}{\{\,\} \mid\!- (\text{fun x -> fun f -> f (f x)}) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

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

- Fifth approximate: **use var rule**, get constraint $\delta \equiv \varphi \to \varepsilon$, Solve with same
- Apply to next sub-proof

$$\frac{\dfrac{\dfrac{\{f{:}\delta;\ x{:}\beta\} \mid\!- \text{f} : \varphi \to \varepsilon \quad \{f{:}\delta;\ x{:}\beta\} \mid\!- \text{f x} : \varphi}{\{f : \delta\,;\, x : \beta\} \mid\!- (\text{f (f x)}) : \varepsilon}}{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}}{\{\,\} \mid\!- (\text{fun x -> fun f -> f (f x)}) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

- Current subst: $\{\delta \equiv \varphi \to \varepsilon\}$

$$\frac{\dfrac{\dfrac{\ldots \quad \{f{:}\varphi \to \varepsilon;\ x{:}\beta\} \mid\!- \text{f x} : \varphi}{\{f : \delta\,;\, x : \beta\} \mid\!- (\text{f (f x)}) : \varepsilon}}{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}}{\{\,\} \mid\!- (\text{fun x -> fun f -> f (f x)}) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

- Current subst: {δ≡φ → ε} Use **App Rule**

{f:φ→ε; x:β}|- f:ζ→φ   {f:φ→ε; x:β}|- x:ζ

… {f:φ → ε; x:β}|- f x : φ

{f : δ ; x : β} |- (f (f x)) : ε

{x : β} |- (fun f -> f (f x)) : γ

{ } |- (fun x -> fun f -> f (f x)) : α

- α ≡ (β → γ); γ ≡ (δ → ε)

Type Inference - Example

- Current subst: {δ≡φ → ε}
- **Var rule**: Solve ζ→φ ≡φ→ε **Unification**

{f:φ→ε; x:β}|- f:ζ→φ   {f:φ→ε; x:β}|- x:ζ

… {f:φ → ε; x:β}|- f x : φ

{f : δ ; x : β} |- (f (f x)) : ε

{x : β} |- (fun f -> f (f x)) : γ

{ } |- (fun x -> fun f -> f (f x)) : α

- α ≡ (β → γ); γ ≡ (δ → ε)

Type Inference - Example

- Current subst: {ζ≡ε, φ≡ε} o {δ≡φ → ε}
- **Var rule**: Solve ζ→φ ≡φ→ε **Unification**

{f:φ→ε; x:β}|- f:ζ→φ   {f:φ→ε; x:β}|- x:ζ

… {f:φ → ε; x:β}|- f x : φ

{f : δ ; x : β} |- (f (f x)) : ε

{x : β} |- (fun f -> f (f x)) : γ

{ } |- (fun x -> fun f -> f (f x)) : α

- α ≡ (β → γ); γ ≡ (δ → ε)

Type Inference - Example

- Current subst: {ζ≡ε, φ≡ε, δ≡ε→ε}
- **Apply to next sub-proof**

(done)…   {f:φ→ε; x:β}|- x:ζ

… {f:φ → ε; x:β}|- f x : φ

{f : δ ; x : β} |- (f (f x)) : ε

{x : β} |- (fun f -> f (f x)) : γ

{ } |- (fun x -> fun f -> f (f x)) : α

- α ≡ (β → γ); γ ≡ (δ → ε)

Type Inference - Example

- Current subst: {ζ≡ε, φ≡ε, δ≡ε→ε}
- **Apply to next sub-proof**

(done)…   {f:ε→ε; x:β}|- x:ε

… {f:φ → ε; x:β}|- f x : φ

{f : δ ; x : β} |- (f (f x)) : ε

{x : β} |- (fun f -> f (f x)) : γ

{ } |- (fun x -> fun f -> f (f x)) : α

- α ≡ (β → γ); γ ≡ (δ → ε)

Type Inference - Example

- Current subst: {ζ≡ε, φ≡ε, δ≡ε→ε}
- **Var rule**: ε≡β

…   {f:ε→ε; x:β}|- x:ε

… {f:φ → ε; x:β}|- f x : φ

{f : δ ; x : β} |- (f (f x)) : ε

{x : β} |- (fun f -> f (f x)) : γ

{ } |- (fun x -> fun f -> f (f x)) : α

- α ≡ (β → γ); γ ≡ (δ → ε)

Type Inference - Example

- Current subst: $\{\varepsilon\equiv\beta\}\circ\{\zeta\equiv\varepsilon,\ \varphi\equiv\varepsilon,\ \delta\equiv\varepsilon\to\varepsilon\}$
- Solves subproof; return one layer

$$\dfrac{\dfrac{\dfrac{\dots \qquad \{f{:}\varepsilon\to\varepsilon;\ x{:}\beta\}\ |\text{-}\ x{:}\varepsilon}{\dots \qquad \{f{:}\varphi\to\varepsilon;\ x{:}\beta\}\ |\text{-}\ f\ x : \varphi}}{\{f : \delta\ ;\ x : \beta\}\ |\text{-}\ (f\ (f\ x)) : \varepsilon}}{\dfrac{\{x : \beta\}\ |\text{-}\ (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}{\{\ \}\ |\text{-}\ (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}}$$

- $\alpha\equiv(\beta\to\gamma);\ \gamma\equiv(\delta\to\varepsilon)$

Type Inference - Example

- Current subst: $\{\varepsilon\equiv\beta,\ \zeta\equiv\beta,\ \varphi\equiv\beta,\ \delta\equiv\beta\to\beta\}$
- Solves this subproof; return one layer

$$\dfrac{\dfrac{\dfrac{\dots}{\dots \qquad \{f{:}\varphi\to\varepsilon;\ x{:}\beta\}\ |\text{-}\ f\ x : \varphi}}{\{f : \delta\ ;\ x : \beta\}\ |\text{-}\ (f\ (f\ x)) : \varepsilon}}{\dfrac{\{x : \beta\}\ |\text{-}\ (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}{\{\ \}\ |\text{-}\ (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}}$$

- $\alpha\equiv(\beta\to\gamma);\ \gamma\equiv(\delta\to\varepsilon)$

Type Inference - Example

- Current subst: $\{\varepsilon\equiv\beta,\ \zeta\equiv\beta,\ \varphi\equiv\beta,\ \delta\equiv\beta\to\beta\}$
- Need to satisfy constraint $\gamma\equiv(\delta\to\varepsilon)$, given subst, becomes: $\gamma\equiv((\beta\to\beta)\to\beta)$

$$\dfrac{\dfrac{\dots}{\{f : \delta\ ;\ x : \beta\}\ |\text{-}\ (f\ (f\ x)) : \varepsilon}}{\dfrac{\{x : \beta\}\ |\text{-}\ (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}{\{\ \}\ |\text{-}\ (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}}$$

- $\alpha\equiv(\beta\to\gamma);\ \gamma\equiv(\delta\to\varepsilon)$

Type Inference - Example

- Current subst:

$\{\gamma\equiv((\beta\to\beta)\to\beta),\varepsilon\equiv\beta,\ \zeta\equiv\beta,\ \varphi\equiv\beta,\ \delta\equiv\beta\to\beta\}$

- Solves subproof; return one layer

$$\dfrac{\dfrac{\dots}{\{f : \delta\ ;\ x : \beta\}\ |\text{-}\ (f\ (f\ x)) : \varepsilon}}{\dfrac{\{x : \beta\}\ |\text{-}\ (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}{\{\ \}\ |\text{-}\ (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}}$$

- $\alpha\equiv(\beta\to\gamma);\ \gamma\equiv(\delta\to\varepsilon)$

Type Inference - Example

- Current subst:

$\{\gamma\equiv((\beta\to\beta)\to\beta),\varepsilon\equiv\beta,\ \zeta\equiv\beta,\ \varphi\equiv\beta,\ \delta\equiv\beta\to\beta\}$

- Need to satisfy constraint $\alpha\equiv(\beta\to\gamma)$ given subst: $\alpha\equiv(\beta\to((\beta\to\beta)\to\beta))$

$$\dfrac{\dfrac{\dots}{\{x : \beta\}\ |\text{-}\ (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \}\ |\text{-}\ (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha\equiv(\beta\to\gamma);$

Type Inference - Example

- Current subst:

$\{\alpha\equiv(\beta\to((\beta\to\beta)\to\beta)),$
$\gamma\equiv((\beta\to\beta)\to\beta),\varepsilon\equiv\beta,\ \zeta\equiv\beta,\ \varphi\equiv\beta,\ \delta\equiv\beta\to\beta\}$

- Solves subproof; return on layer

$$\dfrac{\dfrac{}{\{x : \beta\}\ |\text{-}\ (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \}\ |\text{-}\ (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

Type Inference - Example

- Current subst:

$\{\alpha \equiv (\beta \rightarrow ((\beta\rightarrow\beta) \rightarrow \beta)),$
$\gamma \equiv ((\beta\rightarrow\beta) \rightarrow \beta), \varepsilon\equiv\beta, \zeta\equiv\beta, \varphi\equiv\beta, \delta\equiv\beta\rightarrow\beta\}$

- Done: $\alpha \equiv$ **$(\beta \rightarrow ((\beta\rightarrow\beta) \rightarrow \beta))$**

{ } |- (fun x -> fun f -> f (f x)) : α