Types-2

- Polymorphism
- Type reconstruction (type inference) for a simple PL
- Typing functions
  - Coercion, conversion, reconstruction
- Rich area of programming language research as people try to provide safety assertions about code as part of type systems

Polymorphism

- Motivation: to allow flexibility in implementation with automatic adaptation to correctly typed operations for parameter types used
  - Ease of design (gets rid of special cases)
  - Ease of maintenance (1 copy of code)
  - A cool idea

Polymorphism - Realization

- Ad hoc polymorphism
  - Use coercion to make types work
  - Overloading – same name used for different functions – compiler chooses by context information
- Parametric polymorphism – generics
  - Code is same for range of types, parameterized by the type and instantiated to particular types when code is generated
- True polymorphism –
  - Only one copy of the code! (e.g., ML, Ocaml)
- Question: Do types have to be declared or can they be deduced in a PL allowing polymorphism?

How type reconstruction (type inference) works?

|– <expression> : <type>

1. can always type a constant |– 5.8 : ft/sec

2. can build rules for combining types in expressions

e.g., Distance = Velocity * Time Conversions

$$\frac{|\text{–e1 : ft/sec}, |\text{– e2: sec}}{|\text{– e1*e2 : ft}} \qquad \frac{|\text{– e1:ft/sec}, |\text{– e2: sec/min}}{|\text{– e1*e2 : ft/min}}$$

Velocity = Distance / Time

$$\frac{|\text{– e1: ft}, |\text{– e2: sec}}{|\text{– e1/e2: ft/sec}}$$

Type Reconstruction -1

- See handout for small expression language definition

  Types: τ → Int | Char | Bool … *primitive PL types*

  τ → Pointer(τ) | Tuple (τ,τ) | List(τ) | …*constructed PL types*
  Record(label τ, label τ, ...)

  Expressions syntax: e → <intLiteral> | <listLiteral>|…
  e → varId | (e)
  e → e mod e | e + e | e and e | e or e | not e …
  Boolean/numerical operations
  e → e eq e *comparison operator*

Type Reconstruction -2

e → deref e *pointer operation*
e → fst e | snd e | pair(e,e) *tuple operations* ***tuple constructor***
e → hd e | tail e | cons (e,e) *list operations* ***list constructor***
where <intLiteral> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<listLiteral> → *nil,* etc.

- To perform type reconstruction, we need assumptions for types of constants and then define type deduction rules to type other constructs

Type Reconstruction - 3

- Type rules define the types of results of legal operations

  Constants: c :τ |- c : τ *given in type environment*

  Variables: y: τ |- y: τ *e.g., in declarations*

  Arithmetic: $\frac{\text{|- e1: Int, |- e2: Int}}{\text{|- (e1 mod e2) : Int}}$ *means mod op only applicable to integers*

  Equality: $\frac{\text{|- e1 : τ , |- e2 : τ}}{\text{|- (e1 eq e2) : Bool}}$ *can only compare exprs of same type result is Boolean*

Type Reconstruction - 4

Deref: $\frac{\text{|- e: Pointer(τ)}}{\text{|- deref(e) : τ}}$ *can only apply deref operator to pointer type*

- Examples of use of rules

  fst(1, 2.0) + snd(3.5, 5)

  τ1 = Tuple (Int, Real), τ2 = Tuple (Real, Int)

  fst(τ1) : Int, *snd(τ2)* : Int, therefore + operation is well-typed

---

  fst(1, 2.0) + hd(cons(5, nil))

  τ1 = Tuple(Int, Real), and we want: *τ2* = List(Int)

  but how to get this?

Type Reconstruction - 5

- Need two more rules to type lists:

  [Cons] $\frac{\vdash e1: \tau \ , \vdash e2: List(\tau)}{\vdash cons(e1, e2): List(\tau)}$ (1)

  $\vdash nil: List[\ \_\ ]$ (2) *read this as List of any type*

  or instead use rules (1) and (3):

  $$\frac{\vdash e:\tau}{\vdash cons(e, nil) : List(\tau)} \quad (3)$$

  means lists are made up of homogeneously typed elements, but not necessarily of primitive type *e.g.*, List (Tuple(Int, Bool)) is legal

Typing Statements

- Problem: what to do about typing statements?

  can use special type called *void* for correctly typed statements

$\frac{\vdash y: \tau \ , \vdash e: \tau}{\vdash y:=e : void}$ *Assignment*

$\frac{\vdash s1: void, s2:void}{\vdash s1; s2 :void}$ *Stmt sequence*

$\frac{\vdash b:bool, \vdash s:void}{\vdash \text{if } b \text{ then } s:void}$ *If stmt*

Typing Functions -1

- Want to write a truly polymorphic function and be able to use it on arguments of different types

```
length L = if L=nil then 0 else 1 + length (tl(L));
```

has type signature:

length: List(_) → Int

– Examples from our small expression language

cons : τ → List[τ] → List [τ]

*pair: σ * τ → Tuple(σ,τ)*

fst: Tuple(σ,τ) → σ

*if_then_else: bool * τ * τ → τ*

Typing Functions -2

- Need for type variables to represent unknown types during reconstruction

∀α. List(α) → int is type of SML length function

Type of *deref: ∀β. Pointer(β) → β*

Note: ∀α does not include type *error,* which is used in type checking – more later on this

- Need new inference rule for function application:

$$\frac{\vdash e1: \sigma \rightarrow \tau, \vdash e2: \sigma}{\vdash e1(e2) : \tau}$$

Typing Functions -3

- Functions are usually typed in their curried form
  incr(k,x) = x + k; plus(k), curried incr
  incr: Tuple(int, int) → int *plus: int → (int → int)*
  In curried form can use previous slide's inference rule
- What is a curried form of a function?
  - "A process of transforming a function that takes multiple arguments into a function that takes one argument and returns another function if any arguments are still needed." from https://wiki.haskell.org/Currying
- Currying, an idea from functional programming in which functions are first-class
  - Functions can be values assigned to a variable, passed as parameters, applied to arguments (of the right type)

Curried Form of a Function

- Continuing with our example of incr
  - Incr(K,X) is function of type int x int → int
  - Incr(5,X) returns a value of X+5 (function application)
  - Plus(K) is of type int → (int → int)
  - Plus(5) returns a function that adds 5 to its argument; we write this as lambda(X) = 5 + X (more later on this too)
- Currying is a notion from functional programming in which functions are first-class
  - Functions can be values assigned to a variable, passed as parameters, applied to arguments (of the right type)

Reconstructing Function Types -1
(ASU'86 ed, 6.6)

- High-level view
  1. Introduce new type variables for the function and its parameters.
  2. Setup equations that must hold for these variables based on statements within the function (infer compatible types from uses).
  3. Solve these equations.
     a. If reach a type error, report it.
     b. If can get values for all type variables, then the equations are *consistent.*

Reconstructing Function Types - 2

  c. Note: type value solution process involves using unification to see if two type variables, currently bound to specific types (represented by trees), can be unified to the same type; implementation uses the union-find algorithm
  4. Add a new variable to the type environment to represent this function
     δ = Analyze(fcn_body, E)
- For an example, we will type the SML length function for lists

Analyze (e, E)

- e is expression, E is type environment
- if e is a type variable τ, return E[τ]
- if e is an identifier *id*, return E[*id*]
  - with all ∀ variables renamed and ∀ dropped
    - e.g., ∀ α, α x List(α) --> List(α) is type of *cons*
    - e.g., ∀ α, bool x α x α --> α is type of *if*
    - e.g., ∀ α, α-->β becomes γ-->β, an arbitrary function
- if e is function application, f(e1,…,ek)
  - let t1 - Analyze (e1, E)…
  - let s - Analyze (f, E)
  - introduce fresh type variable, δ
  - add equation (t1 x t2 x…x tk --> δ) = s and return δ
- if e is a function definition, we need to follow the reasoning in this example….

Trace Algm Example -1

Analyze (lng (n) ≡ if (null n) then 0 else (1 + lng(tl n)), E);

Rule 1. Extend E[n] = γ , E[lng] = {$\gamma \rightarrow \delta$}

Rule 2. Analyze function body.

Analyze (if ((null n), 0, (1+lng(tl n))), E).

t1 = Analyze (e1, E) for e1 = (null n) *fcn application*

t11 = Analyze (n) ≈ E[n] = {γ} *identifier*

s11 = Analyze (null) ≈ E[null]= {list $\alpha \rightarrow$ bool} *identifier*

get new type variable β

$\gamma \rightarrow \beta$ = list $\alpha \rightarrow$ bool (1)

return β as type of function application.

Trace Algm Example -2

Analyze (lng (n) ≡ if (null n) then 0 else (1 + lng(tl n)), E);

t2 = Analyze(0,E) ≈ {int} *constant*

t3 = Analyze (1+lng(tl n)) *another fcn application*

t31=Analyze(1,E) ≈ {int}

t32 = Analyze(lng(tl n), E)

t321 = Analyze((tl n),E) *analyze the arg*

t3211 = Analyze(n,E) ≈ {γ} *identifier*

s3211 = Analyze(tl,E) ≈ {list μ → list μ}

new type variable σ

$\gamma \rightarrow \sigma$ = list μ → list μ (2)

return σ as type of function application

s321 = Analyze(lng,E) ≈ {$\gamma \rightarrow \delta$ } *from fcn signature*

new type variable Γ

$\sigma \rightarrow \Gamma = \gamma \rightarrow \delta$ (3)

return Γ as type of function application

Trace Algm Example -3

s31 = Analyze(+,E) ≈ {int * int → int}

new type variable Δ

int * $\Gamma \rightarrow \Delta$ = int * int → int (4)

return Δ

s1 = Analyze(if,E) = {bool * ψ * $\psi \rightarrow \psi$}

new type variable ρ

β * int * $\Delta \rightarrow \rho$ = bool * ψ * $\psi \rightarrow \psi$ (5)

return ρ

Trace Algm Example -4

Rule 3: solve equations using unification using most general unifier

(1) $\gamma \rightarrow \beta$ = list $\alpha \rightarrow$ bool
(2) $\gamma \rightarrow \sigma$ = list $\mu \rightarrow$ list μ
(3) $\sigma \rightarrow \Gamma = \gamma \rightarrow \delta$
(4) int * $\Gamma \rightarrow \Delta$ = int * int $\rightarrow$ int
(5) β * int * $\Delta \rightarrow \rho$ = bool * ψ * $\psi \rightarrow \psi$
β = bool (from 1.)
$\gamma = \sigma$ = list μ (from 2.,3.)
γ = list α (from 1.) (note: list α and list μ are same type)
$\delta = \Gamma = \Delta$ = int (from 3.,4.)
Finally we obtain:
lng: $\gamma \rightarrow \delta$ = list $\mu \rightarrow$ int

Trace Algm Example -5

- In ASU'86 p375
  - Is trace of our algorithm as lines in a table in the same order as our slides
  - Looks like a bottom up traversal of a type tree, typing the subtrees and then going upwards to type higher subtrees