# Types

- What is a type?
- Type reconstruction (inference) for a simple PL
- Type safe programs
- Strong type systems
- Type checking
  - Static versus dynamic



# Types

- Polymorphism
  - Ad hoc: coercion, overloading
  - Parametric: generics
- 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

# What is a type?

- ***Type*: a set of values and meaningful operations on them**
- **Types provide semantic *sanity checks* on programs**
  - **Analogous to units conversions in physics, convert feet per second to inches per minute**
    - **(feet/second) (seconds/minute) (inches/feet)**
  - **How specify types? How check their usage in actual programs?**



# Type Equivalence

- **Governs which constructed types are considered "equivalent" for operations such as assignment**
- **Two main flavors:**
  - **Structural equivalence**
  - **Name equivalence**

# Equality of Structured Types

- *Structural equivalence*: **types are equivalent as terms**
  - **Same primitive type**
  - **Formed by application of same type constructors to structurally equivalent types**
  - **Shortcoming as shown in Pascal:**

    **type salary: int; var s: salary;**

    **type height: int; var y: height**

    ***cannot outlaw s+y by structural equivalence rules.***
  - **Used by Algol-68, Modula-3, ML and C (except for its structs)**



# Equality of Structured Types

- ***Name equivalence:*** **use name of type to assert equivalence**
  - **In Ada: type height: int**

    **var x: list (int)** ***x,y considered same type***

    **var y: list (int)** ***y,s considered different types!***

    **var s: list (height)**
  - **Shortcoming, in Pascal**

    **type cell = record info: int, next: ^cell end;**

    **type link = ^ cell;**

    **var first, last: link;**

    **begin if first.next = last then…** ***comparison isn't valid by either name or struct. eq***

    **types: ^cell link**

**Used by Java, Ada**

# Equality of Structured Types

- ***Declaration equivalence:*** **variables need to be declared in same declaration statement.**

| | |
|---|---|
| **p: ^cell** | ***p,q not compatible types*** |
| **q: ^cell** | ***s,t are compatible types*** |
| **s,t: ^cell** | |

- **Bizarre rule not longer used (ISO Pascal)**



# How type reconstruction (type inference) works?

**$\vdash$ <expression> : <type>**

**1. can always type a constant $\vdash$ 5.8 : ft/sec**

**2. can build rules for combining types in expressions**

**e.g., Distance = Velocity * Time, Conversions**

$$\frac{\vdash e1 : ft/sec, \quad \vdash e2 : sec}{\vdash e1*e2 : ft} \qquad \frac{\vdash e1 : ft/sec, \quad \vdash e2 : sec/min}{\vdash e1*e2 : ft/min}$$

**Velocity = Distance / Time**

$$\frac{\vdash e1 : ft, \quad \vdash e2 : sec}{\vdash e1/e2 : ft/sec}$$

# Type Reconstruction

- **See handout for small expression language definition**

  ***Types:*** $\tau \rightarrow$ **Int | Char | Bool …** ***primitive PL types***

  $\tau \rightarrow$ **Pointer($\tau$) | Tuple($\tau$,$\tau$) | List($\tau$) | …*****constructed PL***

  **Record(label $\tau$, label $\tau$, ...)** ***types***

  ---

  ***Expressions:*** **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

**e → deref e** ***pointer operation*** ***tuple constructor***

**e → fst e | snd e | pair(e,e)** ***tuple operations***

**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**

# Type Reconstruction

- **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:**

$$\frac{\vdash e1: Int,\ \vdash e2: Int}{\vdash (e1 \bmod e2) : Int}$$

***means mod op only applicable to integers***

**Equality:**

$$\frac{\vdash e1 : \tau\ ,\ \vdash e2 : \tau}{\vdash (e1\ eq\ e2) : Bool}$$

***can only compare exprs of same type result is Boolean***



# Type Reconstruction

**Deref:**

$$\frac{\vdash e: Pointer(\tau)}{\vdash deref(e) : \tau}$$

***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

- **Need more rules to type lists:**

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

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

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

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

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



# Definitions (Sethi, Ch 4.9)

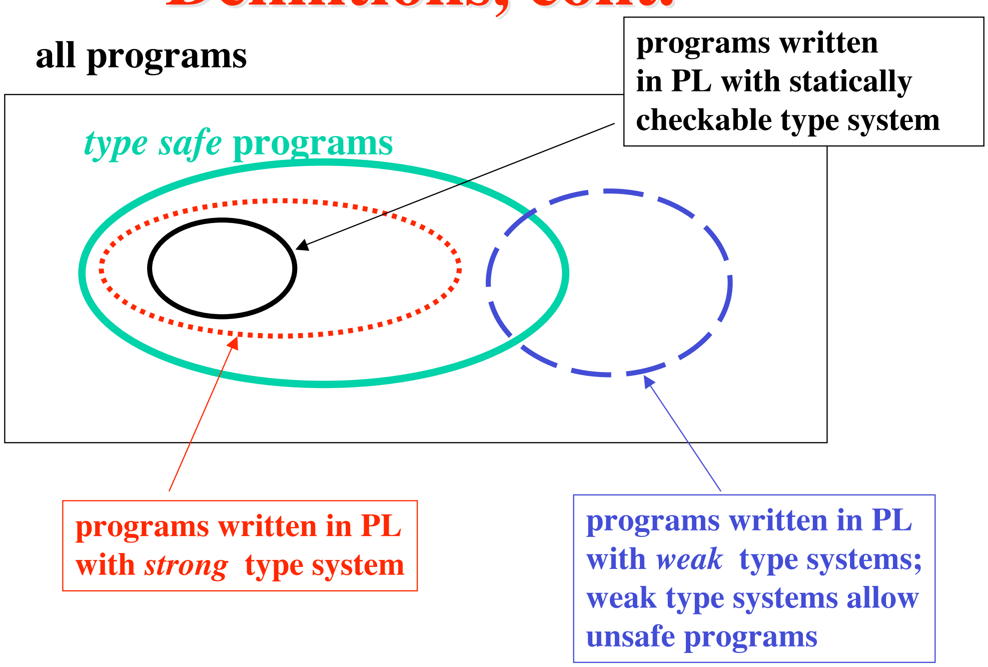
- ***Type safe:*** **program that executes without type errors**
- ***Strong* type system: if it accepts only safe expressions (guaranteed to evaluate without a type error)**
- **PL is *statically typed* if the type of any expression can be fully determined at compile-time. How?**
  - **Explicit declaration, or**
  - **Type reconstruction**

# Definitions, cont.

- PL is *dynamically typed* if during execution type checking occurs
- PL is *strongly typed* (cf Cardelli+Wegner "On Understanding Types, Data Abstraction, Polymorphism", Computer Surveys, 12/85): all expressions are type consistent
  - It is possible to use static and dynamic checking



# Definitions, cont.



# Static Type Checking

- **Points out type errors early**
- **No run-time overhead**
- **Not always possible**
  - **Pascal, Java: array index bounds part of array type; need run-time check for *subscript out of bounds***
- **Highly desirable - key design feature in modern PLs**



# Dynamic Type Checking

- **Incurs run-time overhead plus needs space for type tags**
  - **Operations need to check type tags of their operands before executing**
- **Claim programs are harder to debug**
- **Claim it allows more flexibility in PL design**
  - **Pascal: almost statically typed, except for variant records and array indices**
  - **C: needs dynamic checking for unions; indiscriminate casting thwarts type checking**
  - **Algol68, SML: statically typed (use discriminated unions)**

# Algol68 Example

**from Computing Surveys, June 1976 A. Tanenbaum article on Algol68:**

```
union (int, real, bool) kitchensink;
kitchensink := 3;
kitchensink := 3.14159;
if rndom < .5 then kitchensink := 1
      else kitchensink := 2.76;
fi
case kitchensink in
(int I): print ((“integer”, I));
(real r): print ((“real”, r));
esac
```



# Typing Statements

- **Problem: what to do about typing statements?**

  **use special type called *void***

$$\frac{\vdash y:\tau\ ,\ \vdash e:\tau}{\vdash\ y{:=}e : void} \qquad \frac{\vdash s1: void,\ s2{:}void}{\vdash s1;\ s2 : void} \qquad \frac{\vdash b{:}bool, \vdash s{:}void}{\vdash \text{if } b \text{ then } s{:}void}$$

***Assignment*** ***Stmt sequence*** ***If stmt***

# Typing Functions

- **Want to write a function once 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

- **Need for type variables to represent unknown types during reconstruction**

  $\forall\alpha.\ List(\alpha) \rightarrow int$ **is type of SML length function**

  $deref{:}\ \forall\beta.\ Pointer(\beta) \rightarrow \beta$

  **Note: ∀α does not include type *error*, which is used in type checking**

- **Need new inference rule for function application:**

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

# Typing Functions

- **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**



# Types

**(Cardelli+Wegner Computer Surveys, 12/85)**

- ***Monomorphic*: Conventionally, PL objects have one type**
- ***Polymorphic*: Some PLs allow objects to have more than one type (e.g., *nil* value for lists and pointers)**

# Polymorphism

- *Ad hoc* **(*apparent*) : function appears to work on several different types, but may behave in different ways for different types**
  - *Overloading:* **same name denotes different functions; compiler decides which one by context**
  - *Coercion:* **semantic operation needed to convert an argument to the correct type expected by the function**
    - **Statically or dynamically**
    - **Algol68 only allowed explicit type conversions**



# Polymorphism

- *Parametric:* **function works uniformly on a range of types; (e.g., *cons, length*); often executes the same code no matter what type the arguments are**
  - *Generic functions*: **parameterized template which has to be instantiated to actual parameter values before usage**
    - **Macro-expansion semantics at compile-time**
  - **True parametric polymorphic functions have only 1 copy of code**
    - **ML is the paradigm PL**

# Polymorphism

- **Ada, Pascal are monomorphic, but have**
  - **overloaded arithmetic operators, + * can have mixes of *real* or *int* arguments**
  - **coercion, *int* → *real* allowed**
  - **subtyping, 1..N is subtype of *int***
  - **value sharing, *nil* shared by all pointer types**



# Typing Functions, (ASU 6.6)

- **High-level view**
  1. **Introduce new type variables for the procedure and its parameters.**
  2. **Setup equations that must hold for these variables based on statements within the procedure (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.***

# Typing Functions

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; uses the union-find algorithm

4. Add a new variable to the type environment to represent this function

   $\delta$ = 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 $\tau$, return E[$\tau$]
- if e is an identifier *id*, return E[*id*]
  - with all $\forall$ variables renamed and $\forall$ dropped
    - e.g., $\forall$ $\alpha$, $\alpha$ x List($\alpha$)––>List($\alpha$) is type of *cons*
    - e.g., $\forall$ $\alpha$, bool x $\alpha$ x $\alpha$ -->$\alpha$ is type of *if*
    - e.g., $\forall$ $\alpha$, $\alpha$––>$\beta$ becomes $\gamma$––>$\beta$, 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, $\delta$
  - add equation (t1 x t2 x…x tk --> $\delta$) = s and return $\delta$
- if e is a function definition…..

# Example - Trace Algm

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

**Rule 1. Extend E[n] = $\gamma$ , 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] = {$\gamma$}** ***identifier***

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

**get new type variable $\beta$**

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

**return $\beta$ as type of function application.**



# Example

**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)**

**t3211 = Analyze(n,E) ≈ {$\gamma$}** ***identifier***

**s3211 = Analyze(tl,E) ≈ {list $\mu \rightarrow$ list $\mu$}**

**new type variable $\sigma$**

**$\gamma \rightarrow \sigma$ = list $\mu \rightarrow$ list $\mu$ (2)**

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

**new type variable $\Gamma$**

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

**return $\Gamma$ as type of function application**

# Example

s33 = Analyze(+,E) ≈ {int * int → int}

new type variable Δ

int * Γ → Δ = int * int → int (4)

return Δ

s1 = Analyze(if,E) = {bool * ψ * ψ → ψ}

new type variable ρ

β * int * Δ → ρ = bool * ψ * ψ → ψ (5)

return ρ



# Example

**Rule 3: solve equations using unification using mgu**

(1) γ → β = list α → bool

(2) γ → σ = list μ →list μ

(3) σ → Γ = γ → δ

(4) int * Γ → Δ = int * int → int

(5) β * int * Δ → ρ = bool * ψ * ψ → ψ

β = bool (from 1.)

γ = σ = list μ (from 2.,3.)

γ = list α (from 1.) (note: list α and list μ are same type)

δ = Γ = Δ = int (from 3.,4.)

lng: γ → δ = list μ → int