Types-1

- What is a type?
- “Type safe” programs
- Strong type systems
- Type checking
  - Static versus dynamic
- Polymorphism
  - Ad hoc: coercion, overloading
  - Parametric: generics

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?
Types

• Implicit
  - If variables are typed by usage
    • Prolog, Scheme, Lisp, Smalltalk

• Explicit
  - If declarations bind types to variables at compile time
    • Pascal, Algol68, C, C++, Java

• Mixture
  - Implicit by default but allows explicit declarations
    • Haskell, ML

Type System

• Rules for constructing types
• Rules for determining/inferring the type of expressions
• Rules for type compatibility:
  - In what contexts can values of a type be used (e.g., in assignment, as arguments of functions,...)
• Rules for type equivalence or type conversion
  • Determining (ensuring) that an expression can be used in some context
Types of Expressions

• If \( f \) has type \( S \rightarrow T \) and \( x \) has type \( S \), then \( f(x) \) has type \( T \)
  - type of \( 3 \text{ div } 2 \) is int
  - type of \( \text{round}(3.5) \) is int

• Type error - using wrongly typed operands in an operation
  – \( \text{round}("Nancy") \)
  – \( 3.5 \text{ div } 2 \)
  – “abc” + 3

Type Checking

• Goal: to find out as early as possible, if each procedure and operator is supplied with the correct type of arguments
  - Type error: when a type is used improperly in a context
  - Type checking performed to prevent type errors

• Modern PLs often designed to do type checking (as much as possible) during compilation
When type checking occurs?

- **Compile-time (static)**
  - At compile time, uses declaration information or can infer types from variable uses

- **Run-time (dynamic)**
  - During execution, checks type of object before doing operations on it
    - Uses type tags to record types of variables

- **Combined (compile- and run-time) type checking**
  - Most modern PLs

Type Safety

- A *type safe* program executes on all inputs without type errors
  - Goal of type checking is to ensure type safety
  - Type safe does not mean without errors

```plaintext
read n;
if n>0 then {y:="ab";
  if n<0 then x := y-5;}
```

- Note that assignment to `x` is never executed so program is *type safe* (yet contains an error).
Strong Typing

• Strongly typed PL
  - PL requires all programs to be type checkable
  - PL’s type system only accepts only safe expressions (guaranteed to evaluate without a type error)

• Statically strongly typed PL - compiler allows only programs that can be type checked fully at compile time
  - If the type of any expression can be fully determined at compile-time. How?
    • Explicit declaration, or
    • Type reconstruction (sometimes called type inference)

Strong Typing

• Dynamically strongly typed PL - Operations include code to check run-time types of operands, if type cannot be determined at compile time
  • C++, Java
Definitions, cont.

- *Type safe programs* are programs written in PL with a strong type system:
  
  - Strong type systems ensure type safety, preventing certain kinds of errors at compile time.

- *Weak type systems* allow unsafe programs:
  
  - Programs written in PL with weak type systems; weak type systems allow unsafe programs.

Type Equivalence

- Governs which constructed types are considered “equivalent” for operations such as assignment and copy statements.

- Two main flavors:
  
  - Structural equivalence
  
  - Name equivalence
Types formed by construction

- **Constructive point of view**
  - Primitive types e.g., `int`, `char`, `bool`,
    `enum{red, green, yellow}`
  - Composite/constructed types:
    - reference e.g., `pointerTo(int)`
    - array e.g., `arrayOf(char)` or `arrayOf(char, 20)` or ...
    - record/structure e.g., `record(age:int, name:string)`
    - union e.g. `union(int, pointerTo(char))`
    - list e.g., `list(...)`
    - function e.g., `float → int`
  
  CAN BE NESTED! `pointerTo(arrayOf(pointerTo(char)))`

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)  \(\neq\) \(s\) 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
    ```

  Used by Java

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)
Types

• **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)

(good article on typing by Cardelli+Wegner Computer Surveys, 12/85)

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, but it never caught hold so this solution is not popular
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 Statements

- Problem: what to do about typing statements?
  use special type called void

\[
\begin{align*}
\Gamma & : \tau, \Delta : \tau & \Gamma, \Delta & : \text{void, } \text{void} \\
\Gamma & : \text{void} & \Delta, \Delta & : \text{void} \\
\text{Assignment} & & \text{Stmt sequence} & & \text{If stmt}
\end{align*}
\]