Type Systems in Programming Languages

1

Typing in Programming Languages

- **Typing:** a mechanism for grouping values into types
  - Describe data effectively
  - Prevent nonsensical operations, *e.g.* multiply a string by a set
- Components of this mechanism
  - **Associating a type with every value:** values are grouped into types, and values of the same type exhibit similar behavior under similar operations
    - In complex type systems, certain values can be associated with more than one type.
    - Often, if there is more than one type associated with a value, there is an infinite number of types associated with this value.
  - **Checking rules:** Given a value and an operation, use the type of the value to check whether this operation is allowed on values of this type
    - This becomes more difficult if there are several types associated with the same value
  - **Inference rules:** Given the type of operands, determine:
    - the meaning of an operation
    - the type of the result

2

Significance of Type

- **Type determines meaning (semantics):**
  What will be executed as a result of the following expression?

  ```
  a + b
  ```

  - Integer addition, if `a` and `b` are integers, or
  - Floating point addition, if `a` and `b` are of a floating point type, or
  - Conversion to a common type and then addition, if a and b are of different types.
- **Type determines what's allowed (semantically):**
  Is the following expression legal?

  ```
  X[i]
  ```

  - Yes, if `X` is an array type and `i` is of a discrete type.
    - In C, the above is legal also when `X` is a pointer (and `i` is of a discrete type).
  - No, *e.g.* if `X` is a real number and `i` is a function.

Classifying Type Systems

- **Existence**
  - Does the language include a type system at all?
- **Type equivalence and subtyping**
  - When can one type replace another?
  - Structural/name/declaration type equivalence
- **Strength**
  - How strictly are the typing rules enforced?
- **Time of checking**
  - At what stage is type checking performed?
  - Static *vs.* dynamic typing
- **Responsibility**
  - Is the programmer responsible for type declarations or is it the compiler?
  - Explicit/implicit typing
- **Flexibility**
  - To what extent does the type system restrict the user's expressiveness?
  - Polymorphic typing

4

Existence of a Type System

A language can be

- **Typed**: The set of values can be broken into groups, with more or less uniform behavior under the same operation of values in each group.
  - C, Pascal, ML, Ada, Java, and most other programming languages
- **Untyped**: Each value has its own unique set of permissible operations, and their semantics are particular to the value.
  - **Lisp** (list processing) :
    - All values are S-expressions, which are not much more than binary trees.
    - Elementary operations are: choosing the right and left subtrees of a tree, and combinations and inverse of these operations.
    - Legality of operations is determined by tree structure and values
  - **Mathematica** (a language for symbolic mathematics):
    - Values are mathematical expressions (represented as S-expressions)
    - Legality and semantics of a manipulation of an expression is determined by the structure and semantics of the expression.
- **Degenerate:** The set of values is so simple that it admits only one type, or a small number of types; appears in most scripting languages
  - **BCPL (C's ancestor):** the only data type is a machine word
  - **DOS Batch Language:** the only data type is a string
  - **C-shell, Bourne-shell, AWK, Rexx:** two data types - strings and numbers - with little distinction between them; minimal support for arrays.
  - **Perl:** Several extensions to the type system beyond AWK.

Type Equivalence and Subtyping

Type Equivalence

- Suppose that an operation expects an operand of type *T*, but receives an operand of type *T'*. **Is this an error?**
  - No, if *T'* is a ***subtype*** of *T*
- Two types that are subtypes of each other are called ***equivalent***
- **Caveat**: because the notion of subtype is more refined than the notion of type equivalence, we will find languages where type equivalence is defined but subtyping is not
- Kinds of type equivalence
  - Structural equivalence
  - Name equivalence
  - Declaration equivalence

Structural Equivalence

- **Have the "same" values…. In Algol-68**:
  - If *T* and *T'* are both primitive, then $T \cong T'$ only if they are identical.
  - Else if
    - $T = A \times B$, $T' = A' \times B'$, or
    - $T = A + B$, $T' = A' + B'$, or $T = A + B$, $T' = B' + A'$, or
    - $T = A \to B$, $T' = A' \to B'$

    and $A \cong A'$, $B \cong B'$ => $T \cong T'$
  - Otherwise?
- **Recursive types:**
  - T = Unit + c(A x T)
  - S = Unit + c(A x S)
  - R = Unit + c(A x S)
  - Intuitively T, S and even R are structurally equivalent, but structural equivalence is not easy to define and test for in recursive types.
- **Points of disagreement between languages**:
  - Do records require field name identity to be structurally equivalent?
  - Do arrays require index range identity to be structurally equivalent?

8

An example for (hypothetical) structural equivalence

```
EmployeeData = record
   id: Integer;
   name: String;
   isManager: Boolean;
   ranking: array [1..3] of Chararcter;
end;
```

```
YearlyEmployee = record
    d: EmployeeData;
    salary: Real;
end;
```

```
Employee1 = record
    d: EmployeeData;
    case Integer of
        0:  yearly_salary: Real;
        1:  hourly_rate: Integer
    end;
end;
```

```
HourlyEmployee = record
    d: EmployeeData;
    salary: Integer;
end;
```

```
Employee2 = record
 case Integer of
    0: y: YearlyEmployee;
    1: h: HourlyEmployee;
 end;
end;
```

9

Name Equivalence

- $T \cong T'$ iff T and T' were defined in the same place (original Pascal and Ada):

```
TYPE  T1 = File of Integer;
      T2 = File of Integer;
VAR   f1: T1;
      f2: T2;
Procedure p(Var f: T1);
....
p(f1); (* ok *)
p(f2); (* compile-time error *)
```

Name equivalence across programs

```
program p1(f)
type T = file of Integer;
var  f: T;
begin
 ...
 write(f,...);
 ...
end;
```

```
program p2(f)
type T = file of Integer;
var  f: T;
begin
 ...
 read(f,...); (* Type error *)
 ...
end;
```

- By ***definition*** of Pascal, it follows that
  - two Pascal programs cannot communicate legally through files using any user-defined type, hence the type error above.
  - type `Text`, which is the only predefined type for files in Pascal, allows communication between programs.
- In practice, most ***implementations*** of Pascal do not type check files. Thus, these implementations *subvert* Pascal's type safety, but allow reasonable interfacing.

Declaration Equivalence

- A later Pascal standard (ANSI 1983):
  $T \cong T'$ only if T and T' have the same declaration

```
TYPE  T1 = File of Integer;
      T2 = File of Integer;
VAR   f1: T1;
      f2: T2;
Procedure p(Var f: T1);
....
p(f1); (* ok *)
p(f2); (* ok *)
```

12

Subtyping in Pascal

- A weaker notion than type equivalence: an operation that expects an operand of type *T* but receives an operand of type *T'* is legal also if *T* and *T'* are not equivalent, but *T'* is a *subtype* of *T.*
- **Subtyping in Pascal** - only in one case:
  - if *T=[a..b]* and *T'=[c..d]* then *T'* is a subtype of *T* iff *T'* is a subrange of *T.*
- **Limitation**: there is no way to override this definition. For instance:

```
type
        age = 0..120;
        height = 0..250;
```

`age` will be a subtype of `height` even though we don't want to confuse the two notions.

13

Derived Types in Ada

```
type
age = derived integer range 0..120;
height = derived integer range 0..250;
```

- `age` is not equivalent to `0..120`
- `height` is not equivalent to `0..250`
- Therefore: `age` is not a subtype of `height`

- **In Modula-3**: values of derived types are ***branded***, so that storage in external files will not create communication problems between programs.

14

Subtypes in ADA

```
type
   age = derived integer range 0..120;
   height = derived integer range 0..250;
subtype child_height is height range 0..120;
```

- `age` and `child_height` not equivalent to `0..120`
- `height` is not equivalent to `0..250`
- Therefore: `age` is not a subtype of `height`
- But `child_height` is!
- A *convert* function that expects a height in cm and converts it to inches, can be used with a variable of type `child_height` , but not one of type `age` .

15

More about Subtyping in ADA

- Subtyping of ordered primitive types - using subranges:

```
   subtype Probability is Float range
   0.0..1.0;
```

- Subtyping of array types: using subranges as well:

```
subtype String is array (Integer range
<>) of Character;
subtype String5 is String (1..5);
```

- Subtyping of record types:

```
type Sex is (f,m);
type Person (gender: Sex) is record …
end record;
subtype Female is Person (gender => f);
```

16

Branding

- Sometimes it is not desirable to define two instances of the same type as type equivalent.
  - **Example:** a string describing a person's name and another string describing his or her address are not type equivalent.
- To distinguish two values of the same type we can **brand** these values by adding a tag to it.
- Simple technique for branding – labeled fields:

```
User = Record
    name: String;
    address: String;
end
```

  - In this example, we brand the string types: Address_Type and Name_Type
- Branding is similar to `typedef`'s in C. However, `typedef` merely gives an alias to an existing type and does not define a new type.
- In Modula-3, branding is performed automatically for values of derived types, so their type is maintained also when stored by one program and read by another program.

17

Subtyping as Subclassing

Consider the *class* `Collection` that allows the following operations:

```
insert (what: integer)
is_member (what: integer) : boolean
remove (what: integer) : boolean
write ()
```

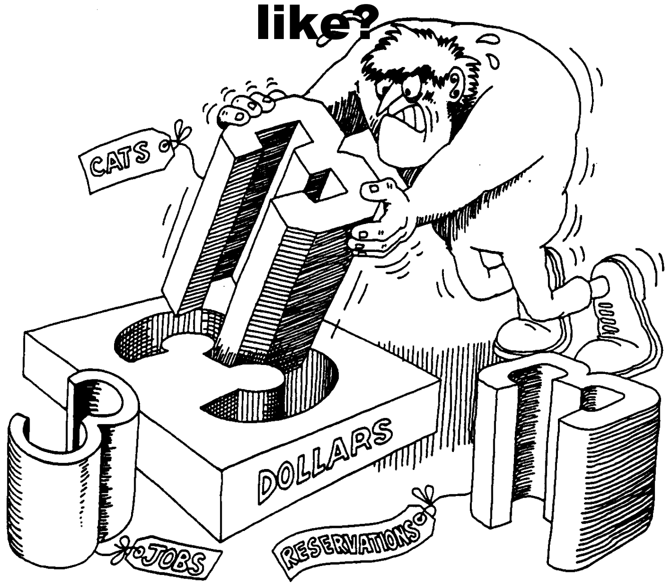
A class `Queue` that allows the same operations plus the following:

```
remove_first () : integer or Unit
```

would be a ***subclass*** of `Collection`.

18

Strong Typing -- What does it look like?



Strong typing prevents mixing abstractions.

Strength of Typing Systems

- **Strongly typed languages**: ML, Eiffel, Modula, Java
  - It is impossible to break the association of a value with a type from within the framework of the language.
  - It is impossible to subject a value to an operation which is not acceptable for its type.
  - **Consequence:** the representation of values can be concealed by the language
- **Weakly typed languages:** values have associated types, but it is possible for the programmer to break or ignore this association.
  - **C** – *we can access the same memory cell in different ways:*
    - *In an array of Booleans, can do boolean operations on the values, or arithmetic ones, etc…*
    - *Parameter passing using pointers*
    - *Access beyond array dimensions (i.e, index value is not of the declared type)*
    - *Union type…misused disjoint union*

Spectrum of Strength

- **S**ome languages are more strongly typed than others:
  - Pascal is more strongly typed than C, with the only ways of breaking the type rules being:
    - *Variant records*
    - *Illegal parameter passing with procedure-valued parameters*
      - *When a procedure B is passed as a parameter to a procedure A, the call in the declaration of A has parameters with types. The types of the parameters of B might not agree with those.*
    - *Through files (in some compilers that just don't check)*

Loopholes in the Type System

Types *usually* hide the fact that a variable is just a box containing bits, however:

Type casting, as in

```
long i, j, *p = &i, *q = &j;
long ij = ((long) p) ^ ((long) q));
```

and *variant records*, as in

```
union {
  float f;
  long l;
} d;
d.f = 3.7;
printf("%ld\n", d.l);
```

allow one to peep into the implementation of types, by subjecting a value to operations not allowed for its type

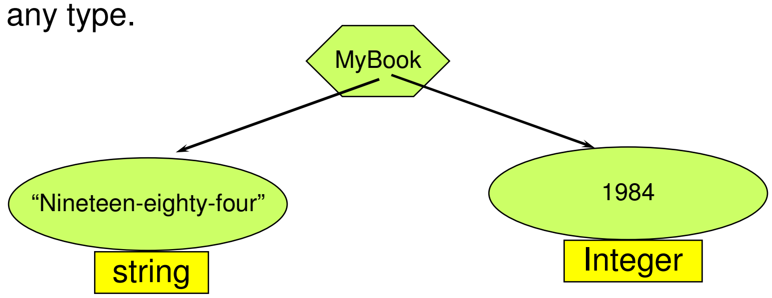
Time of Enforcement

- **Type checking:** Language implementation must ensure that no nonsensical operations occur.
  - **Multiplication:** check that both operands are numeric.
  - **Logical operation:** check that both operands are truth values.
  - **Component selection:** possible only on arrays and records (different implementation in each).
- Type checking must be performed before the operation, but it could be done either at compile-time or at run-time:
  - **Statically typed languages:** type rules are enforced at compile time. Every variable and every formal parameter have an associated type.
    C, Pascal, Eiffel, ML, …
  - **Dynamically typed languages:** type rules are enforced at run-time. *Variables*, and more generally – *expressions*, have no associated type. Only *values* have fixed types. Type checking must be done prior to each operation.
    Smalltalk, Prolog, Snobol, *APL*, Awk, Rexx, ...

23

Dynamic Typing

- Types are associated with values.
  Each value carries a tag, identifying its type.
- **Polymorphic variables:** a variable may contain a value of any type.



- **Advantages**:
  - Arrays don't have to be of a homogeneous type
  - When data isn't uniformly typed, we don't need different variables

Dynamic Typing - Example

```
procedure ReadLiteral(var item);
begin
    read a string of nonblanks;
    if  the string constitutes an integer literal
    then
        item := numeric value of the string
    else
        item := the string itself
end
```

22,10,1996
or
22,OCT,1996

```
read(day); ReadLiteral(month); read(year)
if month is a string then
        case month of {
           "Jan": return 1;
           "Feb": return 2;
           ....
        }
    else
        return month;
end
```

Disadvantages of Dynamic Typing

- Space overhead:
  - **Each value is tagged with type information**
- Slow-down:
  - **Extra data to manipulate**
  - **Run-time checking**
- Security:
  - **Many bugs can be detected by static compile-time checks**

 :=  ;

```
Error: type mismatch
```

Static Typing

- In static typing, each variable and parameter is associated with a type.

Type ⟷ Identifier

  - This usually means that all identifiers should be declared before used. However this is not always the case.
  - A variable may contain only values compatible with its associated type.
- All expressions are guaranteed to be type-consistent:
  - No value will be subject to operations it does not recognize.
  - This allows the compiler to engage in massive optimization.

Why Static Typing?

- Computability theory teaches us that an automatic tool is limited as a programming aid to analyze code in advance
  - Cannot (always) determine if the program stops.
  - Cannot (always) determine if the program is correct.
  - Cannot decide almost any other interesting run time property of a program.
- One thing that *can* be done automatically by code analysis is to make sure that no run time *type error* occurs (for static typing).
- We can use every tiny bit of help in our struggle against the complexity of software!
  - A few other automatic aids are:
    - Garbage collection: automatic memory management (run time)
    - Const correctness: no modification of const parameters (compile time)
    - Pre and post (assert) conditions: partial specification of a function (run time)

Benefits of Static Typing

- Enforce the design decisions
- Prevent run time crashes:
  - Mismatch in # of parameters
  - Mismatch in types of parameters
  - Sending an object an inappropriate message
- Early error detection reduces:
  - Development time
  - Cost
  - Effort
- More efficient and more compact object code
  - Values do not carry along the type tag
  - No need to conduct checks before each operation
  - `type SMALL_COUNTER is range 0 .. 255;`

Limits of Static Typing

- Some senseless operations cannot be statically checked.
- Generic examples:
  - **Division by zero:** no typing system for the integers can prevent this without hitting the halting problem.
    - Checks are realized in run time by using machine-level exceptions. These checks carry no overhead.
  - **Exponentiation of certain pairs of real numbers:** similar problems to division by zero.
    - Realized at run time by the procedure that implements exceptions.
  - **Array references:** it is impossible to detect overflow of array indices.
    - Some hardware support, on some machines, but not without overhead.

Other Times of Enforcement

- **Mixed typing:** There is a variety of possibilities between static and dynamic typing, the two extremes.
  - Mixed typing means that some typing rules are checked at compile time, others are checked during runtime.
  - **Pascal**: Most operations are checked at compile time. Array access is checked during runtime.
    - Type information is still with the variables, but checks must be deferred to run time.
    - Some Pascal compilers have a compilation option that drops array reference checks.
- **"Postmortem typing"**: A different name for weak typing.
  - No checking of type errors is made. If a program makes a type error, then it is allowed to merrily carry along with this error to the bitter end…
  - The programmer is then responsible to remove the type errors from his program

31

Responsibility for Tagging

- **Type inference:** Compilers have the ability to apply type inference rules to determine the type of *expressions*
- Why not apply this also to variables and other entities?
  - **Explicit typing:** programmer is in charge for annotating variables and other entities with type information
    - **This is done by variable type declarations**
  - **Implicit typing:** the compiler infers type information of an entity from the way it is used
    - **No variable type declarations**
  - **Semi-implicit typing:** type is determined from the lexical structure of an identity
    - **No variable declarations**

32

Explicit Typing

- **Example:** variable and parameter declarations in Pascal, Ada, C, etc.
- Type declarations help document programs
  - `X: speed;` (* Good *)
  - `Y: real;` (* Bad *)
  - `Z = 3;` (* Worse *)

Implicit Typing

- The compiler infers the type of an entity from the way it is used
- **Risk**: Inadvertent creation of variables due to typos and spelling errors:
  - **ML's answer:** no variables!
    - **Value declaration**: just like CONST declaration in Pascal
      - ML: `val n = 100`
      - Pascal: `CONST n = 100`
    - **Formal parameters**: declared (without type) in the header of a function.
- **Risk**: Confusing error messages, and type errors
  - **ML's answer:** programmer is allowed to add *type constraints*
- **Risk:** Some complex (recursive and generic) type inference problems are undecidable
  - **ML's answer 1:** careful analysis of the type system to detect when this problem may occur
  - **ML's answer 2:** type constraints

Semi-Implicit Typing

- **Fortran:** All variables which begin with one of the letters `I,J,K,L,M` and `N` are integers; all others are real.
  - **Risk**: inadvertent creation of variables.
  - **Fortran answer:** the declaration `implicit none` excludes automatic creation of variables. The programmer is required to explicitly declare all variables.
- **Basic (older versions):** Suffixes such as %,$ etc. determine the variable's type.
- **Perl**: Essentially the same as Basic.

35

Flexibility of Type System

- Type system should be an aide, not a hurdle:
  - Do not issue type error messages on programs which will not make run time type errors.
  - Allow the use of the same code for many different types.
- We refer to the extent to which a type system has these features as the *flexibility* of the type system.
- Too flexible: doesn't alert programmer to mistaken usage of variables, bad use of actual parameters, etc.
  - Adrienne rocket failed because an actual parameter in inches was used in a procedure with a formal parameter in centimeters….should have been a type error!
- Too inflexible: prevents reuse of code, especially in procedures
- Modern tendency: *polymorphism* – to be studied in great detail later…

36

Inflexibility of Pascal

```
Type T = Integer;
Procedure sort(
   Function comp(a,b:T):Boolean,
   a: array[1..300] of T
); forward;
```

Could *not* be applied to …

- **Arrays of real:** Procedure body and declaration has to be repeated with `T1=Real`
- `array[1..299] of T`: Array is too small.
- `array[1..500] of T`: Array is too big.
- `array[0..299] of T`: Mismatch in indices.
- `array[1..300] of T`: No name equivalence!!!!
  - Pascal is so fussy and inflexible in its type system that even two identical type declarations are considered distinct. A type declaration made at a certain point in a program is equivalent only to itself.

```
Type T = Integer;
     Tarray = array[1..300] of T
Procedure sort(
   Function comp(a,b:T):Boolean,
   a: Tarray;
); forward;
```

37

Responses to Inflexibility

- **C camp:** weak typing
  `int qsort(char *base, int n, int width,int(*cmp)())`
- **Smalltalk/Java camp:** dynamic typing overcomes complex inflexibility problems.
  - In Java dynamic typing is mostly geared towards this problem
- **Ada camp:** polymorphic type systems

```
generic
  type T is private
  with function comp(x: T, y: T)
  procedure sort(a: array(1..max) of T)
  …
procedure int_sort is new sort(int , "<");
```

- The backlash effect:
  - The C camp is leaning toward the Ada camp. ANSI-C, and even more so C++, tend towards strong, yet polymorphic type systems.

```
void qsort(void *base,size_t nel, size_t width,
     int (*compar) (const void *, const void *))
```

  - New additions to Java (e.g., Pizza) introduce generic programming into Java.
  - There were attempts to introduce static typing to Smalltalk!

38

Summary: What's Best?

- **Depends on purpose…**
  - **Light-headed:** a scripting language, designed for small, not-to-be maintained, quick and dirty programs which are supposed to be run only a small number of times with little concern about efficiency:
  - No typing or degenerate typing,
    - Weak typing to remove hassles
    - Dynamic typing to achieve flexibility
    - Semi-implicit typing to reduce programmer time
  - **Software engineering oriented:** programs which are developed by several programmers, maintained and changed, run numerous times, and with efficiency concerns
    - Type system must exist to document and protect the program
    - Strong typing to reduce errors
    - Static typing to enhance efficiency, clarity and robustness
    - No semi-implicit typing to prevent inadvertent creation of variables
    - Flexible type system to allow the programmer to concentrate on the important stuff.

39