Programming Languages

Types

G22.2110
Summer 2008
What is a type?

- A type consists of a set of values
- The compiler/interpreter defines a mapping of these values onto the underlying hardware.
Static vs Dynamic Type Systems

Static vs dynamic

- **Static**
  - Variables have types
  - Compiler ensures that type rules are obeyed at compile time

- **Dynamic**
  - Variables do not have types, values do
  - Compiler ensures that type rules are obeyed at run time

A language may have a mixture; Java has a mostly static type system with some runtime checks.

**Pros and cons**

- Faster: static
dynamic typing requires run-time checks
- More flexible: dynamic
- Easier to refactor code: static
Strong vs weak typing

- A strongly typed language does not allow variables to be used in a way inconsistent with their types (no loopholes)
- A weakly typed language allows many ways to bypass the type system (e.g., pointer arithmetic)

C is a poster child for the latter. Its motto is: “Trust the programmer”.
Scalar Types Overview

- discrete types
  must have clear successor, predecessor

  - integer types
    often several sizes (e.g., 16 bit, 32 bit, 64 bit)
    sometimes have signed and unsigned variants (e.g., C/C++, Ada, C#)
    SML/NJ has a 31-bit integer

  - enumeration types

- floating-point types
  typically 64 bit (double in C); sometimes 32 bit as well (float in C)

- rational types
  used to represent exact fractions (Scheme, Lisp)

- complex
  Fortran, Scheme, Lisp, C99, C++ (in STL)
**Other intrinsic types**

- **boolean**
  Common type; C had no boolean until C99

- **character, string**
  - some languages have no character data type (e.g., Javascript)
  - internationalization support
    - Java: UTF-16
    - C++: 8 or 16 bit characters; semantics implementation dependent
  - string mutability
    most languages allow it, Java does not.

- **void, unit**
  Used as return type of procedures;
  **void**: (C, Java) represents the absence of a type
  **unit**: (ML, Haskell) a type with one value: ( )
Enumeration types: abstraction at its best

- trivial and compact implementation: literals are mapped to successive integers
- very common abstraction: list of names, properties
- expressive of real-world domain, hides machine representation

Examples:

```csharp
    type Suit is (Hearts, Diamonds, Spades, Clubs);
    type Direction is (East, West, North, South);
```

Order of list means that `Spades > Hearts`, etc.

Contrast this with C#:

“arithmetics on enum numbers may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error”
type Fruit is (Apple, Orange, Grape, Apricot);
type Vendor is (Apple, IBM, HP, Dell);

My_PC : Vendor;
Dessert : Fruit;
...
My_PC := Apple;
Dessert := Apple;
Dessert := My_PC;   -- error

Apple is overloaded. It can be of type Fruit or Vendor.
Ada and Pascal allow types to be defined which are subranges of existing discrete types.

```ada
type Sub is new Positive range 2 .. 5;  -- Ada
V: Sub;
```

```pascal
type sub = 2 .. 5;  (* Pascal *)
var v: sub;
```

Assignments to these variables are checked at runtime:

```plaintext
V := I + J;  -- runtime error if not in range
```
Composite Types

- arrays
- records
- variant records, unions
- classes
- pointers, references
- function types
- lists
- sets
- maps
Arrays

- **index types**
  - most languages restrict to an integral type
  - Ada, Pascal, Haskell allow any scalar type

- **index bounds**
  - many languages restrict lower bound:
    - C, Java: 0, Fortran: 1, Ada, Pascal: no restriction

- **when is length determined**
  - Fortran: compile time; most other languages: can choose

- **dimensions**
  - some languages have true multi-dimensional arrays (Fortran)
  - most simulate multi-dimensional arrays quite nicely as arrays of arrays.

- **literals**
  - C/C++ has initializers, but not full-fledged literals
  - Ada: `(23, 76, 14)` Scheme: `#(23, 76, 14)`

- **first-classness**
  - C, C++ does not allow arrays to be returned from functions
Does the language support these?

- **array aggregates**
  
  \[
  A := (1, 2, 3, 10); \quad \text{-- positional}
  \]
  
  \[
  A := (1, \text{others} => 0); \quad \text{-- for default}
  \]
  
  \[
  A := (1..3 => 1, 4 => -999); \quad \text{-- named}
  \]

- **record aggregates**
  
  \[
  R := (\text{name} => "NYU", \text{zipcode} => 10012);
  \]
Similar notion for declarations:

```cpp
int v2[] = { 1, 2, 3, 4 };  // size from initializer
char v3[2] = { 'a', 'z' };  // declared size
int v5[10] = { -1 };  // default: other components = 0
struct School r =
    { "NYU", 10012 };  // record initializer
char name[] = "Algol";  // string literals are aggregates
```

C has no array assignments, so initializer is not an expression (mechanism is less orthogonal)
Related (but distinct) notions:

- a value that denotes a memory location
- a dynamic name that can designate different objects
- a mechanism to separate stack and heap allocation

```plaintext
type Ptr is access Integer; -- Ada: named type
typedef int *ptr;   // C, C++
```
Questions:

- Is it possible to get the address of a variable? Convenient, but aliasing causes optimization difficulties. (the same way that pass by reference does)
  Unsafe if we can get the address of a stack allocated variable

- Is pointer arithmetic allowed?
  unsafe if unrestricted

In C, no bounds checking:

```c
// allocate space for 10 ints
int *p = malloc(10 * sizeof(int));
p += 42;
... *p ...
// out of bounds, but no check
```
type Cell; -- an incomplete type
type Ptr is access Cell; -- an access to it
type Cell is record -- the full declaration
  Value: Integer;
  Next, Prev: Ptr;
end record;
List: Ptr := new Cell'(10, null, null);
... -- A list is just a pointer to its first element
List.Next := new Cell'(15, null, null);
List.Next.Prev := List;
struct cell {
    int value;
    cell *prev;
    cell *next;  // legal to mention name
};  // before end of declaration

struct list;  // incomplete declaration

struct link {
    link *succ;
    list *memberOf;
};  // a pointer to it

struct list {  // full definition
    link *head;  // mutual references
};
Pointers and dereferencing

- Need notation to distinguish pointer from designated object
  - in Ada: `Ptr` vs `Ptr.all`
  - in C: `ptr` vs `*ptr`
  - in Java: no notion of pointer

- For pointers to composite values, dereference can be implicit:
  - in Ada: `C1.Value` equivalent to `C1.all.Value`
  - in C/C++: `c1.value` and `c1->value` are different
A pointer used for low-level memory manipulation, i.e., a memory address. In C, `void` is requisitioned to indicate this. Any pointer type can be converted to a `void *`.

```c
int a[10];
void *p = &a[5];
```

A cast is required to convert back:

```c
int *pi = (int *)p;
double *pd = (double *)p; // no check
```
In C/C++, the notions:

- an array
- a pointer to the first element of an array

are almost the same.

```c
void f (int *p) { ... }
int a[10];
f(a);  // same as f(&a[0])

int *p = new int[4];
... p[0] ...  // first element
... *p ...  // ditto
... 0[p] ...  // ditto

... p[10] ... // past the end; undetected error
```
Pointers and safety

Pointers create aliases: accessing the value through one name affects retrieval through the other:

```c
int *p1, *p2;
...
p1 = new int[10];    // allocate
p2 = p1;            // share
delete[] p1;        // discard storage
p2[5] = ...         // error:
                   // p2 does not denote anything
```
Several possible problems with low-level pointer manipulation:

- dangling references
- garbage (forgetting to free memory)
- freeing dynamically allocated memory twice
- freeing memory that was not dynamically allocated
- reading/writing outside object pointed to
If we can point to local storage, we can create a reference to an undefined value:

```c
int *f () {
    // returns a pointer to an integer
    int local; // variable on stack frame of f
    ...
    return &local; // reference to local entity
}
```

```c
int *x = f ();
...
*x = 5; // stack may have been overwritten
```
A record consists of a set of typed fields. Choices:

- Name or structural equivalence?
  Most statically typed languages choose name equivalence
  ML, Haskell are exceptions
- Does order of fields matter?
  Typically, same answer as previous question.
- Any subtyping relationship with other record types?
  Most statically typed languages say no.
  Dynamically typed languages implicitly say yes.
  This is known as *duck typing*.
A variant record is a record that provides multiple alternative sets of fields, only one of which is valid at any given time.
Need to treat group of related representations as a single type:

```ada
type Figure_Kind is (Circle, Square, Line); type Figure (Kind: Figure_Kind) is record
  Color: Color_Type;
  Visible: Boolean;
  case Kind is
    when Line   => Length: Integer;
      Orientation: Float;
      Start: Point;
    when Square => Lower_Left, Upper_Right: Point;
    when Circle => Radius: Integer;
      Center: Point;
  end case;
end record;
```
C1: Figure(Circle); -- discriminant provides constraint
S1: Figure(Square);
...
C1.Radius := 15;
if S1.Lower_Left = C1.Center then ...

function Area (F: Figure) return Float is
  -- applies to any figure, i.e., subtype
begin
  case F.Kind is
  when Circle => return Pi * Radius ** 2;
  ...
end Area;
L : Figure(Line);
F : Figure; -- illegal, don’t know which kind
P1 := Point;
...
C := (Circle, Red, False, 10, P1);
   -- record aggregate
... C.Orientation ...
   -- illegal, circles have no orientation
C := L;
   -- illegal, different kinds
C.Kind := Square;
   -- illegal, discriminant is constant

Discriminant is a visible constant component of object.
Variants and classes

- discriminated types and classes have overlapping functionalities
- discriminated types can be allocated statically
- run-time code uses less indirection
- compiler can enforce consistent use of discriminants
- adding new variants is disruptive; must modify every case statement
- variant programming: one procedure at a time
- class programming: one class at a time
Free unions can be used to bypass the type model:

```c
union value {
    char *s;
    int i;    // s and i allocated at same address
};
```

Keeping track of current type is programmer’s responsibility. Can use an explicit tag:

```c
struct entry {
    int discr;
    union {
        // anonymous component, either s or i.
        char *s; // if discr = 0
        int i;   // if discr = 1, but system won’t check
    };
};
```
In dynamically-typed languages, only values have types, not names.

\[ S = 13.45 \quad \# \text{a floating-point number} \]
\[ \ldots \]
\[ S = [1,2,3,4] \quad \# \text{now it’s a list} \]

Run-time values are described by discriminated unions. Discriminant denotes type of value.

\[ S = X + Y \quad \# \text{arithmetic or concatenation} \]
Lists, sets and maps

- list: ordered collection of elements
- set: collection of elements with fast searching
- map: collection of (key, value) pairs with fast key lookup

Low-level languages typically do not provide these. High-level and scripting languages do, some as part of a library.

- Perl, Python: built-in, lists and arrays merged.
- C, Fortran, Cobol: no
- C++: part of STL: `list<T>`, `set<T>`, `map<K,V>`
- Java: yes, in library
- Setl: built-in
- ML, Haskell: lists built-in, set, map part of library
- Scheme: lists built-in
- Pascal: built-in sets
  but only for discrete types with few elements, e.g., 32
Function types

- not needed unless the language allows functions to be passed as arguments or returned
- variable number of arguments:
  - C/C++: allowed, type system loophole, Java: allowed, but no loophole
- optional arguments: normally not part of the type.
- missing arguments in call: in dynamically typed languages, typically OK.
Name vs structural

■ name equivalence
  two types are the same only if they have the same name (each type definition introduces a new type)
  carried to extreme in Ada:
  “If a type is useful, it deserves to have a name.”

■ structural equivalence
  two types are equivalent if they have the same structure

Most languages have mixture, e.g., C: name equivalence for records (structs), structural equivalence for almost everything else.
Type equivalence examples

Name equivalence in Ada:

```ada
type t1 is array (1 .. 10) of boolean;
type t2 is array (1 .. 10) of boolean;
v1: t1; v2: t2; -- v1, v2 have different types

x1, x2: array (1 .. 10) of boolean;
-- x1 and x2 have different types too!
```

Structural equivalence in ML:

```ml
type t1 = { a: int, b: real };
type t2 = { b: real, a: int };
(* t1 and t2 are equivalent types *)
```
type student = {
    name: string,
    address: string
}

type school = {
    name: string,
    address: string
}

type age = float;
type weight = float;

With structural equivalence, we can accidentally assign a `school` to a `student`, or a `age` to a `weight`.
Polymorphisms

- **Class polymorphism:**
  The ability to treat a class as one of its superclasses.
  The basis of OOP.

- **Parametric polymorphism:**
  The ability to treat any type uniformly.
  Found in ML, Haskell, and, in a very different form, in C++ templates and Java generics.

- **Subtype polymorphism:**
  The ability to treat a value of a subtype as a value of a supertype.
  Related to subclass polymorphism.
fun length xs =  
    if null xs  
    then 0  
    else 1 + length (tl xs)

length returns an int, and can take a list of any element type, because we don’t care what the element type is. The type of this function is written ’a list -> int.
Subtyping

- A relation between types; similar to but not the same as subclassing.
- Can be used in to different ways:
  - Subtype polymorphism
  - Coercion
Subtype polymorphism and coercion

- subtype polymorphism: ability to *treat* a value of a subtype as a value of a supertype.
- coercion: ability to *convert* a value of a subtype to a value of a supertype.

For example, a record type containing fields \texttt{a}, \texttt{b} and \texttt{c} can be considered a subtype of one containing only \texttt{a} and \texttt{c}. 
Overloading: Multiple definitions for a name, distinguished by their types. Overload resolution: Process of determining which definition is meant in a given use.

- Usually restricted to functions
- Usually only for static type systems
- Related to coercion. Coercion can be simulated by overloading (but at a high cost). If type \(a\) has subtypes \(b\) and \(c\), we can define three overloaded functions, one for each type. Simulation not practical for many subtypes or number of arguments.

Overload resolution based on:
- number of arguments (Erlang)
- argument types
- return type
Ability to declare that a variable will not be changed:

- **C/C++**: `const`
- **Java**: `final`

May or may not affect type system: C++ - yes, Java - no
Type checking and inference

- **Type checking:**
  - Variables are declared with their type.
  - Compiler determines if variables are used in accordance with their type declarations.

- **Type inference:** (ML, Haskell)
  - Variables are declared, but not their type.
  - Compiler determines type of a variable from its usage/initialization.

In both cases, type inconsistencies are reported at compile time.

```plaintext
fun f x = 
  if x = 5
    (* There are two type errors here *)
  then hd x
  else tl x
```