Type Systems

Types

- Denotational view
  - Set of values (e.g., enumerated type)
- Constructive view
  - One of a small collection of built-in types or a composite type (record, array, set, …)
- Abstraction-based view
  - An interface consisting of a set of operations with well-defined and mutually consistent semantics

→Types usually a mixture of these viewpoints

Type Expressions

- Type expressions
  - Used to represent the type of a language construct
  - Describes both language and programmer types
- Examples
  - Basic types: integer, real, character, …
  - Constructed types: arrays, records, pointers, functions, …
- Constructing new types
  - Arrays
  - Records
  - Pointers
  - Functions

Type checking

- Type checker
  - Enforces rules of type system
  - May be strong/weak, static/dynamic
- Static type checking
  - Performed at compile time
  - Early detection, no run-time overhead
  - Not always possible (e.g., A[i])
- Dynamic type checking
  - Performed at run time
  - More flexible, rapid prototyping
  - Overhead to check run-time type tags

Type Checking

- Type equivalence
  - When are the types of two values the same?
- Type compatibility
  - When can a value of type A be used in a context that expects type B?
- Type inference
  - What is the type of an expression, given the types of the operands?

Type Equivalence

- Structural equivalence
  - Based on content
  - Identical type structure/same components in same order
- Name equivalence
  - Each definition (lexical occurrence) introduces a new type (e.g., Java)
  - Strict versus loose (whether or not aliases represent new types)
    - Strict – type A→B represents definition (aliases not equivalent)
    - Loose – type A→B represents declaration or binding (aliases equivalent)
    - Ada allows programmer to specify if an alias is derived (incompatible with base type) or a subtype (compatible)

Structural Equivalence

- Generally -
  - two structs are structurally equivalent if they contain the same number of fields and the corresponding fields in order of declaration are equivalent
  - Arrays are structurally equivalent if they have the same size and each element is structurally equivalent
  - Scalar types are equivalent only to themselves
  - Pointers are structurally equivalent if the types they point to are structurally equivalent
- C and C++ - structurally equivalent except for structs, where fields have to have same names in addition to types (field name is part of type)

Loose Name Equivalence

```
TYPE celsius_temp = REAL;
     fahrenheit_temp = REAL;
VAR  c : celsius_temp;
     f : fahrenheit_temp;
...
f := c;              (* this should probably be an error *)
```

Assuming c and f are equivalent (loose name equivalence/aliases) probably does not reflect what programmer intended

Type Conversion and Casts

- Converting type casts
  - No code needed for structural equivalence
  - Run-time semantic error for intersecting values
  - Possible conversion of low-level representations, e.g., float to integer
- Non-converting type casts
  - E.g., array of characters reinterpreted and pointers or integers, bit manipulation of floats

Type Compatibility and Coercion

- Definition of compatibility language dependent
  - Ada – type S compatible with type T iff
    - S and T are equivalent
    - One is a subtype of the other
    - Both are arrays with same numbers and types of elements in each dimension
- Coercion - Implicit conversion to expected type (specifically allowed by a language, e.g., C and Fortran)
- Type inference – determining the type of an expression
  - Complications – subranges, composite objects

Type Compatibility - Subranges

```
type weekday = {sun, mon, tue, wed, thu, fri, sat};
subtype workday is weekday range mon..fri;
```

```
d : weekday;          -- as above
k : workday;          -- as above
type calendar_column is new weekday;
c : calendar_column
...
k := d;      -- run-time check required
d := k;      -- no check required; every workday is a weekday
c := d;      -- static semantic error;
             -- weekdays and calendar_columns are not compatible
```

Type Coercion

```
short int s;
unsigned long int l;
char c;     /* may be signed or unsigned -- implementation-dependent */
float f;    /* usually IEEE single-precision */
double d;   /* usually IEEE double-precision */
...
s = l;  /* l's low-order bits are interpreted as a signed number. */
l = s;  /* s is sign-extended to the longer length, then
           its bits are interpreted as an unsigned number. */
s = c;  /* c is either sign-extended or zero-extended to s's length;
           the result is then interpreted as a signed number. */
f = l;  /* l is converted to floating-point.  Since f has fewer
           significant bits, some precision may be lost. */
d = f;  /* f is converted to the longer format; no precision lost. */
f = d;  /* d is converted to the shorter format; precision may be lost.
           If d's value cannot be represented in single-precision, the
           result is undefined, but NOT a dynamic semantic error. */
```

General-Purpose Container Objects

- Generic reference, e.g., (void *) in C and C++
- Safety of generic to specific assignments, e.g., Java

```
…
Stack my_stack = new Stack();
String s = "Hello, world";
Foo f = new Foo();
…
my_stack.push(s);
my_stack.push(f);
…
S = (String) my_stack.pop();
// type cast is required, generates exception at run time
// by checking type tag in self-descriptive object
```

Type Classification and Implementation

- Built-in types
  - Integers, characters, booleans, floats
- Enumeration types (first introduced in Pascal)
  e.g., enum weekday(sun,mon,tue,wed,thu,fri,sat);
- Subrange types (also first introduced in Pascal)
  e.g., *type test_score = 0..100;*
- Composite (constructed) types –
  - Records (structures) – introduced by Cobol
  - Variant records (unions) – union of fields
  - Arrays
  - Sets – collection of distinct elements of base type (also introduced by Pascal)
  - Pointers – reference (most often used to implement recursive data types)
  - Lists – recursive sequence of elements
  - Files – represent data on mass storage devices

Records

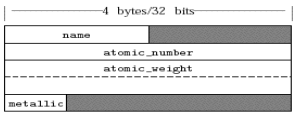
In Pascal, a simple record might be defined as follows:

```
type two_chars = packed array [1..2] of char;
     (* a 'packed' array of char is compatible with a quoted string *)
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```

In C, the corresponding declaration would be

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    char metallic;          /* C has no Boolean type */
};
```

Records – Memory Layout



Compiler may insert holes in the allocation of memory for efficiency of access

Variant Records

```
type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
            (* textual description of principal commercial source *)
        prevalence : real;
            (* percentage, by weight, of Earth's crust *)
      );
      false : (
        lifetime : real;
            (* half-life in seconds of the most stable known isotope *)
      )
end;
```

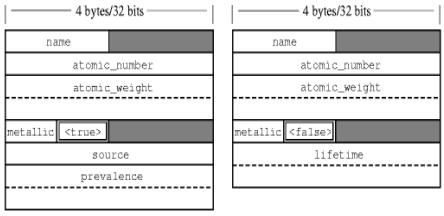
Safety can be checked at run time – e.g., Pascal, Algol 68

Variant Records (Unions)

```
struct element {
        char name[2];
        int atomic_number;
        double atomic_weight;
        char metallic;
        char naturally_occurring;
        union {
                struct {
                        char *source;
                        double prevalence;
                } natural_info;
                double lifetime;
        } extra_fields;
} copper;
```

Type safety cannot be checked – e.g., C

Variant Records – Memory Layout



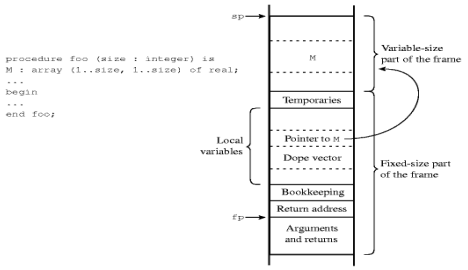
Arrays

- Memory layout strategies
  - Contiguous elements
    - Column major (Fortran)
    - Row major (most other languages)
  - Row pointers – an option in C, used by Java
    - Avoids multiplication
    - Allows rows to be put anywhere
    - Requires extra space for pointers
    - Can have rows of different lengths (e.g., array of strings)

Arrays

- Dimensions, bounds, and allocation
  - Global lifetime, static shape
  - Local lifetime, static shape
  - Local lifetime, shape bound at elaboration time
  - Arbitrary lifetime, shape bound at elaboration time
  - Arbitrary lifetime, dynamic shape
- Use dope vector – runtime descriptor containing bounds and size for each dimension – when shape is not known statically

Local lifetime, shape bound at Elaboration Time E.g., Ada



Pointers

- Reference to an object
  - Variables of built-in Java types employ a value model
  - Variable of user-defined types employ a reference model
- Pointers and single-dimensional arrays in C are interchangeable

Dangling References

- A live pointer that no longer points to a valid object
- May be caught using
  - Tombstones – extra level of indirection, modify tombstone when object is reclaimed
  - Locks and keys – every pointer has a key that is compared to the one stored in the object pointed to

Garbage Collection

- Reference counts
- Mark-and-sweep collection

Input/Output

- Built into language – e.g., Pascal
- Provided by library routines – e.g., C

Types - Overview

- Types
  - Values that share a set of common properties
  - Defined by language and/or programmer
- Type system
  - Set of types in a programming language
  - Rules that use types to specify program behavior
- Example type rules
  - If operands of addition are of type integer, then result is of type integer
  - The result of the unary & operator is a pointer to the object referred to by the operand
- Advantages of typed languages
  - Ensure run-time safety
  - Expressiveness (overloading, polymorphism)
  - Provide information for code generation