22c:111 Programming Language Concepts

Fall 2008

Types I

5.1 Type Errors
5.2 Static and Dynamic Typing
5.3 Basic Types
5.4 NonBasic Types
5.5 Recursive Data Types
5.6 Functions as Types
5.7 Type Equivalence
5.8 Subtypes
5.9 Polymorphism and Generics
5.10 Programmer-Defined Types

Def: A *type* is a collection of values and operations on those values.

Examples:

- The Integer type has values ..., -2, -1, 0, 1, 2, ... and operations +, -, *, /, <, ...
- The Boolean type has values true and false and operations ∧, ∨, ¬.

Computer types have a finite number of values due to fixed size allocation; problematic for numeric types.

Exceptions:

- Smalltalk uses unbounded fractions.
- Haskell type `Integer` represents unbounded integers.

Floating point problems?

Even more problematic is fixed sized floating point numbers:

- 0.2 is not exact in binary.
- So 0.2 * 5 is not exactly 1.0
- Floating point is inconsistent with real numbers in mathematics.

In the early languages, Fortran, Algol, Cobol, all of the types were built in.

If needed a type color, could use integers; but what does it mean to multiply two colors.

Purpose of types in programming languages is to provide ways of effectively modeling a problem solution.

5.1 Type Errors

Machine data carries no type information.

Basically, just a sequence of bits.

Example: 0100 0000 0101 1000 0000 0000 0000 0000

0100 0000 0101 1000 0000 0000 0000 0000

- The floating point number 3.375
- The 32-bit integer 1,079,508,992
- Two 16-bit integers 16472 and 0
- Four ASCII characters: @ X NUL NUL

Def: A *type error* is any error that arises because an operation is attempted on a data type for which it is undefined.

Type errors are common in assembly language programming.

High level languages reduce the number of type errors.

Def: A *type system* is a precise definition of the bindings between the types of a variable, its values, and the possible operations over those values

A type system provides a basis for detecting type errors.

5.2 Static and Dynamic Typing

A type system imposes constraints (such as the values used in an addition must be numeric).

- Cannot be expressed syntactically in EBNF.
- Some languages perform type checking at compile time (eg, C, C++, OCaml).
- Other languages (eg, Perl,Scheme,Python) perform type checking at run time.
- Still others (eg, Java) do both.

Def: A language is *statically typed* if the types of all variables are fixed when they are declared at compile time.

Def: A language is *dynamically typed* if the type of a variable can vary at run time depending on the value assigned.

Can you give more examples of each?

Def: A language is *strongly typed* if its type system allows all type errors in a program to be detected either at compile time or at run time.

Note: A strongly typed language can be either statically or dynamically typed.

Union types are a hole in the type system of many languages (eg, C, C++).

Most dynamically typed languages associate a type with each value.

5.3 Basic Types

Terminology in use with current 32-bit computers:

- Nibble: 4 bits
- Byte: 8 bits
- Half-word: 16 bits
- Word: 32 bits
- Double word: 64 bits
- Quad word: 128 bits

In most languages, the numeric types are finite in size.

So `a + b` may overflow the finite range.

Unlike mathematics:

$$a + (b + c) \neq (a + b) + c$$

Can you see why?

Also in C-like languages, the equality and relational operators produce an `int`, not a Boolean

- `(2 < 4)` evaluates to `0`
- `(2 > 4)` evaluates to `1`
- `if 5 {...} else {...}` is legal, and meaningful, code!

Def: An operator or function is *overloaded* when its meaning varies depending on the types of its operands or arguments or result.

Java: `a+b` (ignoring size)

- integer add
- floating point add
- string concatenation

Mixed mode: one operand an int, the other floating point

Languages that allow mix mode syntax introduce implicit type conversion between values (eg. `3.4 + 1` is treated as `3.4 + intToFloat(1)`)

Def: A type conversion is a *narrowing* conversion if the result type permits fewer bits, thus potentially losing information. Otherwise it is a *widening* conversion.

Should languages ban implicit narrowing conversions?

Why?

5.4 Nonbasic Types

Enumerations

```
enum day {Monday, Tuesday, Wednesday, Thursday,
    Friday, Saturday, Sunday};
enum day myDay = Wednesday;
```

In C/C++ these just define an int range [0..6]

where Monday == 0, Tuesday == 1 and so on

Enumeration types are powerful in Java:

```
for (day d : day.values()) Sytem.out.println(d);
```

They are even more powerful in Ocaml, Haskell as they a special case of *algebraic data types*

(more on them later)

Pointers

C, C++, Ada, Pascal

Java??? OCaml??

The values in a pointer type are memory addresses

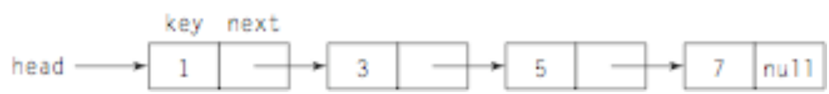
They are used for indirect referencing of data

Operator in C: *

Example

```
struct Node {
    int key;
    struct Node* next;
};
struct Node* head;
```

Fig 5.4: A Simple Linked List in C



Pointers

Bane of reliable software development

Error-prone

Buffer overflow, memory leaks

Particularly troublesome in C

strcpy

```
void strcpy(char *p, char *q) {
   while (*p++ = *q++)  ;
}
```

Pointer Operations

If T is a type and ref T is a pointer:

$\& : T \rightarrow \text{ref } T$

$* : \text{ref } T \rightarrow T$

For an arbitrary variable x:

$*(\&x) = x$

Arrays

```
int a[10];

float x[3][5];  /* odd syntax vs. math */

char s[40];

/* indices: 0 ... n-1 */
```

Array Indexing

Only operation for many languages

Type signature

[ ] : T[ ] x int → T

Example

float x[3] [5];

type of x: float[ ][ ]

type of x[1]: float[ ]

type of x[1][2]: float

Equivalence between arrays and pointers in C/C++

a = &a[0]

If either e1 or e2 is type: ref T

e1[e2] = *(e1 + e2)

Example: a is float[ ] and i int

a[i] = *(a + i)

```c
float sum(float a[ ], int n) {
int i;
float s = 0.0;
for (i = 0; i<n; i++)
     s += a[i];
return s;
```

```c
float sum(float *a, int n) {
int i;
float s = 0.0;
for (i = 0; i<n; i++)
     s += *a++;
return s;
```

Strings

Now so fundamental, directly supported.

In C, a string is a 1D array with the string value terminated by a NULL character (value = 0).

In Java, Perl, Python, a string variable can hold an unbounded number of characters.

Libraries of string operations and functions.

Structures (aka Records)

Analogous to a tuple in mathematics

Collection of elements of different types

Used first in Cobol, PL/I

Absent from Fortran, Algol 60

Common to Pascal-like, C-like, ML-like languages,

Omitted from Java as redundant

```
struct employeeType {
    int id;
    char name[25];
     int age;
     float salary;
     char dept;
};
struct employeeType employee;
...
employee.age = 45;
```

Unions

C: union

Pascal: case-variant record

Logically: multiple views of same storage

Useful in some systems applications

In functional languages, superseded by *recursive data types* (sometimes also called union types or algebraic data types)

```
(* Union type in Pascal *)
type union = record
     case b : boolean of
          true : (i : integer);
          false : (r : real);
     end;
var  u : union, j: integer;
begin
    u := (b => false, r => 3.375);
    j := tagged.i;  (* will generate error *)
```

```
// simulated union type in Java
class Value extends Expression {
    // Value = int intValue | boolean boolValue
    Type t;  int intValue;  boolean boolValue;
    Value(int i) { intValue = i;
        t = new Type(Type.INTEGER);
    }
    Value(boolean b) { boolValue = b;
        t = new Type(Type.BOOLEAN);
    }
```