CSCI312 Principles of Programming Languages

Chapter 5

Types

Xu Liu

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

Example: 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

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.

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).
- Other languages (eg, Python) perform type checking at run time.
- Still others (eg, Java) do both.

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

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

Can you give examples of each?

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.

A strongly typed language can be either statically (e.g., Ada, Java) or dynamically typed (e.g., Python and Perl).

An Example C Program

```
int main(){
  union {int a; float p;} u;
  u.a = -1;
  float x=0;
  x = x + u.p;
  printf ("x=%f\n", x);

  float y=0;
  y = y + (float)u.a;
  printf ("y=%f\n", y);

  return 0;
}
```

00000...00100000000

↑

u.a

u.p

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:

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

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

Overloading

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

Python: a + b (ignoring size)

- integer add
- floating point add
- string concatenation

- *What if a is integer while b is floating point?*

Type Conversion

A type conversion is a *narrowing* conversion if the result type permits fewer bits, thus potentially losing information. E.g., float -> int

Otherwise it is termed a *widening* conversion.

E.g., int -> float

Explicit conversion: 2 + int(1.3);

Implicit conversion: 2 + 1.3;

Should languages use narrowing or widening for implicit conversions?

5.4 Nonbasic Types

Enumeration:

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

In C/C++ the above values of this type are 0, ..., 6.

More powerful in Java:

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

Pointers

C, C++, Ada, Pascal

Java???

Value is a memory address

Indirect referencing

Operator in C: *

Pointer Operations

If *T* is a type and *refT* is a pointer:

- $\& : T \rightarrow refT$. *Eg. &x: returns the address of x*
- $* : refT \rightarrow T$. *Eg. *p: returns the value in the location that p references.*

For an arbitrary variable x:

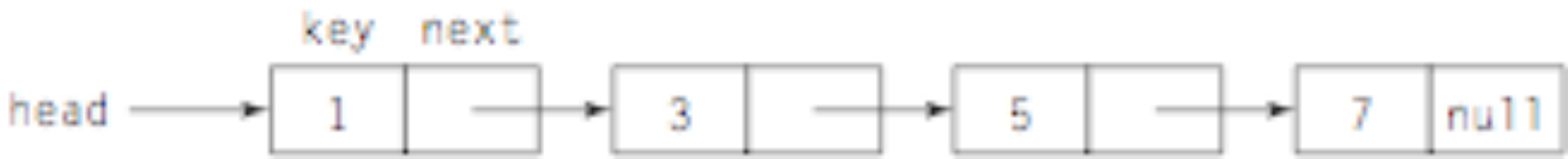
- $*(\&x) = x$

Example

```
int main(){
  int v = 3;
  int * p = &v;

  (*p) = -3;

  printf("v=%d\n", v);

  return 0;
}
```

Pointers are convenient in some cases

Example: Linked List




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

But Error-Prone

E.g. Buffer overflow problem

String copy:

```
while (*p++ == *q++);
```

q points to “a string$”
p points to a 3-char buffer.

Particularly troublesome in C as points and array are regarded the same.

Equivalence between arrays and pointers

int a[100]; // declare an array

- a == &a[0]
- a[i] == *(a + i)

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

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

Arrays and Lists

```
int a[10];
float x[3][5];  /* odd syntax vs. math */
char s[40];
```

Indexing

The only operation for arrays and lists in many languages

Type signature

$[\ ] : T[\ ] \times int \rightarrow T$

Example

float x[3] [5];

type of x: float[ ][ ]

type of x[1]: float[ ]

type of x[1][2]: float

Strings

Now so fundamental, directly supported.

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

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

Libraries of string operations and functions.

Structures

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 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

```
union {int a; float p;} u;
```

Pascal: case-variant record

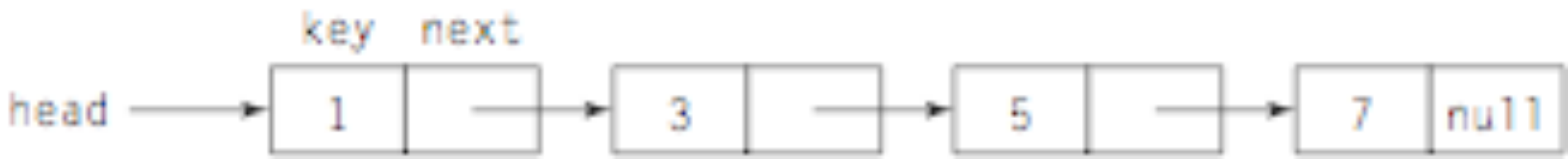
Logically: multiple views of same storage

Useful in some systems applications

Contents

5.5 Recursive Data Type

Example: Linked List




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

Others?

5.6 Functions as Types

Needs example: a function to draw the curve for

$y=f(x)$.

Pascal example:

```
function newton(a, b: real; function f: real): real;
```

Know that f returns a real value, but the arguments to f are unspecified.

Addressed by Java Interface

```
public interface RootSolvable {
        double valueAt(double x);
}

public double Newton(double a, double b,
    RootSolvable f);
```

5.7 Type Equivalence

Pascal Report:

The assignment statement serves to replace the current value of a variable with a new value specified as an expression. ... The variable (or the function) and the expression must be of identical type.

Nowhere does it define *identical type*.

```
struct complex {
    float re, im;
};
struct polar {
    float x, y;
};
struct {
    float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[10];
// which are equivalent types?
```

Kinds of Type Equivalence

Name equivalence

Structural equivalence: # and order of fields of a structure, and the name and type of each field.

Ada, Java: name equivalence.

C: name equivalence for structs and unions, structural equivalence for other constructed types (arrays and pointers). Size of an array doesn't matter.

```
struct complex {
    float re, im;
};
struct polar {
    float x, y;
};
struct {
    float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[10];
// which are equivalent types?
```

Name equivalence: a, b
Structural equivalence:
  a, b, c, d

f, g are equivalent in both cases.

5.8 Subtypes

A subtype is a type that has certain constraints placed on its values or operations.

In Ada subtypes can be directly specified.

```
subtype one_to_ten is Integer range 1 .. 10;
type Day is (Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday);
subtype Weekend is Day range Saturday .. Sunday;
type Salary is delta 0.01 digits 9
            range 0.00 .. 9_999_999.99;
subtype Author_Salary is Salary digits 5
            range 0.0 .. 999.99;
```

Example in Java

```
Integer i = new Integer(3);
...
Number v = i;
...
Integer x = (Integer) v;
//Integer is a subclass of Number,
// and therefore a subtype
```

Polymorphism and Generics

A function or operation is *polymorphic* if it can be applied to any one of several related types and achieve the same result.

An advantage of polymorphism is that it enables code reuse.

Polymorphism

Comes from Greek

Means: having many forms

Example: overloaded built-in operators and functions

+ - * / == != ...

Java: + also used for string concatenation

Ada, C++: define + - ... for new types

Java overloaded methods: number or type of parameters

Example: class PrintStream

print, println defined for:

boolean, char, int, long, float, double, char[ ], String, Object

- Java: instance variable, method
  - *name, name( )*
- Ada generics: generic sort
  - *parametric polymorphism*
  - *type binding delayed from code implementation to compile time*
  - *procedure sort is new generic_sort(integer);*

```
generic
   type element is private;
   type list is array(natural range <>) of element;
   with function ">"(a, b : element) return boolean;

package sort_pck is
   procedure sort (in out a : list);
end sort_pck;
```

```
package sort_pck is
procedure sort (in out a : list) is
begin
    for i in a'first .. a'last - 1 loop
        for j in i+1 .. a'last loop
            if a(i) > a(j) then
                declare t : element;
                begin
                    t := a(i);
                    a(i) := a(j);
                    a(j) := t;
                end;
        end if;
```

Instantiation

```
package integer_sort is
    new generic_sort( Integer, ">" );
```

Programmer-defined Types

Recall the definition of a type:

A set of values and a set of operations on those values.

Structures allow a definition of a representation; problems:

- Representation is not hidden
- Type operations cannot be defined

Defer further until Chapter 12.