

Programming Languages

2nd edition
Tucker and Noonan

Chapter 5 Types

*Types are the Leaven of computer programming;
they make it digestible.*

Robin MILner

Agenda for Today

- Continuing with TYPES
 - Warm-up & Review
 - Arithmetic calculations (AC)
 - Type errors & conversion (TC)
 - Non-basic types (NBT)

Warm-up

- 'Binding' involves what four characteristics?
- In what key way do Strings differ from int, double, char?
- What are the two components of a type?
- Specifying a Lexicon subsumes what two areas?
- List 4 of the "design goals" that characterize a "good" programming language.
- In specifying the Lexicon/Names, what do the two questions "which binding" refer to?

AC: imperfect model

- Integers: $2,147,483,647 + 10 \neq 2,147,483,657$
 - Why?
- Reals: $(0.2 * 5) \neq 1$
 - Why?

← Finite Resources

AC: Range & Precision

- Integers: $2,147,483,647 + 10 \neq 2,147,483,657$
 - 0000 0000 0000 0000 0000 0000 0000 0000
 - Problem: finite range in 32 bits
- Reals: $(0.2 * 5) \neq 1$
 - 0 | 0000 0000 | 001 1001 1001 1001 1001 1001
 - Problem: finite range AND finite precision

TC: Agnostic representation

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

TC: Type errors

- "...any error that arises because an operation is attempted on a data type for which it is undefined."
- A *type system* provides a basis for detecting type errors.

TC: Type systems

A type system can be used to enforce constraints on values and operations (c.f. NL verb agreement)

- Cannot be expressed syntactically in EBNF.
- Some languages type check at compile time (e.g., C).
- Other languages type check at run time (e.g., Perl).

TC: Dynamic v. static typing

- In chapter 4 we discussed dynamic v static BINDING
 - Which binding does a particular token refer to?
 - Where does a particular binding hold?
- $N \leftrightarrow T$: dynamic or static TYPING
 - When is $N \leftrightarrow T$ binding determined / checked?
 - What would be advantages of each?

TC: Strong typing

A *strongly typed* language allows all type errors to be detected, either at compile time or at run time.

Both statically and dynamically typed languages can be strongly typed.

Most dynamically typed languages associate one type with each value.

TC: Ambiguity

- Operator overloading: flip side of dynamic typing
 - Operator meaning determined by type context
 - Type determined by operator context
- When overloaded operator occurs with two different types, which meaning is intended?
 - Language must specify which
 - Other must be converted

TC: Conversions

- Conversions can be narrowing or widening

- **Narrowing**

- Result has fewer bits than original
- E.g., long \rightarrow int; double \rightarrow float; float \rightarrow int

- **Widening**

- Result has same # or more bits than original
- E.g., long \rightarrow int; float \rightarrow int

NBT: Enumerations

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

In C/C++ the above values are equivalent to 0, ..., 6.

More powerful in Java:

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

NBT: Arrays

```
int   a[10];
float x[3][5]; // Q: How many R/C? Why?
char  s[40];
```

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

NBT: Indexing

Type signature

$\beta [n]$ means an array with n elements of type β

Example

```
int x[3]           // 3-elmt array of ints
int x[3][5];      // 5-elmt array of
                  // 3-elmt arrays of int
```

NBT: 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.

NBT: 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

NBT: Structures

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

NBT: Unions

```

type union =
  record
    case b : boolean of
      true : (i : integer);
      false : (r : real);
    end;
var tagged : union;
begin tagged := (b=>false, r=>3.375);
put(tagged.i); -- error currently no i
    
```

NBT: Unions

```

class Value extends Expression { // simulated in Java

  Type type;
  int intValue;
  boolean boolValue;

  Value(int i) {
    intValue = i;
    type = new Type(Type.INTEGER);
  }

  Value(boolean b) {
    boolValue = b;
    type = new Type(Type.BOOLEAN);
  }
    
```

NBT: Pointers

C, C++, Ada, Pascal
Hidden in Java

Value is a memory address; allows for indirect referencing

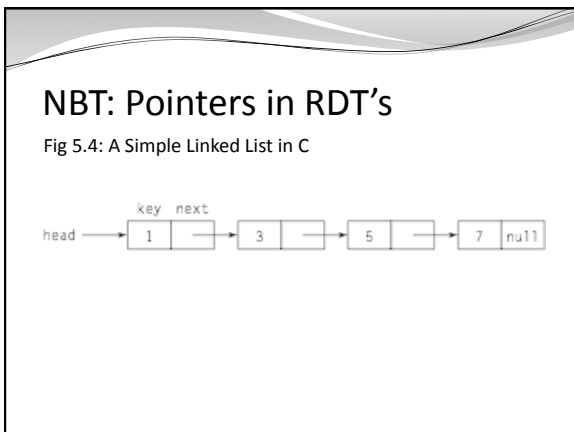
If a is of type int and b of type int*

- &a : address of a
- b* : value stored in b

NBT: Pointers in RDT's

```

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



NBT: Pointer problems

- Bane of reliable software development
- Error-prone
- Buffer overflow, memory leaks
- Particularly troublesome in C

NBT: Indexing & Pointers

Equivalence between arrays and pointers

```
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)
```

NBT: Pointer example 1

```
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;
}
```

NBT: Pointer example 2

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

NBT: Pointer operations

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

```
& : T → ref T
```

```
* : ref T → T
```

For an arbitrary variable x:

```
*(&x) = x
```

5.5 Recursive Data Type

```

data Value = IntValue Integer | FloatValue Float |
           BoolValue Bool | CharValue Char
           deriving (Eq, Ord, Show)
data Expression = Var Variable | Lit Value |
               Binary Op Expression Expression |
               Unary Op Expression
               deriving (Eq, Ord, Show)
type Variable = String
type Op = String
type State = [(Variable, Value)]

```

5.6 Functions as Types

- 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.

```

• 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?

```

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;

- 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 83

- 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.