Types (1^{st} *Part*)

Francesco Nidito

Programmazione Avanzata AA 2005/06

Outline

1. Introduction
2. Type checking
3. Types in the practice
4. Advanced Types

Reference: Micheal L. Scott, “Programming Languages Pragmatics”, Chapter 7

What is a type

- Hardware
    - can manage bits in different ways
    - has no type, but provides operations on numbers and pointers
- Software creates the abstraction of types
- Type
    - defines the memory layout of data
    - defines a set of operations that can be performed on value belonging to that type

Type system

A *type system* consists of

- a mechanism for *defining* types and *associating* them to language structures
- a set of rules for:
  - type equivalence ($Type_A = Type_B$?)
  - type compatibility ($Type_A \in Context_i$?)
  - type inference ($x \in Type_A$?)

Type system rules (Example)

- type equivalence ($Type_A = Type_B$?)
  e.g. Is it safe to cast an integer to a char?

  ```
  integer x := 26;
  char a := (char)x;
  ```

- type compatibility ($Type_A \in Context_i$?)
  e.g. Can I add a string and a real?

  ```
  string s := ‘‘foo’’;
  real x := s + 5.0;
  ```

- type inference ($x \in Type_A$?)
  e.g. For which types of x is f defined?

  ```
  let f x = x + x;;
  ```

What type systems are good for

- Detecting errors
- Enforcing abstraction
- Documentation
- Efficiency

What is type checking

Type checking is the process of ensuring that a program obeys the language's type compatibility rules

Strong vs. weak typing

Strong typing

Values of one type cannot be assigned to variables of another type.
Enables incredibly extensive *static compiler checks*.

Weak typing

Values of one type can be assigned to variables of another type using implicit value conversions.

Strong vs. weak typing (Example)

- Strong typing check returns an error

```
type fruitsalad: integer;
type apple: integer;
type pear: integer;
apple a := 5;
pear p := 3
fruitsalad f := a + p;
```

- Weak typing check goes on

```
type fruitsalad: integer;
type apple: integer;
type pear: integer;
apple a := 5;
pear p := 3
fruitsalad f := a + p; //fruitsalad = 8
```

Dynamic vs. static typing

Dynamic typing

Environment *infers* the type of a variable/expression from the its usage. It can happen both at runtime and compile-time.

Static typing

Programmer must indicate the type of a variable/expression writing it in the code. It's checked at compile-time.

Obviously, in real world they can be mixed!

Dynamic vs. static typing (Example)

- Dynamic typing:

```
s := ‘‘foo’’; //s is string
n := sqrt(42); //n is real
```

- Static Typing:

```
string s := ‘‘foo’’; //s is string
real n := sqrt(42); //n is real
```

Game of types

Non-Typed	Typed		
	Static	Dynamic	
			Strong
			Weak

Types in programming languages

- boolean
- int, long, float, double (signed/unsigned)
- char (1 byte, 2 bytes)
- Enumerations
- Subrange ($n_1..n_2$)
- Pointers
- Composite types
  - struct
  - union
  - array

Type cast

- Type cast operation builds from an expression with type $Type_A$ a new value of type $Type_B$
- Consider the following definitions:

```
int add(int i, int j);
int add2(int i, double j);
```

- Ad the following calls:

```
add(2, 3); //Exact
add(2, (int)3.0); //Explicit cast
add2(2, 3); //Implicit cast
```

Memory layout

- On 32 bits architectures types require from 1 to 8 bytes
- Composite types (e.g. structures) are represented chaining constituent values together
- For performance reasons compilers employ *padding* to align fields to 4 bytes addresses

Memory layout (Example)

```
struct element {
   char name[2];
   int atomicnumber;
   float atomicweight;
   char metallic;
};
```

name		free	free
atomicnumber			
atomicweight			
metallic	free	free	free

Problems with memory layout

- C requires that fields of a struct should be displaced in the same order of the declaration (essential with pointers!)
- Not all languages behaves like this: for instance ML doesn't specify any order
- If the compiler can reorganize fields, "holes" are minimized: for instance packing `name` and `metallic` saves 4 bytes

Union

- Union types allow sharing the same memory area among different types
- The size of the value is the maximum of the size of the constituents

```
union u {
   struct element e;
   int number;
};
```

name		free	free
atomicnumber			
atomicweight			
metallic	free	free	free

number			
free	free	free	free
free	free	free	free
free	free	free	free

Enumerate

- User defined types to increase expressivity
- Values of an enumerate are ordered and can be used as indexes of arrays or collections

```
enum weekday {sun, mon, tue, wed, thu, fri, sat };
```

Array

- Array are *positional* collections of *homogeneous* data
- From an abstract point of view an array is a mapping from an *index type* to an *element type*
- Array's indexes
  - can be fixed (e.g. starting from 0 as in C)
  - can have lower and upper bound (e.g 5..10)
- Array layout of memory is *contiguous*

```
int char[26]; // C/C++

var frequency : array['a'..'z'] of integer; //Pascal
```

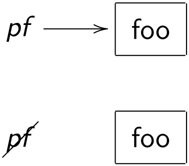
Pointers

- Not a real type, it's a *label*
- A pointer variable is a variable whose value is a *reference* to some object
- A pointer is not an address of memory. It is an high level reference
- One pointer can refer to an already existing object
- A pointer can be created allocating memory for it
- A pointer that was created "must" be destroyed

Problems with pointers: memory leak

- A created pointer must be destroyed to clean memory
- A pointer variable when out of scope is lost
- ...but the pointed object is still in memory
- The pointed object cannot be accessed but uses memory

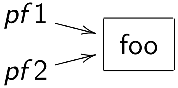
```
{
   foo pf = new foo();
}
```



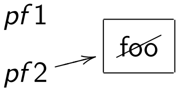
Problems with pointers: dangling reference

- Suppose two pointers pointing to the same object
- When one of the two pointers is destroyed the object is removed from memory
- ...but the second pointer is a live pointer that no longer points to a valid object
- The access to the cleaned object can rise errors

```
foo pf1 := new foo();
foo pf2 := pf1;
```



```
delete(pf1);
```



Abstract data types

- According to the abstraction based view of types a type is an *interface*
- An ADT is a set of *values* and *operations* allowed on it
- Programming languages have mechanisms to define ADT

Abstract data types (Example)

```
struct node {
   int val;
   struct node *next;
};

struct node* next(struct node* l) { return l->next; }

struct node* initNode(struct node* l, int v) {
   l->val = v; l->next = NULL; return l;
}

void append(struct node* l, int v) {
  struct node p = l;
  while (p->next) p = p->next;
  p->next =
  initNode((struct node)malloc(sizeof(struct node)),v);
}
```

Abstract data types limits

- C doesn't provide any mechanism to hide the structure of data types
- A program can access `next` field without using the `next` function
- To hide data and to preserve abstraction we must use a *Class*

Class type

- Class is a *type constructor* like struct or array
- A class combines
  - Data (like struts)
  - Methods (operations on the data)
- A class has two special operations to provide
  - Initialization
  - Finalization

Class type (Example)

```
class Node {
   int val;
   Node m_next;
   Node(int v) { val := v; }
   Node next() { return m_next; }
   void append(int v) {
      Node n := this;
      while (n.m_next != null) n := n.m_next;
      n.m_next := new Node(v);
   }
}
```