

Structuring the Data

Why Use (Static, Strong) Types?

hide machine representation

→ improves programming style and maintainability

consistent use of variables checked by compiler

→ improves readability and correctness

space requirements of variables known at compilation time

→ improves memory efficiency and run-time performance

resolve overloading at compilation time

→ improves readability and performance

Elementary Types

usually: elementary type (not decomposable) = predefined type

“String” in Ada is predefined, but not elementary:

```
type String is array (Positive range <>) of Character
```

enumeration types are elementary, but not predefined:

```
type color = (red, blue, green);
```

Pascal

```
type color is (red, blue, green);
```

Ada

```
enum color { red, blue, green };
```

C

Aggregates and Type Constructors

aggregates are composed of several (elementary) data objects,
e.g., arrays and records

old languages like Fortran, Cobol:
aggregates directly created (without aggregate types)

more recent languages:
new types created from existing types by using type constructors;
aggregates are instances of these types
created by the use of programmer-defined constructors (or sometimes directly)

Cartesian Product

cartesian product $A_1 \times \dots \times A_n$ of n sets A_1, \dots, A_n is the set of all n -tuples (a_1, \dots, a_n) with $a_i \in A_i$

in programming languages: field names instead of indexes (C):

```
typedef struct {  
    int no_of_edges;  
    float edge_size;  
} reg_polygon;  
reg_polygon a_pol = { 3, 3.14 };
```

fields are selected using dot-notation:

```
a_pol.no_of_edges = 4;
```

Finite Mapping

mathematical function maps **domain** into **range**

example: $f: \text{integer} \rightarrow \text{real}$

finite mapping = function with finite domain

definition of mapping as routine is **intensional**

where rules define the values in the range

definition of finite mapping as array is **extensional**

where the values in the range are enumerated

Array Types (Examples)

example in C:

```
char digits[10];  
for (i=0; i<10; i++)  
    digits[i] = ' ';
```

it's an error if an index is not in the domain;
in general, such errors show up at run-time

examples in Pascal:

```
var x: array[2..5] of integer;  
type manufacturer = (ibm, dec, hp, sun);  
type m_data = array[manufacturer] of integer;  
var m_profits, m_empl: m_data;
```

Arrays (Examples)

initialization of arrays in C and Ada:

```
char digits[5] = {'a','b','c','d','e'};  
X: array(INTEGER range 2..6) of INTEGER := (0,2,0,5,-33);
```

multi-dimensional arrays in C and Ada:

```
int y[10][20];  
Y: array(1..27, M..N) of INTEGER;
```

array slicing in Ada:

```
X(2..5) := X(3..6);
```


Associative Data Structures

in dynamically typed languages, array entries can be of different types

example of associative data structure in SNOBOL4:

```
T = TABLE()  
T<'RED'> = 'WAR'  
T<6> = 25  
T<4.6> = 'PEACE'
```

T<6> gives 25 and T<'RED'> gives 'WAR'

Union

definition of union in C:

```
union address {  
    short int offset;  
    long unsigned int absolute;  
};
```

remember which field ist set:

```
enum descriptor {abs, rel};  
typedef struct {  
    address location;  
    descriptor kind;  
} safe_address;
```

Discriminated Union

variant record in Pascal:

```
natural = 0..maxint;  
address_type = (absolute, offset);  
safe_address =  
    record  
        case kind: address_type of  
            absolute: (abs_addr: natural);  
            offset: (off_addr: integer)  
        end  
    end
```

Powerset

powerset = set of all subsets of a set

in programming languages: set is type

example in Pascal:

```
type option = (list, optimize, save, exec);  
    option_set = set of option;
```

```
var active_options: option_set;
```

```
...
```

```
active_options := [optimize, save];
```

```
if exec in active_options then ...
```

Recursive Data Types

examples based on sets:

$$\text{bin_tree} = \{\text{nil}\} \cup (\text{integer} \times \text{bin_tree} \times \text{bin_tree})$$

$$\text{int_list} = \{\text{nil}\} \cup (\text{integer} \times \text{int_list})$$

example of the use of lists in ML:

```
fun find(el, nil) = false
|   find(el, x:xs) =
      if el = x then true
      else find(el, xs) ;
```

Recursive Data Type Examples

examples in C and Ada:

```
typedef struct list {  
    int val;  
    list* next;  
} int_list;  
int_list* head;
```

```
type INT_LIST  
type INT_LIST_REF is access INT_LIST;  
type INT_LIST is  
    record  
        VAL : INTEGER;  
        NEXT: INT_LIST_REF;  
    end;  
HEAD: INT_LIST_REF;
```

Unsafety of Pointers

untyped pointers as in PL/I

→ shall be replaced by typed pointers

arithmetic operations on pointers as in C

→ can be forbidden at the cost of reduced efficiency

dangling references because of scope violations

→ no address operator and all objects on heap; run-time checks difficult

dangling references because of explicit deallocation

→ only garbage collection instead of explicit deallocation

pointers in non-discriminated unions

→ use only discriminated unions

Abstract Data Types

example in C++:

```
class point {  
public:  
    point(int a, int b) { x = a; y = b; }  
    void x_move(int a) { x += a; }  
    void y_move(int b) { y += b; }  
    void reset() { x = 0; y = 0; }  
private:  
    int x, y;  
};
```


Generic Abstract Data Type (C++)

```
template<class T> class Stack {
public:
    Stack(int sz)    { top = s = new T[size = sz]; }
    ~Stack()         { delete[] s; }
    void push(T el)  { *top++ = el; }
    T pop()          { return *--top; }
private:
    int size;
    T *top, *s;
}

void foo() {
    Stack<int> int_st(30);
    Stack<item> item_st(100);
    ...
}
```

Strong and Static Type Systems

type system = set of rules specifying type consistency

strong type system ensures type consistency,

i.e., static type checking ensures type consistency at run-time

static type system associates each expression statically with a type

each static type system is also strong, but not each strong type system is static

Type Compatibility

is type S compatible with type T?
(conformance, equivalence)

name compatibility = only one type definition, otherwise incompatible,

structural compatibility = types have corresponding structures

name compatibility stronger than structural compatibility

because of abstraction through (informal) intentions expressed by names

C uses structural equivalence except for structs,

Ada uses (almost only) name compatibility:

```
IA: array (1..100) of INTEGER;
```

```
IB: arrau (1..100) of INTEGER;  -- not compatible with IA
```

Structural Type Compatibility

many possible ways to define structural type compatibility:

```
type s1 is struct {  
    int y;  
    int w;  
}  
type s2 is struct {  
    int y;  
    int w;  
}  
type s3 is struct {  
    int y;  
}
```

Type Coercion

in some languages some types are implicitly changed when the compiler detects an incompatibility

example in C:

```
int x;  
float z;  
...  
x = x + z;          // coercion of x to float, result to int  
x = x + (int)z;     // explicit type cast, no coercion
```

Names and Constraints in Ada

```
type IntVector is array (Integer range <>) of Integer;
```

```
type VarRec (IsChar: Boolean) is
```

```
    record X: Float;
```

```
        case IsChar of
```

```
            when False => Y: Integer;
```

```
            True  => U: Char;
```

```
        end case;
```

```
    end record;
```

```
subtype Vec100 is IntVector(0..99);    -- compat. with IntVector
```

```
subtype VarRecChar is VarRec(True);    -- compat. with VarRec
```

```
subtype MyInt is Integer range -9..9;  -- compat. with Integer
```

Polymorphic Types

monomorphic: each variable and expression has exactly one type

polymorphic: variables and expressions can have several types

kinds of polymorphism:

universal polymorphism: parametric polymorphism (= genericity)
 inclusion polymorphism (= subtyping)

ad-hoc polymorphism: overloading
 coercion

Scalar Types in Ada

```
type Small_Int is range -10..10;    -- user-defined integer
type Two_Digit is mod 100;         -- modulo numb. (0..99)
```

```
subtype Natural  is Integer range 0..INTEGER'LAST;
subtype Positive is Integer range 1..INTEGER'LAST;
```

```
type Celsius    is new Integer; -- incompatible to Integer
type Fahrenheit is new Integer;
```

```
type MyFloat is digits 10;    -- user-defined floating point
type Fix_Pt  is delta 0.01 range 0.0..100.0; -- fixed point
type Dec_Pt  is delta 0.01 digits 3; -- accurate fixed point
```

```
type Week is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
type Daily is array (Week) of Integer;
type At_Work is array (Week range Mon..Fri) of Integer;
```


Discriminated Union in Ada

```
type Address_Type is (Absolute, Offset);
type Safe_Address is
  record (Kind: Address_Type := Absolute)
    case Kind is
      when Absolute =>
        Abs_Addr: Natural;
      when Offset =>
        Off_Addr: Integer;
    end case;
  end record;
```

Pointers in Ada

```
T: Tree_Ref;  
...  
T := new Bin_Tree_Node;  
T.all := (Info => 0, Left => null, Right => null)  
  
type Message_Routine is access procedure(M: String);  
Give_Message: Message_Routine;  
...  
Give_Message := Print_This'Access;  
Give_Message.all("This is not an error");  
  
Structure: array (1..10) of aliased Component;  
type ComponentPtr is access all Component;  
Mine, Yours: ComponentPtr;  
...  
Mine := Structure(1)'Access;  
Yours := Structure(2)'Access;
```

Modularity and Programming in the Large

Package Specification in Ada

```
package Dictionary is
    procedure Insert(C:String; I:Integer);
    function Lookup(C:String) return Integer;
end Dictionary;
```

Package Body in Ada

```
package body Dictionary is
  type Node;
  type Node_Ptr is access Node;
  type Node is record
    Name: String;
    Id:   Integer;
    Next: Node_Ptr;
  end record;
  Root: Node_Ptr;
  procedure Insert(C:String; I:Integer) is begin ...
  end Insert;
  function Lookup(C:String) return Integer is begin ...
  end Lookup;
begin
  Root := null;
end Dictionary;
```

Use of Package in Ada

```
with Dictionary;  
use Dictionary;  
procedure Main is  
    Code: Integer;  
begin  
    Insert("volleyball", 1);  
    Insert("football", 2);  
    ...  
    Code := Lookup("football");  
    ...  
end Main;
```

Use of Package in Ada

```
with X;
package T is
    C: Integer;
    procedure D(...);
end T;

package body T is
    ...
    ...
    ...
end T;

with T;
procedure U(...) is
    ...
    ... T.D(...) ...
    ... T.C ...
end U;

with T;
use T;
procedure U(...) is
    ...
    ... D(...) ...
    ... C ...
end U;
```

Package Specification with Private Part

```
package Dictionary is
  type Dict is private;
  procedure Insert(D: in out Dict; C: String; I: Integer);
  function Lookup(D: Dict; C: String) return Integer;
private
  type Node;
  type Node_Ptr is access Node;
  type Node is record
    Name: String;
    Id:   Integer;
    Next: Node_Ptr;
  end record;
  type Dict is new Node_Ptr;
end Dictionary;
```


Module in ML

```
structure Dictionary =  
struct  
  exception NotFound;  
  
  val create: (string * int) list = nil;  
  
  fun insert(c:string, i:int, nil:(string*int)list) = [(c,i)]  
    | insert(c, i, (cc,ii)::cs) =  
      if c = cc then (c,i)::cs  
      else (cc,ii)::insert(c,i,cs);  
  
  fun lookup(c:string, nil:(string*int)list) = raise NotFound  
    | lookup(c, (cc,ii)::cs) =  
      if c = cc then ii  
      else lookup(c,cs);  
  
end;
```

Use of Module in ML

```
val d = Dictionary.create
val e = Dictionary.insert("X", 7, d);
...
Dictionary.lookup("X", e);
```

Signature in ML

```
signature DictLookupSig = sig
  exception NotFound;
  val lookup: string * (string * int) list -> int
end;
```

Module Constrained by Signature in ML

```
structure LookupDict: DictLookupSig = Dictionary;  
val l = LookupDict.create;           (* not allowed *)  
val d = Dictionary.create;  
val e = Dictionary.insert("X", 7, d);  
val v = LookupDict.lookup("X", e);  
val k = LookupDict.insert("Y", 8, e); (* not allowed *)
```

Generic Module in ML

```
structure Dictionary =  
struct  
  exception NotFound;  
  
  val create = nil;  
  
  fun insert(c, i, nil) = [(c,i)]  
    | insert(c, i, (cc,ii)::cs) =  
      if c = cc then (c,i)::cs  
      else (cc,ii)::insert(c,i,cs);  
  
  fun lookup(c, nil) = raise NotFound  
    | lookup(c, (cc,ii)::cs) =  
      if c = cc then ii  
      else lookup(c,cs);  
  
end;
```