Data Types

Data Types

- *Expressions* in a programming language represent the *values* of computation
- These values can be *grouped* according to their nature

Data Types

- At the machine level the *bit representations* of values must be manipulated by the *correct operations*
- Usually there are *distinct machine operations* for integer and real arithmetic, bit-wise arithmetic, and character manipulation

Data Types

- A high-level programming language usually provides more *abstract* collections of values
- Most of the time, the programmer prefers to compute more *abstract values*, e.g., *integers with infinite size*
- Sometimes the programmer wants or needs to manipulate the *words* or *bytes* of the underlying machine, e.g., *system programming*

Data Types

- A challenging aspect for programming language design is to find the *right mechanisms* to allow the programmer to *create* and *manipulate* values appropriate to the problem at hand
- This must be accomplished while considering all the competing pressures of language design: *simplicity*, *efficiency*, *generality*, etc.

Static Type Checking

- The introduction of *types* opens up the possibility of *static type-checking* (i.e., at compile time)
- A language is said to be *strongly typed* if all type checking can be done at compile time
- As a result, there are *fewer programming errors*, and the compiler can produce *better code* for the language

Type Completeness

- A language is said to be *type complete* if all the values in the language have *equal status*; in some languages values of certain types are restricted
- In Ada, one can pass records and arrays as parameters, but not functions
- In Pascal, one cannot declare variables of function types

Scalar Types

- A *scalar type* is a type whose elements consist of indivisible entities
- These types are also called *base types*, *primitive types*, or *unstructured types*
- Examples are integer, real, boolean, complex, character, string, atoms, and pointers (address)

Structured Types

- A *structured type* is a compound type
- Examples are arrays, records, tuples, lists, sets, functions, and pointers (to values of a type)
- There are two kinds of structured types: *heterogeneous*, whose elements are of different types, and *homogeneous*, whose elements are of the same type

Structured Types

- Arrays have homogeneous elements, but *dynamic* selectors, called indices, and records have heterogeneous elements, but *static* field selectors
- The case of *homogeneous* elements and *static* selection is just a special case of records
- Several languages without static type checking have data structures of *heterogeneous* elements and *dynamic* selectors

Structured Types

- Examples include *tables* in Icon and *association lists* in LISP
- These data structures associate a value of any type to a *dynamically computable key*
- They abandon *constant time access* to the elements in favor of *dynamically changing size*

Association Lists

- An *association list* is a list of *pairs*

```
(SETQ today '((temp 103) (pressure (120 60))
              (pulse 72)))
```

- The function ASSOC is a function with two arguments: a *key* and an *association list*
- It finds an element of the association list whose first component is equal to the key

```
(ASSOC 'temp today)
    (temp 103)
```

Types in Ada

```
type MONTH_TYPE is (Jan, Feb, Mar, April, May, June,
                    July, Aug, Sep, Oct, Nov, Dec);
type INT_POINTER is access INTEGER;
type COORD is record x: INTEGER; y: FLOAT end;
type TABLE is array (1 .. 5) of INTEGER;
type MATRIX is array (INTEGER range <>,
                      INTEGER range <>) of FLOAT;
```

Dynamic Arrays

- *Dynamic arrays* are arrays whose size can be determined at run time and can be changed at run time (ALGOL 68)
- *Semidynamic* arrays are arrays whose size can be determined at run time but cannot be changed at run time (Ada, C)

An Example

```
A: declare
   u1: constant INTEGER := input (1, 20);
   u2: constant INTEGER := input (1, 50);
   m: MATRIX (1..u1, 1..u2);
begin
   for i in m'RANGE(1) loop
      for j in m'RANGE(2) loop
         --  m(i, j)
      end loop;
   end loop;
end A;
```

An Example

```
subtype POSITIVE is
      INTEGER range 1 .. INTEGER'LAST;
type STRING is array (POSITIVE range <>)
      of CHARACTER;

word: STRING := "queue";
word := "q";                    -- size is still 5
```

Function Types

ML	int * real -> bool	int -> unit
Haskell	(Int, Float) -> Bool	Int -> ( )
ANSI C	int f (int x, float y)	void f (int x)
ALGOL 68	proc (int, real) bool	proc (int) void

Modula-3

```
PROCEDURE(x:INTEGER,y:REAL):BOOLEAN
PROCEDURE(x:INTEGER)
```

Constructors

- It is possible in Ada to *construct* arrays and records all at once, and not piece by piece

```
type Date is
   record
      Day: INTEGER range 1 .. 31;
      Month: MONTH_TYPE;
      Year: INTEGER range 0 .. 4000;
   end record;
A: DATE := (4, July, 1776);
B: DATE := (Day=>4, Month=>July, Year=>1776);
C: DATE := (Month=>July, Day=>4, Year=>1776);
D: DATE := (2+2, July, 3552/2);
```

Type Insecurities

- The purpose of a *compile-time type system* is to *avoid* interpreting the bits in computer memory incorrectly at run time without encoding the type of every object in those bits
- This requires *careful* design of the language
- Look at three Pascal constructs that prevent compile-time type checking: *subranges*, *variant records*, and *function types*

An Example - PL/I

```
POINT:  PROCEDURE OPTION (MAIN);
   DECLARE
      1 NODE BASED (P),
        2 INFO FIXED,
        2 LINK POINTER,
      1 STUDENT BASED (Q),
        2 NAME CHARACTER (30),
        2 GPA FLOAT,
      X POINTER;
   ALLOCATE NODE;
   ALLOCATE STUDENT;
   X = P;
   X ->GPA = 3.75;
END POINT;
```

An Example - Pascal

```
program Main (input, output);
   var  p: ^integer;  q: ^real;  x: ^integer;
begin
   new (p);
   new (q);
   x := p;
   x^ := 3.75;
end.
```

Subranges

```
program Main (input, output);
var
   wide: 1 .. 100;
   narrow: 10 .. 20;
   farout: 150 .. 300;
begin
   narrow := farout;          { Illegal! }
   wide := narrow;            { Legal! }
   narrow := wide;            { Unknown! }
end.
```

Variant Records - Pascal

```
type
   option = (a, b);
   vrt = record case tag: option of
      a: (f1: real);  b: (f2: integer)
   end
var  x: vrt
begin
   x.f1 := 12.65;                {w/o regard to tag}
   x.tag := b;
   if x.f2 = 32 then (* … *)  {w/o regard to tag}
end.
```

Variant Records - Euclid

```
var  x: vrt(a);        {tag cannot change at runtime}
var  y: vrt(b);        {tag cannot change at runtime}
var  z: vrt(any);      {tag may change at runtime}

z := x;
case discriminating w = z on tag of  {test at runtime}
   a => { use w as if it were declared vrt(a) }
   b => { use w as if it were declared vrt(b) }
end case       {static check is possible within case}
```

Variant Records - ML

```
datatype  vrt = f1 of real | f2 of int;

case x of                {test at runtime}
   f1 r => (* … r : real … *)
  | f2 i => (* … i : int … *)
{static check is possible within case}
```

Variant Records - Ada

```
type vrt (US_Address: Boolean := true) is
   record
      Name, Street, City: STRING(1..30);
      case  US_Address is
         when true =>  State: STRING(1..2);
                       Zip: INTEGER range 0..99999;
         when false =>  Country: STRING(1..20);
      end case
   end record
Addr1: vrt(US_Address => true);
Addr2: vrt(US_Address => false);
Addr3: vrt;
```

Variant Records - Ada

```
Addr1.US_Address := false;     {Illegal}
Addr1 := Addr2;                {Illegal}
Addr1 := Addr3;                {runtime check}
Addr3 := Addr2;                {Legal}
Addr1.State := "TX";           {Legal}
Addr2.State := "TX";           {Illegal}
Addr3.State := "TX";           {runtime check}
case Addr3.US_Address is       {test at runtime}
   when true => Addr3.Zip := 76203;
   when false => Addr3.Country := "Canada";
end case       {static check is possible within case}
```

Function Types

```
program Main (input, output);
   procedure Print (x: integer; procedure P);
   begin
      if x=1 then P(2.1) else P(3.2, 9.3);
   end;
   procedure Print1 (x: real);
   begin  writeln('Print1: x = ', x);  end;
   procedure Print2 (x, y: real);
   begin  writeln('Print2: x = ', x, 'y = ', y);  end;
begin
   Print(1, Print1);  Print(2, Print2);
end.
```

Type Equivalence

```
declare
   type BLACK is INTEGER;
   type WHITE is INTEGER;
   B: BLACK;  W: WHITE;  I: INTEGER;
begin
   W := 5;
   B := W;
   I := B;
end;
```

Type Equivalence

- *Name equivalence*: types with the same name
  none of the assignments are legal
- *Structural equivalence*: types with the same structure
  all of the assignments are legal

Name Equivalence

```
Outer: declare
   type BLACK is INTEGER;
   B: BLACK;
   Inner: declare
      type BLACK is INTEGER;
      A: BLACK;
   begin
   B := A;    -- types in distinct scopes are distinct
   end Inner;
end Outer;
```

Name Equivalence

```
declare
   N: INTEGER;
   type INDEX is INTEGER range 1..10;
   I: INDEX;
begin
   N := I;          -- illegal, different types
   I := I + 1;      -- illegal, + is for type INTEGER
end;
```

Name Equivalence

```
declare
   type point      is record first, second: real end record;
   type complex is record first, second: real end record;
   var p: point;
   var z: complex;
begin
   z := p;                    -- illegal, different types
   z.first := p.first;
   z.second := p.second;   -- same effect
end;
```

Name Equivalence

```
type
   T = record a: integer; b: char end;
var
   x, y: array [1..2] of record a: integer; b: char end;
                          { anonymous types }
   z:    array [1..2] of T;
   u, v: array [1..2] of record a: integer; b: char end;

  { x and y are equivalent, u and v are equivalent,
    but no others are equivalent
    -- declaration equivalence }
```

Structural Equivalence

```
type T1 is record
   X: INTEGER;
   N: access T1;
end record;

type T2 is record
   X: INTEGER;
   N: access T2;
end record;
```

```
type T3 is record
   X: INTEGER;
   N: access record
      X: INTEGER;
      N: access T3;
   end record;
end record;
```

Structural Equivalence

```
type T4 is record
   X: INTEGER;
   N: access T2;
end record;

type T5 is record
   N: access T5;
   X: INTEGER;
end record;
```

```
type T6 is record
   Y: INTEGER;
   N: access T6;
end record;
```

Structural Equivalence

- Testing structural equivalence is not easy
- C uses structural equivalence, but when this becomes difficult, i.e., with **struct**, C uses name equivalence
- C++ uses name equivalence, but **typedef** does not introduce a new type name, just a synonym
- Name equivalence is sometimes favored because it is simpler and safer

Name Equivalence in Ada

- Ada uses name equivalence
- The type of an object always be a *name* or a *constrained array*; this avoids anonymous types
- Use *subtypes* to avoids introducing new range types
- Use *derived types* to introduce isomorphic new types

Subtypes

- A *subtype* declaration in Ada does *not* introduce a new type at all
- The purpose of an Ada subtype is the *runtime* guarantee that the program will notice if the value is out of range

Subtypes

```
declare
   N: INTEGER;
   subtype INDEX is INTEGER range 1..10;
   I: INDEX;
begin
   N := I;
   I := N;      -- possible runtime error, but type correct
   I := I + 1;
end;
```

Derived Types

- A *derived type* declaration in Ada introduces a *new* type
- Derived types create an *isomorphic copy* of another type--same operators and constants, i.e., these operators and constants are *overloaded*
- *Implicit coercions* are not allowed in Ada

Derived Types

```
declare
   type Centimeters is new INTEGER;
   type Meters is new INTEGER;
   Length1: Centimeters;
   Length2: Meters;
begin
   Length1 := Length1 + Length2;        -- illegal
   Length1 := 5 + 2 * Length1;  -- 2 & 5 overloaded
                                -- + & * overloaded
end;
```

Derived Types

```
function "+" (cm: Centimeters, m: Meters)
        return Centimeters is
begin
   return (cm + Centimeters(100 * INTEGER(m)));
end "+";

function "+" (m: Meters, cm: Centimeters)
        return Centimeters is
begin
   return (cm + Centimeters(100 * INTEGER(m)));
end "+";
```

Overloading in C++

```
int  f(int i1, int i2) { return 8; }                  // f1
double  f(int i1, double d2) { return 3.5; }      // f2

int x = f(1, 2);                 // f1
double y = f(1, 2.0);          // f2
int z = f(1.3, 2);              // f1, an implicit coercion to int
```

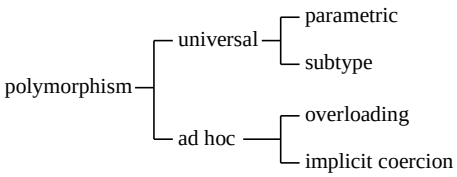
Polymorphism

- A function is said to be *polymorphic* if it can assume *more than one type*

```
f(x) = x
function f(x: int): int
function f(x: boolean): boolean
function f(x: real): real

fun length(x) = if null(x) then 0
                else 1 + length(tl x)
```

Polymorphism



Polymorphism

- With *ad hoc polymorphism* we have *different code* for the different manifestations of a function name
- With *universal polymorphism* the appropriate arguments to a function must be from a uniform collection of data structure; the *same code* works, regardless of the type of the arguments

Ad Hoc Polymorphism

- *Overloading* occurs when the same name is used for different functions, like "+" in Ada
- The function "+" is used for integer and real numbers, and for user-defined types
- *Implicit coercion* occurs when values of one type are routinely converted to another so that a value will have a type appropriate for context
- Many languages permit mixed-mode expressions like 822.34 + 4

Overloading

- The signatures of two overloaded functions must be different
- Functions differing in only the return type may not be overloaded

Implicit Coercion

- Coercions are called *widenings* or *promotions* if there is a natural mapping that loses no information, e.g., from an integer to a real number
- Coercions are called *narrowings* if there is more than one reasonable mapping or mappings lose information, e.g., from a real to an integer number

An Example

```
class  tiny {
   char v;
   int assign(int i)
     {return (v=(i&~63) ? (cerr << “range error”, 0 : i);}
public:
   tiny(int i) { void assign(i); }
   tiny(tiny &t) { v = t.v }
   int operator=(tiny &t) { return v = t.v; }
   int operator=(int i) { return assign(i); }
   operator int( ) { return v; }
};
```

An Example

```
int main( )
{
   tiny c1 = 2;
   tiny c2 = 62;
   tiny c3 = c2 - c1;     // c3 = 60
   tiny c4 = c3;          // range check avoid
   int i = c1 + c2;       // c1, c2 converted to int, i = 64
   c1 = c2 + 2 * c1;      // c1, c2 converted to int, error
   c2 = c1 - i;           // c1converted to int, error
   c3 = c2;               // range check avoid
}
```

Universal Polymorphism

- Universal polymorphism can be divided into *parametric polymorphism* and *inclusion polymorphism*
- These two differ in what *facet* of data structures they take advantage of
- In both cases the part of the data structure that is *not relevant* to a function is permitted to be filled by *different structures with different types*

Parametric Polymorphism

- *Parametric polymorphism* takes advantage of the fact that data structures are built out of other data structures. Any value is built from many *layers* of constructors. Not all the layers may be relevant to a particular *function*
- With parametric polymorphism functions may have *type parameters* as well as *value parameter*

An Example

```
generic
   type Type is private;
function Id (X: in Type) return Type is
begin
   return X;
end;

function IntId is new Id (INTEGER);
function FloatId is new Id (FLOAT);
```

Generics

- The Ada mechanism for generics does not generalize well

```
type GenericList is record
      Element: Type;
      Next: access GenericList;
end record;
```

We can put it neither inside nor outside the function

Type System of mini-ML

- The BNF definition for type system

  <*type*> ::= <*type variable*>
  | int
  | bool
  | <*type*> -> <*type*>

- The BNF definition for expressions

  <*expr*> ::= <*identifier*>
  | (fn <*identifier*> => <*expr*>)
  | <*expr*> <*expr*>
  | if <*expr*> then <*expr*> else <*expr*>

Type System of mini-ML

- The *strings* derived by the type system are called *judgments*
- A *typing judgment* is a triple of the form

  $A \vdash e : \tau$

  where A is a type assignment, e is an expression, and τ is a type
- A *type assignment* is a mapping from identifiers to types defining type bindings

Type System of mini-ML

- Conditional statement

$$\frac{A \vdash e_1 : \text{bool} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- Identifier

$$\frac{}{A \vdash x : A(x)} \quad \text{if } x \in \text{Dom}(A)$$

Type System of mini-ML

- Function definition

$$\frac{A[x \alpha \tau_1] \vdash e_1 : \tau_2}{A \vdash (\text{fn } x \Rightarrow e_1) : \tau_1 \rightarrow \tau_2}$$

- Function application

$$\frac{A \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad A \vdash e_2 : \tau_1}{A \vdash e_1\ e_2 : \tau_2}$$

An Example

$\varnothing$ |– (fn x => x) : int -> int is derivable

$$\frac{\varnothing[x\ \alpha\ \text{int}\ ] \vdash x : \text{int}}{\varnothing \vdash (\text{fn}\ x \Rightarrow x) : \text{int} \rightarrow \text{int}}$$

A |– (fn x => x x) : τ is not derivable

An Example

A_0 = {<true, bool>, <0, int>, <1, int>}
A_1 = {<x, 'a>, <true, bool>, <0, int>, <1, int>}

$$\frac{\dfrac{\dfrac{}{A_1 \vdash \text{true} : \text{bool}} \quad \dfrac{}{A_1 \vdash 0 : \text{int}} \quad \dfrac{}{A_1 \vdash 1 : \text{int}}}{A_1 \vdash (\text{if true then 0 else 1}) : \text{int}}}{A_0 \vdash (\text{fn}\ x \Rightarrow \text{if true then 0 else 1}) : \text{'a} \rightarrow \text{int}}$$

Polymorphic Expressions

- An expression is said to be *polymorphic* if its type contains *type variables*; otherwise, it is said to be *monomorphic*
- A type variable stands for *any* type; each type is *an instance* of a type variable
- The goal of *type reconstruction* is to determine the *most general type* of expressions by their use

Type Checking

- Type checking can be viewed as proving theorems in the deductive system defined by the four preceding rules
- There is an algorithm or theorem prover, called TypeOf, for computing the type of expressions

Type Checking

- Theorem (correctness): If TypeOf(A, e) succeeds with τ, then $A \vdash e : \tau$ is derivable from the typing rules
- Theorem (completeness): If $A \vdash e : \sigma$ is derivable from the typing rules, then TypeOf(A, e) succeeds with τ and σ is an instance of τ

Let Construct in ML

- We extend mini-ML to include a *let* construct
  let <*identifier*> = <*expr*>$_1$ in <*expr*>$_2$ end
- The let construct is *computationally* equivalent to function application
  (fn <*identifier*> => <*expr*>$_2$) (<*expr*>$_1$)
- The let construct is *not* equivalent to function application as far as *type checking* is concerned

An Example

- The ML type system successfully finds the type int * bool for the let expression
  let f = (fn x => x) in (f 2, f true) end
  because the let construct permits the type variable 'a in 'a->'a to be *instantiated differently* for each application

An Example

- The ML type system fails to find the type for the expression
  (fn f => (f 2, f true)) (fn x => x)
  because we must find *one* type that works for both applications

Inclusion Polymorphism

- *Inclusion polymorphism* takes advantage of the fact that not all fields of records may be relevant to a function
- Inclusion polymorphism introduces the notion of *subtype* and *type hierarchy*

Type System of Amber

- Record types are written in curly braces
  {k: Int, l: String}
- Variant types are written in square brackets
  [k: Int, l: String, m: Bool]
- Function types are written $dom \rightarrow ran$ where *dom* is the type of the domain and *ran* is the type of the range
  {k: Int, l: String} $\rightarrow$ {m: Int}

Type System of Amber

- Amber types is organized in a hierarchy based on the *structure* of each type or the *subtype* relation
- The purpose of the subtype relation is to define a *relaxed form of type compatibility*

Subtype

- We say that τ is a *subtype* of σ, written $\tau \leq \sigma$, if τ is equal to σ or one of the following three cases hold
- A record r_1 is a *subtype* of r_2 , if every field in r_2 is present in r_1 and for the fields common to both records the type of r_1's field is a subtype of r_2's field. Or,
  $\{l_i: \tau_i\} \leq \{m_j : \sigma_j\}$ if for all l_i and m_j
  $m_j \in \{l_i\}$ & $(l_i = m_j \Rightarrow \tau_i \leq \sigma_j)$

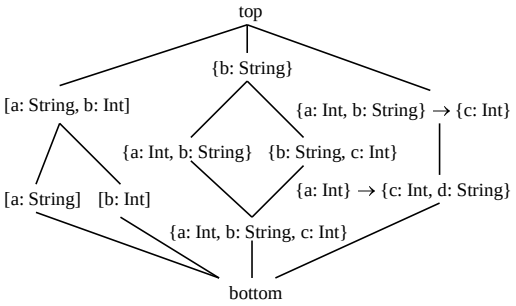
Subtype

- The subtype rule for *variant records*:
  $[l_i: \tau_I] \le [m_j : \sigma_j]$ if for all l_i and m_j
  $l_i \in \{m_j\}$ & $(l_i = m_j \Rightarrow \tau_i \le \sigma_j)$
- The subtype rule for *function types*:
  $\tau_1 \to \sigma_1 \le \tau_2 \to \sigma_2$ if
  $\sigma_1 \le \sigma_2$ & $\tau_2 \le \tau_1$

Type Hierarchy

- If we add two fictitious types, one greater than all others, called *top*, and one less than all others, called *bottom*, then the subtype relation forms a *lattice*
- *Inclusion polymorphism* means that a value with type τ can also be considered as belonging to any type higher than τ in the lattice

Type Hierarchy



Object-Oriented Languages

- It is possible to capture aspects of *object-oriented languages* using inclusion polymorphism
- *Objects* are records; *methods* are fields with function values
- Subtype relation is established using *inheritance* mechanism

An Example

```
TYPE
   O1 = OBJECT a: INTEGER METHODS m( ); END;
   O2 = O1 OBJECT b: INTEGER END;
PROCEDURE P (self: O1) = BEGIN … self.a … END P;
PROCEDURE Q (self: O2) = BEGIN … self.a …self.b ... END P;
TYPE
   T1 = O1 OBJECT OVERRIDES m := P END;
   T2 = O2 OBJECT OVERRIDES m := Q END;
   T3 = O2 OBJECT OVERRIDES m := P END;
   T4 = O1 OBJECT OVERRIDES m := Q END;     (* error *)
```

An Example

```
VAR
   v0 := NEW (O1);            (* create an object of O1 *)
   v1 := NEW (O1, m := P);    (* dynamically override m with P *)
   v1 := NEW (T1);            (* same effect as previous one *)
   v2 := NEW (O2, m := Q);    (* dynamically override m with P *)
   v2 := NEW (T2);            (* same effect as previous one *)
   v3 := NEW (O2, m := P);    (* dynamically override m with P *)
   v3 := NEW (T3);            (* same effect as previous one *)
   v4 := NEW (O1, m := Q);    (* compile-time error *)
BEGIN
   v1.m( );
END;
```