Lesson 24

• Composite data types (cont’d)
Summary

• Data Types in programming languages
• Type system, Type safety, Type checking
  – Equivalence, compatibility and coercion
• Primitive and composite types
  – Discrete and scalar types
  – Tuples and records
  – Arrays
  – Unions
  – Pointers
  – Recursive types
A brief overview of composite types

• We review type constructors in several languages corresponding to the following mathematical concepts:
  – Cartesian products (records and tuples)
  – mappings (arrays)
  – disjoint unions (algebraic data types, unions)
  – recursive types (lists, trees, etc.)
Mappings

• We write $m : S \rightarrow T$ to state that $m$ is a **mapping** from set $S$ to set $T$. In other words, $m$ maps every value in $S$ to some value in $T$.

• If $m$ maps value $x$ to value $y$, we write $y = m(x)$. The value $y$ is called the **image** of $x$ under $m$.

• Some of the mappings in $\{u, v\} \rightarrow \{a, b, c\}$:

  \[
  m_1 = \{u \rightarrow a, \ \ v \rightarrow c\} \\
  m_2 = \{u \rightarrow c, \ \ v \rightarrow c\} \\
  m_3 = \{u \rightarrow c, \ \ v \rightarrow b\}
  \]

  image of $u$ is $c$, image of $v$ is $b$
Arrays (1)

- **Arrays** (found in all imperative and OO PLs) can be understood as mappings.
- If the array’s elements are of type $T$ (*base type*) and its index values are of type $S$, the array’s type is $S \rightarrow T$.
- An array’s **length** is the number of components, $\#S$.
- Basic operations on arrays:
  - **construction** of an array from its components
  - **indexing** – using a *computed* index value to select a component.

so we *can* select the $i$th component
Arrays (2)

- An array of type $S \rightarrow T$ is a finite mapping.
- Here $S$ is nearly always a finite range of consecutive values $\{l, l+1, \ldots, u\}$. This is called the array’s index range.

In C and Java, the index range must be $\{0, 1, \ldots, n-1\}$. In Pascal and Ada, the index range may be any scalar (sub)type other than real/float.

- We can generalise to $n$-dimensional arrays. If an array has index ranges of types $S_1, \ldots, S_n$, the array’s type is $S_1 \times \ldots \times S_n \rightarrow T$. 
When is the index range known?

• A **static array** is an array variable whose index range is fixed by the program code.

• A **dynamic array** is an array variable whose index range is fixed at the time when the array variable is created.
  – In Ada, the definition of an array type must fix the index *type*, but need not fix the index *range*. Only when an array variable is created must its index range be fixed.
  – Arrays as formal parameters of subroutines are often dynamic (eg. *conformant arrays* in Pascal)

• A **flexible** (or **fully dynamic**) **array** is an array variable whose index range is not fixed at all, but may change whenever a new array value is assigned.
Example: C static arrays

- Array variable declarations:
  ```c
  float v1[] = {2.0, 3.0, 5.0, 7.0};  // index range is {0, ..., 3}
  float v2[10];  // index range is {0, ..., 9}
  ```

- Function:
  ```c
  void print_vector (float v[], int n) {
  // Print the array v[0], ..., v[n-1] in the form “[... [...]]”.
  int i;
  printf("[%f", v[0]);
  for (i = 1; i < n; i++)
    printf(" %f", v[i]);
  printf("]");
  }

  ...
  print_vector(v1, 4);  print_vector(v2, 10);
  ```

A C array doesn’t know its own length!
Example: Ada dynamic arrays

• Array type and variable declarations:

```ada
type Vector is
  array (Integer range <>) of Float;

v1: Vector(1 .. 4) := (1.0, 0.5, 5.0, 3.5);
v2: Vector(0 .. m) := (0 .. m => 0.0);
```

• Procedure:

```ada
procedure print_vector (v: in Vector) is
  -- Print the array v in the form “[... [...]]”.
begin
  put('[');  put(v(v'first));
  for i in v'first + 1 .. v'last loop
    put(' ');  put(v(i));
  end loop;
  put(']');
end;
... print_vector(v1);  print_vector(v2);
```
Example: Java flexible arrays

- Array variable declarations:
  ```java
  float[] v1 = {1.0, 0.5, 5.0, 3.5};  // index range is {0, ..., 3}
  float[] v2 = {0.0, 0.0, 0.0};      // index range is {0, ..., 3}
  ...
  v1 = v2;                            // v1’s index range is now {0, ..., 2}
  ```

- Method:
  ```java
  static void printVector (float[] v) {
    // Print the array v in the form "[... ... ...]"
    System.out.print("[" + v[0]);
    for (int i = 1; i < v.length; i++)
      System.out.print(" " + v[i]);
    System.out.print("]");
  }
  ...
  printVector(v1);  printVector(v2);
  ```
  Enhanced for:
  ```java
  for (float f : v)
    System.out.print(" " + f)
  ```
Array allocation

- **static array, global lifetime** — If a static array can exist throughout the execution of the program, then the compiler can allocate space for it in *static global memory*

- **static array, local lifetime** — If a static array should not exist throughout the execution of the program, then space can be allocated *in the subroutine’s stack frame* at runtime.

- **dynamic array, local lifetime** — If the index range is known at runtime, the array can still be allocated *in the stack*, but in a variable size area

- **fully dynamic** — If the index range can be modified at runtime it has to be allocated *in the heap*
Allocation of dynamic arrays on stack

-- Ada:
procedure foo (size : integer) is
  M : array (1..size, 1..size) of real;
  ...
begin
  ...
end foo;

// C99:
void foo(int size) {
  double M[size][size];
  ...
}

Figure 7.6
Elaboration-time allocation of arrays in Ada or C99.
Here M is a square two-dimensional array whose bounds are determined by a parameter passed to foo at run time. The compiler arranges for a pointer to M and a dope vector to reside at static offsets from the frame pointer. M cannot be placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays or records are easily accommodated.

Several languages, including Snobol, Icon, and all the scripting languages, allow strings—arrays of characters—to change size after elaboration time. Java and C# provide a similar capability (with a similar implementation), but describe the semantics differently: string variables in these languages are references to immutable string objects:

String s = "short"; // This is Java; use lowercase 'string' in C#
...

s = s + " but sweet"; // + is the concatenation operator

Here the declaration String s introduces a string variable, which we initialize with a reference to the constant string "short". In the subsequent assignment, + creates a new string containing the concatenation of the old s and the constant " but sweet"; s is then set to refer to this new string, rather than the old.

12
Arrays: memory layout

• Contiguous elements
  – column major - only in Fortran
  – row major
    • used by everybody else

• Row pointers
  – an option in C, the rule in Java
  – allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  – avoids multiplication
  – nice for matrices whose rows are of different lengths
    • e.g. an array of strings
  – requires extra space for the pointers
Arrays’ memory layout in C

```c
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...'
    days[2][3] == 's';  /* in Tuesday */
```

```c
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...'
    days[2][3] == 's';  /* in Tuesday */
```

- Address computation varies a lot
- With contiguous allocation part of the computation can be done statically
Strings

• A **string** is a sequence of 0 or more characters.
• Usually ad-hoc syntax is supported
• Some PLs (ML, Python) treat strings as *primitive*.
• Haskell treats strings as *lists* of characters. Strings are thus equipped with general list operations (length, head selection, tail selection, concatenation, ...).
• Ada treats strings as *arrays* of characters. Strings are thus equipped with general array operations (length, indexing, slicing, concatenation, ...).
• Java treats strings as *objects*, of class `String`.
Disjoint Union (1)

• In a **disjoint union**, a value is chosen from one of several different types.

• Let $S + T$ stand for a set of disjoint-union values, each of which consists of a *tag* together with a *variant* chosen from either type $S$ or type $T$. The tag indicates the type of the variant:

\[
S + T = \{ \text{left } x \mid x \in S \} \cup \{ \text{right } y \mid y \in T \}
\]

– *left $x$* is a value with tag *left* and variant $x$ chosen from $S$

– *right $x$* is a value with tag *right* and variant $y$ chosen from $T$.

• We write **left $S + right T$** (instead of $S + T$) when we want to make the tags explicit.
Disjoint Union (2)

- Basic operations on disjoint-union values in $S + T$:
  - **construction** of a disjoint-union value from its tag and variant
  - **tag test**, to determine whether the variant was chosen from $S$ or $T$
  - **projection**, to recover either the variant in $S$ or the variant in $T$.

- **Algebraic data types** (Haskell), **discriminated records** (Ada), **unions** (C) and **objects** (Java) can all be understood in terms of disjoint unions.

- We can generalise to multiple variants: $S_1 + S_2 + \ldots + S_n$. 
Example: Haskell/ML algebraic data types

• Type declaration:

```haskell
data Number = Exact Int | Inexact Float
```

Each Number value consists of a tag, together with either an Integer variant (if the tag is `Exact`) or a Float variant (if the tag is `Inexact`).

• Application code:

```haskell
pi = Inexact 3.1416
rounded :: Number -> Integer
rounded num =
  case num of
    Exact i -> i
    Inexact r -> round r
```
Variant records (unions)

- **Origin:** Fortran I *equivalence statement:* variables should share the same memory location
- **C’s *union* types**
- **Motivations:**
  - Saving space
  - Need of different access to the same memory locations for system programming
  - Alternative configurations of a data type

```fortran
Fortran I -- equivalence statement
integer i
real r
logical b
equivalence (i, r, b)
```

```c
C -- union
union {
    int i;
    double d;
    _Bool b;
};
```
Variant records (unions) (2)

• In Ada, Pascal, unions are *discriminated* by a tag, called *discriminant*

• Integrated with records in Pascal/Ada, not in C

ADA – discriminated variant

```ada
type Form is
  (pointy, circular, rectangular);

type Figure (f: Form := pointy) is record
  x, y: Float;
  case f is
    when pointy => null;
    when circular => r: Float;
    when rectangular => w, h: Float;
  end case;
end record;
```
Using discriminated records in Ada

- Application code:
  
  ```ada
  box: Figure :=
  (rectangular, 1.5, 2.0, 3.0, 4.0);

  function area (fig: Figure) return Float is
  begin
    case fig.f is
      when pointy =>
        return 0.0;
      when circular =>
        return 3.1416 * fig.r**2;
      when rectangular =>
        return fig.w * fig.h;
    end case;
  end;
  ```

  discriminated-record construction
  discriminated-record construction
  tag test
  tag test
  projection
  projection
(Lack of) Safety in variant records

- Only Ada has strict rules for assignment: tag and variant have to be changed *together*.
- For *nondiscriminated unions* (Fortran, C) no runtime check: responsibility of the programmer.
- In Pascal the tag field can be modified independently of the variant. Even worse: the tag field is optional.
- Unions not included in Modula 3, Java, and recent OO languages: replaced by classes + inheritance.
Example: Java objects (1)

• Type declarations:

```java
class Point {
    private float x, y;
    ...
    // methods
}

class Circle extends Point {
    private float r;
    ...
    // methods
}

class Rectangle extends Point {
    private float w, h;
    ...
    // methods
}
```

The class `Circle` and `Rectangle` inherit `x` and `y` from the `Point` class.

```java
class Point { 
        private float x, y;
        ...
    // methods
    }

class Circle extends Point { 
        private float r;            inherits x and y from Point
        ...
    // methods
    }

class Rectangle extends Point { 
        private float w, h;          inherits x and y from Point
        ...
    // methods
    }
```
Example: Java objects (2)

• Methods:

```java
class Point {
    ...
    public float area() {
        return 0.0;
    }
}
class Circle extends Point {
    ...
    public float area() {  // overrides Point’s area() method
        return 3.1416 * r * r;
    }
}
class Rectangle extends Point {
    ...
    public float area() {  // overrides Point’s area() method
        return w * h;
    }
}
```
Example: Java objects (3)

- Application code:

```java
Rectangle box =
    new Rectangle(1.5, 2.0, 3.0, 4.0);

float a1 = box.area();
Point it = ...;
float a2 = it.area();
```

- it can refer to a Point, Circle, or Rectangle object calls the appropriate area() method
Value model vs. reference model

- What happens when a composite value is assigned to a variable of the same type?
- **Value model** (aka *copy semantics*): all components of the composite value are copied into the corresponding components of the composite variable.
- **Reference model**: the composite variable is made to contain a reference to the composite value.
- **Note**: this makes no difference for basic or immutable types.

- C and Ada adopt copy semantics.
- Java adopts value model for primitive values, reference model for objects.
- Functional languages usually adopt the reference model
Example: Ada value model (1)

- Declarations:
  ```
  type Date is
  record
    y: Year_Number;
    m: Month;
    d: Day_Number;
  end record;
  
  dateA: Date := (2004, jan, 1);
  dateB: Date;
  ```

- Effect of copy semantics:

  ```
  dateB := dateA;
  dateB.y := 2005;
  ```
Example: Java reference model (1)

- Declarations:

```java
class Date {
    int y, m, d;
    public Date (int y, int m, int d)
    {
        ...
    }
}
Date dateR = new Date(2004, 1, 1);
Date dateS = new Date(2004, 12, 25);
```

- Effect of reference semantics:

```java
dateS = dateR;
dateR.y = 2005;
```
Ada reference model with pointers (2)

• We can achieve the *effect* of reference model in Ada by using explicit *pointers*:

```ada
    type Date_Pointer is access Date;
    Date_Pointer dateP = new Date;
    Date_Pointer dateQ = new Date;

    ...
    dateP.all := dateA;
    dateQ := dateP;
```
Java value model with cloning (2)

- We can achieve the *effect* of copy semantics in Java by cloning:

```java
Date dateR = new Date(2004, 4, 1);
dateT = dateR.clone();
```
Pointers

• Thus in a language adopting the *value model*, the *reference model* can be simulated with the use of pointers.
• A **pointer** (value) is a reference to a particular variable.
• A pointer’s **referent** is the variable to which it refers.
• A **null pointer** is a special pointer value that has no referent.
• A pointer is essentially the address of its referent in the store, but it also has a *type*. The type of a pointer allows us to infer the type of its referent.
• Pointers mainly serve two purposes:
  – efficient (sometimes intuitive) access to elaborated objects (as in C)
  – dynamic creation of linked data structures, in conjunction with a heap storage manager
Dangling pointers

• A **dangling pointer** is a pointer to a variable that has been destroyed.

• Dangling pointers arise from the following situations:
  – where a pointer to a heap variable still exists after the heap variable is destroyed by a deallocator
  – where a pointer to a local variable still exists at exit from the block in which the local variable was declared.

• A deallocator immediately destroys a heap variable. All existing pointers to that heap variable become dangling pointers.

• Thus deallocators are inherently unsafe.
Dangling pointers in languages

• C is highly unsafe:
  - After a heap variable is destroyed, pointers to it might still exist.
  - At exit from a block, pointers to its local variables might still exist (e.g., stored in global variables).

• Ada and Pascal are safer:
  - After a heap variable is destroyed, pointers to it might still exist.
  - But pointers to local variables may not be stored in global variables.

• Java is very safe:
  - It has no deallocator.
  - Pointers to local variables cannot be obtained.

• Functional languages are even safer:
  - they don’t have pointers
Example: C dangling pointers

- Consider this C code:

```c
struct Date { int y, m, d;};
struct Date* dateP, dateQ;
dateP = (struct Date*)malloc(sizeof (struct Date));
dateP->y = 2004; dateP->m = 1; dateP->d = 1;
dateQ = dateP;
free(dateQ);
printf("%d", dateP->y);
dateP->y = 2005;
```

allocates a new heap variable

makes `dateQ` point to the same heap variable as `dateP`

deadallocates that heap variable (`dateP` and `dateQ` are now dangling pointers)
Techniques to avoid dangling pointers

- **Tombstones**
  - A pointer variable refers to a *tombstone* that in turn refers to an object
  - If the object is destroyed, the tombstone is marked as “expired”
Locks and Keys

- Heap objects are associated with an integer (lock) initialized when created.
- A valid pointer contains a key that matches the lock on the object in the heap.
- Every access checks that they match.
- A dangling reference is unlikely to match.
Pointers and arrays in C

• In C, an array variable is a pointer to its first element

  ```c
  int *a == int a[]
  int **a == int *a[]
  ```

• BUT equivalences don't always hold
  – Specifically, a declaration allocates an array if it specifies a size for the first dimension, otherwise it allocates a pointer

  ```c
  int **a, int *a[]  // pointer to pointer to int
  int *a[n],  // n-element array of row pointers
  int a[n][m],  // 2-d array
  ```

• Pointer arithmetics: operations on pointers are scaled by the base type size. All these expressions denote the third element of `a`:

  ```c
  a[2]     (a+2)[0]     (a+1)[1]     2[a]     0[a+2]
  ```
C pointers and recursive types

• C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

  int *a[n], n-element array of pointers to integer
  int (*a)[n], pointer to n-element array of integers

• Compiler has to be able to tell the size of the things to which you point
  – So the following aren't valid:

    int a[][] bad
    int (*a)[] bad
Recursive types: Lists

• A **recursive type** is one defined in terms of itself, like lists and trees.

• A **list** is a sequence of 0 or more component values.

• The **length** of a list is its number of components. The **empty list** has no components.

• A non-empty list consists of a **head** (its first component) and a **tail** (all but its first component).

• A list is **homogeneous** if all its components are of the same type. Otherwise it is **heterogeneous**.
List operations

• Typical list operations:
  – length
  – emptiness test
  – head selection
  – tail selection
  – concatenation.
Example: Haskell lists

- Type declaration for integer-lists:
  ```haskell
data IntList = Nil | Cons Int IntList
```

  - Some IntList constructions:
    - Nil
    - Cons 2 (Cons 3 (Cons 5 (Cons 7 Nil)))

  - Actually, Haskell has built-in list types:
    - [Int] [String] [[Int]]

  - Some list constructions:
    - [] [2,3,5,7] ["cat","dog"] [[1],[2,3]]

  - Built-in operator for cons
    - 
Example: Ada lists

• Type declarations for integer-lists:

  ```adalog
  type IntNode;
type IntList is access IntNode;
type IntNode is record
    head: Integer;
tail: IntList;
  end record;
  ```

- An IntList construction:

  ```adalog
  new IntNode'(2,
    new IntNode'(3,
      new IntNode'(5,
        new IntNode'(7, null)))
  ```

mutually recursive
Example: Java lists

• Class declarations for generic lists:

```java
class List<E> {
    public E head;
    public List<E> tail;
    public List<E> (E el, List<E> t) {
        head = h;  tail = t;
    }
}
```

- A list construction:

```java
List<Integer> list =
    new List<Integer>(2,
        new List<Integer>(3,
            new List<Integer>(5, null)));
```