

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Composite Types

CS 208 - Bucknell University

Spring 2012

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Outline

- 1 Enum, Subrange: $S \subseteq T$
- 2 Cartesian Product: $S \times T$
- 3 Mapping, Arrays: $S \rightarrow T$
- 4 Disjoint Union: $S + T$
- 5 Recursive Types: $S = f(S)$

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Enum, Subrange

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Enum, Subrange

Motivation: Sometimes, primitive types are too large for a particular problem. **Enum** or **Subrange** limit the use of a type to certain values. The aim is to refine the modelling of a problem.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Enum, Subrange

Motivation: Sometimes, primitive types are too large for a particular problem. **Enum** or **Subrange** limit the use of a type to certain values. The aim is to refine the modelling of a problem.

- enum in C:

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

Enum, Subrange

Motivation: Sometimes, primitive types are too large for a particular problem. **Enum** or **Subrange** limit the use of a type to certain values. The aim is to refine the modelling of a problem.

- enum in C:

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

- subrange in Ada:

```
type Day_Number is range 1 .. 31;  
type Population is range 0 .. 1e10;
```

Enum, Subrange

Motivation: Sometimes, primitive types are too large for a particular problem. **Enum** or **Subrange** limit the use of a type to certain values. The aim is to refine the modelling of a problem.

- enum in C:

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

- subrange in Ada:

```
type Day_Number is range 1 .. 31;  
type Population is range 0 .. 1e10;
```

Type representation:

Color \subseteq *int*

Day_Number \subseteq *Int*

Population \subseteq *Float*

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Cartesian Product

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Cartesian Product

Definition: Consider two types of values S and T .
The **cartesian product** of S and T is:

$$S \times T = \{ (x, y) \mid x \in S \text{ and } y \in T \}$$

There are two basic operations:

- **construction** of a pair from two component values
- **selection** of the first or the second component

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: Records in Ada

```
type Month is ( jan, feb, mar, apr, may, jun,
                jul, aug, sep, oct, nov, dec );
type Day_Number is range 1 .. 31;
type Date is
    record
        m: Month;
        d: Day_Number;
    end record;
```

Example: Records in Ada

```
type Month is ( jan, feb, mar, apr, may, jun,
                jul, aug, sep, oct, nov, dec );
type Day_Number is range 1 .. 31;
type Date is
    record
        m: Month;
        d: Day_Number;
    end record;
```

- Construction of a record:

```
someday: Date := (m => jan, d => 1);
```

Example: Records in Ada

```
type Month is ( jan, feb, mar, apr, may, jun,
                jul, aug, sep, oct, nov, dec );
type Day_Number is range 1 .. 31;
type Date is
    record
        m: Month;
        d: Day_Number;
    end record;
```

- Construction of a record:

```
someday: Date := (m => jan, d => 1);
```

- Selection of the first or second component of the record:

```
put(someday.m + 1);  put("/");  put(someday.d);
someday.d := 29;    someday.m := feb;
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: Struct in C++ (also in C)

```
enum Month { jan, feb, mar, apr, may, jun, jul,
            jul, aug, sep, oct, nov, dec };
struct Date{
    Month m;
    byte d;
}
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: Struct in C++ (also in C)

```
enum Month { jan, feb, mar, apr, may, jun, jul,
            jul, aug, sep, oct, nov, dec };
struct Date{
    Month m;
    byte d;
}
```

- Construction of a struct:

```
struct Date someday = { jan, 1 };
```

Example: Struct in C++ (also in C)

```
enum Month { jan, feb, mar, apr, may, jun, jul,
            jul, aug, sep, oct, nov, dec };
struct Date{
    Month m;
    byte d;
}
```

- Construction of a struct:

```
struct Date someday = { jan, 1 };
```

- Selection of the first or second component of the record:

```
printf("%d/%d", someday.m + 1, someday.d);
someday.d = 29;      someday.m = feb;
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Mapping, Arrays

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Mappings, Arrays

Definition: Consider two types S and T .
A **mapping** m associates to a value x in S a value y in T such that $y = m(x)$. Notice the similarity with functions, we model it:

$$S \rightarrow T = \{ m \mid \text{if } x \in S \text{ then } m(x) \in T \}$$

We call the elements in S the **indexes**.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Mappings, Arrays

Definition: Consider two types S and T .

A **mapping** m associates to a value x in S a value y in T such that $y = m(x)$. Notice the similarity with functions, we model it:

$$S \rightarrow T = \{ m \mid \text{if } x \in S \text{ then } m(x) \in T \}$$

We call the elements in S the **indexes**.

An **array** is a mapping for which the set of indexes is a subset of S .

Mappings, Arrays

Definition: Consider two types S and T .
A **mapping** m associates to a value x in S a value y in T such that $y = m(x)$. Notice the similarity with functions, we model it:

$$S \rightarrow T = \{ m \mid \text{if } x \in S \text{ then } m(x) \in T \}$$

We call the elements in S the **indexes**.

An **array** is a mapping for which the set of indexes is a subset of S .
The basic operations are:

- Construction of an array given its components.
- Selection of a particular component given its index.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: Arrays in C++ & Ada

Arrays in C++:

```
bool p[] = {true, false, false};  
p[1] = !p[1];
```

Example: Arrays in C++ & Ada

Arrays in C++:

```
bool p[] = {true, false, false};  
p[1] = !p[1];
```

Arrays in Ada:

```
type Color is (red, blue, green);  
type Pixel is array (Color) of Boolean;  
p: Pixel := (red    => true ,  
             green => false ,  
             blue   => false );  
p(c) := not p(c);
```

Arrays more expressive

In some programming languages arrays can be indexed not only by **integers** but also by **string** (sometimes **float**). In order to index properly, the type needs to have basic operations of comparison, equality, lesser, greater:

(*int*, ==, <, >)

(*string*, ==, lexicographic order)

Arrays more expressive

In some programming languages arrays can be indexed not only by **integers** but also by **string** (sometimes **float**). In order to index properly, the type needs to have basic operations of comparison, equality, lesser, greater:

$(int, ==, <, >)$

$(string, ==, \text{lexicographic order})$

In some programming languages an array can be indexed by multiple types.

Example in PHP:

```
arr = array( 6 => 5,  
            13 => 9,  
            "a" => 42 );
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Arrays vs Functions

Tradeoff between **Space** and **Time**.

- Arrays write in the **memory** all the associations of the mapping in order to have a **fast access**.
- Functions do not write the associations on memory, and need to recompute values for every index.

Arrays vs Functions

Tradeoff between **Space** and **Time**.

- Arrays write in the **memory** all the associations of the mapping in order to have a **fast access**.
- Functions do not write the associations on memory, and need to recompute values for every index.

Remark. the **syntax** gives a hint (Example in C):

```
int array1[12];  
int array1Func(Month m){ ... }  
  
array1[7] == array1Func(jul);
```

Arrays vs Functions

Tradeoff between **Space** and **Time**.

- Arrays write in the **memory** all the associations of the mapping in order to have a **fast access**.
- Functions do not write the associations on memory, and need to recompute values for every index.

Remark. the **syntax** gives a hint (Example in C):

```
int array1[12];  
int array1Func(Month m){ ... }  
  
array1[7] == array1Func(jul);  
  
string array2[12,31];  
string array2Func(Month m, int d){ ... }  
array2[7,4] == array2Func(jul,4);
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Disjoint Union

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Disjoint Union

Definition: A **disjoint union** of two types S and T creates a set of values as follows:

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

where "left" and "right" are tags indicating the origin of values. The result is a type containing both the values of each type S and T .

Disjoint Union

Definition: A **disjoint union** of two types S and T creates a set of values as follows:

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

where "left" and "right" are tags indicating the origin of values. The result is a type containing both the values of each type S and T .

The operations on disjoint union are:

- construction
- tag test
- projection

Example: algebraic datatypes in Haskell

In Haskell, a new datatype for numbers:

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

- Construction:

```
let pi = Inexact 3.1416
```

- tag test and projection: (assuming num is of type Number)

```
let rounded num =  
  case num of  
    Exact i   → i  
    Inexact r → intOfFloat r
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: discriminated records in Ada

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: discriminated records in Ada

```
type Accuracy is (exact, inexact);  
type Number (acc: Accuracy) is  
  record  
    case acc of  
      when exact    => ival: Integer;  
      when inexact => rval: Float;  
    end case;  
end record;
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: discriminated records in Ada

```
type Accuracy is (exact, inexact);  
type Number (acc: Accuracy) is  
  record  
    case acc of  
      when exact    => ival: Integer;  
      when inexact => rval: Float;  
    end case;  
  end record;
```

Mathematical representation:

Number = exact Integer + inexact Float

Example: discriminated records in Ada

- Construction

```
pi: constant Number :=  
    (acc => inexact, rval => 3.1416);
```

- tag test and projection:

```
function rounded (num: number) return Integer is  
    case num.acc is  
        when exact    => return num.ival;  
        when inexact => return Integer(num.rval);  
    end case;  
end;
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: discriminated records in Ada (continue)

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

Example: discriminated records in Ada (continue)

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

Type description:

Figure = *pointy*(Float × Float)
+ *circular*(Float × Float × Float)
+ *rectangular*(Float × Float × Float × Float)

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: Objects in Java

How do you encode in Java these *discriminated records* of Ada.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Example: Objects in Java

How do you encode in Java these *discriminated records* of Ada.
Class and subclass

```
class Point { private float x, y; }  
class Circle extends Point  
  { private float r; }  
class Rectangle extends Point  
  { private float w, h; }
```

Example: Objects in Java

How do you encode in Java these *discriminated records* of Ada.
Class and subclass

```
class Point { private float x, y; }  
class Circle extends Point  
  { private float r; }  
class Rectangle extends Point  
  { private float w, h; }
```

Type description:

Figure = *pointy*($Float \times Float$)
+ *circular*($Float \times Float \times Float$)
+ *rectangular*($Float \times Float \times Float \times Float$)
+ ...

(1)

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Recursive Types

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Lists in Java

How do you program simply linked list in Java ?

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Lists in Java

How do you program simply linked list in Java ?

```
class IntList {
    public int elem;
    public IntList succ;

    public IntList (int elem, IntList succ){
        this.elem = elem; this.succ = succ;
    }
}

IntList square = new IntList(1, new IntList(4,
    new IntList(9, new IntList(16, null))));
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Lists in Java

How do you program simply linked list in Java ?

```
class IntList {
    public int elem;
    public IntList succ;

    public IntList (int elem, IntList succ){
        this.elem = elem; this.succ = succ;
    }
}

IntList square = new IntList(1, new IntList(4,
    new IntList(9, new IntList(16, null))));
```

Type description:

IntList = Null + (elem int × succ IntList)

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Lists in Haskell

Type description:

$$\mathit{IntList} = \mathit{Nil} + \mathit{Cons} (\mathit{int} \times \mathit{IntList})$$

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Lists in Haskell

Type description:

$$\text{IntList} = \text{Nil} + \text{Cons} (\text{int} \times \text{IntList})$$

```
data IntList = Nil | Cons (Int, IntList)
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

General Definition

Definition: A recursive type is of the form

$$\begin{aligned} R &= (\dots) \\ &+ (\dots R \dots) \\ &+ (\dots R \dots) \end{aligned}$$

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

General Definition

Definition: A recursive type is of the form

$$\begin{aligned} R &= (\dots) \\ &+ (\dots R \dots) \\ &+ (\dots R \dots) \end{aligned}$$

Recursive types are the best for describing structured information like programs, structured documents, etc.

General Definition

Definition: A recursive type is of the form

$$\begin{aligned} R = & (\dots) \\ & + (\dots R \dots) \\ & + (\dots R \dots) \end{aligned}$$

Recursive types are the best for describing structured information like programs, structured documents, etc.

Basic example, consider arithmetic expressions:

$$\begin{aligned} Expr = & Const \ Int \\ & + Sum \ (Expr \times Expr) \\ & + Prod \ (Expr \times Expr) \end{aligned}$$

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Activity 1

In Java, write the set of classes implementing the following type of arithmetic expressions.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Activity 1

In Java, write the set of classes implementing the following type of arithmetic expressions.

```
class Expr {}

class Const extends Expr{
    public int val;
    public Const(int val){
        this.val = val;
    }
}
```

```
class Sum extends Expr{  
    public Expr lSum  
    public Expr rSum;  
  
    public Sum(Expr lSum, Expr rSum){  
        this.lSum = lSum; this.rSum = rSum;  
    }  
}  
  
class Prod extends Expr{  
    public Expr lProd;  
    public Expr rProd;  
  
    public Prod (Expr lProd, Expr rProd){  
        this.lProd = lProd; this.rProd = rProd;  
    }  
}
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Activity 2

Modify these classes in order to implement an interpreter of arithmetic expressions.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Activity 2

Modify these classes in order to implement an interpreter of arithmetic expressions.

```
class Expr{  
    public int interpret(){return 0;}  
}
```

```
class Const extends Expr{  
    public int val;  
    public Const(int val){  
        this.val = val;  
    }  
}
```

```
    public int interpret(){  
        return this.val;  
    }  
}
```

```
class Sum extends Expr{
  ...
  public int interpret(){
    int x = lSum.interpret();
    int y = rSum.interpret();
    return x + y;
  }
}
```

```
class Prod extends Expr{
  ...
  public int interpret(){
    int x = lProd.interpret();
    int y = rProd.interpret();
    return x * y;
  }
}
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Acitivity

Consider the following composite type representing a certain kind of expressions:

$$\begin{aligned} \text{Expr} = & \text{Const Int} \\ & + \text{Op} (\text{String} \times \text{Expr} \times \text{Expr}) \\ & + \text{IfThenElse} (\text{Cond} \times \text{Expr} \times \text{Expr}) \end{aligned}$$
$$\begin{aligned} \text{Cond} = & \text{Bool boolean} \\ & + \text{Neg Cond} \\ & + \text{ExprComp} (\text{String} \times \text{Expr} \times \text{Expr}) \\ & + \text{BooleanOp} (\text{String} \times \text{Cond} \times \text{Cond}) \end{aligned}$$

- 1 Propose a set of classes modelizing the constructed type above.
- 2 Implement an interpreter of expression returning the integer of its evaluation.

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Same example in Haskell

```
data Expr = Const Int  
          | Sum   (Expr, Expr)  
          | Prod  (Expr, Expr)
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Same example in Haskell

```
data Expr = Const Int
          | Sum   (Expr, Expr)
          | Prod  (Expr, Expr)

interpret :: Expr -> Int
interpret x =
  case x of
    Const i      -> i
    Sum   (e1, e2) -> (interpret e1) + (interpret e2)
    Prod  (e1, e2) -> (interpret e1) * (interpret e2)
```

Enum, Subrange: $S \subseteq T$
Cartesian Product: $S \times T$
Mapping, Arrays: $S \rightarrow T$
Disjoint Union: $S + T$
Recursive Types: $S = f(S)$

Conclusion

Classes can be used to implement all the composite types (cartesian product, disjoint union, recursive type), they are very powerful in this regard. Nevertheless, the encoding is not always convenient. Other languages have disjoint union and recursive types as more primitive in the language (Ada, Haskell, ML, OCaml, F#). Depending on the task you want to implement you may choose the adequate language.