Chapter 11
A Fourth Look At ML

Type Definitions

- Predefined, but not primitive in ML:

```
datatype bool = true | false;
```

- Type constructor for lists:

```
datatype 'element list = nil |
   :: of 'element * 'element list
```

- Defined for ML *in ML*

Defining Your Own Types

- New types can be defined using the keyword **`datatype`**
- These declarations define both:
  - *type constructors* for making new (possibly polymorphic) types
  - *data constructors* for making values of those new types

Defining Your Own Types: Enumerations

```
- datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype day = Fri | Mon | Sat | Sun | Thu | Tue | Wed
- fun isWeekDay x = not (x = Sat orelse x = Sun);
val isWeekDay = fn : day -> bool
- isWeekDay Mon;
val it = true : bool
- isWeekDay Sat;
val it = false : bool
```

- New types can be defined using the keyword **`datatype`**
- The example above defined an enumerated type called **`day`** and its members, **`Mon`** through **`Sun`**
- **`day`** is the new **type constructor** and **`Mon`**, **`Tue`**, etc. are the new **data constructors**

day → type

Mon . Tue ... → data value

No Parameters 没有参数

```
- datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype day = Fri | Mon | Sat | Sun | Thu | Tue | Wed
```

- The type constructor **`day`** takes no parameters: it is not polymorphic, there is only one **`day`** type
- The data constructors **`Mon`**, **`Tue`**, etc. take no parameters: they are constant values of the **`day`** type
- Capitalize the names of data constructors

5

Strict Typing

```
- datatype flip = Heads | Tails;
datatype flip = Heads | Tails
- fun isHeads x = (x = Heads);
val isHeads = fn : flip -> bool
- isHeads Tails;
val it = false : bool
- isHeads Mon;
Error: operator and operand don't agree [tycon mismatch]
  operator domain: flip
  operand:         day
```

- ML is strict about these new types, just as you would expect
- Unlike C **`enum`**, no implementation details are exposed to the programmer

6

Data Constructors In Patterns

```
fun isWeekDay Sat = false
|   isWeekDay Sun = false
|   isWeekDay _ = true;
```

- You can use the data constructors in patterns
- In this simple case, they are like constants
- But we will see more general cases next

7

Data constructors with parameters

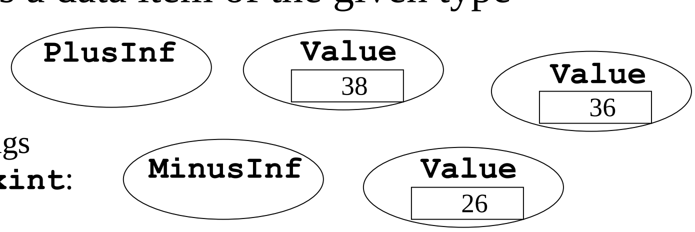
- You can add a parameter of any type to a data constructor, using the keyword **`of`**:

```
datatype exint = Value of int | PlusInf | MinusInf;
```

+∞ -∞

- In effect, such a constructor is a **wrapper** that contains a data item of the given type

Some things of type **`exint`**:



8

Data constructors with parameters

```
- datatype exint = Value of int | PlusInf | MinusInf;
datatype exint = MinusInf | PlusInf | Value of int
- PlusInf;
val it = PlusInf : exint
- MinusInf;
val it = MinusInf : exint
- Value;
val it = fn : int -> exint
- Value 3;
val it = Value 3 : exint
```

extended integer 扩展整数.

How ML reports the types of the data constructors.

- **`Value`** is a data constructor that takes a parameter: the value of the **`int`** to store
- It looks like a function that takes an **`int`** and returns an **`exint`** containing that **`int`**

9

A `Value` Is Not an `int`

```
- val x = Value 5;
val x = Value 5 : exint
- x+x;
Error: overloaded variable not defined at type
  symbol: +
  type: exint
```

- **`Value 5`** is an **`exint`**
- It is not an **`int`**, though it contains one
- How can we get the **`int`** out again?
- By pattern matching…

10

Patterns With Data Constructors

```
- val (Value y) = x;
val y = 5 : int
```

- To recover a data constructor's parameters, use pattern matching
- Note that this example only works because **`x`** actually is a **`Value`** here

11

An Exhaustive Pattern 详尽的模式

```
- val s = case x of
=           PlusInf => "infinity" |
=           MinusInf => "-infinity" |
=           Value y => Int.toString y;
val s = "5" : string
```

- An **`exint`** can be a **`PlusInf`**, a **`MinusInf`**, or a **`Value`**
- Unlike the previous example, this one says what to do for all possible values of **`x`**

12

Pattern-Matching Function

```
- fun square PlusInf = PlusInf
= |   square MinusInf = PlusInf
= |   square (Value x) = Value (x*x);
val square = fn : exint -> exint
- square MinusInf;
val it = PlusInf : exint
- square (Value 3);
val it = Value 9 : exint
```

- Pattern-matching function definitions are especially important when working with your own datatypes

13

Exception Handling (A Peek)

```
- fun square PlusInf = PlusInf
= |   square MinusInf = PlusInf
= |   square (Value x) = Value (x*x)
=        handle Overflow => PlusInf;
val square = fn : exint -> exint
- square (Value 10000);
val it = Value 100000000 : exint
- square (Value 100000);
val it = PlusInf : exint
```

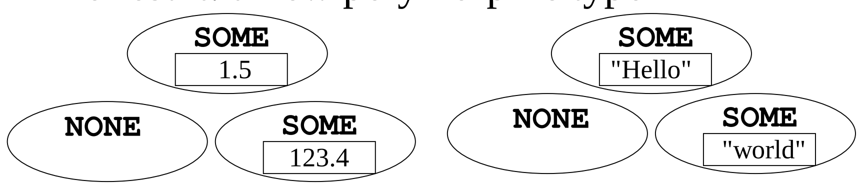
溢出情况
→ 给 plus infinity

- Patterns are also used in ML for exception handling, as in this example

14

Type Constructors With Parameters

- Type constructors can also use parameters:
  `datatype 'a option = NONE | SOME of 'a;`
- The parameters of a type constructor are type variables, which are used in the data constructors
- The result: a new polymorphic type



Values of type **`real option`**

Values of type **`string option`**

15

Parameter Before Name

```
- SOME 4;
val it = SOME 4 : int option
- SOME 1.2;
val it = SOME 1.2 : real option
- SOME "pig";
val it = SOME "pig" : string option
```

- Type constructor parameter comes before the type constructor name:
  `datatype 'a option = NONE | SOME of 'a;`
- We have types **`'a option`** and **`int option`**, just like **`'a list`** and **`int list`**

16

Uses for **`option`**

- Predefined type constructor in ML
- Used by predefined functions (or your own) when the result is not always defined

```
- fun optdiv a b =
=    if b = 0 then NONE else SOME (a div b);
val optdiv = fn : int -> int -> int option
- optdiv 7 2;
val it = SOME 3 : int option
- optdiv 7 0;
val it = NONE : int option
```


堆.

Longer Example: **`bunch`**

```
datatype 'x bunch =
    One of 'x |
    Group of 'x list;
```

可能是字符,数字也可能是列表.

- An **`'x bunch`** is either a thing of type **`'x`**, or a list of things of type **`'x`**
- As usual, ML infers types:

```
- One 1.0;
val it = One 1.0 : real bunch
- Group [true,false];
val it = Group [true,false] : bool bunch
```

Example: Polymorphism

```
- fun size (One _) = 1
= |   size (Group x) = length x;
val size = fn : 'a bunch -> int
- size (One 1.0);
val it = 1 : int
- size (Group [true,false]);
val it = 2 : int
```

- ML can infer **bunch** types, but does not always have to resolve them, just as with **list** types

解析

Example: No Polymorphism

```
- fun sum (One x) = x
= |   sum (Group xlist) = foldr op + 0 xlist;
val sum = fn : int bunch -> int
- sum (One 5);
val it = 5 : int
- sum (Group [1,2,3]);
val it = 6 : int
```

- We applied the **+** operator (through **foldr**) to the list elements
- So ML knows the parameter type must be **int bunch**

Recursively Defined Type Constructors

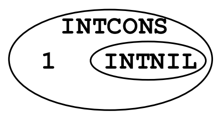
- The type constructor being defined may be used in its own data constructors:

```
datatype intlist =
  INTNIL |
  INTCONS of int * intlist;
```

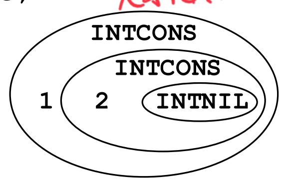
不断从整体中分离元素出来

Some values of type **`intlist`**:

the empty list



the list [1]

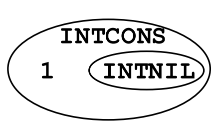


the list [1,2]

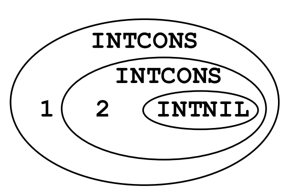
Constructing Those Values

```
- INTNIL;
val it = INTNIL : intlist
- INTCONS (1,INTNIL);
val it = INTCONS (1,INTNIL) : intlist
- INTCONS (1,INTCONS(2,INTNIL));
val it = INTCONS (1,INTCONS (2,INTNIL)) : intlist
```

the empty list



the list [1]



the list [1,2]

An **intlist** Length Function

```
fun intlistLength INTNIL = 0
  | intlistLength (INTCONS(_,tail)) =
      1 + (intListLength tail);

fun listLength nil = 0
  | listLength (_::tail) =
      1 + (listLength tail);
```

- A length function
- Much like you would write for native lists
- Except, of course, that native lists are not always lists of integers…

参数列表类型

Parametric List Type

```
datatype 'element mylist =
  NIL |
  CONS of 'element * 'element mylist;
```

- A parametric list type, almost like the predefined **list**
- ML handles type inference in the usual way:

```
- CONS(1.0, NIL);
val it = CONS (1.0,NIL) : real mylist
- CONS(1, CONS(2, NIL));
val it = CONS (1,CONS (2,NIL)) : int mylist
```

Some **mylist** Functions

```
fun myListLength NIL = 0
  | myListLength (CONS(_,tail)) =
      1 + myListLength(tail);

fun addup NIL = 0
  | addup (CONS(head,tail)) =
      head + addup tail;
```

- This now works almost exactly like the predefined **list** type constructor
- Of course, to add up a list you would use **foldr**…

25

A **foldr** for **mylist**

```
fun myfoldr f c NIL = c
  | myfoldr f c (CONS(a,b)) =
      f(a, myfoldr f c b);
```

- Definition of a function like **foldr** (P141) that works on **'a mylist**
- Can now add up an **int mylist x** with:
  myfoldr (op +) 0 x
- One remaining difference: **::** is an operator and **CONS** is not

26

Defining Operators (A Peek)

- ML allows new operators to be defined
- A right-associative binary operator **CONS** with a precedence level of 5:

```
- infixr 5 CONS;
infixr 5 CONS
- 1 CONS 2 CONS NIL;
val it = 1 CONS 2 CONS NIL : int mylist
```

- Now you can write `1 CONS 2 CONS NIL` just as you can write `1 :: 2 :: NIL`.

Questions?