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
Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to 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.
Example

- 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

- **day** is the new type constructor and Mon, Tue, etc. are the new data constructors
- Why “constructors”? In a moment we will see how both can have parameters...
No Parameters

- 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
Strict Typing

- `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
```

- MIL is strict about these new types, just as you would expect
- Unlike Cenum, no implementation details are exposed to the programmer
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
Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to ML
You can add a parameter of any type to a data constructor, using the keyword of:

\[ \text{datatype exint} = \text{Value of int} \mid \text{PlusInf} \mid \text{MinusInf}; \]

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

Some things of type \text{exint}:

- Value 38
- Value 36
- Value 26
- Value 26
- Value 38
- Value 38
- Value 38
- Value 38
- Value 38
- 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

- **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`
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...
Patterns With Data Constructors

- `val (Value y) = x;`
  Warning: binding not exhaustive
  Value y = ...
  `val y = 5 : int`

- To recover a data constructor’s parameters, use pattern matching

- So `Value` is no ordinary function: ordinary functions can't be pattern-matched this way
An Exhaustive Pattern

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

- Like most uses of the match construct, you get a warning if it is not exhaustive
- An `exint` can be a `PlusInf`, a `MinusInf`, or a `Value`
- This one says what to do in all cases
Pattern-Matching Function

```plaintext
- 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
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

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

We’ll see it in Java, but skip it in ML
Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to ML
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:
- NONE
- SOME 1.5
- SOME 123.4

Values of type string option:
- NONE
- SOME "Hello"
- SOME "world"
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
Uses For \texttt{option}

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

```ml
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
```
Longer Example: \texttt{bunch}

\begin{verbatim}
datatype 'x bunch = 
  One of 'x | 
  Group of 'x list;
\end{verbatim}

- An \texttt{'x bunch} is either a thing of type \texttt{'x}, or a list of things of type \texttt{'x}
- As usual, ML infers types:

\begin{verbatim}
- One 1.0;
  val it = One 1.0 : real bunch
- Group [true,false];
  val it = Group [true,false] : bool bunch
\end{verbatim}
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
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
Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to ML
The type constructor being defined may be used in its own data constructors:

\[
\text{datatype intlist = INTNIL | INTCONS of int * * intlist;}
\]

Some values of type \text{intlist}:

- \text{INTNIL}
- \text{INTCONS
  \begin{itemize}
  \item 1
  \item INTNIL
  \end{itemize}
  \text{the list } [1]
- \text{INTCONS
  \begin{itemize}
  \item 1
  \item 2
  \item INTNIL
  \end{itemize}
  \text{the list } [1,2]
Constructing Those Values

- \texttt{INTNIL};
val it = \texttt{INTNIL} : \texttt{intlist}
- \texttt{INTCONS (1,INTNIL)};
val it = \texttt{INTCONS (1,INTNIL)} : \texttt{intlist}
- \texttt{INTCONS (1,INTCONS(2,INTNIL))};
val it = \texttt{INTCONS (1,INTCONS (2,INTNIL))} : \texttt{intlist}

\begin{itemize}
  \item \texttt{INTNIL}  \quad \textit{the empty list}
  \item \texttt{INTCONS 1 \texttt{INTNIL}}  \quad \textit{the list [1]}
  \item \texttt{INTCONS 1 2 \texttt{INTNIL}}  \quad \textit{the list [1,2]}
\end{itemize}
An \textit{intlist} Length Function

\begin{verbatim}
fun intlistLength INTNIL = 0
    | intlistLength (INTCONS(_,tail)) =
        1 + (intListLength tail);

fun listLength nil = 0
    | listLength (_:::tail) =
        1 + (listLength tail);
\end{verbatim}

\begin{itemize}
    \item A length function
    \item Much like you would write for native lists
    \item Except, of course, that native lists are not always lists of integers…
\end{itemize}
Parametric List Type

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

mylist;

A parametric list type, almost like the predefined list

• MIL 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...
**A foldr For mylist**

\[
\text{fun myfoldr } f \ c \ NIL = c \\
| \text{myfoldr } f \ c \ (\text{CONS}(a,b)) = f(a, \text{myfoldr } f \ c \ b);
\]

- Definition of a function like `foldr` that works on `'a mylist`
- Can now add up an `int mylist x` with:
  \[
  \text{myfoldr } (\text{op } +) \ 0 \ x
  \]
- One remaining difference: `::` is an operator and \text{CONS} is not
Defining Operators (A Peek)

- ML allows new operators to be defined
- Like this:

```ml
- infixr 5 CONS;
infixr 5 CONS
- 1 CONS 2 CONS NIL;
val it = 1 CONS 2 CONS NIL : int
mylist
```
Polymorphic Binary Tree

datatype 'data tree =
    Empty |
    Node of 'data tree * 'data * 'data tree;

Some values of type int tree:

the emptytree

the tree 2

the tree 2

1 3
Constructing Those Values

- `val treeEmpty = Empty;
val treeEmpty = Empty : 'a tree
- `val tree2 = Node(Empty, 2, Empty);
val tree2 = Node (Empty, 2, Empty) : int tree
- `val tree123 = Node(Node(Empty, 1, Empty),
= 2,
= Node(Empty, 3, Empty));`
Increment All Elements

```plaintext
fun incall Empty = Empty
  | incall (Node(x,y,z)) = Node(incall x, y+1, incall z);

val it = Node (Node (Empty,2,Empty),
             3,
             Node (Empty,4,Empty)) : int tree
```
Add Up The Elements

fun sumall Empty = 0
| sumall (Node(x,y,z)) =
    sumall x + y + sumall z;

- sumall tree123;
val it = 6 : int
Convert To List (Polymorphic)

```ocaml
fun listall Empty = nil
  | listall (Node(x,y,z)) =
      listall x @ y :: listall z;

- listall tree123;
val it = [1,2,3] : int list
```
Tree Search

fun isinmtree x Empty = false
    | isinmtree x (Node(left,y,right)) =
        x=y
        orelse isinmtree x left
        orelse isinmtree x right;

- isinmtree 4 tree123;
val it = false : bool
- isinmtree 3 tree123;
val it = true : bool
Outline

■ Enumerations
■ Data constructors with parameters
■ Type constructors with parameters
■ Recursively defined type constructors
■ Farewell to ML
That's All

- That's all the ML we will see
- There is, of course, a lot more
- A few words about the parts we skipped:
  - records (like tuples with named fields)
  - arrays, with elements that can be altered
  - references, for values that can be altered
  - exception handling
More Parts We Skipped

- support for encapsulation and data hiding:
  - structures: collections of datatypes, functions, etc.
  - signatures: interfaces for structures
  - functors: like functions that operate on structures, allowing type variables and other things to be instantiated across a whole structure
More Parts We Skipped

- **API**: the standard basis
  - predefined functions, types, etc.
  - Some at the top level but most in structures: `Int.maxInt`, `Real.Math.sqrt`, `List.nth`, etc.
More Parts We Skipped

- eXene: an ML library for applications that work in the X window system
- the Compilation Manager for building large ML projects

Other dialects besides Standard ML

- OCaml
- Concurrent ML (CML) extensions
Functional Languages

- ML supports a function-oriented style of programming
- If you like that style, there are many other languages to explore, like Lisp and Haskell