Processadors de Llenguatge II

Functional Paradigm II

Pratt A.7

Robert Harper’s SML tutorial (Sec II)

Rafael Ramirez
Dep Tecnologia
Universitat Pompeu Fabra
User-Defined Types

- How to define the new type.

- The principle in the use of data types (constructing via the data constructors, and deconstructing via pattern matching) is the same principle that you learnt in lists.
User-Defined Types

In SML,

- the list is a predefined type constructor. The :: and [] are predefined data constructors.
- The value 1::(2::[]) has a type of int list. Which can be depicted conceptually as

```
|::|::|[]|
```

- Functions on list, deconstruct it by pattern matching...

```haskell
fun len [] = 0
  | len (x::xs) = 1 + len xs
```

The function len pattern-matches by checking the data constructor first. Then it will know what things are stored.

If it's a '::' then it knows it can take out a head and tail.

If it's a '[ ]', then it knows that there is no data stored there.
User-Defined Types

In SML

- the list is a predefined type constructor.
- The :: and [] are predefined data constructors.
- The value 1::(2::[]) has a type of int list. Which can be depicted conceptually as

```
-> :: -> :: -> []
```
- Functions on list, deconstruct it by pattern matching...
  ```
  fun len [] = 0 
  |   len (x::xs) = 1 + len xs
  ```
- You can define your own type:
  ```
  datatype 'a mylist
    = mynil
    | mycons of 'a * ('a mylist)
  ```
- Now, mylist is a user-defined type constructor.
- The mycons and mynil are the data constructors.
- The value mycons(1,mycons(2,mynil)) has a type of int mylist. Which can be depicted conceptually as

```
  mycons 1 2
  mycons mynil
```
- Functions on list, deconstruct it by pattern matching...
  ```
  fun mylen mynil = 0 
  |   mylen (mycons(x,xs))
      = 1 + mylen xs
  ```
In defining a new datatype, you define

- A new type or a new type constructor.
- One or more data constructors.
User-Defined Types

```
datatype 'a mylist
  = mynil
  | mycons of 'a * ('a mylist)

datatype myintlist
  = intnil
  | intcons of int * myintlist
```

In defining a new datatype, you define
- A new type or a new type constructor.
- One or more data constructors.
User-Defined Types

datatype `a mylist
  = mynil
  | mycons of `a * (`a mylist)

mycons(1,mycons(2,mynil)) : int mylist

Data constructors

Type constructor

Data expression

Type expression
User-Defined Types

datatype ‘a mylist
    = mynil
    | mycons of ‘a * (‘a mylist)

mycons(1,mycons(2,mynil)) : int mylist
mycons(1.1,mycons(2.2,mynil)) : real mylist
mycons(“a”,mycons(“b”,mynil)):string mylist

This is an example of PARAMETRIC POLYMORPHISM. It is when the type takes in another type as parameter.
User-Defined Types

datatype `a mylist
  = mynill
  | mycons of `a * (`a mylist)

In general, the BNF for datatype declaration is:

<DataDecl> ::= datatype [tyvar] tynamel =
  <dataAlt> {`|` <dataAlt>}

<dataAlt> ::= <Constructor>
  | <Constructor> of <type> {`*` <type>}

<Constructor> ::= <Identifier>
<tynamel> ::= <Identifier>
<tyvar> ::= <Identifier>
User-Defined Types – Binary Tree

Defining an integer binary tree:

```
datatype IntTree
  = Leaf of int
  | Node of IntTree * int * IntTree
```

```
<DataDecl> ::= datatype [<tyvar>] <tynname> =
  <dataAlt> {'|' <dataAlt>}

<dataAlt> ::= <Constructor>
  | <Constructor> of <type> {* <type>}

<Constructor> ::= <Identifier>
<tynname> ::= <Identifier>
<tyvar> ::= <Identifier>
```
User-Defined Types – Binary Tree

- Defining an integer binary tree:

  ```
  datatype IntTree
      = Leaf of int
      | Node of IntTree * int * IntTree
  ```

- The above will declare
  - A new type name: `IntTree`
  - 2 new data constructors: `Leaf`, `Node`.
Defining an integer binary tree:

datatype IntTree
    = Leaf of int
    | Node of IntTree * int * IntTree

Constructing a tree using the user-defined data constructors:

• Leaf 26
• Node (Leaf 26, 30, Leaf 10)
• Node (Leaf 26, 30, Node (Leaf 10, 3, Leaf 4))

All will be of type IntTree
User-Defined Types – Binary Tree

Defining an integer binary tree:

```haskell
datatype IntTree = Leaf of int |
                Node of IntTree * int * IntTree
```

Deconstructing a tree through pattern matching:

```haskell
fun bsearch (Leaf y) _ = y
|   bsearch (Leaf _) _ = ~1
|   bsearch (Node (l,x,r)) y =
    if (y = x) then x  (* return the value *)
    else if (y < x) then bsearch l y
    else bsearch r y;
```
User Defined DataTypes

Parametric Polymorphism

- Defining a binary tree of some type:

```haskell
datatype 'a tree
    = Leaf of 'a
    | Node of ('a tree) * 'a * ('a tree)
```

- Now we can have trees containing any type of object, but all objects stored must be the same type.
User Defined DataTypes

Parametric Polymorphism

- Defining a binary tree of some type:

\[
\text{datatype ‘a tree} \\
= \text{Leaf of ‘a} \\
| \text{Node of (‘a tree) * ‘a * (‘a tree)}
\]

Constructing a tree using the user-defined data constructors:

- Leaf 1.33 : real tree
- Leaf 1 : int tree
- Node (Leaf 26, 30, Leaf 10) : int tree
- Node (Leaf [1], [2,3], Leaf [4]) : int list tree
User Defined Data Types

Parametric Polymorphism

Defining a binary tree of some type:

```
datatype 'a tree
    = Leaf of 'a
    | Node of ('a tree) * 'a * ('a tree)
```

Deconstructing a tree through pattern-matching:

```
fun height (Leaf _) = 1
|    height (Node (l,_,r)) = 1 + max(height l, height r)
```

```
height (Node (Node (Leaf 1,10,Leaf 2), 20 (Leaf 3)))
```

Will evaluate to ?.
User Defined DataTypes

Parametric Polymorphism

Defining a binary tree of some type:

```ocaml
datatype 'a tree = Leaf of 'a | Node of ('a tree) * 'a * ('a tree)
```

Deconstructing a tree through pattern-matching:

```ocaml
fun height (Leaf _) = 1
| height (Node (l,_,r)) = 1 + max(height l, height r)
```

height (Node (Node (Leaf 1,10,Leaf 2), 20 (Leaf 3)))

Will evaluate to 3.
**Types in ML**

Every expression will evaluate to a value. By the word "value", we do not merely restrict ourselves to numbers, characters or booleans. For example, a list \([1, 2, 3]\) is also considered as a value.

In as much as

\[1 :: \text{int} \]

so

\[[1, 2, 3] :: \text{int list} \]

Value

Type (which is a set of values)
Every value has a type.

Therefore every expression written in ML “can be assigned a type” – which is the type of the value it will evaluate to.

If the expression that is written “has a type” or “can be assigned a type”, we say that:

- “The expression is well-typed”; or
- “The expression is typable (by the ML type system)”. 
Types in ML – Type Check vs Infer

Type checking vs Type inferencing

- In most programming languages, we declare the type of an identifier, and the type system **CHECKS** that the program is type-safe.

- In ML, (most of the time) we need not declare the type of an identifier / expression, because the type system **INFERS** the type of the expression in the process of checking for type-safety.
However, if you insist on declaring the type of each expression, you may do so.

\[
\text{<Exp>} ::= \text{<Constant>} \ [\text{: <type}>] \\
| \quad \text{<Exp}_1 \ <op> \ <Exp}_2 \ [\text{: <type}>] \\
| \quad \text{if} \ <Exp}_0 \ \text{then} \ <Exp}_1 \ \text{else} \ <Exp}_2 \ [\text{: <type}>] \\
| \quad \text{let} \ {\langle\text{Decl}\rangle} \ \text{in} \ <Exp> \ \text{end} \ [\text{: <type}>] \\
| \quad <Exp}_1 \ <Exp}_2 \ [\text{: <type}>] \\
| \quad \text{fn} \ <\text{param}> \ => \ <Exp> \ [\text{: <type}>] \\
| \quad \text{<TupleExp>} \ [\text{: <type}>] \\
| \quad \text{<ListExp>} \ [\text{: <type}>] \\
| \quad \text{<Identifier>} \ [\text{: <type}>]
\]
Types in ML – BNF for types

\[
\text{<type> ::= int | real | bool | char | string}
\]

Primitive types names

- You have seen integers, reals and booleans.
- Here are examples of \texttt{char} and \texttt{string} types.
  - \texttt{"a"};
  val it = "a" : char
  - "a" : char;
  val it = "a" : char
  - ["a", "b", "c"];
  val it = ["a", "b", "c"] : char list
  - "a";
  val it = "a" : string
  - "a":string;
  val it = "a" : string
  - ["a","b","c"];
  val it = ["a","b","c"] : string list
Types in ML – BNF for types

\[
\text{<type>} ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{char} \mid \text{string} \\
\mid \text{<type>} \rightarrow \text{<type>}
\]

- **Function Types**

  - Here are some examples:
    - \((\text{fn } x \Rightarrow x + 1) : \text{int} \rightarrow \text{int};\)
    - \text{val it} = \text{fn} : \text{int} \rightarrow \text{int}
    - \text{fun add} x y z = x + y + z;
    - \text{val add} = \text{fn} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}
    - \text{val add2} = (\text{add} 0) : \text{int} \rightarrow \text{int} \rightarrow \text{int}
    - \text{val add2} = \text{fn} : \text{int} \rightarrow \text{int} \rightarrow \text{int}
    - \text{val add3} = (\text{add} 5 6) : \text{int} \rightarrow \text{int}
    - \text{val add3} = \text{fn} : \text{int} \rightarrow \text{int}
Types in ML – BNF for types

\[
\text{<type>} ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{char} \mid \text{string} \\
\mid \text{<type>} \rightarrow \text{<type>} \\
\mid \text{<type>} \ast \text{<type>} \{ \ast \text{<type>} \}
\]

- **Tuples (product types)**

- Here are some examples:
  - \((1,2) : \text{int} \ast \text{int}\);
  - \(\text{val it} = (1,2) : \text{int} \ast \text{int}\)
  - \((1.2,3) : \text{real} \ast \text{int}\);
  - \(\text{val it} = (1.2,3) : \text{real} \ast \text{int}\)
  - \(((\#"a",2), 3.3 , ("abc",true)) : ((\text{char} \ast \text{int}) \ast \text{real} \ast (\text{string} \ast \text{bool}))\);
  - \(\text{val it} = ((\#"a",2),3.3,("abc",true)) : (\text{char} \ast \text{int}) \ast \text{real} \ast (\text{string} \ast \text{bool})\)
Types in ML – BNF for types

```
<type> ::= int | real | bool | char | string
    | <type> -> <type>
    | <type> * <type> { * <type> }
    | <typename>
```

- (User-Defined) Type names:
  - Shown earlier with user-defined `mylist` and `intTree`.

```ml
datatype IntTree
    = Leaf of int
    | Node of IntTree * int * IntTree
```
Types in ML – BNF for types

```
<type> ::= int | real | bool | char | string
    | <type> -> <type>
    | <type> * <type> { * <type> }
    | <typename>
    | <type> <typecons>
    | <typevar>
```

- Type constructors and type variables
  - Examples: Shown earlier with ‘a mylist.
  - Here ‘a is a type variable, and mylist is a type constructor,
Curried vs Uncurried Functions

- How many arguments does this function take?
  \[
  \textbf{fun} \ f(x, y) = x + 2 \times y;
  \]
- Ans: 1 argument, which is pattern-matched to a pair.
  \[
  \textbf{val} \ f = \textbf{fn} : \text{int} \times \text{int} \rightarrow \text{int}
  \]
Curried vs Uncurried Functions

How many arguments does this function take?

```
fun f x y = x + 2 * y ;
```

Look at the type of the function:

```
val f = fn : int -> int -> int
```

Note that

1. the `->` associates to the right. So the type is:

```
val f = fn : int -> (int -> int)
```

2. Application associates to the left. So

```
f 1 2 = ((f 1) 2)
```
Curried vs Uncurried Functions

So from the type what can you deduce? How many arguments?

```plaintext
val f = fn : int -> (int -> int)
```

- If you mean in the immediate sense, ONE argument is taken in by f.
  - `f` a function which takes in an integer and returns a function.

- But if you mean ‘eventually, when fully-applied, how many arguments does it take in’, then f takes in a total of TWO arguments.
  - `f` takes in an integer (1\textsuperscript{st} argument) which returns a function, which will take in another integer (2\textsuperscript{nd} argument), and return an integer.
Curried vs Uncurried Functions

How many arguments does the function really take?

\[
\text{fun } f \ (x, y) = x + 2 \times y ;
\]

pattern 1: a pair

Ans: 1 argument which is pattern-matched to a pair

\[
\text{fun } f \ x \ y = x + 2 \times y ;
\]

pattern 1 \hspace{1cm} \text{pattern 2}

Ans: Eventually, when fully-applied, 2 arguments
Curried vs Uncurried Functions

You can convert any uncurried function to a curried version. This process is known as currying. ‘curry’ is named after Haskell B. Curry

\[
\text{fun } f(x, y) = x + 2 \times y; \quad \text{UNCURRED FUNCTION}
\]

\[
\text{fun } f x y = x + 2 \times y; \quad \text{CURRIED FUNCTION}
\]

The same

\[
\text{fun } f x = (\text{fn } y => x + 2 \times y);
\]

\[
\text{val } f = (\text{fn } x => (\text{fn } y => x + 2 \times y));
\]
Curried vs Uncurried Functions

Curried functions provide extra flexibility to the language because it enables partial application of a function.

```ml
fun twice f x = f (f x);
let
  val inc2 = twice (fn x => x + x)
in  (inc2 1) + (inc2 2)
end;

val it = 12 ;
```
Curried vs Uncurried Functions

Curried functions provide extra flexibility to the language because it enables partial application of a function.

```plaintext
fun add_ver1 (x,y) = x+y; (* Uncurried *)
fun add_ver2 x y = x+y; (* Curried *)

map (add_ver1 10) [1,2,3,4,5] (* will not work *)
map (add_ver2 10) [1,2,3,4,5] (* this will work *)
```