CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

Version: 1

March 25, 2020

Plan

- **Announcements**
  - HW6 is due Friday
- **Last time**
  - ML intro
- **Today**
  - Programming with constructed data and types

ML -- The Five Questions

- Syntax: definitions, expression, patterns, types
- Values: num/string/bool, record/tuple, algebraic data types
- Environments: names stand for values (and types)
- Evaluation: uScheme + case and pattern matching
- Initial Basis: medium size; emphasizes lists
- (Question Six: type system – a coming attraction)

Foundation: Data/Values in ML

- **Base types: int, real, bool, char, string**
- **Functions**
- **Constructed data**
  - Tuples: pairs, triples, etc.
  - (Records with named fields)
  - Lists and other algebraic data types

Algebraic Datatypes

- **Enumerated types**
  - Datatypes can define an enumerated type and associated values

```
datatype suit = heart | diamond | spade | club
```

- **"suit" is the name of a new type**
- **Data constructors heart, diamond, space, and club are values of that type**
- **Data constructors are separated by vertical bars**

Algebraic Datatypes

- **Pattern matching**
  - Datatypes are deconstructed using pattern matching

```
fun toString heart = "heart"
  | toString diamond = "diamond"
  | toString spade = "spade"
  | toString club = "club"

val suitName = toString heart
```

Data constructors can take arguments!


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

- What is the name of the new type?
- What data constructors are in the above?
- What is the parameter to the Node data constructor?
- What are some example values made with the data constructors?
  - Leaf
  - Node (9, Leaf, Leaf)
- What are the type(s) of those values?

Tree Example


```
val tempty = Leaf
val t1 = Node (1, tempty, tempty)
val t2 = Node (2, t1, t1)
val t3 = Node (3, t2, t2)
```

- ➔ **Tree diagram**

- **Questions**
  - **What is the in-order traversal of t3?**
    - [1, 2, 1, 3, 1, 2, 1]
  - **What is the pre-order traversal of t3?**
    - [3, 2, 1, 1, 2, 1, 1]

Deconstruct values with pattern matching


```
fun inOrder Leaf = []
  | inOrder (Node (v, left, right)) =
      (inOrder left) @ [v] @ (inOrder right)
val il3 = inOrder t3

fun preOrder Leaf = []
  | preOrder (Node (v, left, right)) =
      v :: (preOrder left) @ (preOrder right)
val pl3 = preOrder t3
```

- **Notes**
  - IntTree is monomorphic because it has a single type
  - Note though that the inOrder and preOrder functions only care about the structure of the tree, not the payload value
- **Questions**
  - What does @ do?
  - How would we implement postOrder? (postOrder left) @ (postOrder right) @ [v]

Polymorphic datatypes!


```
datatype 'a tree = Child
                 | Parent of 'a * 'a tree * 'a tree
```

- **Notes**
  - **Polymorphic datatypes are written using type variables that can be instantiated with any type**
  - `tree` **is a type constructor (written in post-fix notation), which means it produces a type when applied to a type argument**
  - **Examples:**
    - `int tree` **is a tree of integers**
    - `bool tree` **is a tree of booleans**
    - `int list tree` **is a tree of a list of integers**
  - `'a` **is a type variable: it can represent any type**
- **Questions**
  - **What are the data constructors? Child, Parent**
  - **Create an example tree with at least two parents.**
    - **Parent (42, Parent(123, Child, Child)), Parent(456, Child, Child)))**

Pattern Matching with Polymorphism


```
datatype 'a tree = Child
                 | Parent of 'a * 'a tree * 'a tree

fun inOrder Child = []
  | inOrder (Parent(v,left,right)) =
      (inOrder left) @ [v] @ (inOrder right)
```

- **Questions**
  - Finish the rest of the above?
  - What name(s) are being introduced into the type environment above?
  - What name(s) are being introduced into the value environment above?
  - Are datatype declarations inductive? How can you tell whether they are or not?
  - How is polymorphism different than overloading? Polymorphism is more like templates in C++, Java Generics

Datatype Exercise

Define algebraic data types for SX_1 and SX_2, where

$$SX_1 = ATOM \cup LIST(SX_1)$$

$$SX_2 = ATOM \cup \{\,(\texttt{cons}\ \ v_1\ \ v_2) \mid v_1 \in SX_2, v_2 \in SX_2\}$$

(take *ATOM*, with ML type `atom` as given)

- **Question**
  - What are some example values of sx1? LIST1 ([ATOM1 blah]),
    - LIST1 ([ATOM1 foo, LIST1 ([ATOM1 blah])])
  - How can you declare a datatype for sx2?

```
datatype atom = blah | foo
datatype sx1 = ATOM1 of atom
             | LIST1 of sx1 list

datatype sx2 = ATOM2 of atom
             | LIST2 of sx2 * sx2
```

Datatype Exercise cont…

- **Question**
  - What patterns would we use for sx1?

```
datatype ATOM1 = blah | foo
datatype sx1 = ATOM1 of atom
             | LIST1 of sx1 list


fun toString (ATOM1(blah)) = "blah"
           | (ATOM1(foo)) = "foo"
           | (LIST1([])) = ""
           | (LIST1(x::xs)) =
                (toString x) ^ (toString LIST1(xs))
```

Tuple Pattern Matching

- **Question**
  - Indicate what the variables in the pattern will be bound to.

```
val (x,y) = (1,2) (* x=1, y=2 *)

val (n,xs) = (3,[1,2,3]) (* n=3, xs=[1,2,3] *)

val (x::xs) = [1,2,3]  (* x=1, xs=[2,3] *)

val (_::xs) = [1,2,3]  (* xs=[2,3] *)

val (_::xs) = [3]  (* xs=[] *)
```

Case Expressions also use pattern matching

- **New language construct: `case` expression**

```
fun length xs =
  case xs
    of []       => 0
     | (x::xs) => 1 + length xs
```

- **At top level, `fun` is better than `case`**

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

Case works for any datatype

- **At top level, `fun` is better than `case`**

```
fun toStr t =
    case t
      of Leaf => "Leaf"
       | Node(v,left,right) => "Node"
```

- **Question: how do we rewrite case using fun?**

```
fun toStr ??
```

Exception Handling

- **Syntax**
  - `Declaration: exception exn`
  - `Introduction: raise where e: exn`
  - `Elimination: e1 handle pat => e2`
- **Informal Semantics**
  - Alternative to normal termination
  - Can happen in any expression
  - Tied to function call: if evaluation of body raises exn, call raises exn
  - Handler uses pattern matching

```
e handle pat1 => e1 | pat2 => e2
```

ML traps and pitfalls

- **Order of clauses matters**

```
fun take n (x::xs) = x :: take (n-1) xs
  | take 0 xs = []
  | take n [] = []
(* what goes wrong? *)
```

- **Gotcha – overloading**

```
- fun plus x y = x + y;
> val plus = fn : int -> int -> int
- fun plus x y = x + y : real;
> val plus = fn : real -> real -> real
```

- **Gotcha – equality types, ''a is equality type var**

```
- (fn (x,y) => x=y);
> val ''a it = fn : ''a * ''a -> bool
```