CSCI-344
Programming Language Concepts (Section 3)

Lecture 12

Algebraic data types

Instructor: Hossein Hojjat

September 28, 2016

Where we are

Done:

- ML: Overview

This session:

- Algebraic data types

Defining new types

- SML has a number of built-in data types and associated operators
  - `int`, `bool`, `string`
- We sometimes need to define our own data types to express some kind of data
- Algebraic data types allow us to add our own abstractions

```
datatype Bool = True | False
```

- `Bool`: name of the new data type
- `True` and `False`: data constructors of the type `Bool`
- Data constructors are separated by vertical bars

"The type `Bool` can have a value of `True` or `False`"

Why "Algebraic"?

- It is based on the framework of algebraic specifications in mathematics

Pattern Matching

- We can define functions by pattern matching

```
fun not False = True
  | not True = False;
```

```
fun toString False = "False"
  | toString True = "True";
```

- Data types are **deconstructed** using pattern matching

Recursive Definition

- How can we define Natural numbers?
- We do not have something like this:

```
datatype Nat = 0 | 1 | 2 ...
```

Recursive Definition

- How can we define Natural numbers?
- We do not have something like this:

```
datatype Nat = 0 | 1 | 2 ...
```

- Answer: Use recursion!

```
datatype Nat = Zero | Succ of Nat
```

- This is number 3:
- `Succ(Succ(Succ(Zero)))`
- This way you can define any natural number

Type parameters

- Data constructor can take arguments and then produce a new value

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- `intTree` is the name of a new type
- There are two data constructors: `Empty` and `Node`
- Nodes take a tuple of three arguments:
  - value at the node
  - left and right subtrees
- The keyword **of** separates the name of the data constructor and the type of its argument
- When fully applied, data constructors have the type of the defining datatype (ie, `intTree`)

intTree

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- Function that adds up the values in the nodes of a tree

```
fun sum Empty = 0
  | sum (Node(n,left,right))=
        (sum left) + n + (sum right);
```

intTree

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- Function that adds up the values in the nodes of a tree

```
fun sum Empty = 0
  | sum (Node(n,left,right))=
        (sum left) + n + (sum right);
```

- Function that returns the height of a tree
- For example,

  `height(Node(1,Node(2,Empty,Empty),Empty))=2`

intTree

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- Function that adds up the values in the nodes of a tree

```
fun sum Empty = 0
  | sum (Node(n,left,right))=
        (sum left) + n + (sum right);
```

- Function that returns the height of a tree
- For example,
  `height(Node(1,Node(2,Empty,Empty),Empty))=2`

```
fun max(x,y) = if x < y then y else x;
fun height(Empty) = 0
  | height(Node(n,left,right)) =
      1 + max(height(left),height(right));
```

intTree

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- Function that computes in-order traversal a tree
- The @ symbol denotes append in ML

```
fun inOrder Empty = []
  | inOrder (Node (v, left, right)) =
      (inOrder left) @ [v] @ (inOrder right);
```

intTree

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- Function that computes in-order traversal a tree
- The @ symbol denotes append in ML

```
fun inOrder Empty = []
  | inOrder (Node (v, left, right)) =
      (inOrder left) @ [v] @ (inOrder right);
```

- Function that computes pre-order traversal a tree

```
fun preOrder Empty = []
  | preOrder (Node (v, left, right)) =
      v :: (preOrder left) @ (preOrder right);
```

intTree

```
datatype intTree = Empty | Node of int * intTree * intTree
```

- Function that computes in-order traversal a tree
- The @ symbol denotes append in ML

```
fun inOrder Empty = []
  | inOrder (Node (v, left, right)) =
      (inOrder left) @ [v] @ (inOrder right);
```

- Function that computes pre-order traversal a tree

```
fun preOrder Empty = []
  | preOrder (Node (v, left, right)) =
      v :: (preOrder left) @ (preOrder right);
```

- `inOrder` and `preOrder` only care about the structure of the tree, not the payload value at each node

Polymorphic Data Types

- `intTree` is monomorphic because it has a single type
- Polymorphic datatypes are written using type variables that can be instantiated with any type

```
datatype 'a Tree = Empty | Node of 'a * 'a Tree * 'a Tree
```

- `'a` is a type variable: it can represent any type
- `Tree` is a type constructor (written in post-fix notation)
- It produces a type when applied to a type argument
  - `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

Polymorphic Data Types

- `intTree` is monomorphic because it has a single type
- Polymorphic datatypes are written using type variables that can be instantiated with any type

```
datatype 'a Tree = Empty | Node of 'a * 'a Tree * 'a Tree
```

Example:

```
- val empty = Empty;
- val t1 = Node(1, Empty, Empty);
- val t2 = Node(2,empty,t1);
- val t3 = Node("x", empty,empty);
```

Is this one ok?

```
- val t4 = Node("y",empty,t1);
```

Polymorphic Data Types

- `intTree` is monomorphic because it has a single type
- Polymorphic datatypes are written using type variables that can be instantiated with any type

```
datatype 'a Tree = Empty | Node of 'a * 'a Tree * 'a Tree
```

- `Empty` and `Node` are data constructors
- `Empty` takes no arguments and produces a value of type `'a tree`
- `Node` gets a value of type `'a` and two `'a Tree`s and produces a `'a Tree`

Recursive Data Types

- A recursive data type references itself (like `Tree`)
- Algebraic data types may or may not be recursive

```
datatype 'a List = Nil | Cons of 'a * ('a List)
```

Recursive Data Types

- A recursive data type references itself (like `Tree`)
- Algebraic data types may or may not be recursive

```
datatype 'a List = Nil | Cons of 'a * ('a List)
```

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)

Recursive Data Types

- A recursive data type references itself (like `Tree`)
- Algebraic data types may or may not be recursive

```
datatype 'a List = Nil | Cons of 'a * ('a List)
```

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)

Answer:

```
datatype sx1 = ATOM1 of atom
             | LIST1 of sx1 list
datatype sx2 = ATOM2 of atom
             | PAIR2 of sx2 * sx2
```

Case Expressions

- Yet another way to do pattern matching

```
case expr of
    pat1 => ...
|   pat2 => ...
```

- Example: check if the length of the arguments list is at least two
- The _ pattern matches everything!

```
fun atLeastTwo(l) =
  case l of
      [] => false
  | _::[] => false
    | _ => true
```

Case Expressions

- You can transform any **`case`** construct into a function
- Following code snippets are semantically equivalent

```
case x of
  1 => "one"
| _ => "not one"
```

```
fun case_example 1 = "one"
  | case_example _ = "not one";
case_example(x)
```

Summary

- Algebraic Data Types
- Polymorphic Types

Next Lecture

- Introduction to type theory