CSCI 334:
Principles of Programming Languages

Lecture 4: ML, part 2

Instructor: Dan Barowy

Williams

Topics

Pattern matching

Algebraic data types

Option type

Your to-dos

1. Lab 2, **due Sunday 9/25 by 10pm** (partner lab).
2. Reading quiz, **due Wednesday 9/21 by 10pm**.

Announcements

- CS Colloquium this **Friday, Sept 23 @ 2:35pm in Wege Auditorium (TCL 123)**

Michael Greenberg (Stevens Institute of Tech)

Formal Support for the POSIX Shell

The POSIX shell is a widely deployed, powerful tool for managing computer systems. The shell is the expert's control panel, a necessary tool for configuring, compiling, installing, maintaining, and deploying systems. Even though it is powerful, critical infrastructure, the POSIX shell is maligned and misunderstood. Its power and its subtlety make for a dangerous combination.

How can we support the POSIX shell? I'll describe two recent lines of work---Smoosh, a formal, mechanized, executable small-step semantics for the POSIX shell---and ffs---a tool for helping users manipulate semi-structured data (like JSON and YAML) in the shell.

Free your mind

Freeing your mind is difficult

Pattern Matching

Pattern matching

```
let rec product nums =
  if (nums = []) then
    1
  else
    (List.head nums)
    * product (List.tail nums)
```

Using **patterns**…

```
let rec product nums =
  match nums with
  | []    -> 1
  | x::xs -> x * product xs
```

Pattern matching

A **pattern** is built from

- **values**,
- (de)**constructors**,
- and **variables**

Tests whether values match “pattern”

If yes, values bound to variables in pattern

Pattern matching

```
let rec product nums =
  if (nums = []) then
    1
  else
    (List.head nums)
    * product (List.tail nums)
```

Using **patterns**…

```
let rec product nums =
  match nums with
  | []     -> 1
  | x::xs -> x * product xs
```

Activity: Pattern matching on integers

Write a function `listOfInts` that returns a list of integers from **zero** to `n`.

```
let rec listOfInts n =
  match n with
  | 0 -> [0]
  | i -> i :: listOfInts (i - 1)
```

Oops! This returns the list backward.

Let’s flip it around.

Revisiting local declarations

Let’s fix our code the lazy way…

```
let listOfInts n =
  let rec li n =
    match n with
    | 0 -> [0]
    | i -> i :: listOfInts (i - 1)
  li n |> List.rev
```

… by defining a function inside our function.

Sidebar: breakpoint debugging

Pattern matching on lists

- Remember, a list is one of two things:
  - `[]`
  - `<first elem> :: <rest of elems>`
  - E.g., `[1; 2; 3] = 1::[2,3] = 1::2::[3] = 1::2::3::[]`
- Can define function by cases…

```
let rec length xs =
  match xs with
  | []    -> 0
  | x::xs -> 1 + length xs
```

Activity: Pattern matching on tuples

Write a **function** that computes the **Cartesian product** of two sets, represented by lists:

$$A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

Hint: I find it helpful to think about **base cases** first.

```
let rec cartesianProduct xs ys =
  match xs,ys with
  | []    ,_   -> []
  | _     ,[] -> []
  | x::xs',_   ->
    let zs = ys |> List.map (fun y -> (x,y))
    zs @ cartesianProduct xs' ys
```

Patterns in declarations

- Patterns can be used in place of variables
- Most basic pattern form
  - `let <pattern> = <exp>`
- Examples
  - `let x = 3`
  - `let tuple = ("moo", "cow")`
  - `let (x,y) = tuple`
  - `let myList = [1; 2; 3]`
  - `let w::rest = myList`
  - `let v::_ = myList`

Algebraic Data Types*

*not to be confused with Abstract Data Types!

Algebraic Data Type

An **algebraic data type** is a composite data type, made by combining other types in one of two different ways:

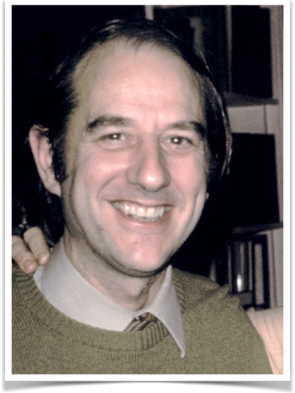
- by **product**, or
- by **sum**.

You've already seen **product types**: tuples and records.

So-called b/c the set of all possible values of such a type is the cartesian product of its component types.

We'll focus on **sum types**.

Algebraic Data Types



- Invented by Rod Burstall at University of Edinburgh in '70s.
- Part of the HOPE programming language.
- Not useful without pattern matching.
- Like peanut butter and chocolate, they are "better together."

A “move” function in a game

A “move” function in a game (Java)

```
public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public … move(int x, int y, int dir) {
  switch (dir) {
    case NORTH: ...
    case ...
  }
}
```

A “move” function in a game (Java)

Discriminated Union (sum type)

```
type Direction =
     North | South | East | West;
let move coords dir =
  match coords,dir with
  |(x,y),North -> (x,y - 1)
  |(x,y),South -> (x,y + 1)
```

- Above is an “incomplete pattern”
- ML will warn you when you’ve missed a case!
  - “proof by exhaustion”

Parameters

```
type Shape =
    | Rectangle of float * float
    | Circle of float
```

- Pattern match to extract parameters

```
let s = Rectangle(1.0,4.0)
match s with
| Rectangle(w,h) -> …
| Circle(r) -> …
```

Named parameters

```
type Shape =
    | Rectangle of width: float * height: float
    | Circle of radius: float
```

- Names are really only useful for initialization, though.

```
let s = Rectangle(height = 1.0, width = 4.0)
```

ADTs can be recursive and generic

```
type MyList<'a> =
      | Empty
      | NonEmpty of head: 'a * tail: MyList<'a>
```

```
> NonEmpty(2, Empty);;
    val it : MyList<int> = NonEmpty (2,Empty)
```

Avoiding errors with patterns

- Another example: handling errors.
- SML has exceptions (like Java)
- But an alternative, easy way to handle many errors is to use the option type:

```
type option<'a> =
| None
| Some of 'a
```

Avoiding errors with patterns

```
let divide quot div =
  match div with
  | 0 -> None
  | _ -> Some (float quot/float div)
```

Avoiding errors with patterns

```
> divide 6 7;;
val it : float option = Some 0.8571428571

> divide 6 0;;
val it : float option = None

>
```

option type

- Why `option`?
- `option` is a data type;

  not handling errors is a static type error!

Recap & Next Class

Today:

Pattern matching
Algebraic data types
Option type

Next class:

Higher order functions