CSE341: Programming Languages

Lecture 5
Pattern-Matching

Dan Grossman
Fall 2011

Review

Datatype bindings and pattern-matching so far:

```
datatype t = C1 of t1 | C2 of t2 | … | Cn of tn
```

Adds type `t` and constructors `Ci` of type `ti->t`

- `Ci v` is a value

```
case e of p1 => e1 | p2 => e2 | … | pn => en
```

- Evaluate `e` to a value
- If `pi` is the first pattern to match the value, then result is evaluation of `ei` in environment extended by the match
- Pattern `Ci(x1,…,xn)` matches value `Ci(v1,…,vn)` and extends the environment with `x1` to `v1` … `xn` to `vn`
- This lecture: many more kinds of patterns and ways to use them

Recursive datatypes

Datatype bindings can describe recursive structures

- Arithmetic expressions from last lecture
- Linked lists, for example:

```
datatype my_int_list = Empty
                     | Cons of int * my_int_list

val x = Cons(4,Cons(23,Cons(2008,Empty)))

fun append_my_list (xs,ys) =
   case xs of
       Empty => ys
     | Cons(x,xs’) => Cons(x, append_my_list(xs’,ys))
```


Options are datatypes

Options are just a predefined datatyping binding

- `NONE` and `SOME` are constructors, not just functions
- So use pattern-matching not `isSome` and `valOf`

```
fun inc_or_zero intoption =
   case intoption of
        NONE => 0
      | SOME i => i+1
```


Lists are datatypes

Don’t use `hd`, `tl`, or `null` either

- `[]` and `::` are constructors too
- (strange syntax, particularly *infix*)

```
fun sum_list intlist =
   case intlist of
        [] => 0
      | head::tail => head + sum_list tail

fun append (xs,ys) =
   case xs of
        [] => ys
      | x::xs’ => x :: append(xs’,ys)
```


Why pattern-matching

- Pattern-matching is better for options and lists for the same reasons as for all datatypes
  - No missing cases, no exceptions for wrong variant, etc.
- We just learned the other way first for pedagogy
- So why are `null` and `tl` predefined then?
  - For passing as arguments to other functions (next week)
  - Because sometimes they’re really convenient
  - But not a big deal: could define them yourself with case

Each-of types

So far have used pattern-matching for one of types because we *needed* a way to access the values

Pattern matching also works for records and tuples:
- The pattern **`(x1,…,xn)`** matches the tuple value **`(v1,…,vn)`**
- The pattern **`{f1=x1, …, fn=xn}`** matches the record value **`{f1=v1, …, fn=vn}`** (and fields can be reordered)

Example

This is poor style, but based on what I told you so far, the only way to use patterns
- Works but poor style to have one-branch cases

```
fun sum_triple triple =
   case triple of
     (x, y, z) => x + y + z

fun sum_stooges stooges =
   case stooges of
     {larry=x, moe=y, curly=z} => x + y + z
```


Val-binding patterns

- New feature: A val-binding can use a pattern, not just a variable
  - (Turns out variables are just one kind of pattern, so we just told you a half-truth in lecture 1)

```
val p = e
```

- This is great for getting (all) pieces out of an each-of type
  - Can also get only parts out (see the book or ask later)
- Usually poor style to put a constructor pattern in a val-binding
  - This tests for the one variant and raises an exception if a different one is there (like **`hd`**, **`tl`**, and **`valOf`**)

Better example

This is reasonable style
- Though we will improve it one more time next
- Semantically identical to one-branch case expressions

```
fun sum_triple triple =
   let val (x, y, z) = triple
   in
       x + y + z
   end

fun sum_stooges stooges =
   let val {larry=x, moe=y, curly=z} = stooges
   in
       x + y + z
   end
```


A new way to go

- For homework 2:
  - Do not use the # character
  - You won't need to write down any explicit types
- These are related
  - Type-checker can use patterns to figure out the types
  - With just **`#foo`** it can't "guess what other fields"

Function-argument patterns

A function argument can also be a pattern
- Match against the argument in a function call

```
fun f p = e
```

Examples:

```
fun sum_triple (x, y, z) =
    x + y + z

fun sum_stooges {larry=x, moe=y, curly=z} =
    x + y + z
```

Hmm

A function that takes one triple of type `int*int*int` and returns an `int` that is their sum:

```
fun sum_triple (x, y, z) =
    x + y + z
```

A function that takes three `int` arguments and returns an `int` that is their sum

```
fun sum_triple (x, y, z) =
    x + y + z
```

See the difference? (Me neither.) ☺

The truth about functions

- In ML, every function takes exactly one argument (*)
- What we call multi-argument functions are just functions taking one tuple argument, implemented with a tuple pattern in the function binding
  - Elegant and flexible language design
- Enables cute and useful things you can't do in Java, e.g.,

```
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left(rotate_left t)
```

* "Zero arguments" is the unit pattern `()` matching the unit value `()`

One-of types in function bindings

As a matter of *taste,* I personally have never loved this syntax, but others love it and you're welcome to use it:

```
fun f p1 = e1
  | f p2 = e2
  …
  | f pn = en
```

Example:

```
fun eval (Constant i) = i
  | eval (Add(e1,e2)) =
       (eval e1) + (eval e2)
  | eval (Negate e1) =
        ~ (eval e1)
```

As a matter of *semantics*, it's syntactic sugar for:

```
fun f x = e1
  case x of
    p1 => e1
  | p2 => e2
    …
```


More sugar

By the way, conditionals are just a predefined datatype and if-expressions are just syntactic sugar for case expressions

```
datatype bool = true | false

if e1 then e2 else e3

case e1 of true => e2 | false => e3
```


Nested patterns

- We can nest patterns as deep as we want
  - Just like we can nest expressions as deep as we want
  - Often avoids hard-to-read, wordy nested case expressions
- So the full meaning of pattern-matching is to compare a pattern against a value for the "same shape" and bind variables to the "right parts"
  - More precise recursive definition coming after examples
- Examples:
  - Pattern `a::b::c::d` matches all lists with >= 3 elements
  - Pattern `a::b::c::[]` matches all lists with 3 elements
  - Pattern `((a,b),(c,d))::e` matches all non-empty lists of pairs of pairs

Useful example: zip/unzip 3 lists

```
fun zip3 lists =
   case lists of
        ([],[],[]) => []
      | (hd1::tl1,hd2::tl2,hd3::tl3) =>
              (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
      | _ => raise ListLengthMismatch

fun unzip3 triples =
   case triples of
        [] => ([],[],[])
      | (a,b,c)::tl =>
          let val (l1, l2, l3) = unzip3 tl
          in
              (a::l1,b::l2,c::l3)
          end
```

More examples in the code for the lecture

(Most of) the full definition

The semantics for pattern-matching takes a pattern *p* and a value *v* and decides (1) does it match and (2) if so, what variable bindings are introduced.

Since patterns can nest, the definition is elegantly recursive, with a separate rule for each kind of pattern. Some of the rules:

- If *p* is a variable *x*, the match succeeds and *x* is bound to *v*
- If *p* is _, the match succeeds and no bindings are introduced
- If *p* is *(p1,…,pn)* and *v* is *(v1,…,vn)*, the match succeeds if and only if *p1* matches *v1*, …, *pn* matches *vn*. The bindings are the union of all bindings from the submatches
- If *p* is *C p1*, the match succeeds if *v* is *C v1* (i.e., the same constructor) and *p1* matches *v1*. The bindings are the bindings from the submatch.
- … (there are several other similar forms of patterns)