CSE341: Programming Languages

Lecture 5
Pattern-Matching

Dan Grossman
Fall 2011
Review

Datatype bindings and pattern-matching so far:

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

Adds type \( t \) and constructors \( C_i \) of type \( t_i \rightarrow t \)
- \( C_i \ v \) is a value

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

- Evaluate \( e \) to a value
- If \( p_i \) is the first pattern to match the value, then result is evaluation of \( e_i \) in environment extended by the match
- Pattern \( C_i(x_1,...,x_n) \) matches value \( C_i(v_1,...,v_n) \) and extends the environment with \( x_1 \) to \( v_1 \) ... \( x_n \) to \( v_n \)
- 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:

```plaintext
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**

```haskell
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*)

```haskell
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 \((x_1, \ldots, x_n)\)
  matches the tuple value \((v_1, \ldots, v_n)\)
- The pattern \(\{f_1=x_1, \ldots, f_n=x_n\}\)
  matches the record value \(\{f_1=v_1, \ldots, f_n=v_n\}\)
  (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

```haskell
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`)

Fall 2011  CSE341: Programming Languages  9
Better example

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

```plaintext
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:

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

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

```latex
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.,

  ```ml
  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

```plaintext
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 \texttt{a::b::c::d} matches all lists with \( \geq 3 \) elements
  – Pattern \texttt{a::b::c::[]} matches all lists with 3 elements
  – Pattern \texttt{( (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 $(p_1,\ldots,p_n)$ and $v$ is $(v_1,\ldots,v_n)$, the match succeeds if and only if $p_1$ matches $v_1$, $\ldots$, $p_n$ matches $v_n$. The bindings are the union of all bindings from the submatches
- If $p$ is $C\ p_1$, the match succeeds if $v$ is $C\ v_1$ (i.e., the same constructor) and $p_1$ matches $v_1$. The bindings are the bindings from the submatch.
- ... (there are several other similar forms of patterns)