CSci 4223
Principles of Programming Languages

Lecture 6

Prof. Clarkson
Spring 2013

Review

- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions
- Today:
  - Type synonyms
  - More datatypes
  - Pattern matching (awesome generalization of case expressions)

---

Type synonym

- A type synonym is a new kind of binding
  - It’s not a datatype binding, just happens to have similar syntax
  - Type name = t
  - Creates another name for a type
  - The type and the name are interchangeable in every way
  - REPL can mix-and-match: might pick one or the other
    - Kind of like it how it is free to reorder field names of records

---

Syntax and semantics of datatype bindings (for now)

- Syntax:

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

- Type-checking rules:
  - If t1, t2, ..., tn are types in current static environment extended with t,
    then t is a type
  - Extend static environment with t and functions C1 of type t1 -> t

- Evaluation rules:
  - For binding itself, none. Types aren’t evaluated.
  - After binding, C1 v is a value
Syntax and semantics of case expressions (for now)

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

- Evaluation rules:
  - Evaluate e to a value
  - If pi is the first pattern to match the value, then evaluate ei to value vi and return vi
  - Evaluation of ei is in dynamic environment extended by the match
    - Pattern matches value if it "looks like" the value
      - Pattern Ci(x1,...,xn) matches value Ci(v1,...,vn) and extends the environment with x1 bound to v1, ... xn to vn
      - Pattern _ matches any value

1. If expressions are just syntactic sugar

- If expressions exist only in the surface syntax of the language
- Early pass in compiler actually replaces them with case expressions, then proceeds to compile the case expression instead:

  ```
  if e0 then e1 else e2
  ```

  becomes...

  ```
  case e0 of true => e1 | false => e2
  ```

  because...

  ```
  datatype bool = false | true
  ```

2. Options are just datatypes

- Options are just a predefined datatyping binding

  ```
  datatype 'a option = NONE | SOME 'a
  ```

  - NONE and SOME are constructors, not just functions
  - "'a" means "any type"; we’ll look at this more carefully in next class

- Better to use pattern-matching than isSome and valOf

  ```
  fun intopt_to_string(x:int option) =
  case x of
    NONE => ""
  | SOME(i) => Int.toString(i)
  ```

3. Lists are just datatypes

- We could have coded up lists ourselves:

  ```
  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:my_int_list,ys:my_int_list) =
  case xs of
    Empty => ys
  | Cons(x,xs') => Cons(x, append_my_list(xs',ys))
  ```

  But much better to reuse well-known, widely-understood implementation ML already provides.
3. Lists are just datatypes

datatype 'a list = nil | :: of 'a * 'a list

fun sum_list (ls : int list) = 
  case ls of
    [] => 0
  | h:t => h + sum_list t

fun append (xs : int list, ys : int list) = 
  case xs of
    [] => ys
  | x:xs' => x :: append(xs', ys)

From now on, won’t use hd, tl, or null to destruct lists

Why use pattern-matching instead of functions?

- 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 learned the other way first only to keep things simple for you
- So why are null, hd, tl predefined then?
  - Because sometimes they’re really convenient (more next week)
  - But could define them yourself with case

Pattern matching each-of types

- One-of types: we introduced pattern-matching because as a way to:
  - discriminate between variants
  - access data carried by variants
- Each-of types: pattern matching also works to access data
  - 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

fun sum_triple (triple:int*int*int) = 
  case triple of
    (x, y, z) => x + y + z

type stooges = {larry:int, moe:int, curly:int}

fun sum_stooges (s:stooges) = 
  case s of
    \{larry=x, moe=y, curly=z\} => x + y + z

- Works, but poor style to have one-branch cases. We can do better...

Val-binding patterns

- New feature: A val-binding can use a pattern
  - (Variables are just one kind of pattern)
  - Access all/some of data inside each of type
- Note:
  - Implies it’s possible to do this (e.g.)
    
    ```
    val [h,t] = lst
    ```
  - Tests for the one variant (cons) and raises an exception if a different one is there (nil) — so it works like hd, tl
  - That’s why it’s a bad idea to use a constructor pattern in a val-binding

Better example

fun sum_triple (triple:int*int*int) = 
  let val \(s\) = triple
  in
    x + y + z
  end

fun sum_stooges (s:stooges) = 
  let val \(s\) = s
  in
    x + y + z
  end

- Semantically identical to one-branch case expressions
- Improved style, but we can do even better...
Function-argument patterns

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

Examples:

```plaintext
fun f p = p
```

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

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

- No argument types; type-checker can infer them
- So sometimes ML programmers omit them; various opinions on whether that's good style

Hmm

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

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

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

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

See the difference? (Me neither) © … more on this next class

For homework 2

- Do not use selectors (e.g., #1, #field, etc.)
  - Use pattern-matching instead
- You won’t need to write down any explicit types
  - But it will certainly help you