CSE 341: Programming Languages

Hal Perkins

Spring 2011

Lecture 5— Pattern-matching, one-argument functions, tail-recursion, accumulators

Review: datatypes and pattern-matching

Evaluation rules for `datatype` bindings and `case` expressions:

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

Adds constructors `Ci` where `Ci v` is a value (and `Ci` has type `ti->t`).

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

- Evaluate `e` to `v`
- If `pi` is the first pattern to *match* `v`, then result is evaluation of `ei` in environment extended by the match.
- If `C` is a constructor of type `t1 * ... * tn -> t`, then `C(x1,...,xn)` is a pattern that matches `C(v1,...,vn)` and the match extends the environment with `x1` to `v1` ... `xn` to `vn`.
- Coming soon: more kinds of patterns.

Expression trees

```
datatype arith_exp = Constant of int
                   | Negate of arith_exp
                   | Add of arith_exp * arith_exp
```

Think of values of type `arith_exp` as trees where nodes are

- `Constant` with one `int` child
- `Negate` with one child that can be any `arith_exp` tree.
- `Add` with two children that can be any `arith_exp` trees.

In general, a type describes a set of values, which are often trees. One-of types give you different variants for nodes.

Constructors evaluate arguments to values (trees) and create bigger values (i.e., taller trees).

Where we're going

So far, case gives us what we *need* to use datatypes:

- A (combined) way to test variants and extract values

In fact, pattern-matching is far more general and elegant:

- Can use it for datatypes already in the top-level environment (e.g., lists and options and bools)
- Can use it for each-of types (today)
- Can have deep (nested) patterns (next time)

Why patterns?

Even without more pattern forms, this design has advantages over functions for "testing and destructing" (e.g., `null`, `hd`, and `tl`):

- easier to check for missing and redundant cases
- more concise syntax by combining "test, destruct, and bind"
- you can easily define testing and destructing in terms of pattern-matching

In fact, case expressions are the preferred way to test variants and extract values for all of ML's "one-of" types, including predefined ones (`[]` and `::` just funny syntax).

So: *Don't* use functions `hd`, `tl`, `null`, `isSome`, `valOf` on homework 2

Teaser: These functions are useful for *passing to other functions*

Tuple/record patterns

You can also use patterns to extract fields from tuples and records: pattern `{f1=x1, ..., fn=xn}` (or `(x1,...,xn)`) matches `{f1=v1, ..., fn=vn}` (or `(v1,...,vn)`).

For record-patterns, field-order does not matter.

This is better style than `#1` and `#foo`, and it means you do not (ever) need to write function-argument types.

Instead of a case with one pattern, better style is a pattern directly in a `val` binding.

- Or a function argument, which is what we have been doing the whole time with (allegedly) multi-argument functions!

Now where are we

Could use a short break from pattern-matching

- Deep (nested) patterns on Friday (along with course motivation)

Rest of today: Tail recursion, accumulators, function-call efficiency

Section tomorrow: Some key features that will come up in minor ways on homework 2:

- type synonyms (e.g., `type card = suit * rank`)
- `'a` and `''a` types and one type being "more general than another" (full lecture on polymorphism later)
- using = for comparing tuples and datatypes
- creating and raising (a.k.a. throwing) exceptions

Recursion

You should now have the hang of recursion:

- It's no harder than using a loop (whatever that is)
- It's much easier when you have multiple recursive calls (e.g., with functions over trees)

But there are idioms you should learn for *elegance*, *efficiency*, and *understandability*.

Today: using an *accumulator*.

Accumulator lessons

- Accumulators can reduce the depth of recursive calls that are not *tail calls*
- Key idioms:
  - Non-accumulator: compute recursive results and combine
  - Accumulator: use recursive result as new accumulator
  - The base case becomes the initial accumulator

You will use recursion in non-functional languages—this lesson still applies.

Tail calls

If the result of `f(x)` is the "immediate result" for the enclosing function body, then `f(x)` is a *tail call*.

More precisely, a tail call is a call in *tail position*:

- In `fun f(x) = e`, `e` is in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (not `e1`). (Similar for case).
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (not any binding expressions).
- Function-call arguments are not in tail position.
- ...

So what?

Why does this matter?

- Implementation takes space proportional to depth of function calls ("call stack" must "remember what to do next")
- But in functional languages, implementation must ensure tail calls eliminate the caller's space
- Accumulators are a systematic way to make some functions tail recursive
- "Self" tail-recursive is very loop-like because space does not grow