CSE 341:
Programming Languages

Autumn 2005

Lecture 3 — Let bindings, pattern preview, options, and benefits of no mutation

Let bindings

Motivation: Functions without local variables can be poor style and/or really inefficient.

Syntax: `let b1 b2 ... bn in e end` where each `bi` is a *binding*.

Typing rules: Type-check each `bi` and `e` in context including previous bindings. Type of whole expression is type of `e`.

Evaluation rules: Evaluate each `bi` and `e` in environment including previous bindings. Value of whole expression is result of evaluating `e`.

Elegant design worth repeating:

- Let-expressions can appear anywhere an expression can.
- Let-expressions can have any kind of binding.
  - Local functions can refer to any bindings *in scope*.

More than style

Exercise: hand-evaluate `bad_max` and `good_max` for lists `[1,2]` `[1,2,3]`, and `[3,2,1]`.

Extra Credit Exercise: As a function of n, how long will it take to calculate

- `bad_max([1, 2, ..., n])`?
- `bad_max([n, n-1, ..., 1])`?

Summary and general pattern

Major progress: recursive functions, pairs, lists, let-expressions

Each has a syntax, typing rules, evaluation rules.

Functions, pairs, and lists are very different, but we can describe them in the same way:

- How do you create values? (function definition, pair expressions, empty-list and `::`)
- How do you use values? (function application, `#1` and `#2`, `null`, `hd`, and `tl`)

Boolean operations

In ML the "and" and "or" operations are named `andalso` and `orelse`.

Example:

```
val x = 10;
val y = 0;
val z = if x>2 andalso y>2 then 3.0 else 4.0;
val w = if x>2 orelse y>2 then 3.0 else 4.0;
```

Patterns – Sneak Preview

In ML patterns provide a useful way of defining functions, often more readable than using conditionals. (You can use them for HW 1 if you like!)

```
(* return the result of reversing a list *)
fun reverse(xs) = if xs=[] then []
                  else reverse(tl(xs)) @ [hd(xs)]

(* definition of reverse using patterns to test for
   the empty list, and also to pick the list apart *)
fun preverse([]) = []
| preverse(x::xs) = preverse(xs) @ [x]
```

Options

Options provide a way of representing a value that might or might not be present.

- Create a `t option` with `NONE` or `SOME e` where `e` has type `t`.
- Use a `t option` with `isSome` and `valOf`

Why not just use a list with zero or one element? An interesting style trade-off:

- Options better express purpose, enforce invariants on callers, maybe faster.
- But cannot use functions on options with lists that are already constructed for some other purpose.