CSE 341:
Programming Languages

Hal Perkins
Spring 2011
Lecture 3—Lack of Mutation, let bindings, options
List Review

- Build lists: [], ::, and shorthand \([e_1,e_2,\ldots,e_n]\]
- Use lists: null, hd, tl
- Types: Each list has elements of the same type. Examples:
  
  int list
  
  (int*int) list
  
  ((int*int) list) list

- So what are the typing rules for [], ::, null, hd, and tl?
- Functions that build or use lists are usually recursive
  - And/or use other recursive functions
  - Elegant algorithms by “thinking high-level” (e.g., append)
Sharing

Recall \( \text{append}([2,4],[5,3,0]) \) evaluates to \([2,4,5,3,0]\).

Similarly, \( t1\ [9,7,4,2] \) evaluates to \([7,4,2]\).

Do the results share, i.e., alias the arguments?

Example: \( \text{val } x=[2,4]; \text{ val } y=[5,3,0]; \text{ val } z=\text{append}(x,y) \)

\[
\begin{align*}
x & \quad 2 \rightarrow 4 \\
y & \quad 5 \rightarrow 3 \rightarrow 0 \\
z & \quad 2 \rightarrow 4 \\
\end{align*}
\]

versus

\[
\begin{align*}
x & \quad 2 \rightarrow 4 \\
y & \quad 5 \rightarrow 3 \rightarrow 0 \\
z & \quad 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 0 \\
\end{align*}
\]
Sharing, good or bad?

Java programmer’s view:

- A never-ending *obsession* with what is shared. This obsession is *necessary* because everything is mutable.

- Sharing is wrong if you don’t want a mutation of “one list” to “affect the other” and right if you do.

- So sometimes make copies just to avoid sharing in case some other code might do a mutation.
Sharing, good or bad?

ML programmer’s view:

- It is actually *impossible* to tell if there is sharing or not!
- So stop worrying and just write append; all lists \([2,4,5,3,0]\) behave the same no matter what they do or do not share with.
- Amount of sharing is just a “space optimization”
  - Usually good to share.
  - \(t\)1 shares, which makes it very fast \(O(1)\).
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.
  - Better style than passing around unchanging arguments.
More than style

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

Moral: Repeating expensive (recursive) computations is not just bad style; it is the wrong algorithm performance-wise.
Options

“Options are like lists that can have at most one element.”

• Create a \( t \) option with NONE or SOME \( e \) where \( e \) has type \( t \).
• Use a \( t \) option with isSome and \( \text{valOf} \)

Why not just use lists? An interesting style trade-off:

• Options better express purpose, enforce invariants on callers, maybe faster.
• But cannot use functions for lists already written.
Summary and general pattern

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

Each has a syntax, typing rules, evaluation rules.

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

- How do you create values?
  - function definition; pair expressions; [] and ::; NONE and SOME

- How do you use values?
  - function application; #1 and #2; null, hd, and tl; isSome and valOf

Soon: much better ways to use pairs and lists (pattern-matching)