CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 3— ML lists, local bindings, and the lack of mutation

Lists

We can have pairs of pairs of pairs... but we still "commit" to the amount of data when we write down a type.

Lists can have *any* number of elements:

- `[]` is the empty list
- More generally, `[v1,v2,...,vn]` is a length n list
- If `e1` evaluates to `v` and `e2` evaluates to a list `[v1,v2,...,vn]`, then `e1::e2` evaluates to `[v,v1,v2,...,vn]`.
- `null e` evaluates to true if and only if `e` evaluates to `[]`
- If `e` evaluates to `[v1,v2,...,vn]`, then `hd e` evaluates to `v1` and `tl e` evaluates to `[v2,...,vn]`.
  - If `e` evaluates to `[]`, a *run-time exception* is raised (this is different than a type error; more on this later)

List types

A given list's elements must all have the same type.

If the elements have type `t`, then the list has type `t list`. Examples: `int list`, `int*int list`, `int list list`.

What are the type rules for `::`, `null`, `hd`, and `tl`?

- Possible exceptions do not affect the type.

Hmmm, that does not explain the type of `[]` ?

- It can have any type, which is indicated via `'a list`.
- That is, we can build a list of any type from `[]`.
- *Polymorphic* types are 3 weeks ahead of us.
  - Teaser: `null`, `hd`, and `tl` are not keywords!

Recursion again

Functions over lists that depend on all list elements will be recursive:

- What should the answer be for the empty list?
- What should they do for a non-empty list? (In terms of answer for the tail of the list.)

Functions that produce lists of (potentially) any size will be recursive:

- When do we create a small (e.g., empty) list?
- How should we build a bigger list out of a smaller one?

Local variables

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

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

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

Meaning: Each `bi` is evaluated and added to the environment for subsequent bindings and `e`.

The whole expression evaluates to whatever `e` evaluates to (and the bindings are not part of any environment outside of the expression).

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*.

Summary and general pattern

Major progress: functions (including recursion), pairs, lists, and 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 bindings, pair expressions, empty-list and `::`)
- How do you use values? (function application, `#1` and `#2`, `null`, `hd`, and `tl`)

This (and conditionals) is enough for your homework though:

- `andalso` and `orelse` help a bit (see section)
- Soon: much better ways to use pairs and lists (pattern-matching)

You want to *change* something?

There is no way to *mutate* (assign to) a binding, pair component, or list element.

How could the *lack* of a feature make programming easier?
In this case:

- Amount of sharing is indistinguishable
  - Aliasing irrelevant to correctness!
- Bindings are invariant across function application
  - Mutation breaks compositional reasoning, a (the?) intellectual tool of engineering