CSE341: Programming Languages

Lecture 3
Local Bindings;
Options;
Benefits of No Mutation

Brett Wortzman
Summer 2019

Slides originally created by Dan Grossman
Review

Huge progress already on the core pieces of ML:

• Types: int bool unit t1*…*tn t list t1*…*tn->t
  – Types “nest” (each t above can be itself a compound type)

• Variables, environments, and basic expressions

• Functions
  – Build: fun x0 (x1:t1, …, xn:tn) = e
  – Use: e0 (e1, …, en)

• Tuples
  – Build: (e1, …, en)
  – Use: #1 e, #2 e, …

• Lists
  – Build: [] e1::e2
  – Use: null e  hd e  tl e
Today

- The big thing we need: local bindings
  - For style and convenience
  - A big but natural idea: nested function bindings
  - For efficiency (**not** “just a little faster”)

- One last feature for Problem 11 of Homework 1: options

- Why **not having mutation** (assignment statements) is a valuable language feature
  - No need for you to keep track of sharing/aliasing, which Java programmers must obsess about
Let-expressions

3 questions:

• Syntax: \texttt{let b1 b2 \ldots bn in e end}
  – Each \texttt{bi} is any \textit{binding} and \texttt{e} is any \textit{expression}

• Type-checking: Type-check each \texttt{bi} and \texttt{e} in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of \texttt{e}.

• Evaluation: Evaluate each \texttt{bi} and \texttt{e} in a dynamic environment that includes the previous bindings.
  Result of whole let-expression is result of evaluating \texttt{e}.
It is an expression

A let-expression is *just an expression*, so we can use it *anywhere* an expression can go
Silly examples

```haskell
fun silly1 (z : int) =
  let val x = if z > 0 then z else 34
      val y = x+z+9
  in
    if x > y then x*2 else y*y
  end
fun silly2 () =
  let val x = 1
  in
    (let val x = 2 in x+1 end) +
    (let val y = x+2 in y+1 end)
  end
```

*silly2* is poor style but shows let-expressions are expressions
  - Can also use them in function-call arguments, if branches, etc.
  - Also notice shadowing
What’s new

• What’s new is **scope**: where a binding is in the environment
  – *In* later bindings and body of the let-expression
    • (Unless a later or nested binding shadows it)
    – *Only in* later bindings and body of the let-expression

• **Nothing else is new:**
  – Can put any binding we want, even function bindings
  – Type-check and evaluate just like at “top-level”
Any binding

According to our rules for let-expressions, we can define functions inside any let-expression

\[
\texttt{let } b_1 \ b_2 \ \ldots \ b_n \ \texttt{in} \ e \ \texttt{end}
\]

This is a natural idea, and often good style
(Inferior) Example

```plaintext
fun countup_from1 (x : int) = 
    let fun count (from : int, to : int) = 
        if from = to 
        then to :: [] 
        else from :: count(from+1,to) 
    in 
        count (1,x) 
    end
```

• This shows how to use a local function binding, but:
  – Better version on next slide
  – `count` might be useful elsewhere
Better:

```plaintext
fun countup_from1_better (x : int) = 
  let fun count (from : int) = 
    if from = x 
    then x :: [] 
    else from :: count(from+1) 
  in 
    count 1 
  end
```

• Functions can use bindings in the environment where they are defined:
  – Bindings from “outer” environments
  • Such as parameters to the outer function
  – Earlier bindings in the let-expression

• Unnecessary parameters are usually bad style
  – Like `to` in previous example
Nested functions: style

• Good style to define helper functions inside the functions they help if they are:
  – Unlikely to be useful elsewhere
  – Likely to be misused if available elsewhere
  – Likely to be changed or removed later

• A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later
Avoid repeated recursion

Consider this code and the recursive calls it makes

- Don’t worry about calls to `null`, `hd`, and `tl` because they do a small constant amount of work

```haskell
fun bad_max (xs : int list) = 
  if null xs
  then 0 (* horrible style; fix later *)
  else if null (tl xs)
    then hd xs
  else if hd xs > bad_max (tl xs)
    then hd xs
  else bad_max (tl xs)

let x = bad_max [50,49,...,1]
let y = bad_max [1,2,...,50]
```
Fast vs. unusable

if \texttt{hd \hspace{1pt} xs} > \texttt{bad\_max (tl \hspace{1pt} xs)}
then \texttt{hd \hspace{1pt} xs}
else \texttt{bad\_max (tl \hspace{1pt} xs)}
Math never lies

Suppose one `bad_max` call’s if-then-else logic and calls to `hd`, `null`, `tl` take $10^{-7}$ seconds

- Then `bad_max [50, 49, ..., 1]` takes $50 \times 10^{-7}$ seconds
- And `bad_max [1, 2, ..., 50]` takes $1.12 \times 10^8$ seconds
  - (over 3.5 years)
  - `bad_max [1, 2, ..., 55]` takes over 1 century
  - Buying a faster computer won’t help much 😊

The key is not to do repeated work that might do repeated work that might do…

- Saving recursive results in local bindings is essential…
Efficient max

```ml
fun good_max (xs : int list) =
  if null xs
  then 0 (* horrible style; fix later *)
  else if null (tl xs)
    then hd xs
  else
    let val tl_ans = good_max(tl xs)
    in
      if hd xs > tl_ans
      then hd xs
      else tl_ans
    end
```
Fast vs. fast

let val tl_ans = good_max(tl xs) in
  if hd xs > tl_ans then hd xs else tl_ans end
Options

• `t option` is a type for any type `t`
  – (much like `t list`, but a different type, not a list)

Building:
• `NONE` has type `'a option` (much like `[]` has type `'a list`)
• `SOME e` has type `t option` if `e` has type `t` (much like `e::[]`)

Accessing:
• `isSome` has type `'a option` -> `bool`
• `valOf` has type `'a option` -> `'a` (exception if given `NONE`)
fun better_max (xs : int list) = 
  if null xs 
  then NONE 
  else 
    let val tl_ans = better_max(tl xs) 
    in 
      if isSome tl_ans 
      andalso valOf tl_ans > hd xs 
      then tl_ans 
      else SOME (hd xs) 
    end 

val better_max = fn : int list -> int option

- Nothing wrong with this, but as a matter of style might prefer not to do so much useless "valOf" in the recursion
Example variation

fun better_max2 \((xs : \text{int list})\) =
  if null xs
  then NONE
  else let (* ok to assume \(xs\) nonempty b/c local *)
    fun max_nonempty \((xs : \text{int list})\) =
      if null (tl xs)
      then hd xs
      else let val tl_ans = max_nonempty (tl xs)
       in
       if hd xs > tl_ans
       then hd xs
       else tl_ans
       end
       in
       SOME (max_nonempty xs)
  end
Cannot tell if you copy

In ML, these two implementations of `sort_pair` are indistinguishable

- But only because tuples are immutable
- The first is better style: simpler and avoids making a new pair in the then-branch
- In languages with mutable compound data, these are different!
Suppose we had mutation…

```scala
val x = (3, 4)
val y = sort_pair x

somehow mutate #1 x to hold 5

val z = #1 y
```

• What is \( z \)?
  – Would depend on how we implemented `sort_pair`
    • Would have to decide carefully and document `sort_pair`
  – But without mutation, we can implement “either way”
    • No code can ever distinguish aliasing vs. identical copies
    • No need to think about aliasing: focus on other things
    • Can use aliasing, which saves space, without danger
An even better example

```ocaml
fun append (xs : int list, ys : int list) = 
  if null xs
  then ys
  else hd (xs) :: append (tl(xs), ys)
val x = [2,4]
val y = [5,3,0]
val z = append(x,y)
```

(x can’t tell, but it’s the first one)

```
x    2 ——— 4
y    5 ——— 3 ——— 0
z    2 ——— 4
```

```
x    2 ——— 4
y    5 ——— 3 ——— 0
z    2 ——— 4 ——— 5 ——— 3 ——— 0
```
ML vs. Imperative Languages

• In ML, we create aliases all the time without thinking about it because it is *impossible* to tell where there is aliasing
  – Example: \texttt{tl} is constant time; does not copy rest of the list
  – So don’t worry and focus on your algorithm

• In languages with mutable data (e.g., Java), programmers are *obsessed* with aliasing and object identity
  – They have to be (!) so that subsequent assignments affect the right parts of the program
  – Often crucial to make copies in just the right places
    • Consider a Java example…
Java security nightmare (bad code)

class ProtectedResource {
    private Resource theResource = ...;
    private String[] allowedUsers = ...;
    public String[] getNotAllowedUsers() {
        return allowedUsers;
    }
    public String currentUser() { ... }
    public void useTheResource() {
        for(int i=0; i < allowedUsers.length; i++) {
            if(currentUser().equals(allowedUsers[i])) {
                ... // access allowed: use it
                return;
            }
        }
        throw new IllegalAccessException();
    }
}
Have to make copies

The problem:

```java
p.getAllowedUsers()[0] = p.currentUser();
p.useTheResource();
```

The fix:

```java
public String[] getAllowedUsers() {
    ... return a copy of allowedUsers ... 
}
```

Reference (alias) vs. copy doesn’t matter if code is immutable!