CSE341: Programming Languages
Lecture 3
Local bindings, Options, Benefits of No Mutation
Dan Grossman
Fall 2011

Review
Huge progress in 2 lectures on the core pieces of SML:
• Types: int bool unit $t_1 \ldots t_n$ $t$ list $t_1 \ldots t_n \rightarrow t$
  – Types "nest" (each $t$ above can be itself a compound type)
• Variables and environments
• Functions
  – Build: $\text{fun } x_0 \ (x_1 : t_1, \ldots, x_n : t_n) = e$
  – Use: $e_0 \ (e_1, \ldots, e_n)$
• Tuples
  – Build: $(e_1, \ldots, e_n)$
  – Use: $\#_1 e$, $\#_2 e$, ...
• Lists
  – Build: $[]$, $e_1 :: e_2$
  – Use: null $e$, hd $e$, tl $e$

Today
• The big thing we need: local bindings
  – For style and convenience
  – For efficiency (not "just a little faster")
  – A big but natural idea: nested function bindings
• One last feature for last problem 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
The construct for introducing local bindings is just an expression, so we can use it anywhere we can use an expression
3 questions:
• Syntax: $\text{let } b_1 b_2 \ldots b_n \ \text{in } e \ \text{end}$
  – Each $b_i$ is any binding and $e$ is any expression
• Type-checking: Type-check each $b_i$ and $e$ in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of $e$.
• Evaluation: Evaluate each $b_i$ and $e$ in a dynamic environment that includes the previous bindings.
  Result of whole let-expression is result of evaluating $e$.

Silly examples
fun silly1 (z : int) =
  let val x = if $z > 0$ then $z$ else 34
  val $y = x + 9$
  in
    if $x > y$ then $x ^ 2$ else $y * y$
  end
fun silly2 () =
  let val $x = 1$
  in
    $(\text{let } \text{val } x = 2 \ \text{in } x + 1 \ \text{end}) +$
    $(\text{let } \text{val } y = x + 2 \ \text{in } y + 1 \ \text{end})$
  end

silly2 is poor style but shows let-expressions are expressions
• Could also use them in function-call arguments, parts of conditionals, 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”
Nested functions, part 1

• 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

(Inferior) Example

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:
  – Will show a better version next
  – count might be useful elsewhere

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

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

let x = bad_max [50,49,…,1]  
let y = bad_max [1,2,…,50]

Fast vs. unusable

| bm [50,...] | bm [49,...] | bm [48,...] | bm [1] |
| bm [1,...] | bm [2,...] | bm [3,...] | bm [50] |
| bm [2,...] | bm [3,...] | bm [3,...] | bm [3,...] | bm [50] |

Math never lies

Consider the code and recursive calls it makes:
  – Don’t worry about calls to null, hd, and tl because they do a small constant amount of work

>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

• This shows how to use a local function binding, but:
  – Will show a better version next
  – count might be useful elsewhere

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

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

let x = bad_max [50,49,…,1]  
let y = bad_max [1,2,…,50]

Fast vs. unusable

| bm [50,...] | bm [49,...] | bm [48,...] | bm [1] |
| bm [1,...] | bm [2,...] | bm [3,...] | bm [50] |
| bm [2,...] | bm [3,...] | bm [3,...] | bm [3,...] | bm [50] |

Math never lies

Consider the code and recursive calls it makes:
  – Don’t worry about calls to null, hd, and tl because they do a small constant amount of work

>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

• This shows how to use a local function binding, but:
  – Will show a better version next
  – count might be useful elsewhere
**Efficient max**

```haskell
fun good_max (lst : int list) = 
  if null lst 
  then 0 (* horrible style; fix later *) 
  else if null (tl lst) 
  then hd lst 
  else 
    let val tl_ans = good_max(tl lst) 
    in 
      if hd lst > tl_ans 
      then hd lst 
      else tl_ans 
    end
```

**Fast vs. fast**

```haskell
let val tl_ans = good_max(tl lst) 
  in 
    if hd lst > tl_ans 
    then hd lst 
    else tl_ans 
  end
```

**Options**

Having `good_max` return 0 for the empty list is really awful
- Could raise an exception (see section this week)
- Could return a zero-element or one-element list
  - That works but is poor style because the built-in support for `options` expresses this situation directly
    - `t option` is a type for any type `t`
      - (much like `t list`, but a different type, not a list)
    Building:
      - `NONE` has type `t option`
      - `SOME e` has type `t option` if `e` has type `t` (much like `e::[]`)
    Accessing:
      - `isSome` has type `t option -> bool`
      - `valOf` has type `t option -> t` (exception if given `NONE`)

**Example**

```haskell
fun better_max (lst : int list) = 
  if null lst 
  then NONE 
  else 
    let val tl_ans = better_max(tl lst) 
    in 
      if isSome tl_ans 
      andalso valOf tl_ans > hd lst 
      then tl_ans 
      else SOME (hd lst) 
    end
```

**A valuable non-feature: no mutation**

Those are all the features you need (and should use) on hw1

Now learn a very important non-feature
- Huh?? How could the *lack* of a feature be important?
- When it lets you know things other code will *not* do with your code and the results your code produces

A major aspect and contribution of functional programming:

**Example variation**

```haskell
fun better_max2 (lst : int list) = 
  if null lst 
  then NONE 
  else let (* ok to assume lst nonempty b/c local *) 
    fun max_nonempty (lst : int list) = 
      if null (tl lst) 
      then hd lst 
      else 
        let val tl_ans = max_nonempty(tl lst) 
        in 
          if hd lst > tl_ans 
          then hd lst 
          else tl_ans 
        end 
    in 
      SOME (max_nonempty lst) 
    end
```
Suppose we had mutation...

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

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

Interface vs. implementation

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 Java, you make copies like the second one all the time

```scala
fun sort_pair (pr : int * int) =
  if #1 pr > #2 pr
    then pr
  else (#2 pr, #1 pr)

fun sort_pair (pr : int * int) =
  if #1 pr > #2 pr
    then (#1 pr, #2 pr)
  else (#2 pr, #1 pr)
```

An even better example

```scala
fun append (lst1 : int list, lst2 : int list) =
  if null lst1
    then lst2
  else hd (lst1) :: append (tl(lst1), lst2)

val x = [2,4]
val y = [5,3,0]
val z = append(x,y)
```

```
x → 2 4
y → 5 3 0
z → 2 4
```

( can’t tell, but it’s the first one)

ML vs. Java on mutable data

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

Java security nightmare (bad code)

```java
class ProtectedResource {
  private Resource theResource = ...;
  private String[] allowedUsers = ...;
  public String[] getAllowedUsers() {
    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 IllegalAccessExcpetion();
  }
}
```

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 ... 
}
```