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 t1*…*tn t list t1*…*tn->t`**
  - Types "nest" (each **`t`** above can be itself a compound type)
- Variables and environments
- 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
  - 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: `let b1 b2 … bn in e end`
  - Each ***bi*** is any *binding* and ***e*** is any *expression*
- Type-checking: Type-check each ***bi*** and ***e*** in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of ***e***.
- Evaluation: Evaluate each ***bi*** 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
        (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

- 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

Nested functions, better

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

- Usually bad style to have unnecessary parameters
  - Like `to` in the previous example

```
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
```

Avoid repeated recursion

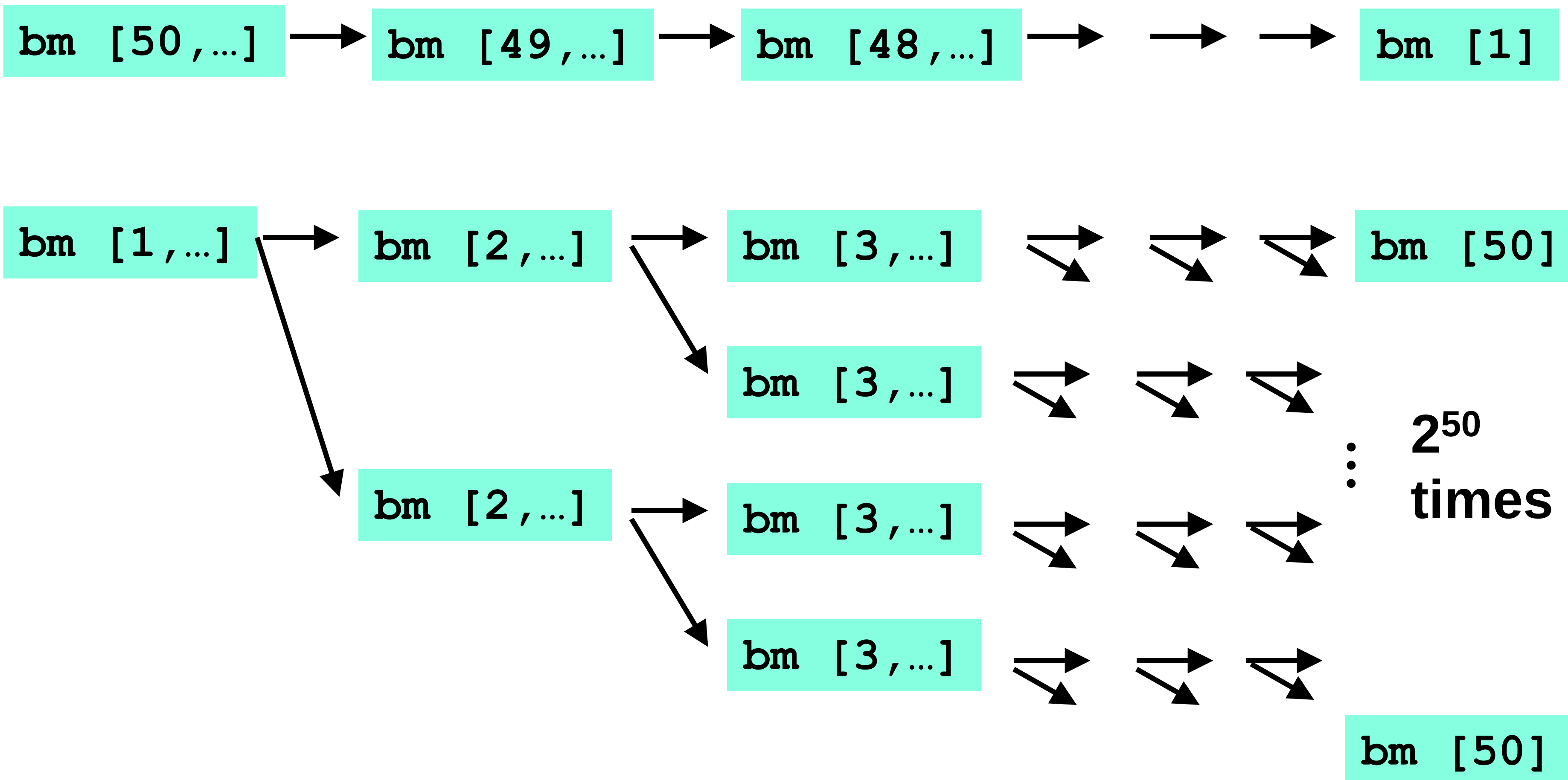
Consider this code and the recursive calls it makes

- 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

```
if hd lst > bad_max (tl lst)
then hd lst
else bad_max (tl lst)
```



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 x 10^{-7} seconds
- And **`bad_max [1,2,…,50]`** takes 2.25 x 10^{8} seconds
  - (over 7 years)
  - **`bad_max [55,54,…,1]`** takes over 2 centuries
  - 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

```
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

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

gm [50,…] → gm [49,…] → gm [48,…] → → → gm [1]

gm [1,…] → gm [2,…] → gm [3,…] → → → gm [50]

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 **`'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`**)

Example

```
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
```

```
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 (lst : int list) =
    if null lst
    then NONE
    else let (* ok to assume lst nonempty b/c local *)
             fun max_nonempty (lst : int lst) =
                 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
```

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:

Not being able to assign to (a.k.a. mutate) variables or parts of tuples and lists

Suppose we had mutation…

```
val x = (4,3)
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

Interface vs. implementation

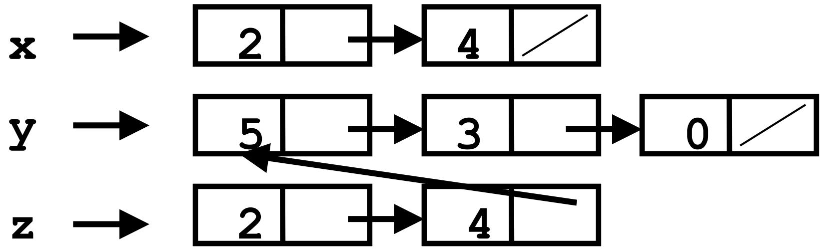
In ML, these two implementations of **`sort_pair`** are indistinguishable

- 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

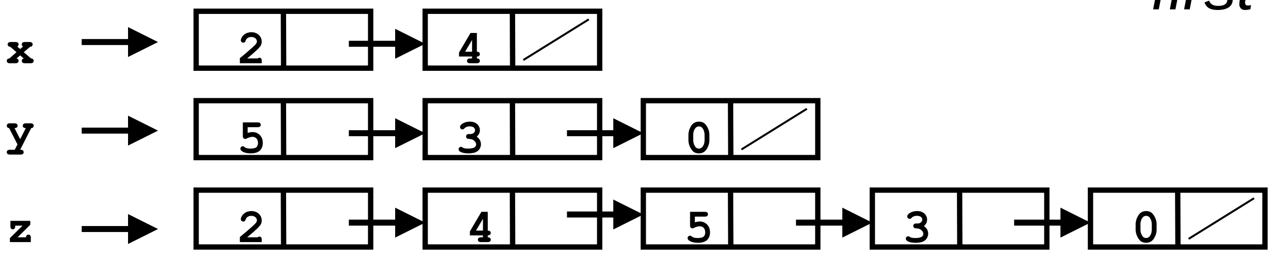
```
fun sort_pair (pr : int * int) =
  if #1 pr > #2 pr
  then pr
  else (#2 pr, #1 pr)

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

```
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)
```



or



(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)

```
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:

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

The fix:

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