Lecture 08: Programming Languages [CMPU-235]
September 26, 2017

Sameer S Pradhan
Vassar College

Currying

Curying and *partial* application

- Recall every ML function takes exactly one argument
- Previously encoded n arguments via one n-tuple
- Another way: Take one argument and return a function that takes another argument and...
  - Called "currying" after famous logician Haskell Curry

vs.

Haskell B. Curry

1900-1982

Curry-Howard isomorphism

Types are *logical formulas*
Programs are *logical proofs*

```
fn x => x : 'a -> 'a
```

Currying and *partial* application

Currying is much prettier than we have indicated so far

- Can write `e1 e2 e3 e4` in place of `((e1 e2) e3) e4`
- Can write `fun f x y z = e` in place of

  `fun f x = fn y => fn z => e`

```
fun sorted3 x y z = z >= y andalso y >= x
val true_ans = sorted3 7 9 11
val is_non_negative = sorted3 0 0
```

Result is a little shorter and prettier than the tupled version:

```
fun sorted3 (x,y,z) = z >= y andalso y >= x
val true_ans = sorted3(7,9,11)
fun is_non_negative x = sorted3(0,0,x)
```

Syntactic Sugar

In addition to being sufficient multi-argument functions and pretty, currying is useful because partial application is convenient

Example: Often use higher-order functions to create other functions

```
fun fold f acc xs =
    case xs of
      []      => acc
    | x::xs'  => fold f (f(acc,x)) xs'

fun sum xs = fold (fn (x,y) => x+y) 0 xs
```

Return to the *fold*

- So the SML standard library is fond of currying iterators
  - See types for **`List.map`**, **`List.filter`**, **`List.foldl`**, etc.
  - So calling them as though arguments are tupled won't work

- Another example is **`List.exists`**:

```
fun exists predicate xs =
    case xs of
      []      => false
    | x::xs' => predicate xs
                 orelse exists predicate xs'

val no = exists (fn x => x=7) [4,11,23]
val has_seven = exists (fn x => x=7)
```

Another example

Currying and partial application can be convenient even without higher-order functions

```
fun zip xs ys =
   case (xs,ys) of
        ([],[]) => []
      | (x::xs',y::ys') => (x,y)::(zip xs' ys')
      | _ => raise Empty

fun range i j =
    if i>j then [] else i :: range (i+1) j

val countup = range 1 (* partial application *)

fun add_number xs = zip (countup (length xs)) xs
```

More combining functions

- What if you want to curry a tupled function or vice-versa?
- What if a function's arguments are in the wrong order for the partial application you want?

Naturally, it's easy to write higher-order wrapper functions

– And their types are neat logical formulas

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

Scope—Lexical, Dynamic

Scope and Shadowing

A brain-teaser for scope and shadowing:

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- Line 2 defines a function that, when called, evaluates body **`x+y`** in environment where **`x`** maps to **`1`** and **`y`** maps to the argument
- Call on line 5:
  - Looks up **`f`** to get the function defined on line 2
  - Evaluates **`x+y`** in current environment, producing **7**
  - Calls the function, which evaluates the body in the old environment, producing **8**

Scoping

- We know function bodies can use any bindings in scope
- But now that functions can be passed around: In scope where?

When the function was defined
(not when it was called)

- There are lots of good reasons for this semantics
  - Discussed after explaining what the semantics is
- For HW, exams, and competent programming, you must "get this"
- This semantics is called *lexical scope*

Lexical vs Dynamic Scope

Rule of lexical scope: The body of a function is evaluated in the old dynamic environment that existed at the time the function was **defined**, not the current environment when the function is called.

Rule of dynamic scope: The body of a function is evaluated in the current dynamic environment at the time the function is **called**, not the old dynamic environment that existed at the time the function was defined.

(In both, environment is extended to map function argument to passed value.)

Closures

How can functions be evaluated in old environments that aren't around anymore?

- The language implementation keeps them around as necessary

Can define the semantics of functions as follows:

- A function value has two parts
  - The code (obviously)
  - The environment that was current when the function was defined
- This is a "pair" but unlike ML pairs, you cannot access the pieces
- All you can do is call this "pair"
- This pair is called a *function closure*
- A call evaluates the code part in the environment part (extended with the function argument)

Example of Lexical Scope

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- Line 2 creates a closure and binds **f** to it:
  - Code: "argument **y** and body **x+y**"
  - Environment: "**x** maps to **1**"
    - (Plus whatever else is in scope, including **f** for recursion)
- Line 5 calls that closure with **7** as argument
  - In function body, **x** maps to **1** and **y** maps to **7**
- So **z** is bound to **8**

Another Example of Lexical Scope

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Evaluating line 4 binds to **g** to a closure:
  - Code: "argument **z** and body **x+y+z**"
  - Environment: "**y** maps to **4**, **x** maps to **5** (shadowing), ..."
  - So this closure will always add **9** to its argument
- So line 6 binds **15** to **z**

Another Example of Lexical Scope

```
(* 1 *) fun f g =
(* 1a *)     let val x = 3
(* 1b *)     in g 2 end
(* 2 *) val x = 4
(* 3 *) fun h y = x + y
(* 4 *) val z = f h
```

- Evaluating line 3 binds **h** to a closure:
  - Code: "argument **y** and body **x+y**"
  - Environment: "**x** maps to **4**, **f** maps to another closure, ..."
  - So **h**'s closure will always add **4** to its argument
- So line 4 binds **6** to **z**
- Line 1a is irrelevant (and bad style)

Example of Dynamic Scope

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- At line 5, current dynamic environment maps **x** to **3** and **y** to **4**
- Line 5 calls **f** with argument **7**
  - body of **f** is evaluated in current dynamic environment, with **y** bound to argument value **7**
  - So **x** is **3** and **y** is **7**
  - And result is **10**
- Finally, **z** is bound to **10**

Another Example of Dynamic Scope

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Line 4 calls **`f`** with argument **`4`**
  - Current dynamic environment maps **`x`** to **`3`**
  - Body of **`f`** is evaluated in current dynamic environment at line 4, extended to map **`y`** to **`4`**
  - **`let`** binding on 2a extends dynamic environment to map **`x`** to **`4`**
  - Body of **`let`** is a function, and functions are values. So body evaluates just to itself (**`fn z => x+y+z`**). That's what is returned.
  - So line 4 binds **`g`** to (**`fn z => x+y+z`**).

Another Example of Dynamic Scope

```
(* 1 *) fun f g =
(* 1a *)     let val x = 3
(* 1b *)     in g 2 end
(* 2 *) val x = 4
(* 3 *) fun h y = x + y
(* 4 *) val z = f h
```

- Line 4 calls **f** with argument **h**
  - Body of **f** is evaluated in current dynamic environment, which maps **x** to **4**.
  - **let** binding shadows **x**, makes it bind to **3**.
  - Body of **let** calls **g** with argument **2**
  - But **g** is bound to **h**, so **h** is called with argument **2**

Another Example of Dynamic Scope

```
(* 1 *) fun f g =
(* 1a *)     let val x = 3
(* 1b *)     in g 2 end
(* 2 *) val x = 4
(* 3 *) fun h y = x + y
(* 4 *) val z = f h
```

- Line 4 calls **f** with argument **h**
  - ...
  - But **g** is bound to **h**, so **h** is called with argument **2**
    - Body of **h** is evaluated in current dynamic environment, which maps **x** to **3**
    - So body of **h** evaluates to **3+2** which is **5**
  - Body of **f** therefore evaluates to **5**
- Finally, **z** is bound to **5**
- So an changing an unused binding can change result of computation! ☹

Semantcs of function calls

```
e0 e1
```

Evaluation rule:

1. In current dynamic environment, evaluate `e0` to a function closure. The code part of the closure takes an argument; call it `x`.
2. In current dynamic environment, evaluate argument `e1` to value `v`.
3. Take environment part of closure. Extend it to map `x` to `v`. Call the resulting dynamic environment E.
4. Evaluate code part of closure in dynamic environment E, resulting in a value `v'`.
5. Return `v'` as the result of the function call.

Why Lexical Scope?

1. Function meaning does not depend on variable names used

Example:

- Lexical scope: Can change body to use **q** instead of **x** without changing final value of **z**

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

Why Lexical Scope?

1. Function meaning does not depend on variable names used

Example:

- Lexical scope: Can change body to use **q** instead of **x** without changing final value of **z**
- Dynamic scope: Run-time error; **q** isn't in scope at line 6!

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val q = y+1
(* 2b *)     in fn z => q+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6  (* error! *)
```

q isn't bound in dynamic environment at call site

Why Lexical Scope?

2. Functions can be type-checked & reasoned about where defined

Example:

- Lexical scope: Could change line 3 to `val x = "hi"` without changing result of computation

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

Why Lexical Scope?

2. Functions can be type-checked & reasoned about where defined

Example:

- Lexical scope: Could change line 3 to `val x = "hi"` without changing result of computation

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = "hi"
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

Why Lexical Scope?

2. Functions can be type-checked & reasoned about where defined

Example:

- Lexical scope: Could change line 3 to `val x = "hi"` without changing result of computation
- Dynamic scope: Run-time error; can't add string to int!

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = "hi"
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6   (* Error! *)
```

Dynamic environment at call site binds `x` to string, so code at 2b tries to add string to int.

Why Lexical Scope?

3. Unused bindings can be eliminated

(see Example 3, earlier)

Why Lexical Scope?

4. Closures can easily store the data they need

```
fun greaterThanX x = fn y => y > x
fun noNegatives xs = filter(greaterThanX ~1, xs)
```

Does Dynamic Scope Exist?

- Lexical scope for variables is definitely the right default
  - Very common across languages
- Dynamic scope is occasionally convenient in some situations
  - So some languages (e.g., Perl, Racket) have special ways to do it
  - In some languages, it's the norm (e.g., Emacs LISP, LaTeX)
  - But most languages just don't have it
- If you squint some, exception handling is like dynamic scope:
  - **`raise e`** transfers control to the current innermost handler
  - Does not have to be syntactically inside a **`handle`** expression (and usually isn't)

A note on closures and performance: Recomputation

These both work and rely on using variables in the environment

```
fun allShorterThan1 (xs,s) =
    filter(fn x => String.size x < String.size s,
           xs)

fun allShorterThan2 (xs,s) =
    let val i = String.size s
    in filter(fn x => String.size x < i, xs) end
```

The first one computes **`String.size`** once per element of **`xs`**

The second one computes **`String.size s`** once per list

- Nothing new here: let-bindings are evaluated when encountered and function bodies evaluated when *called*