CS 3110

Lecture 8: Closures

Prof. Clarkson

Fall 2014

Today's music: Selections from *Doctor Who* soundtracks, series 5-7

Review

Dynamic semantics:

- how expressions evaluate
- *substitution model*: substitute value for variable in let expressions, function calls, etc.
- *environment model*: maintain a data structure that binds variables to values

Today:

- semantics of function calls in environment model

Question #1

How much of PS2 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I'm done!!!

Review: the core of OCaml

Essential sublanguage of OCaml:

```
e ::= c | (op) | x | (e1, …, en)
      | C e
      | e1 e2
      | fun x -> e
      | let x = e1 in e2
      | match e0 with pi -> ei
```

Missing, unimportant: records, lists, options, declarations, patterns in function arguments and let bindings, `if`

Missing, important: `rec`

Review: evaluation

- Expressions evaluate to values in a dynamic environment

```
env :: e --> v
```

- Evaluation is meaningless if expression does not type check
- Values are a *syntactic subset* of expressions:

```
v ::= c | (op) | (v1, …, vn)
      | C v
      | fun x -> e
```

Review: function values

Anonymous functions **`fun x-> e`** are values

```
env :: (fun x -> e) --> (fun x -> e)
```

Review: let expressions

To evaluate `let x = e1 in e2` in environment **`env`**

Evaluate the binding expression **`e1`** to a value **`v1`** in environment **`env`**

```
env :: e1 --> v1
```

Extend the environment to bind **`x`** to **`v1`**

```
env' = env + {x=v1}
```

(newer bindings temporarily *shadow* older bindings)

Evaluate the body expression **`e2`** to a value **`v2`** in environment **`env'`**

```
env' :: e2 --> v2
```

Return `v2`

Function application v1.0

To evaluate `e1 e2` in environment **`env`**

Evaluate `e2` to a value **`v2`** in environment **`env`**

```
env :: e2 --> v2
```

Note: right to left order, like tuples, which matters in the presence of side effects

Evaluate `e1` to a value **`v1`** in environment **`env`**

```
env :: e1 --> v1
```

Note that **`v1`** *must be a function value* **`fun x -> e`**
*because function application type checks**

Extend environment to bind formal parameter **`x`** to actual value **`v2`**

```
env' = env + {x=v2}
```

Evaluate body **`e`** to a value **`v`** in environment **`env'`**

```
env' :: e --> v
```

Return `v`

*Or a built-in operator **`(op)`**. In which case, immediately apply **`(op)`** to **`v2`** and return result.

Function application rule v1.0

If `env :: e2 --> v2`

and `env :: e1 --> (fun x -> e)`

and `env + {x=v2} :: e --> v`

then `env :: e1 e2 --> v`

Function application example

```
let f = fun x -> x in f 0 --> 0
```

1. Evaluate binding expression `fun x->x` to a value in empty environment
   – (already is a value)
2. Extend environment to bind `f` to `fun x->x`
3. Evaluate let-body expression `f 0` in environment
   `env = {f=fun x->x}`
   1. Evaluate argument `0` to a value
      - (already is a value)
   2. Evaluate `f` to a value
      - By variable rule, `f` evaluates to `env(f)`, i.e., `fun x->x`
   3. Extend environment to map formal parameter `x` to actual value `0`, i.e., `env' = {f=(fun x->x), x=0}`
   4. Evaluate function body `x` to value
      - By variable rule, `x` evaluates to `env'(x)`, i.e., `0`
   5. Return `0`

Function application example

`let f = fun x -> x in f 0 --> 0`

Another way of expressing the previous slide:

1. By function value rule,
   `{} :: fun x -> x --> fun x -> x`
2. By constant rule, **`{f = fun x -> x} :: 0 --> 0`**
3. By variable rule,
   `{f = fun x -> x} :: f --> fun x-> x`
4. By variable rule, **`{f=fun x -> x, x=0} :: x --> 0`**
5. By function application rule with 2 and 3 and 4,
   `{f=fun x -> x} :: f 0 --> 0`
6. By **`let`** rule with 1 and 5,
   `let f = fun x -> x in f 0 --> 0`

Hard example

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
  f 0
```

What does our dynamic semantics say it evaluates to?

What does OCaml say?

What do YOU say?

Question #2

What do you think this expression should evaluate to?

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
  f 0
```

A. 1

B. 2

Hard example: OCaml

What does OCaml say this evaluates to?

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
  f 0
- : int = 1
```

Hard example: our semantics

What does our semantics say?

```
let x = 1 in
{x=1} let f = fun y -> x in
{x=1,f=fun y->x} let x = 2 in
  {x=2,f=fun y->x} f 0
```

```
{x=2,f=fun y->x} :: f 0 --> ?
```

1. Evaluate `0` to a value, i.e., `0`
2. Evaluate `f` to a value, i.e., `fun y->x`
3. Extend environment to map parameter:
   `{x=2, f=(fun y->x), y=0}`
4. Evaluate body `x` in that environment
5. Return `2`

`2 <> 1`

Why different answers?

Two different rules for variable scope:

- Rule of *dynamic scope* (our semantics)
- Rule of *lexical scope* (OCaml)

Dynamic scope

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.

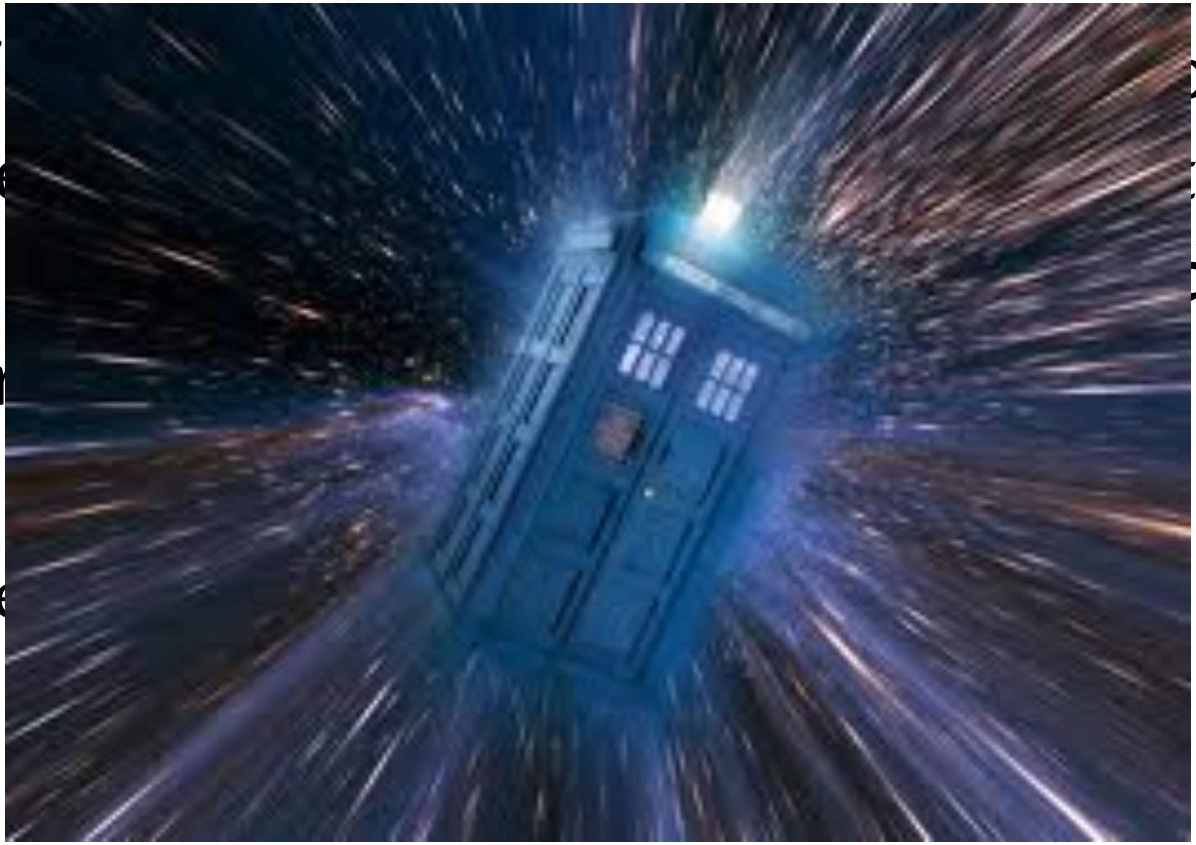
- Causes our semantics to use latest binding of **`x`**
- Thus return 2

Lexical 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.

- Causes OCaml to use earlier binding of **x**
- Thus return 1

Lexical scope

Rule of  on is evaluate hat existed **d**, not the curr is called.

- Cause
- Thus

Scope

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.

– Causes our semantics to use latest binding of **x**
– Thus return 2

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.

– Causes OCaml to use earlier binding of **x**
– Thus return 1

(In both, environment is extended to map formal parameter to actual value.)

Why would you want one vs. the other? Let's come back to that...

Implementing time travel

Q: How can functions be evaluated in old environments?
A: The language implementation keeps them around as necessary

- A function value is really a data structure that has two parts:
  - The code (obviously)
  - The environment that was current when the function was defined
    - Gives meaning to all the *free variables* of the function body
  - Like a "pair"
    - But you cannot access the pieces, or directly write one down in the language syntax
    - All you can do is call it
  - This data structure is called a *function closure*
- A function application:
  - evaluates the code part of the closure
  - in the environment part of the closure
  - extended to bind the function argument

Hard example revisited

```
(* 1 *) let x = 1
(* 2 *) let f = fun y -> x
(* 3 *) let x = 2
(* 4 *) let z = f 0
```

With lexical scope:

- Line 2 creates a closure and binds `f` to it:
  - Code: `fun y -> x`
  - Environment: `{x=1}`
- Line 4 calls that closure with `0` as argument
  - In function body, `y` maps to `0` and `x` maps to `1`
- So `z` is bound to `1`

Question #3

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

What value does **z** have with lexical scope?

A. 1

B. 5

C. 7

D. 8

E. 10

Question #3

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

- Line 2 creates a closure and binds `f` to it:
  - Code: `fun y -> x+y`
  - Environment: `{x=1}`
- 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`

Question #3

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

What value does **z** have with lexical scope?

A. 1

B. 5

C. 7

D. 8

E. 10

Question #4

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

What value does **z** have with **dynamic** scope?

A. 1

B. 5

C. 7

D. 8

E. 10

Question #4

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

- At line 5, environment is **`{x=3,y=4}`**
- Line 5 calls **`f`** with argument **`7`**
  - body of **`f`** is evaluated in current environment,
    - but with **`y`** bound to argument value **`7`**
    - argument binding shadows previous binding
  - So **`x`** is **`3`** and **`y`** is **`7`** and result of call is **`10`**
- Finally, **`z`** is bound to **`10`**

Question #4

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

What value does **z** have with dynamic scope?

A. 1

B. 5

C. 7

D. 8

E. 10

Closure notation

```
<<code, environment>>
```

e.g.,

```
<<fun y -> x+y, {x=1}>>
```

With lexical scoping, well-typed programs are guaranteed never to have any variables in the code body other than function argument and variables bound by closure environment.

Function application v2.0

To evaluate **e1 e2** in environment **env**

Evaluate **e2** to a value **v2** in environment **env**

```
env :: e2 --> v2
```

Evaluate **e1** to a value **v1** in environment **env**

```
env :: e1 --> v1
```

Note that **v1** *must be a function closure* **<<fun x -> e, env'>>***

Extend closure environment to bind formal parameter **x** to actual value **v2**

```
env'' = env' + {x=v2}
```

Evaluate body **e** to a value **v** in environment **env''**

```
env'' :: e --> v
```

Return **v**

*Or a built-in operator **(op)**. In which case, immediately apply **(op)** to **v2** and return result.

Function application rule v2.0

```
If env :: e2 --> v2
and env :: e1 -->
      <<fun x -> e,env'>>
and env' + {x=v2} :: e --> v
then env :: e1 e2 --> v
```

Function values v2.0

Anonymous functions **`fun x-> e`** are closures

```
env :: (fun x -> e) -->
              <<fun x -> e, env>>
```

Lexical vs. dynamic scope

- Consensus after decades of programming language design is that **lexical scope is the right choice**

- Dynamic scope is convenient in some situations
  - Some languages use it as the norm (e.g., Emacs LISP, LaTeX)
  - Some languages have special ways to do it (e.g., Perl, Racket)
  - But most languages just don't have it

- Exception handling resembles dynamic scope:
  - **`raise e`** transfers control to the "most recent" exception handler
  - like how dynamic scope uses "most recent" binding of variable

Why lexical scope?

1. Programmer can freely change names of local variables

```
(* 1 *) let x = 1
(* 2 *) let f y =
          let x = y + 1 in
          fun z -> x+y+z
(* 3 *) let x = 3
(* 4 *) let w = (f 4) 6
```

Lexical scope: evaluates to 15
Dynamic scope: evaluates to 13

```
(* 1 *) let x = 0
(* 2 *) let f y =
          let q = y + 1 in
          fun z -> q+y+z
(* 3 *) let x = 3
(* 4 *) let w = (f 4) 6
```

Lexical scope: evaluates to 15
Dynamic scope: run-time error

- (f 4) --> fun z -> q+y+z
- (fun z -> q+y+z) 6 --> q+y+6
- at line 4, env. doesn't bind q

Why lexical scope?

2. Type checker can prevent run-time errors

```
(* 1 *) let x = 1
(* 2 *) let f y =
          let x = y + 1 in
          fun z -> x+y+z
(* 3 *) let x = 3
(* 4 *) let w = (f 4) 6
```

Lexical scope: evaluates to 15
Dynamic scope: evaluates to 13

```
(* 1 *) let x = 0
(* 2 *) let f y =
          let x = y + 1 in
          fun z -> x+y+z
(* 3 *) let x = "hi"
(* 4 *) let w = (f 4) 6
```

Lexical scope: evaluates to 15
Dynamic scope: run-time error

- (f 4) --> fun z -> x+y+z
- (fun z -> x+y+z) 6 --> x+y+6
- at line 4, env. binds x to string; can't add string to int

Progress

```
e ::= c | (op) | x | (e1, …, en)
     | C e
     | e1 e2
     | fun x -> e
     | let x = e1 in e2
     | match e0 with pi -> ei
```

But we still don't have **`let rec`**

Recursive functions

```
e ::= c | (op) | x | (e1, …, en)
    | C e
    | e1 e2
    | fun x -> e
    | let x = e1 in e2
    | match e0 with pi -> ei
    | let rec f x = e1 in e2
```

Let rec expressions

To evaluate `let rec f x = e1 in e2` in environment **`env`**

don't evaluate the binding expression **`e1`**

Extend the environment to bind **`f`** to a *recursive closure*

```
env' = env +
   {f=<<f, fun x -> e1, env>>}
```

Evaluate the body expression **`e2`** to a value **`v2`** in environment **`env'`**

```
   env' :: e2 --> v2
```

Return `v2`

Function application v3.0

To evaluate `e1  e2` in environment **`env`**

Evaluate **`e2`** to a value **`v2`** in environment **`env`**

```
env :: e2 --> v2
```

Evaluate **`e1`** to a value **`v1`** in environment **`env`**

```
env :: e1 --> v1
```

Note that **`v1`** *must be a recursive closure* **`cl=<<f, fun x -> e, env'>>`**
or a closure **`<<fun x -> e, env'>>`**

Extend closure environment to bind formal parameter **`x`** to actual value **`v2`** and (if present) function name **`f`** to the closure

```
env'' = env' + {x=v2,f=cl}
```

That's where the recursion happens: name is bound to "itself" inside call

Evaluate body **`e`** to a value **`v`** in environment **`env''`**

```
env'' :: e --> v
```

Return **`v`**

Closures in OCaml

```
clarkson@chardonnay ~/share/ocaml-4.02.0/
bytecomp
$ grep Kclosure *.ml
bytegen.ml:          (Kclosure(lbl, List.length
fv) :: cont)
bytegen.ml:            (Kclosurerec(lbls,
List.length fv) ::
emitcode.ml:  | Kclosure(lbl, n) -> out
opCLOSURE; out_int n; out_label lbl
emitcode.ml:  | Kclosurerec(lbls, n) ->
instruct.ml:  | Kclosure of label * int
instruct.ml:  | Kclosurerec of label list * int
printinstr.ml:  | Kclosure(lbl, n) ->
printinstr.ml:  | Kclosurerec(lbls, n) ->
```

Closures in OCaml

- *Closure conversion* is an important phase of compiling many functional languages
- Expands on ideas we've seen here
  - Many optimizations possible
  - Especially, better handling of recursive functions

Closures in Java

- Nested classes can simulate closures
  - Used everywhere for Swing GUI! http://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html#innerClasses
  - You've done it yourself already in 2110
- Java 8 adds higher-order functions and closures
- Can even think of OCaml closures as resembling Java objects:
  - closure has a single method, the code part, that can be invoked
  - closure has many fields, the environment part, that can be accessed

Closures in C

- In C, a *function pointer* is just a code pointer, period. No environment.
- To simulate closures, a common **idiom**:

  Define function pointers to take an extra, explicit environment argument

  - But without generics, no good choice for type of list elements or the environment
  - Use `void*` and various type casts…
- From Linux kernel: http://lxr.free-electrons.com/source/include/linux/kthread.h#L13

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Upcoming events

- **PS2 is due tonight at 11:59 pm**
- Clarkson permanent(?) office hours: Tuesday & Thursday 3-4 pm

This is closure.

THIS IS 3110