CS 3110

Lecture 8: Closures

Prof. Clarkson
Spring 2015

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

Have your registered your iClicker for this semester?

A. Oops...
B. Not sure
C. Yes

https://atcsupport.cit.cornell.edu/pollsrvc/
iClicker data

• What gets recorded: "serial number XYZ voted with button W"
  – so the raw data is all there...

• What we need to give you credit for those votes: map from NetID to serial numbers

• Registration is what gives us that map!

• Suggestion: write down all the serial numbers you use so that even if you lose remote, we can give you credit
Review: the core of OCaml

Essential sublanguage of OCaml:

$$e ::= v \mid \text{C } e \mid (e_1, \ldots, e_n) \mid e_1 + e_2 \mid x \mid e_1 \ e_2 \mid \text{let } x = e_1 \ \text{in } e_2 \mid \text{match } e_0 \ \text{with } p_i \rightarrow e_i$$

$$v ::= \text{c} \mid \text{fun } x \rightarrow e \mid \text{C } v \mid (v_1, \ldots, v_n)$$

In recitation, pared this down even further to tuples/datatypes with only two components/constructors
Match expressions

To evaluate

```latex
\text{match } e_0 \text{ with } \\\nn \hspace{1cm} p_1 \rightarrow e_1 \\
| \ldots \\
| p_n \rightarrow e_n
```
in environment $\text{env}$

Evaluate expression $e_0$ to value $v_0$ in $\text{env}$

Find the first pattern $p_i$ that matches $v_0$

That match produces new bindings $b$

i.e., $v_0 = p_i\{v_1/x_1\}\{v_2/x_2\}\ldots \{v_n/x_n\}$

and $b = \{x_1=v_1, x_2=v_2, \ldots, x_n=v_n\}$

Evaluate expression $e_i$ to value $v_i$ in environment $\text{env}+b$

Return $v_i$
**Match expression rule**

\[
\text{env} :: \text{match e0 with pi -> ei || vi}
\]

if \(\text{env} :: e0 \mid \mid v0\)
and \(pi\) is the first pattern to match \(v0\)
and that match produces bindings \(b\)
and \(\text{env}+b :: ei \mid \mid vi\)

Example:
\[
\{\} :: \text{match 42 with x -> x || 42}
\]
because \(\{\} :: 42 \mid \mid 42\)
and \(x\) is the first pattern that matches \(42\)
and that match produces binding \(\{x=42\}\)
and \(\{x=42\} :: x \mid \mid 42\)
Progress

e ::=  v | C e | (e₁, ..., en) | e₁ + e₂
    | x | e₁ e₂
    | let x = e₁ in e₂
    | match e₀ with pi -> ei

v ::=  c | fun x -> e | C v | (v₁, ..., vn)
Review: function values

Anonymous functions \texttt{fun x -> e} are values

\texttt{env :: (fun x -> e) || (fun x -> e)}
Review: let expressions

To evaluate \( \text{let } x = e_1 \text{ in } e_2 \) in environment \( \text{env} \)

Evaluate the binding expression \( e_1 \) to a value \( v_1 \) in environment \( \text{env} \)

\[
\text{env} :: e_1 \ || \ v_1
\]

Extend the environment to bind \( x \) to \( v_1 \)

\[
\text{env'} = \text{env} + \{ x=v_1 \}
\]

(newer bindings temporarily \textit{shadow} older bindings)

Evaluate the body expression \( e_2 \) to a value \( v_2 \) in environment \( \text{env'} \)

\[
\text{env'} :: e_2 \ || \ v_2
\]

Return \( v_2 \)
Review: let vs. application

These two expressions mean the same thing:

•  let x = e1 in e2
•  (fun x -> e2) e1
Function application v1.0

To evaluate \( e_1 \ e_2 \) in environment \( env \)
Evaluate \( e_1 \) to a value \( v_1 \) in environment \( env \)
\[
env :: e_1 \mid \mid v_1
\]
Note that \( v_1 \) must be a function value \( \text{fun} \ x \rightarrow e \)
because function application type checks
Evaluate \( e_2 \) to a value \( v_2 \) in environment \( env \)
\[
env :: e_2 \mid \mid v_2
\]
Extend environment to bind formal parameter \( x \) to actual value \( v_2 \)
\[
env' = env + \{ x=v_2 \}
\]
Evaluate body \( e \) to a value \( v \) in environment \( env' \)
\[
env' :: e \mid \mid v
\]
Return \( v \)
Function application rule v1.0

\[
env :: e_1 \; e_2 \iff v \\
\text{if } env :: e_1 \iff (\text{fun } x \rightarrow e) \\
\text{and } env :: e_2 \iff v_2 \\
\text{and } env+\{x=v_2\} :: e \iff v
\]

Example:

\[
\{\} :: (\text{fun } x \rightarrow x) \; 1 \iff 1 \\
b/c \{\} :: (\text{fun } x \rightarrow x) \iff (\text{fun } x \rightarrow x) \\
\text{and } \{\} :: 1 \iff 1 \\
\text{and } \{\}+\{x=1\} :: x \iff 1
\]
Hard example

```ocaml
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?

```ocaml
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?

```ocaml
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
```

```ocaml
{x=2,f=(fun y->x)} :: f 0 || ???
```

1. Evaluate $f$ to a value, i.e., $\text{fun y->x}$
2. Evaluate $0$ to a value, i.e., $0$
3. Extend environment to map parameter:
   $\{x=2, f=(\text{fun y->x}), y=0\}$
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 so far)
• 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 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 OCR to use earlier binding of x.
- Thus return 1.
**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
  – Code+env is 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

```ocaml
(* 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` bound to 0 and `x` bound to 1
- So `z` ends up being bound to 1
Question #3

What value does \( z \) have with lexical scope?

A. 1  
B. 5  
C. 7  
D. 8  
E. 10
Question #3

\[
\begin{align*}
(* 1 *) \ & \text{let } x = 1 \\
(* 2 *) \ & \text{let } f \ y = x + y \\
(* 3 *) \ & \text{let } x = 3 \\
(* 4 *) \ & \text{let } y = 4 \\
(* 5 *) \ & \text{let } z = f \ (x + y)
\end{align*}
\]

- 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, \( x \) bound to 1 and \( y \) bound 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

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
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}>>

N.B. Can't write this in OCaml syntax
Function application v2.0

To evaluate $e_1$ $e_2$ in environment $env$

Evaluate $e_1$ to a value $v_1$ in environment $env$

$$env :: e_1 || v_1$$

*Note that $v_1$ must be a function closure $<<\text{fun } x \rightarrow e, env'>>$*

Evaluate $e_2$ to a value $v_2$ in environment $env$

$$env :: e_2 || v_2$$

Extend closure environment to bind formal parameter $x$ to actual value $v_2$

$$env'' = env' + \{ x = v_2 \}$$

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

$$env'' :: e || v$$

Return $v$
Function application rule v2.0

\[
\begin{align*}
\text{env} &::= \text{e1 e2} \mid \mid \text{v} \\
\quad &\text{if } \text{env} ::= \text{e1} \mid |

\quad &\langle\langle \text{fun x -\to e}, \text{env'} \rangle\rangle \\
\quad &\text{and } \text{env} ::= \text{e2} \mid \mid \text{v2} \\
\quad &\text{and } \text{env'} + \{x=v2\} ::= \text{e} \mid \mid \text{v}
\end{align*}
\]
Anonymous functions $\text{fun } x \rightarrow e$ are closures

$\text{env} :: (\text{fun } x \rightarrow e) ||$

$<<\text{fun } x \rightarrow e, \text{env}}>>$
Lexical vs. dynamic scope

• Consensus after decades of programming language design is that **lexical scope is the right choice**
  – programmers free to change names of local variables
  – type checker can prevent more run-time errors

• 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
Progress

\[ e ::= v \mid C\ e \mid (e_1, \ldots, e_n) \mid e_1 + e_2 \]
\[ \mid x \mid e_1 \ e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]
\[ \mid \text{match } e_0 \text{ with } \pi \rightarrow e_i \]

\[ v ::= c \mid \text{fun } x \rightarrow e \mid C\ v \mid (v_1, \ldots, v_n) \]

(and there's now a special kind of value, a closure, that can't appear in programs but does get produced during evaluation)
Closures in OCaml

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

To evaluate `let rec f x = e1 in e2` in environment `env`
do\’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`
To evaluate $e_1 \ e_2$ in environment $env$

Evaluate $e_1$ to a value $v_1$ in environment $env$

$$env :: e_1 \mid \mid v_1$$

Note that $v_1$ must be a recursive closure $cl=<<f, \ \text{fun} \ x \rightarrow e, \ env'>>$

or a closure $<<\text{fun} \ x \rightarrow e, \ env'>>$

Evaluate $e_2$ to a value $v_2$ in environment $env$

$$env :: e_2 \mid \mid v_2$$

Extend closure environment to bind formal parameter $x$ to actual value $v_2$ and (if present) function name $f$ to the closure

$$env'' = env' + \{x=v_2, f=cl\}$$

That’s where the recursion happens: name is bound to “itself” inside call

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

$$env'' :: e \mid \mid v$$

Return $v$