Closures

Prof. Clarkson
Fall 2015

Today’s music: Selections from Doctor Who soundtracks by Murray Gold
Review

Previously in 3110:
• Interpreters: ASTs, evaluation, parsing
• Formal syntax: BNF
• Formal semantics:
  – dynamic: small-step substitution model
  – static semantics

Today:
• More formal dynamic semantics: large-step, environment model
Prelim 1

- Next Thursday, 5:30-7 and 7:30-9
- For further details, see Piazza @852
Review: big-step semantics

• *Big-step semantics:* we model just the reduction from the original expression to the final value

• Judgement is written $e \implies v$
  read as $e$ *takes a big step to* $v$

• **Goal:** $e \implies v$ if and only if $e \longrightarrow^* v$
Variables

• What does a variable name evaluate to?
  \[ x \Rightarrow ??? \]

• Trick question: we don’t have enough information to answer it

• Need to know what value variable was bound to
  – e.g., \textbf{let} \( x = 2 \) \textbf{in} \( x+1 \)
  – e.g., \textbf{(fun} \( x \rightarrow x+1 \) \textbf{)} 2
  – e.g., \textbf{match} 2 \textbf{with} \( x \rightarrow x+1 \)
  – All evaluate to 3, but we reach a point where we need to know binding of \( x \)

• Until now, \textbf{we've never needed this}, because we always \textbf{substituted} before we ever get to a variable name
Variables

OCaml doesn't actually do substitution

\[(\text{fun } x \rightarrow 42) \ 0\]

waste of runtime resources to do substitution inside 42

Instead, OCaml lazily substitutes by maintaining

*dynamic environment*
Dynamic environment

- Dictionary of bindings of all current variables
- Changes throughout evaluation:
  - No bindings at $:
    $\text{let x = 42 in}$
    $\text{let y = false in}$
    $e$
  - One binding $[x=42]$ at $:$
    $\text{let x = 42 in}$
    $\text{let y = false in}$
    $e$
  - Two bindings $[x=42, y=false]$ at $:$
    $\text{let x = 42 in}$
    $\text{let y = false in}$
    $e$
Variable evaluation

To evaluate $x$ in environment $env$

Look up value $v$ of $x$ in $env$

Return $v$

Type checking guarantees that variable is bound, so we can’t ever fail to find a binding in dynamic environment
Evaluation judgement

Extended notation:
\(<env, e> \Rightarrow v\)

Meaning: in dynamic environment $env$, expression $e$ takes a big step to value $v$

$<env, e>$ is called a machine configuration
Variable evaluation

<env, x> ==> v
if v = env(x)

env(x):
• meaning: the value to which env binds x
• think of it as looking up x in dictionary env
Redo: evaluation with environment

\(<\text{env}, \, v\> \implies v\)

\(<\text{env}, \, e_1 + e_2\> \implies v
if \text{ \textless env, \, e_1\textgreater} \implies i_1
and \text{ \textless env, \, e_2\textgreater} \implies i_2
and v is the result of
primitive operation \, i_1 + i_2
Let expressions

To evaluate \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ in } \texttt{environment } env

Evaluate the binding expression \texttt{e_1} to a value \texttt{v_1} in environment \texttt{env}

\[ \langle env, e_1 \rangle =\to v_1 \]

Extend the environment to bind \texttt{x} to \texttt{v_1}

\[ env' = env[x\rightarrow v_1] \quad \text{new notation} \]

Evaluate the body expression \texttt{e_2} to a value \texttt{v_2} in extended environment \texttt{env'}

\[ \langle env', e_2 \rangle =\to v_2 \]

Return \texttt{v_2}
Let expression evaluation rule

\(<env, \text{let } x=e1 \text{ in } e2> \Rightarrow v2\) 
if \(<env, e1> \Rightarrow v1\) 
and \(<env[x->v1], e2> \Rightarrow v2\)

Example: (let [] be the empty environment)

\(<[],\text{let } x = 42 \text{ in } x> \Rightarrow 42\)

Because...

• \(<[], 42> \Rightarrow 42\)
• and \(<[][x->42], x> \Rightarrow 42\)
  – Because \([x=42](x)=42\)
Function values v1.0

Anonymous functions are values:

\(<env, \text{fun} \ x \ \rightarrow \ e> \implies \text{fun} \ x \ \rightarrow \ e>\)
To evaluate $e_1 \ e_2$ in environment $env$

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

$\langle env, e_1 \rangle \Rightarrow v_1$

Note that $v_1$ must be a function value $\text{fun} \ x -> e$

because function application type checks

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

$\langle env, e_2 \rangle \Rightarrow v_2$

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 $env'$

$\langle env', e \rangle \Rightarrow v$

Return $v$
Function application rule v1.0

\[ <\text{env}, e_1 \ e_2> \implies v \]
\[ \text{if } <\text{env}, e_1> \implies \text{fun } x \rightarrow e \]
\[ \text{and } <\text{env}, e_2> \implies v_2 \]
\[ \text{and } <\text{env}[x\rightarrow v_2], e> \implies v \]

Example:
\[ <[] , (\text{fun } x \rightarrow x) \ 1> \implies 1 \]
\[ \text{b/c } <[] , \text{fun } x \rightarrow x> \implies \text{fun } x \rightarrow x \]
\[ \text{and } <[] , 1> \implies 1 \]
\[ \text{and } <[ ] [x\rightarrow1] , x> \implies 1 \]
Scope

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?
What do you think this expression should evaluate to?

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

A. 1
B. 2
Scope: 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
```
Scope: 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 \( \mathbf{f} \) to a value, i.e., \( \mathbf{fun} \ y \rightarrow \mathbf{x} \)
2. Evaluate \( \mathbf{0} \) to a value, i.e., \( \mathbf{0} \)
3. Extend environment to map parameter:
   \[ [x=2, \ f=(fun \ y->x), \ y=0] \]
4. Evaluate body \( \mathbf{x} \) 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
– Thus return 1
Lexical vs. dynamic scope

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

• Dynamic scope is useful 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
Implementing time travel

Q: How can functions be evaluated in old environments?

A: The language implementation keeps old environments around as necessary
Implementing time travel

A function value is really a data structure that has two parts:
- The code (obviously), an expression `e`
- The environment `env` that was current when the function was defined
- We'll notate that data structure as `{e | env}

```
{e | env}  is like a pair
```
- But you cannot write OCaml syntax to access the pieces
- And you cannot directly write it in OCaml syntax

This data structure is called a function closure
("and that my friend is what they call closure")
To evaluate $e_1$  $e_2$  in environment $env$
Evaluate $e_1$ to a value $v_1$ in environment $env$
  $<env,e_1> \Rightarrow v_1$
  *Note that $v_1$ must be a function closure* \{fun $x$  $->$  $e$  |  defenv\}
Evaluate $e_2$ to a value $v_2$ in environment $env$
  $<env,e_2> \Rightarrow v_2$
Extend closure environment to bind formal parameter $x$ to actual value $v_2$
  $env' = defenv[x->v_2]$
Evaluate body $e$ to a value $v$ in environment $env'$
  $<env',e> \Rightarrow v$
Return $v$
Function application rule v2.0

\(<env, e_1 \ e_2> \Longrightarrow v\)
    if \(<env, e_1> \Longrightarrow \{fun\ x \rightarrow e \mid defenv\}\"
    and \(<env, e_2> \Longrightarrow v_2\"
    and \(<defenv[x\rightarrow v_2], e> \Longrightarrow v\)
Function values v2.0

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

\[
<\text{env}, \text{fun x -> e}>
\]

\[
===> \{\text{fun x -> e} \mid \text{env}\}
\]
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:  | Kclosure of label list * int
printinstr.ml: | Kclosure(lbl, n) ->
printinstr.ml: | Kclosurerec(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
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](http://lxr.free-electrons.com/source/include/linux/kthread.h#L13)
Upcoming events

• [today] A3 due
• [Mon, Tue] Fall Break
• [Wed] Prelim 1 review
• [Thu am] lecture cancelled
• [Thu pm] Prelim 1 at 5:30 and 7:30 pm

This is closure.

THIS IS 3110