Closures

Prof. Clarkson
Fall 2016

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
Review: Dynamic semantics

Two different models of evaluation:

• **Small-step substitution model:** substitute value for variable in body of `let` expression
  – And in body of function, since `let x = e1 in e2` behaves the same as `(fun x -> e2) e1`
  – What we’ve done so far; good mental model for evaluation
  – Not efficient: too much substitution at run time
  – Not really what OCaml does

• **Big-step environment model:** keep a data structure around that binds variables to values
  – What we’ll do now; also a good mental model
  – At the heart of what OCaml really does
New evaluation judgement

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

• Suppose $e \longrightarrow e' \longrightarrow \ldots \longrightarrow v$

• We'll abstract that fact to $e \Longrightarrow v$
  – forget about all the intermediate expressions
  – new notation means $e$ evaluates (down) to $v$, equiv. $e$ takes a big step to $v$
  – textbooks use down arrows: $e \Downarrow v$

• **Goal**: $e \Longrightarrow v$ if and only if $e \longrightarrow^* v$
  – Another 4110 theorem
Values

• Values are already done:
  – Evaluation rule: $v \implies v$

• Constants are values
  – 42 is a value, so $42 \implies 42$
  – true is a value, so $true \implies true$
Operator evaluation

\[ e_1 + e_2 \implies v \]

if \( e_1 \implies i_1 \)

and \( e_2 \implies i_2 \)

and \( v \) is the result of primitive operation \( i_1 + i_2 \)

e.g.,

\[ \text{true} \land \text{false} \implies \text{false} \]

\[ 1 + 2 \implies 3 \]

\[ 1 + (2+3) \implies 6 \]
Variables

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

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

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

• Until now, \texttt{we've never needed this}, because we always \texttt{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 \text{ in}
    \text{let } y = \text{false in}
    e
  – One binding \([x=42]\) at $:
    \text{let } x = 42 \text{ in}
    $ \text{let } y = \text{false in}
    e
  – Two bindings \([x=42, y=\text{false}]\) at $:
    \text{let } x = 42 \text{ in}
    \text{let } y = \text{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:
\(<env, e> \implies v\)

Meaning: in dynamic environment \texttt{env}, expression \texttt{e} takes a big step to value \texttt{v}

\(<env, e>\) is called a \textit{machine configuration}\n
Variable evaluation

\(<env, \ x> \implies v\)

\(if \ v = \ env(x)\)

\(env(x)\):

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

\[\langle \text{env}, \text{v}\rangle \implies \text{v} \]

\[\langle \text{env}, \text{e}_1 + \text{e}_2\rangle \implies \text{v} \]

\[
\text{if } \langle \text{env}, \text{e}_1\rangle \implies \text{i}_1
\]

\[
\text{and } \langle \text{env}, \text{e}_2\rangle \implies \text{i}_2
\]

\[\text{and } \text{v} \text{ is the result of primitive operation } \text{i}_1+\text{i}_2\]
Let expressions

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

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

\[
<\texttt{env}, e_1> \implies v_1
\]

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

\[
\texttt{env}' = \texttt{env}[x\rightarrow v_1]
\]

new notation

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

\[
<\texttt{env}', e_2> \implies v_2
\]

Return \( v_2 \)
Let expression evaluation rule

\[<\text{env}, \text{let } x=e_1 \text{ in } e_2> \implies v_2 \]
if \[<\text{env}, e_1> \implies v_1\]
and \[<\text{env}[x\rightarrow v_1], e_2> \implies v_2\]

Example: (let [] be the empty environment)

\[<[], \text{let } x = 42 \text{ in } x> \implies 42\]

Because...

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

Anonymous functions are values:

<env, fun x -> e> ==> fun x -> e
To evaluate \( e_1 \ e_2 \) in environment \( \text{env} \)

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

\(<\text{env}, e_1> \implies v_1\)

Note that \( v_1 \) must be a function value \( \text{fun} \ x \to e \) because function application type checks

Evaluate \( e_2 \) to a value \( v_2 \) in environment \( \text{env} \)

\(<\text{env}, e_2> \implies v_2\)

Extend environment to bind formal parameter \( x \) to actual value \( v_2 \)

\( \text{env'} = \text{env}[x->v_2] \)

Evaluate body \( e \) to a value \( v \) in environment \( \text{env'} \)

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

Return \( v \)
Function application rule v1.0

\( \langle \text{env}, e_1 \ e_2 \rangle \implies v \)

if \( \langle \text{env}, e_1 \rangle \implies \text{fun } x \rightarrow e \)
and \( \langle \text{env}, e_2 \rangle \implies v_2 \)
and \( \langle \text{env} [x \rightarrow v_2], e \rangle \implies v \)

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

```ocaml
g  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
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 \(f\) 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 \texttt{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: name of variable shouldn't matter to meaning of program
  – 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, an expression e
- The environment env that was current when the function was defined
- We'll notate that data structure as \{e \mid \text{env}\}

\{e \mid \text{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
Function application v2.0

orange = changed from v1.0

To evaluate e1 e2 in environment env
Evaluate e1 to a value v1 in environment env
  <env,e1> ==> v1
  Note that v1 must be a function closure {fun x -> e | defenv}
Evaluate e2 to a value v2 in environment env
  <env,e2> ==> v2
Extend closure environment to bind formal parameter x to actual value v2
  env' = defenv[x->v2]
Evaluate body e to a value v in environment env'
  <env',e> ==> v
Return v
Function application rule v2.0

\[
\langle \text{env}, \ e_1 \ e_2 \rangle \implies v \\
\text{if } \langle \text{env}, \ e_1 \rangle \implies \{ \text{fun } x \rightarrow e \mid \text{defenv} \} \\
\text{and } \langle \text{env}, \ e_2 \rangle \implies v_2 \\
\text{and } \langle \text{defenv}[x\rightarrow v_2], \ e \rangle \implies v
\]
Function values v2.0

Anonymous functions \texttt{fun } x \rightarrow e \text{ are closures:}

\begin{align*}
\langle \texttt{env}, \texttt{fun x \rightarrow e} \rangle \\
\Rightarrow \{ \texttt{fun x \rightarrow e} \mid \texttt{env} \}
\end{align*}
Closures in OCaml

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) ->
instruct.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
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
Upcoming events

• [Friday] MS0 due, no late submissions

This is closure.

THIS IS 3110