The Environment Model

Nate Foster
Spring 2018
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

• **Small-step substitution model:** substitute value for variable in body of `let` and functions
  – Good mental model
  – Not efficient: too much substitution at run time

• **Big-step environment model:** maintain a dictionary that binds variables to values
  – What OCaml really does
New evaluation relation

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

• Suppose $e \rightarrow e' \rightarrow \ldots \rightarrow v$
• Abstract to $e \Rightarrow v$
  – forget intermediate expressions
  – read as $e$ evaluates down to $v$, equiv. $e$ big-steps to $v$
  – textbook notation: $e \downarrow v$

• **Goal:** for all expressions $e$ and values $v$, $e \Rightarrow v$ if and only if $e \rightarrow^* v$
  – A 4110 theorem
Values

• Constants are already done evaluating
  – 42 ==> 42
  – true ==> true

• In fact, all values big-step to themselves
  v ==> v
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.,
"big" ^ "red" \implies "bigred"

1 + 2 \implies 3

1 + (2+3) \implies 6
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., \( \text{let } x = 2 \text{ in } x+1 \)
  – e.g., \( \text{(fun } x \rightarrow x+1) \ 2 \)
  – e.g., \( \text{match } 2 \text{ with } x \rightarrow x+1 \)
  – All evaluate to 3, but we reach a point where we need to know binding of \( x \)

• Until now, we've never needed this, because we always *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:false\} at $:
    
    ```
    \text{let } x = 42 \text{ in }
    \text{let } y = \text{false in}
    $ e
    ```
Variable evaluation

To evaluate \texttt{x} in environment \texttt{env}

Look up value \texttt{v} of \texttt{x} in \texttt{env}

Return \texttt{v}

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

Extended notation:

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

Meaning: in dynamic environment \text{env}, expression \text{e} big-steps to value \text{v}

\[ \langle \text{env}, \ e \rangle \] is called a \textit{machine configuration}
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

\[ <\text{env}, \; v> \implies v \]

\[ <\text{env}, \; e_1 \; + \; e_2> \implies v \]

if \[ <\text{env}, \; e_1> \implies i_1 \]

and \[ <\text{env}, \; e_2> \implies i_2 \]

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

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

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

\[
<\text{env}, \ e1> \implies v1
\]

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

\[
\text{env}' = \text{env}[x \rightarrow v1] \quad \text{new notation}
\]

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

\[
<\text{env}', \ e2> \implies v2
\]

Return `v2`
Let expression evaluation rule

\[
\langle \text{env}, \text{let } x=\text{e1} \text{ in } \text{e2} \rangle \implies v2 \\
\text{if } \langle \text{env}, \text{e1} \rangle \implies v1 \\
\text{and } \langle \text{env}[x \to v1], \text{e2} \rangle \implies v2
\]

Example: (let {} be the empty environment)

\[
\langle {} \rangle, \text{let } x = 42 \text{ in } x \rangle \implies 42
\]

Because...

• \[
\langle {} \rangle, 42 \rangle \implies 42
\]

• and \[
\langle {} \rangle [x \to 42], x \rangle \implies 42
\]

  - Because \{x:42\}(x)=42
Function values v1.0

Anonymous functions are values:

<env, fun x -> e> ==> fun x -> e
Function application v1.0

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 \( \text{fun} \ x \rightarrow e \) 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 \( v_2 \)

\[
\text{env}' = \text{env}[x \rightarrow v_2]
\]

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

\[
<\text{env}', e> \implies v
\]

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

\[<\text{env},\text{e1 e2}> \implies v\]

if \(<\text{env, e1}> \implies \text{fun } x \rightarrow e\)
and \(<\text{env, e2}> \implies v2\)
and \(<\text{env}[x \rightarrow v2], e> \implies v\)

Example:
\(<\{\}, (\text{fun } x \rightarrow x) 1> \implies 1\)
b/c \(<\{\}, \text{fun } x \rightarrow x> \implies \text{fun } x \rightarrow x\)
and \(<\{\}, 1> \implies 1\)
and \(<\{\} [x \rightarrow 1], x> \implies 1\)
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 \texttt{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
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 called a function closure 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 , env|)

(|e , env|) is like a pair
• But indivisible: you cannot write OCaml syntax to access the pieces
• And inexpressible: you cannot directly write it in OCaml syntax
orange = changed from v1.0

To evaluate \( e_1 \) \( e_2 \) in environment \( env \)
Evaluate \( e_1 \) to a value \( v_1 \) in environment \( env \)
\(<env,e_1> \implies v_1\)
Note that \( v_1 \) must be closure (\(|\text{fun } x \rightarrow e \text{ , defenv}|\))
Evaluate \( e_2 \) to a value \( v_2 \) in environment \( env \)
\(<env,e_2> \implies v_2\)
Extend closure environment to bind formal parameter \( x \) to actual value \( v_2 \)
\( \text{env}' = \text{defenv}[x \rightarrow v_2] \)
Evaluate body \( e \) to a value \( v \) in environment \( \text{env}' \)
\(<\text{env}',e> \implies v\)
Return \( v \)
Function application rule v2.0

\[(\text{env}, e_1 e_2) \implies v\]

if \[(\text{env}, e_1) \implies \]

\[\left( |\text{fun } x \rightarrow e \ , \ \text{defenv} | \right)\]

and\[(\text{env}, e_2) \implies v_2\]

and \[(\text{defenv}[ x \rightarrow v_2 ], e) \implies v\]
Anonymous functions `fun x -> e` are closures:

```<env, fun x -> e> ==\> (|fun x -> e , env|)```
# Closures in OCaml bytecode compiler

https://github.com/ocaml/ocaml/search?q=kclosure

## Results in ocaml/ocaml

### bytecomp/instruct.ml

Showing the top match  Last indexed on Sep 15, 2016

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Krestart</td>
</tr>
<tr>
<td>64</td>
<td>Kgrab of int  (* number of arguments *)</td>
</tr>
<tr>
<td>65</td>
<td>Kclosure of label * int</td>
</tr>
</tbody>
</table>

### bytecomp/printinstr.ml

Showing the top match  Last indexed on Sep 15, 2016

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Kgrab n --&gt; fprintf pfpp &quot;$tgrab %i&quot; n</td>
</tr>
<tr>
<td>36</td>
<td>Kclosure(lbl, n) --&gt;  fprintf pfpp &quot;$tclosure L%i, %i&quot; lbl n</td>
</tr>
</tbody>
</table>

### bytecomp/instruct.mli

Showing the top match  Last indexed on Aug 10

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td>Kgrab of int  (* number of arguments *)</td>
</tr>
<tr>
<td>85</td>
<td>Kclosure of label * int</td>
</tr>
<tr>
<td>86</td>
<td>Kclosurer of label list * int</td>
</tr>
<tr>
<td>87</td>
<td>Koffsetclosure of int</td>
</tr>
</tbody>
</table>
Closures in Java

• Nested classes can simulate closures
  – Used everywhere for Swing GUI!
    http://docs.oracle.com/javase/tutorial/uiswing/event
    s/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
Interpreter for expr. lang.

See `interp4.ml` in code for this lecture for implementation with closures
Upcoming events

• [today] A4 out
• [tomorrow] spring break starts!