The Environment Model

Prof. Clarkson
Fall 2017

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

• **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 \implies 42
  – true \implies true

• In fact, all values big-step to themselves
  \[ v \implies v \]
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?
  \[ x \implies ??? \]
• Trick question: we don’t have enough information to answer it
• Need to know what value variable was 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 3, but we reach a point where we need to know binding of \( x \)
• Until now, 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 
\textit{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 relation

Extended notation:

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

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

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

\[
<\text{env}, \ x> \implies v
\]
\[
\text{if } v = \text{env}(x)
\]

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

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

\[ \langle \text{env}, \ e_1 + e_2 \rangle \implies v \]
  if \[ \langle \text{env}, \ e_1 \rangle \implies i_1 \]
  and \[ \langle \text{env}, \ e_2 \rangle \implies i_2 \]
  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 environment } env

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

\[
<env, \ e_1> \implies v_1
\]

\textbf{Extend} the environment to bind \( x \) to \( v_1 \)

\[
env' = env[x \to v_1]
\]

\textit{new notation}

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

\[
<env', \ e_2> \implies v_2
\]

Return \( v_2 \)
Let expression evaluation rule

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

Example: \(\langle \{\}, \text{let } x = 42 \text{ in } x \rangle \implies 42\)

Because...

\bullet \quad \langle \{\}, 42 \rangle \implies 42
\bullet \quad \text{and } \langle \{\}[x \rightarrow 42], x \rangle \implies 42
   \quad \text{– Because } \{x:42\}(x) = 42
Function values v1.0

Anonymous functions are values:

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

\[
<env, e_1 \ e_2> \implies \ v
\]

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

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 \[
\langle\{\} [x \rightarrow 1], x\rangle \implies 1
\]
Question

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
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., fun y->x
2. Evaluate 0 to a value, i.e., 0
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 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 variables to be bound to their value at the time of declaration.
– 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 called a function closure that has two parts:

• The code, an expression \( e \)
• The environment \( \text{env} \) that was current when the function was defined
• We'll notate that data structure as \( (|e, \text{env}|) \)

\( (|e, \text{env}|) \) 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
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 \( \langle \text{fun } x \rightarrow e \rangle \) \( \text{defenv} \)

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 \( v_2 \)

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

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

\[
<env',e> \implies v
\]

Return \( v \)
Function application rule v2.0

\[ \langle \text{env}, e_1, e_2 \rangle \implies v \]
  
  if \[ \langle \text{env}, e_1 \rangle \implies \]
  
  \[ (\text{|fun x -> e , defenv|}) \]

  and \[ \langle \text{env}, e_2 \rangle \implies v_2 \]

  and \[ \langle \text{defenv}[x \rightarrow v_2], e \rangle \implies v \]
Function values v2.0

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

\[<\text{env}, \text{fun } x \to e>\]
\[\Rightarrow (|\text{fun } x \to e \ , \ \text{env}|)\]
Closures in OCaml bytecode compiler

https://github.com/ocaml/ocaml/search?q=kclosure
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)
Interpreter for expr. lang.

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

• [today or tomorrow] A4 out

This is closure.

THIS IS 3110