The Environment Model

Prof. Clarkson
Fall 2018

Today's music: Selections from Doctor Who soundtracks by Murray Gold
Attendance question

Does there exist a well-typed program $e$ such that $e \rightarrow \ast x + 42$?

A. Yes
B. No
Review

Previously in 3110:
• Interpreters
• Small-step substitution model

Today:
• Large-step environment model
Review

• **Small-step substitution model:** substitute value for variable
  – Good mental model
  – Not efficient: too much substitution at run time

• **Big-step environment model:** maintain a dictionary that binds variables to values
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
  – \text{true} \implies \text{true}

• In fact, all values big-step to themselves
  \mathbf{v} \implies \mathbf{v}
Operator evaluation

e1 + e2 ==> v

if e1 ==> i1

and e2 ==> i2

and v is the result of primitive operation i1 + i2
Variables

What does a variable name evaluate to?

\[ x \implies ??? \]

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

OCaml doesn't actually do substitution

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

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 $:
    \[
    \begin{align*}
    &\$ \ \text{let} \ x = 42 \ \text{in} \\
    &\text{let} \ y = \text{false} \ \text{in} \\
    &e
    \end{align*}
    \]
  - One binding \(\{x:42\}\) at $:
    \[
    \begin{align*}
    &\text{let} \ x = 42 \ \text{in} \\
    &\$ \ \text{let} \ y = \text{false} \ \text{in} \\
    &e
    \end{align*}
    \]
  - Two bindings \(\{x:42, y: false\}\) at $:
    \[
    \begin{align*}
    &\text{let} \ x = 42 \ \text{in} \\
    &\text{let} \ y = \text{false} \ \text{in} \\
    &\$ \ e
    \end{align*}
    \]
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:

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

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

\[<\text{env}, \ e>\] 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

\text{primitive operation} \ i_1 + i_2
Let expressions

To evaluate \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ in environment } env\\
Evaluate the binding expression \texttt{e}_1 \texttt{ to a value } v_1 \texttt{ in environment } env\\
\texttt{<env, } e_1 \texttt{> } \implies \texttt{ v}_1\\
Extend the environment to bind \( x \) to \( v_1 \)\\
\texttt{env'} = env[x \rightarrow v_1] \quad \textit{new notation}\\
Evaluate the body expression \texttt{e}_2 \texttt{ to a value } v_2 \texttt{ in extended environment } env'\\
\texttt{<env', } e_2 \texttt{> } \implies \texttt{ v}_2\\
Return \texttt{ v}_2
Let expression evaluation rule

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

if \[
\langle \text{env}, e1 \rangle \implies v1
\]

and \[
\langle \text{env}[x \rightarrow v1], e2 \rangle \implies v2
\]
Anonymous functions are values:

\[
<env, \text{fun } x \rightarrow e> \iff \text{fun } x \rightarrow e
\]
To evaluate $e_1 \ e_2$ in environment $env$

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

$$<env, e_1> \implies v_1$$

*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 $env$

$$<env, e_2> \implies v_2$$

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

$$env' = env[x \rightarrow v_2]$$

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

$$<env', e> \implies v$$

Return $v$
Function application rule v1.0

<env,e1 e2> ==> v
if <env,e1> ==> fun x -> e
and <env,e2> ==> v2
and <env[x -> v2],e> ==> v
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?

```ml
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., \( \text{fun } y \rightarrow x \)
2. Evaluate \( 0 \) to a value, i.e., \( 0 \)
3. Extend environment to map parameter:
   \( \{x:2, f:(\text{fun } y \rightarrow x), y:0\} \)
4. Evaluate body \( x \) in that environment
5. Return \( 2 \)

\( 2 \lt<\) \( 1 \)
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 OCaml to use earlier binding of x
– 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
Function application v2.0

orange = changed from v1.0

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

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

$$<env,e_1> \Rightarrow v_1$$

*Note that $v_1$ must be closure $(|\text{fun } x \rightarrow e, \text{defenv}|)$*

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

$$<env,e_2> \Rightarrow v_2$$

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 $env'$

$$<env',e> \Rightarrow v$$

Return $v$
Function application rule v2.0

<env, e1 e2> ==> v
  if<env, e1> ==> 
    (|fun x -> e , defenv|)
and<env, e2> ==> v2
and<defenv[x -> v2], e> ==> v
Function values v2.0

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

```plaintext
<env, fun x -> e>  
===>  ( |fun x -> e , 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/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
Upcoming events

• N/A

This is closure.

THIS IS 3110