CS 3110

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 --> e' --> ... --> v`
- Abstract to `e ==> 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 ==> v` if and only if `e -->* 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

`e1 + e2 ==> v`

if `e1 ==> i1`

and `e2 ==> i2`

and `v` *is the result of primitive operation* `i1 + i2`

e.g.,

```
"big" ^ "red" ==> "bigred"
 1 + 2 ==> 3
 1 + (2+3) ==> 6
```

Variables

- What does a variable name evaluate to?

  `x` `==>` **`???`**

- Trick question: we don't have enough information to answer it
- Need to know what value variable was *bound* to
  - e.g., **`let x = 2 in x+1`**
  - e.g., **`(fun x -> x+1) 2`**
  - e.g., **`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 **substituted** before we ever get to a variable name

Variables

OCaml doesn't actually do substitution

```
(fun x -> 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 $:
    ```
    $ let x = 42 in
      let y = false in
      e
    ```
  - One binding **{x:42}** at $:
    ```
      let x = 42 in
    $ let y = false in
      e
    ```
  - Two bindings **{x:42,y:false}** at $:
    ```
      let x = 42 in
      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 relation

Extended notation:

`<env, e> ==> v`

Meaning: in dynamic environment **`env`**, expression **`e`** big-steps 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

<env, v> ==> v

<env, e1 + e2> ==> v

if **<env, e1> ==> i1**

and **<env, e2> ==> i2**

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

Let expressions

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

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

```
<env, e1> ==> v1
```

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

```
env' = env[x -> v1]
```
new notation

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

```
<env', e2> ==> v2
```

Return `v2`

Let expression evaluation rule

```
<env, let x=e1 in e2> ==> v2
  if <env, e1> ==> v1
  and <env[x -> v1], e2> ==> v2
```

Example: (let **{ }** be the empty environment)

```
<{},let x = 42 in x> ==> 42
```

Because...

- **<{}, 42> ==> 42**
- and **<{}[x -> 42], x> ==> 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 **e1 e2** in environment **env**

Evaluate **e1** to a value **v1** in environment **env**

```
<env,e1> ==> v1
```

Note that **v1** *must be a function value* **fun x -> e**
because function application type checks

Evaluate **e2** to a value **v2** in environment **env**

```
<env,e2> ==> v2
```

Extend environment to bind formal parameter **x** to actual value **v2**

```
env' = env[x -> v2]
```

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

```
<env',e> ==> 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
```

Example:

```
<{},(fun x -> x) 1> ==> 1
   b/c <{},fun x -> x> ==> fun x -> x
   and <{},1> ==> 1
   and <{}[x -> 1], x> ==> 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 n is
evaluat that
existed l, not
the curr n is
called.

– Cause
– 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

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 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

```
<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:

```
<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

OCaml

Showing the top match   Last indexed on Sep 15, 2016

```
  | Krestart
  | Kgrab of int                          (* number of arguments *)
  | Kclosure of label * int
```

bytecomp/printinstr.ml

OCaml

Showing the top match   Last indexed on Sep 15, 2016

```
  | Kgrab n -> fprintf ppf "\tgrab %i" n
  | Kclosure(lbl, n) ->
      fprintf ppf "\tclosure L%i, %i" lbl n
```

bytecomp/instruct.mli

OCaml

Showing the top match   Last indexed on Aug 10

```
  | Kgrab of int                          (* number of arguments *)
  | Kclosure of label * int
  | Kclosurerec of label list * int
  | Koffsetclosure of int
```

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

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!