CS4450/7450
AoPL, Chapter 3: Variables

Principles of Programming Languages

Dr. William Harrison

University of Missouri

November 1, 2016

Announcements

- HW4 due today (already)
- We're starting to use William Cook's online textbook, *Anatomy of Programming Languages*. It is available here. We're in Chapter 3.

Object vs. Meta

In each of the interpreters we have written so far, there have been really **two** languages:

```
module ArithAST where

data Op     = Plus | Minus | Times | Div
data Exp    = Const Int | Aexp Op Exp Exp
```

Object vs. Meta

In each of the interpreters we have written so far, there have been really **two** languages:

```haskell
module ArithAST where

data Op    = Plus | Minus | Times | Div
data Exp   = Const Int | Aexp Op Exp Exp
```

- Object Language: this is the language being studied
  - Here, that of arithmetic expressions
- Metalanguage: the language of description/implementation
  - Here, Haskell

What is a Variable?

- **Variable**: a symbol referring to a value
- Full code here:

```
data Exp  = Number    Int
          | Add       Exp Exp
          | Subtract  Exp Exp
          | Multiply  Exp Exp
          | Divide    Exp Exp
          | Variable  String   -- added
  deriving (Eq, Show)
```

- **Binding:** Association of a variable with a value
  - sometimes written $x \mapsto v$

Mutation

- "Mathematical variables" vary by *context*

Mutation

- "Mathematical variables" vary by *context*
  - E.g., x and x vary because they are in different contexts:

$$x^2 + 2x + 9 = 0$$
$$x^2 - 5x - 7 = 0$$

Mutation

- "Mathematical variables" vary by *context*
  - E.g., x and x vary because they are in different contexts:

$$x^2 + 2x + 9 = 0$$
$$x^2 - 5x - 7 = 0$$

- "Program Variables" *mutate* — they are really *containers*
- For example, x contains a number of values in the same context:

```
x = 0; while (x++ < 10) { ... }
```

Substitution

Substitution replaces a variable with a value in an expression.

Substitution	Expression	Produces
$x \mapsto 5$	$x + 2$	$5 + 2$
$x \mapsto 5$	2	2
$x \mapsto 5$	x	5
$x \mapsto 5$	$x * x + x$	$5 * 5 + 5$
$x \mapsto 5$	$x + y$	$5 + y$

Substitution

Substitution replaces a variable with a value in an expression.

Substitution	Expression	Produces
$x \mapsto 5$	$x + 2$	$5 + 2$
$x \mapsto 5$	2	2
$x \mapsto 5$	x	5
$x \mapsto 5$	$x * x + x$	$5 * 5 + 5$
$x \mapsto 5$	$x + y$	$5 + y$

N.b., if the variable names don't match, they are left alone

Substitution Implementation

The following code implements this behavior:

```
substitute1:: (String, Int) -> Exp -> Exp
substitute1 (var, val) exp = subst exp where
  subst (Number i)      = Number i
  subst (Add a b)       = Add (subst a) (subst b)
  subst (Subtract a b)  = Subtract (subst a) (subst b)
  subst (Multiply a b)  = Multiply (subst a) (subst b)
  subst (Divide a b)    = Divide (subst a) (subst b)
  subst (Variable name) = if var == name
                          then Number val
                          else Variable name
```

Can run tests using this code.

Multiple Bindings

- Obviously, some expressions have multiple variables; e.g., $2 * x + y$ with $x \mapsto 3$ and $y \mapsto -2$.

Multiple Bindings

- Obviously, some expressions have multiple variables; e.g., $2 * x + y$ with $x \mapsto 3$ and $y \mapsto -2$.
- **Environment**: collection of multiple bindings

Multiple Bindings

- Obviously, some expressions have multiple variables; e.g., $2 * x + y$ with $x \mapsto 3$ and $y \mapsto -2$.
- **Environment**: collection of multiple bindings
- Represent Environments in Haskell with

```
type Env = [(String,Int)]

-- for expression above
e1 = [("x",3),("y",-2)]
```

Multiple Bindings

- Obviously, some expressions have multiple variables; e.g., $2 * x + y$ with $x \mapsto 3$ and $y \mapsto -2$.
- **Environment**: collection of multiple bindings
- Represent Environments in Haskell with

```
type Env = [(String,Int)]

-- for expression above
e1 = [("x",3),("y",-2)]
```

- Looking up variables in an Env

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup a [] = Nothing
lookup a ((a',b):bs)
     | a==a'      = Just b
     | otherwise = lookup a bs
```

Substitution Implementation

The substitution function is easily modified to work with environments rather than single bindings:

```
substitute :: Env -> Exp -> Exp
substitute env exp = subst exp where
  subst (Number i)        = Number i
  subst (Add a b)         = Add (subst a) (subst b)
  subst (Subtract a b)    = Subtract (subst a) (subst b)
  subst (Multiply a b)    = Multiply (subst a) (subst b)
  subst (Divide a b)      = Divide (subst a) (subst b)
  subst (Variable name) =
    case lookup name env of
      Just val -> Number val
      Nothing  -> Variable name
```

Can run tests using this code.

Local variables

- All variables have been defined so far *outside* the expression itself
- Useful to allow variables to be defined *within* an expression

Local variables

- All variables have been defined so far *outside* the expression itself
- Useful to allow variables to be defined *within* an expression
- Most PLs support this with local variables; e.g., x and y in the following:

```
let x = 3 in let y = x*2 in x + y
```

Local variables

- All variables have been defined so far *outside* the expression itself
- Useful to allow variables to be defined *within* an expression
- Most PLs support this with local variables; e.g., x and y in the following:

```
let x = 3 in let y = x*2 in x + y
```

- Variable declaration expression can be represented by adding another case to the definition of expressions:

```
data Exp = ...
         | Declare String Exp Exp
```

Scope

Scope of a Variable Declaration

is the portion of the code text where that declaration holds.

Scope of y

```
let y = 7 in
  let x = 3 in
    5 + (let x = 2 in x + y) * x
```

Scope of first x

```
let y = 7 in
  let x = 3 in
    5 + (let x = 2 in x + y) * x
```

Scope of second x

```
let y = 7 in
  let x = 3 in
    5 + (let x = 2 in x + y) * x
```

Processing Declarations

Substituting into Variable Declarations

```
substitute1 (var, val) exp = subst exp
  ...
  subst (Declare x exp body)  = Declare x (subst exp)
      body'
    where body' = if x == var
                  then body
                  else subst body
```

Processing Declarations

Substituting into Variable Declarations

```
substitute1 (var, val) exp = subst exp
  ...
  subst (Declare x exp body)  = Declare x (subst exp)
       body'
    where body' = if x == var
                  then body
                  else subst body
```

Evaluating Variable Declarations using Substitution:

```
evaluate (Declare x exp body)
     = evaluate (substitute1 (x, evaluate exp) body)
```

Undefined Variable Errors

New Kind of Error

Attempting to evaluate an expression containing a variable that does not have a value.

For example, this pseudocode contains undefined variables:

```
x + 3
var x = 2; x * y
(var x = 3; x) * x
```

Static vs Dynamic Properties

Static Property

A **static property** of a program can be determined by examining the text of the program but without executing or evaluating it.

Dynamic Property

A **dynamic property** of a program can only be determined by evaluating the program.

Static vs Dynamic Properties

Static Property

A **static property** of a program can be determined by examining the text of the program but without executing or evaluating it.

Dynamic Property

A **dynamic property** of a program can only be determined by evaluating the program.

- *variable is undefined*: static property of the program:
  - whether it is undefined depends only on program text, not upon the particular data that the program is manipulating.

Static vs Dynamic Properties

Static Property

A **static property** of a program can be determined by examining the text of the program but without executing or evaluating it.

Dynamic Property

A **dynamic property** of a program can only be determined by evaluating the program.

- *variable is undefined*: static property of the program:
  - whether it is undefined depends only on program text, not upon the particular data that the program is manipulating.
- *divide by zero error* depends on particular data that the program is manipulating.
  - As a result, divide by zero is a dynamic error.

Static vs. Dynamic

Static Property

A **static property** of a program can be determined by examining the text of the program but without executing or evaluating it.

Dynamic Property

A **dynamic property** of a program can only be determined by evaluating the program.

- Might be possible to identify, just from examining the text of a program, that it will always divide by zero.
- Alternatively, may be that the code containing an undefined variable is never executed at runtime.
- Thus, boundary between static and dynamic errors is not absolute.

Summary


```
data Exp = Number     Int
         | Add        Exp Exp
            ...
         | Divide     Exp Exp
         | Variable   String
         | Declare    String Exp Exp

substitute1 (var, val) exp = subst exp where
  subst (Number i)       = Number i
  subst (Add a b)        = Add (subst a) (subst b)
            ...
  subst (Divide a b)     = Divide (subst a) (subst b)
  subst (Variable name) = if var == name
                            then Number val
                            else Variable name
  subst (Declare x exp body)  = Declare x (subst exp) body'
    where body' = if x == var
                  then body
                  else subst body

evaluate :: Exp -> Int
evaluate (Number i)       = i
evaluate (Add a b)        = evaluate a + evaluate b
            ...
evaluate (Divide a b)     = evaluate a `div` evaluate b
evaluate (Declare x exp body) = evaluate (substitute1 (x, evaluate exp) body)
```

Evaluation using substitution

Step	Result

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst $\mathtt{x} \mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst `x` $\mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

eval bound expr	$2 + 1 \Rightarrow 3$

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst $\mathtt{x} \mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

eval bound expr	$2 + 1 \Rightarrow 3$
subst $\mathtt{y} \mapsto 3$	`var z = 3 + 2;` `2 * 3 * z`

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst $\mathtt{x} \mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

eval bound expr	$2 + 1 \Rightarrow 3$
subst $\mathtt{y} \mapsto 3$	`var z = 3 + 2;` `2 * 3 * z`
eval bound expr	$3 + 2 \Rightarrow 5$

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst $\mathtt{x} \mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

eval bound expr	$2 + 1 \Rightarrow 3$
subst $\mathtt{y} \mapsto 3$	`var z = 3 + 2;` `2 * 3 * z`
eval bound expr	$3 + 2 \Rightarrow 5$
subst. $\mathtt{z} \mapsto 5$	`2 * 3 * 5`

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst $\mathtt{x} \mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

eval bound expr	$2 + 1 \Rightarrow 3$
subst $\mathtt{y} \mapsto 3$	`var z = 3 + 2;` `2 * 3 * z`
eval bound expr	$3 + 2 \Rightarrow 5$
subst. $\mathtt{z} \mapsto 5$	`2 * 3 * 5`
eval body	`2 * 3 * 5` $\Rightarrow$ `30`

Evaluation using substitution

Step	Result

initial expression	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`

eval bound expr.	$2 \Rightarrow 2$

subst `x` $\mapsto 2$	`var y = 2 + 1;`
	`var z = y + 2;`
	`2 * y * z`

eval bound expr	$2 + 1 \Rightarrow 3$
subst `y` $\mapsto 3$	`var z = 3 + 2;` `2 * 3 * z`
eval bound expr	$3 + 2 \Rightarrow 5$
subst. `z` $\mapsto 5$	`2 * 3 * 5`
eval body	`2 * 3 * 5` $\Rightarrow$ `30`

Correct, albeit not terribly efficient. Why?

Evaluating expressions in an environment

```
evaluate :: Exp -> Env -> Int
evaluate (Number i) env     = i
evaluate (Add a b) env
       = evaluate a env + evaluate b env
evaluate (Subtract a b) env
       = evaluate a env - evaluate b env
evaluate (Multiply a b) env
       = evaluate a env * evaluate b env
evaluate (Divide a b) env
       = evaluate a env `div` evaluate b env
evaluate (Variable x) env    = fromJust (lookup x env)
evaluate (Declare x exp body) env = evaluate body newEnv
  where newEnv = (x, evaluate exp env) : env

fromJust :: Maybe a -> a
fromJust (Just a) = a
fromJust Nothing  = error "Doh!"
```

Shorthand


```
var x = 2;
var y = x + 1;
var z = y + 2;
x * y * z
```

is shorthand for:

```
Declare "x" (Number 2)
 (Declare "y" (Add (Variable "x") (Number 1))
   (Declare "z" (Add (Variable "y") (Number 2))
     (Multiply (Variable "x")
       (Multiply (Variable "y")
                 (Variable "z")))))
```

Evaluating with Environments

Environment	Evaluation

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr 2*}

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* 2}

$\emptyset$	`2` $\Rightarrow$ `2` { *add new binding* `x` $\mapsto$ `2`}

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* 2}

$\emptyset$	$2 \Rightarrow 2$ { *add new binding* $x \mapsto 2$}

$x \mapsto 2$	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{ *eval bound expr* `x + 1` }

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* 2}

$\emptyset$	$2 \Rightarrow 2$ { *add new binding* $\mathtt{x} \mapsto 2$}

$\mathtt{x} \mapsto 2$	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{ *eval bound expr* `x + 1` }

$\mathtt{x} \mapsto 2$	$\mathtt{x} + 1 \Rightarrow 3$ { *add new binding* $\mathtt{y} \mapsto 3$}

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* 2}

$\emptyset$	$2 \Rightarrow 2$ { *add new binding* $\mathtt{x} \mapsto 2$}

$\mathtt{x} \mapsto 2$	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{ *eval bound expr* `x + 1` }

$\mathtt{x} \mapsto 2$	$\mathtt{x} + 1 \Rightarrow 3$ { *add new binding* $\mathtt{y} \mapsto 3$}

$\mathtt{y} \mapsto 3, \mathtt{x} \mapsto 2$	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* $\mathtt{y} + 2$}

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* 2}

$\emptyset$	$2 \Rightarrow 2$ { *add new binding* $\mathtt{x} \mapsto 2$}

$\mathtt{x} \mapsto 2$	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{ *eval bound expr* `x + 1` }

$\mathtt{x} \mapsto 2$	$\mathtt{x} + 1 \Rightarrow 3$ { *add new binding* $\mathtt{y} \mapsto 3$}

$\mathtt{y} \mapsto 3, \mathtt{x} \mapsto 2$	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* $\mathtt{y} + 2$}

$\mathtt{y} \mapsto 3, \mathtt{x} \mapsto 2$	$\mathtt{y} + 2 \Rightarrow 5$ {*add new binding for* `z`; *eval body of var decl* }

Evaluating with Environments

Environment	Evaluation

$\emptyset$	`var x = 2;`
	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr 2*}

$\emptyset$	$2 \Rightarrow 2$ { *add new binding* $x \mapsto 2$}

$x \mapsto 2$	`var y = x + 1;`
	`var z = y + 2;`
	`x * y * z`
	{ *eval bound expr* `x + 1` }

$x \mapsto 2$	$x + 1 \Rightarrow 3$ { *add new binding* $y \mapsto 3$}

$y \mapsto 3, x \mapsto 2$	`var z = y + 2;`
	`x * y * z`
	{*eval bound expr* $y + 2$}

$y \mapsto 3, x \mapsto 2$	$y + 2 \Rightarrow 5$ {*add new binding for* z; *eval body of var decl* }
$z \mapsto 5, y \mapsto 3, x \mapsto 2$	$x * y * z \Rightarrow 30$

* Text has an obvious typo.

"Shadowing"

Environment	Evaluation

"Shadowing"

Environment	Evaluation
$\emptyset$	`var x = 9; var x = x * x; x + x` {*eval bound expr* `9`}

"Shadowing"

Environment	Evaluation
$\emptyset$	`var x = 9; var x = x * x; x + x` $\{$*eval bound expr* `9`$\}$
$\emptyset$	`9` $\Rightarrow$ `9` $\{$*add* `x` $\mapsto$ `9`, *eval body of var decl*$\}$

“Shadowing”

Environment	Evaluation
$\emptyset$	`var x = 9; var x = x * x; x + x` $\{$*eval bound expr* `9`$\}$
$\emptyset$	`9` $\Rightarrow$ `9` $\{$*add* `x` $\mapsto$ `9`, *eval body of var decl*$\}$
`x` $\mapsto$ `9`	`var x = x * x; x + x` $\{$*eval bound expr* $x * x\}$

"Shadowing"

Environment	Evaluation
$\emptyset$	`var x = 9; var x = x * x; x + x` {*eval bound expr* `9`}
$\emptyset$	`9` $\Rightarrow$ `9` {*add* `x` $\mapsto$ `9`, *eval body of var decl*}
`x` $\mapsto$ `9`	`var x = x * x; x + x` {*eval bound expr* $x * x$}
`x` $\mapsto$ `9`	{*add* `x` $\mapsto$ `81`, *eval body of var decl*}

"Shadowing"

Environment	Evaluation
$\emptyset$	`var x = 9; var x = x * x; x + x` {*eval bound expr* `9`}
$\emptyset$	$9 \Rightarrow 9$ {*add* $x \mapsto 9$, *eval body of var decl*}
$x \mapsto 9$	`var x = x * x; x + x` {*eval bound expr* $x * x$}
$x \mapsto 9$	{*add* $x \mapsto 81$, *eval body of var decl*}
$x \mapsto 81, x \mapsto 9$	$x + x \Rightarrow 162$

"Shadowing"

Environment	Evaluation
$\emptyset$	`var x = 9; var x = x * x; x + x` {*eval bound expr* `9`}
$\emptyset$	$9 \Rightarrow 9$ {*add* $x \mapsto 9$, *eval body of var decl*}
$x \mapsto 9$	`var x = x * x; x + x` {*eval bound expr* $x * x$}
$x \mapsto 9$	{*add* $x \mapsto 81$, *eval body of var decl*}
$x \mapsto 81, x \mapsto 9$	$x + x \Rightarrow 162$

- Final environment contains two bindings for `x`, but the leftmost one is used? Why?

Extension to Booleans and Conditionals
Full code here

- Extend values:

```
data Value   =  IntV  Int
             |  BoolV Bool
 deriving (Eq)
```

- Extend abstract syntax:

```
data BinaryOp = Add | Sub | Mul | Div | And | Or
             | GT | LT | LE | GE | EQ
                    deriving (Show, Eq)

data UnaryOp = Neg | Not deriving (Show, Eq)

data Exp  = Literal   Value
          | Unary     UnaryOp Exp
          | Binary    BinaryOp Exp Exp
          | If        Exp Exp Exp
          | Variable  String
          | Declare   String Exp Exp
```

Helper Functions

What are their types?


```
unary Not  (BoolV b)  = BoolV (not b)
unary Neg  (IntV i)   = IntV (-i)

binary Add (IntV a)   (IntV b)   = IntV  (a + b)
binary Sub (IntV a)   (IntV b)   = IntV  (a - b)
binary Mul (IntV a)   (IntV b)   = IntV  (a * b)
binary Div (IntV a)   (IntV b)   = IntV  (a `div` b)
binary And (BoolV a)  (BoolV b)  = BoolV (a && b)
binary Or  (BoolV a)  (BoolV b)  = BoolV (a || b)
binary LT  (IntV a)   (IntV b)   = BoolV (a < b)
binary LE  (IntV a)   (IntV b)   = BoolV (a <= b)
binary GE  (IntV a)   (IntV b)   = BoolV (a >= b)
binary GT  (IntV a)   (IntV b)   = BoolV (a > b)
binary EQ  a          b          = BoolV (a == b)
```

Defining Conditional


```
evaluate (If a b c) env =
  let BoolV test = evaluate a env in
    if test then evaluate b env
            else evaluate c env
```

Defining Conditional


```
evaluate (If a b c) env =
  let BoolV test = evaluate a env in
    if test then evaluate b env
            else evaluate c env
```

What happens if `a` doesn't evaluate to a `BoolV`?