Elements of Programming Languages

Lecture 4: Variables, binding and substitution

James Cheney

University of Edinburgh

October 6, 2015

Variables

- A variable is a symbol that can stand for another expression.
- Often written $x, y, z, \ldots$.
- Let's extend $\mathsf{L}_{\mathsf{Arith}}$ with variables:

$$e ::= e_1 + e_2 \mid e_1 \times e_2 \mid n \mid x \in \mathit{Var}$$

- Here, x is shorthand for an arbitrary variable in Var, the set of expression variables

Substitution

- A variable can "stand for" another expression.
- What does this mean precisely?
- Suppose we have $x + 1$ and we want x to "stand for" 42.
- We should be able to *replace* x everywhere in $x + 1$ with 42:

$$x + 1 \rightsquigarrow 42 + 1$$

- Another example: if y "stands for" $x + 1$ then

$$x + y + 1 \rightsquigarrow x + (x + 1) + 1$$

- (Remember that we insert parentheses when necessary to disambiguate in abstract syntax expressions.)

Substitution

- Let's introduce a notation for this *substitution* operation:

Definition (Substitution)

Given e_1, x, e_2, the *substitution of e_2 for x in e_1* is an expression written $e_1[e_2/x]$.

- For $\mathsf{L}_{\mathsf{Arith}}$ with variables, define substitution as follows:

$$\begin{array}{rcl} n[e/x] & = & n \\ x[e/x] & = & e \\ y[e/x] & = & y \quad (x \neq y) \\ (e_1 + e_2)[e/x] & = & e_1[e/x] + e_2[e/x] \\ (e_1 \times e_2)[e/x] & = & e_1[e/x] \times e_2[e/x] \end{array}$$

Scope

- As we all know from programming, we can *reuse* variable names:

```
def foo(x: Int) = x + 1
def bar(x: Int) = x * x
```

- The occurrences of `x` in `foo` have nothing to do with those in `bar`
- Moreover the following code is equivalent (since `y` is not already in use in `foo` or `bar`):

```
def foo(x: Int) = x + 1
def bar(y: Int) = y * y
```


Scope

Definition (Scope)

The *scope* of a variable name is the collection of program locations in which occurrences of the variable refer to the same thing.

- I am being a little casual here: "refer to the same thing" doesn't necessarily mean that the two variable occurrences evaluate to the same value at run time.
- For example, the variables could refer to a shared *reference cell* whose value changes over time.

Scope, Binding and Bound Variables

- Certain occurrences of variables are called *binding*
- Again, consider

```
def foo(x: Int) = x + 1
def bar(y: Int) = y * y
```

- The occurrences of `x` and `y` on the left-hand side of the definitions are *binding*
- The other occurrences are called *bound*
- Binding occurrences define scopes: two bound variables are in the same scope if they are bound by the same binding occurrence.

Dynamic vs. static scope

- The terms *static* and *dynamic* scope are sometimes used.
- In **static scope**, the scope and binding occurrences of all variables can be determined from the program text, **without** actually running the program.
- In **dynamic scope**, this is not necessarily the case: the scope of a variable can depend on the context in which it is evaluated **at run time**.
- We will have more to say about this later when we cover functions
  - but for now, the short version is: Static scope good, dynamic scope bad.

Simple scope: let-binding

- For now, we consider a very basic form of scope: let-binding.

$$e ::= \cdots \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

- We define L_{Let} to be L_{If} extended with variables and `let`.
- In an expression of the form $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, we say that x is *bound* in e_2
- Intuition: let-binding allows us to use a variable x as an abbreviation for some other expression:

$$\texttt{let}\ x = 1 + 2\ \texttt{in}\ 3 \times x \rightsquigarrow 3 \times (1 + 2)$$

Free variables

- The set of *free variables* of an expression is defined as:

$$\begin{aligned}
FV(n) &= \emptyset \\
FV(x) &= \{x\} \\
FV(e_1 \oplus e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) &= FV(e) \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) &= FV(e_1) \cup (FV(e_2) - \{x\})
\end{aligned}$$

  where $X - Y$ is the set of elements of X that are not in Y

$$\{x, y, z\} - \{y\} = \{x, z\}$$

- (Recall that $e_1 \oplus e_2$ is shorthand for several cases.)
- Examples:

$$FV(x + y) = \{x, y\} \qquad FV(\texttt{let}\ x = y\ \texttt{in}\ x) = \{y\}$$
$$FV(\texttt{let}\ x = x + y\ \texttt{in}\ z) = \{x, y, z\}$$

Alpha-Equivalence

- Two expressions that are equivalent "modulo consistent renaming of bound variables" are called *alpha-equivalent*
- For L_{Let} we can define alpha-equivalence as follows:

Alpha-equivalence for L_{Let} ($e_1 \equiv_\alpha e_2$)

$$\frac{}{v \equiv_\alpha v} \qquad \frac{}{x \equiv_\alpha x} \qquad \frac{e_1 \equiv_\alpha e_1' \quad e_2 \equiv_\alpha e_2'}{e_1 \oplus e_2 \equiv_\alpha e_1' \oplus e_2'}$$

$$\cdots \quad \frac{e_1 \equiv_\alpha e_1' \quad e_2[z/x] \equiv_\alpha e_2'[z/y] \quad z \notin FV(e_2) \cup FV(e_2')}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \equiv_\alpha \texttt{let}\ y = e_1'\ \texttt{in}\ e_2'}$$

- Structural equality except for `let`
- For `let`, we compare the e_1s and replace the bound names with fresh names and compare the e_2s

Alpha-equivalence: examples

To illustrate, here are some examples of equivalent terms:

$$x \equiv_\alpha x \qquad (\texttt{let}\ x = y\ \texttt{in}\ x) \equiv_\alpha (\texttt{let}\ z = y\ \texttt{in}\ z)$$

$$\begin{aligned}
&(\texttt{let}\ y = 1\ \texttt{in}\ \texttt{let}\ x = 2\ \texttt{in}\ x + y) \\
\equiv_\alpha\ &(\texttt{let}\ w = 1\ \texttt{in}\ \texttt{let}\ z = 2\ \texttt{in}\ z + w)
\end{aligned}$$

and here are some inequivalent terms:

$$x \not\equiv_\alpha y \qquad (\texttt{let}\ x = y\ \texttt{in}\ x) \not\equiv_\alpha (\texttt{let}\ y = x\ \texttt{in}\ y)$$

$$\begin{aligned}
&(\texttt{let}\ y = 1\ \texttt{in}\ \texttt{let}\ x = 2\ \texttt{in}\ x + y) \\
\not\equiv_\alpha\ &(\texttt{let}\ y = 1\ \texttt{in}\ \texttt{let}\ y = 2\ \texttt{in}\ y + y)
\end{aligned}$$

Types and variables

- Once we add variables to our language, how does that affect typing?
- Consider

  $$\texttt{let } x = e_1 \texttt{ in } e_2$$

  When is this well-formed? What type does it have?
- Consider a variable on its own: what type does it have?
- **Different occurrences of the same variable in different scopes could have different types.**
- We need a way to *keep track of* the types of variables

Types for variables and let, informally

- Suppose we have a way of keeping track of the types of variables (say, some kind of map or table)
- When we see a variable x, look up its type in the map.
- When we see a $\texttt{let } x = e_1 \texttt{ in } e_2$, find out the type of e_1. Add the information that x has type τ_1 to the map, and check e_2 using the augmented map.
- Note: The local information about x's type should not persist beyond typechecking its scope e_2.

Types for variables and let, informally

- For example:

  $$\texttt{let } x = 1 \texttt{ in } x + 1$$

  is well-formed: we know that x must be an `int` since it is set equal to 1, and then $x + 1$ is well-formed because x is an `int` and 1 is an `int`.
- On the other hand,

  $$\texttt{let } x = 1 \texttt{ in if } x \texttt{ then } 42 \texttt{ else } 17$$

  is not well-formed: we again know that x must be an `int` while checking `if` x `then` 42 `else` 17, but then when we check that the conditional's test x is a `bool`, we find that it is actually an `int`.

Type Environments

- We write Γ to denote a *type environment*, or a finite map from variable names to types, often written as follows:

  $$\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n$$

- In Scala, we can use the built-in type `ListMap[Variable,Type]` for this.
- Moreover, we write $\Gamma(x)$ for the type of x according to Γ and $\Gamma, x : \tau$ to indicate extending Γ with the mapping x to τ.

Types for variables and let, formally

- We now generalize the ideal of well-formedness:

Definition (Well-formedness in a context)

We write $\Gamma \vdash e : \tau$ to indicate that e is well-formed at type τ (or just "has type τ") in context Γ.

- The rules for variables and let-binding are as follows:

$\boxed{\Gamma \vdash e : \tau}$ **for L_{Let}**

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2}$$

Types for variables and let, formally

- We also need to generalize the L_{If} rules to allow contexts:

$\boxed{\Gamma \vdash e : \tau}$ **for L_{If}**

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \oplus : \tau_1 \times \tau_2 \to \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

$$\frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

- This is straightforward: we just add Γ everywhere.
- The previous rules are special cases where Γ is empty.

Examples, revisited

We can now typecheck as follows:

$$\frac{\frac{}{\vdash 1 : \texttt{int}} \quad \frac{\frac{}{x : \texttt{int} \vdash x : \texttt{int}} \quad \frac{}{x : \texttt{int} \vdash 1 : \texttt{int}}}{x : \texttt{int} \vdash x + 1 : \texttt{int}}}{\vdash \texttt{let } x = 1 \texttt{ in } x + 1 : \texttt{int}}$$

On the other hand:

$$\frac{\frac{}{\vdash 1 : \texttt{int}} \quad \frac{x : \texttt{int} \vdash x : \texttt{bool} \quad \cdots}{x : \texttt{int} \vdash \texttt{if } x \texttt{ then } 42 \texttt{ else } 17 : ??}}{\vdash \texttt{let } x = 1 \texttt{ in if } x \texttt{ then } 42 \texttt{ else } 17 : ??}$$

is not derivable because the judgment $x : \texttt{int} \vdash x : \texttt{bool}$ isn't.

Evaluation for `let` and variables

- One approach: whenever we see $\texttt{let } x = e_1 \texttt{ in } e_2$,
  1. evaluate e_1 to v_1
  2. replace x with v_1 in e_2 and evaluate that

$\boxed{e \Downarrow v}$ **for L_{If}**

$$\frac{e_1 \Downarrow v_1 \quad e_2[v_1/x] \Downarrow v_2}{\texttt{let } x = e_1 \texttt{ in } e_2 \Downarrow v_2}$$

- Note: We always substitute values for variables, and do not need a rule for "evaluating" a variable
- This evaluation strategy is called *eager*, *strict*, or (for historical reasons) *call-by-value*
- This is a design choice. We will revisit this choice (and consider alternatives) later.

Substitution-based interpreter

```
type Variable = String
...
case class Var(x: Variable) extends Expr
case class Let(x: Variable, e1: Expr, e2: Expr)
  extends Expr
...
def eval(e: Expr): Value = e match {
  ...
  Let(x,e1,e2) =>
    val v = eval e1
    val e2' = subst(e2,val2expr(v),x)
    eval e2'
  }
```

- Note: No case for `Var(x)`; need to convert `Value` to `Expr`

Substitution revisited

- Consider the following two alpha-equivalent terms:

$$(\texttt{let}\ x = 1\ \texttt{in}\ x + y) \equiv_\alpha (\texttt{let}\ z = 1\ \texttt{in}\ z + y)$$

  Intuition: the choice of bound name x (or z) does not matter, as long as we avoid other names
- Now consider what happens if we substitute:

$$(\texttt{let}\ x = 1\ \texttt{in}\ x + y)[x/y] = \texttt{let}\ x = 1\ \texttt{in}\ x + x$$

- But

$$(\texttt{let}\ z = 1\ \texttt{in}\ z + y)[x/y] = \texttt{let}\ z = 1\ \texttt{in}\ z + x$$

- These are not alpha-equivalent!
- Substituting for x under a binding of x leads to *variable capture*

Capture-avoiding substitution

- To fix this problem, substitution needs to *avoid capture*
- For $\mathrm{L}_{\mathrm{Let}}$, this works as follows:

$$(\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[e/x] = \texttt{let}\ y = e_1[e/x]\ \texttt{in}\ e_2'$$
$$\text{where } e_2' = \begin{cases} e_2 & (y = x) \\ e_2[e/x] & (y \notin FV(e)) \end{cases}$$

- Note: The above cases are non-exhaustive
- But it is always safe to rename to a completely fresh name $z \notin FV(e, e_1, e_2)$

$$\texttt{let}\ y = e_1\ \texttt{in}\ e_2 \equiv_\alpha \texttt{let}\ z = e_1\ \texttt{in}\ e_2[z/y]$$

  so that the second case applies

Example, revisited

- Now consider the example:

$$(\texttt{let}\ x = 1\ \texttt{in}\ x + y)[x/y]$$

  Neither case of capture-avoiding substitution for `let` applies. But we can α-rename:

$$(\texttt{let}\ x = 1\ \texttt{in}\ x{+}y)[x/y] \equiv_\alpha (\texttt{let}\ w = 1\ \texttt{in}\ w{+}y)[x/y]$$

  Now the second case applies:

$$(\texttt{let}\ w = 1\ \texttt{in}\ w + y)[x/y] \quad = \quad \texttt{let}\ w = 1\ \texttt{in}\ w + x$$

- Capture-avoiding substitution is partial on expressions, but total and well-defined on alpha-equivalence classes of expressions.

Alternative semantics: environments

- Another common way to handle variables is to use an *environment*
- An environment σ is a partial function from variables to values (e.g. a Scala `ListMap[Variable,Value]`).
- We add σ as an argument to the evaluation judgment:

$\boxed{\sigma, e \Downarrow v}$

$$\frac{}{\sigma, v \Downarrow v} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 \times e_2 \Downarrow v_1 \times_{\mathbb{N}} v_2}$$

$$\cdots \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma[x = v], e_2 \Downarrow v_2}{\sigma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \qquad \frac{}{\sigma, x \Downarrow \sigma(x)}$$

- Coursework 1 asks you to implement such an interpreter.

Variables and Substitution Scope and Binding Types Semantics

Summary

- Today we've covered:
  - Basics of variables, scope, and binding
  - Free variables, alpha-equivalence, and substitution
  - Let-binding and how it affects typing and semantics

  Next time:
  - Functions and function types
  - Recursion