Elements of Programming Languages

Lecture 4: Variables, scope, and substitution

James Cheney

University of Edinburgh

October 4, 2016

Variables

- A variable is a symbol that can 'stand for' a value.
- Often written $x, y, z, \ldots$.
- Let's extend L_{If} with variables:

$$\begin{array}{rcl} e & ::= & n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 \times e_2 \\ & \mid & b \in \mathbb{B} \mid e_1 == e_2 \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\ & \mid & x \in Var \end{array}$$

- Here, x is shorthand for an arbitrary variable in Var, the set of expression variables
- Let's call this language L_{Var}

Aside: Operators, operators everywhere

- We have now considered several *binary operators*

$$+ \quad \times \quad \wedge \quad \vee \quad \approx$$

- as well as a unary one ($\neg$)
- It is tiresome to write their syntax, evaluation rules, and typing rules explicitly, every time we add to the language
- We will sometimes represent such operations using *schematic* syntax $e_1 \oplus e_2$ and rules:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \oplus e_2 \Downarrow v_1 \oplus_{\mathbb{A}} v_2} \qquad \frac{\vdash e_1 : \tau' \quad \vdash e_2 : \tau' \quad \oplus : \tau' \times \tau' \to \tau}{\vdash e_1 \oplus e_2 : \tau}$$

- where $\oplus : \tau' \times \tau' \to \tau$ means that operator $\oplus$ takes arguments τ', τ' and yields result of type τ
- (e.g. $+ : \texttt{int} \times \texttt{int} \to \texttt{int}$, $== : \tau \times \tau \to \texttt{bool}$)

Substitution

- We said "A variable can 'stand for' a value."
- 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:

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

- Similarly, if x "stands for" 3 then

$$\texttt{if } x == y \texttt{ then } x \texttt{ else } y \rightsquigarrow \texttt{if } 3 == y \texttt{ then } 3 \texttt{ else } y$$

Substitution

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

Definition (Substitution)

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

- For L_{Var}, define substitution as follows:

$$
\begin{array}{rcl}
v_0[v/x] & = & v_0 \\
x[v/x] & = & v \\
y[v/x] & = & y \quad (x \neq y) \\
(e_1 \oplus e_2)[v/x] & = & e_1[v/x] \oplus e_2[v/x] \\
(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2)[v/x] & = & \texttt{if } e[v/x] \texttt{ then } e_1[v/x] \\
 & & \texttt{ else } e_2[v/x]
\end{array}
$$

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*
- Binding occurrences define scopes: the occurrences of `x` and `y` on the right-hand side are *bound*
- Any variables not in scope of a binder are called *free*
- Key idea: Renaming all binding and bound occurrences in a scope *consistently* (avoiding name clashes) should not affect meaning

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:

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

Equivalence up to consistent renaming

- We wish to consider expressions *equivalent* if they have the same binding structure
- We can *rename* bound names to get equivalent expressions:

$$\texttt{let}\ x = y + z\ \texttt{in}\ x == w \equiv \texttt{let}\ u = y + z\ \texttt{in}\ u == w$$

- But some renamings change the binding structure:

$$\texttt{let}\ x = y + z\ \texttt{in}\ x == w \not\equiv \texttt{let}\ w = y + z\ \texttt{in}\ w == w$$

- Intuition: Renaming to u is fine, because u is not already "in use".
- But renaming to w changes the binding structure, since w was already "in use".

Freshness

- We say that a variable x is *fresh* for an expression e if there are no free occurrences of x in e.
- We can define this using rules as follows:

$x \# e$

$$\frac{}{x \# v} \qquad \frac{x \neq y}{x \# y} \qquad \frac{x \# e_1 \quad x \# e_2}{x \# e_1 \oplus e_2} \qquad \frac{x \# e \quad x \# e_1 \quad x \# e_2}{x \# \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{x \# e_1}{x \# \texttt{let}\ x = e_1\ \texttt{in}\ e_2} \qquad \frac{x \neq y \quad x \# e_1 \quad x \# e_2}{x \# \texttt{let}\ y = e_1\ \texttt{in}\ e_2}$$

- Examples:

$$x \# \texttt{true} \qquad x \# y \qquad x \# \texttt{let}\ x = 1\ \texttt{in}\ x$$

Renaming

- We will also use the following *swapping* operation to rename variables:

$$
\begin{aligned}
x(y\leftrightarrow z) &= \begin{cases} y & \text{if } x = z \\ z & \text{if } x = y \\ x & \text{otherwise} \end{cases} \\
v(y\leftrightarrow z) &= v \\
(e_1 \oplus e_2)(y\leftrightarrow z) &= e_1(y\leftrightarrow z) \oplus e_2(y\leftrightarrow z) \\
(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2)(y\leftrightarrow z) &= \texttt{if } e(y\leftrightarrow z) \texttt{ then } e_1(y\leftrightarrow z) \\
&\quad \texttt{else } e_2(y\leftrightarrow z) \\
(\texttt{let } x = e_1 \texttt{ in } e_2)(y\leftrightarrow z) &= \texttt{let } x(y\leftrightarrow z) = e_1(y\leftrightarrow z) \\
&\quad \texttt{in } e_2(y\leftrightarrow z)
\end{aligned}
$$

- Example:

$$(\texttt{let } x = y \texttt{ in } x + z)(x\leftrightarrow z) = \texttt{let } z = y \texttt{ in } z + x$$

Alpha-conversion

- We can now define "consistent renaming".
- Suppose $y \# e_2$. Then we can rename a `let`-expression as follows:

$$\texttt{let } x = e_1 \texttt{ in } e_2 \rightsquigarrow_\alpha \texttt{let } y = e_1 \texttt{ in } e_2(x\leftrightarrow y)$$

- This is called *alpha-conversion*.
- Two expressions are *alpha-equivalent* if we can convert one to the other using alpha-conversions.

Examples

- Examples:

$$
\begin{aligned}
& \texttt{let } x = y + z \texttt{ in } x == w \\
\rightsquigarrow_\alpha\ & \texttt{let } u = y + z \texttt{ in } (x == w)(x\leftrightarrow u) \\
=\ & \texttt{let } u = y + z \texttt{ in } u(x\leftrightarrow u) == w(x\leftrightarrow u) \\
=\ & \texttt{let } u = y + z \texttt{ in } u == w
\end{aligned}
$$

  since $u \# (x == w)$.
- But

  $\texttt{let } x = y{+}z \texttt{ in } x == w \not\rightsquigarrow_\alpha \texttt{let } w = y{+}z \texttt{ in } w == w$

  because w already appears in $x == w$.

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. Suppose that type is τ_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 $\texttt{if}\ x\ \texttt{then}\ 42\ \texttt{else}\ 17$, 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.
  - *hey, maybe that's why the Lab has all that stuff about* `ListMaps`*!*
- 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:

$\Gamma \vdash e : \tau$ 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:

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

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

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

- 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:

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

On the other hand:

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

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

Evaluation for `let` and variables

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

$e \Downarrow v$ for L_{Let}

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

- 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 {
  ...
  case Let(x,e1,e2) => {
    val v = eval(e1);
    val e2vx = subst(e2,v,x);
    eval(e2vx)
  }
```

- Note: No case for `Var(x)`.

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:

$\sigma, e \Downarrow v$

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

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

- Assignment 2 will ask you to implement such an interpreter.

Summary

- Today we've covered:
  - Variables that can be replaced with values
  - Scope and binding, alpha-equivalence
  - Let-binding and how it affects typing and semantics

  Next time:
  - Functions and function types
  - Recursion