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:

$$e ::= n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 \times e_2 \mid b \in \mathbb{B} \mid e_1 == e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid x \in \text{Var}$$

Here, $x$ is shorthand for an arbitrary variable in $\text{Var}$, the set of expression variables. Let’s call this language $L_{\text{Var}}$

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

Aside: Operators, operators everywhere
We have now considered several binary operators

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

- 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:

$$\begin{align*}
  e_1 \downarrow v_1 & \quad e_2 \downarrow v_2 \\
  \vdash e_1 : \tau' & \quad \vdash e_2 : \tau' \quad \vdash : \tau' \times \tau' \rightarrow \tau \\
  e_1 \oplus e_2 \downarrow v_1 \oplus_A v_2 & \quad \vdash e_1 \oplus e_2 : \tau
\end{align*}$$

- where $\oplus : \tau' \times \tau' \rightarrow \tau$ means that operator $\oplus$ takes arguments $\tau', \tau'$ and yields result of type $\tau$
- (e.g. $+: \text{int} \times \text{int} \rightarrow \text{int}, == : \tau \times \tau \rightarrow \text{bool}$)
**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 $L_{\text{Var}}$, define substitution as follows:
  
  $v_0[v/x] = v_0$
  $x[v/x] = v$
  $y[v/x] = y$ (if $x \neq y$)
  $(e_1 \oplus e_2)[v/x] = e_1[v/x] \oplus e_2[v/x]$
  $(\text{if } e \text{ then } e_1 \text{ else } e_2)[v/x] = \text{if } e[v/x] \text{ then } e_1[v/x] \text{ else } e_2[v/x]$

**Scope**

- As we all know from programming, we can *reuse* variable names:
  
  ```java
  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流`):
  
  ```java
  def foo(x: Int) = x + 1
  def bar(y: Int) = y * y
  ```

**Scope, Binding and Bound Variables**

- Certain occurrences of variables are called *binding*
- Again, consider
  
  ```java
  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
Variables and Substitution
Scope and Binding
Types and evaluation
Variables and Substitution
Scope and Binding
Types and evaluation

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 runtime.
- 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.

\[ e ::= \cdots \mid x \mid \text{let } x = e_1 \text{ in } e_2 \]

- We define \( L_{let} \) to be \( L \) extended with variables and 1let.
- In an expression of the form \( \text{let } x = e_1 \text{ in } e_2 \), 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:

\[ \text{let } x = 1 + 2 \text{ in } 3 \times x \leadsto 3 \times (1 + 2) \]

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:

\[
\begin{align*}
\text{let } x = y + z \text{ in } x &= w & \equiv & \text{let } u = y + z \text{ in } u &= w \\
\text{let } x = y + z \text{ in } x &= w & \not\equiv & \text{let } w = y + z \text{ in } w &= w \\
\text{let } x = y + z \text{ in } x &= w & & & \text{let } y = e_1 \text{ in } e_2 \\
\text{Intuition: Renaming to } u \text{ is fine, because } u \text{ is not already “in use”}.
\end{align*}
\]

- 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:

\[
\begin{align*}
\text{x} \# \text{e} \\
\equiv & \quad \text{x} \neq \text{y} \quad \text{x} \# \text{e}_1 \quad \text{x} \# \text{e}_2 \\
\equiv & \quad x \# \text{if } e \text{ then } e_1 \text{ else } e_2 \\
\equiv & \quad \text{let } x = e_1 \text{ in } e_2 \\
\equiv & \quad \text{let } y = e_1 \text{ in } e_2
\end{align*}
\]

- Examples:

\[ x \# \text{true} \quad x \# \text{y} \quad x \# \text{let } x = 1 \text{ in } x \]
Renaming

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

\[ x(y \leftrightarrow z) = \begin{cases} 
  y & \text{if } x = z \\
  z & \text{if } x = y \\
  x & \text{otherwise}
\end{cases} \]

\[ v(y \leftrightarrow z) = v \]

\[(e_1 \oplus e_2)(y \leftrightarrow z) = e_1(y \leftrightarrow z) \oplus e_2(y \leftrightarrow z)\]

\[(\text{if } e \text{ then } e_1 \text{ else } e_2)(y \leftrightarrow z) = \begin{cases} 
  e(y \leftrightarrow z) & \text{if } e \text{ is } e_1(y \leftrightarrow z) \\
  e_2(y \leftrightarrow z) & \text{else}
\end{cases} \]

\[(\text{let } x = e_1 \text{ in } e_2)(y \leftrightarrow z) = \begin{cases} 
  \text{let } x(y \leftrightarrow z) = e_1(y \leftrightarrow z) \text{ in } e_2(y \leftrightarrow z) & \text{if } x = e_1 \\
  \text{let } x(y \leftrightarrow z) = e_1(y \leftrightarrow z) \text{ in } e_2(y \leftrightarrow z) & \text{else}
\end{cases} \]

- Example:

\[(\text{let } x = y \text{ in } x + z)(x \leftrightarrow z) = \text{let } z = y \text{ in } z + x\]

Alpha-conversion

- We can now define “consistent renaming”.

- Suppose \( y \neq e_2 \). Then we can rename a let-expression as follows:

\[ \text{let } x = e_1 \text{ in } e_2 \overset{\alpha}{\Rightarrow} \text{let } y = e_1 \text{ in } e_2(x \leftrightarrow y) \]

- This is called \textit{alpha-conversion}.

- Two expressions are \textit{alpha-equivalent} if we can convert one to the other using alpha-conversions.

Examples

- Examples:

\[ \text{let } x = y + z \text{ in } x == w \]

\[ \overset{\alpha}{\Rightarrow} \text{let } u = y + z \text{ in } (x == w)(x \leftrightarrow u) \]

\[ = \text{let } u = y + z \text{ in } u(x \leftrightarrow u) == w(x \leftrightarrow u) \]

\[ = \text{let } u = y + z \text{ in } u == w \]

since \( u \neq (x == w) \).

- But

\[ \text{let } x = y + z \text{ in } x == w \not\overset{\alpha}{\Rightarrow} \text{let } w = y + z \text{ in } w == w \]

because \( w \) already appears in \( x == w \).

Types and variables

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

- Consider

\[ \text{let } x = e_1 \text{ in } e_2 \]

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 `let \( x = e_1 \) in e_2`, find out the type of \( e_1 \). Suppose that type is \( \tau_1 \). Add the information that \( x \) has type \( \tau_1 \) 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 \( e_2 \).

For example:

\[
\text{let } x = 1 \text{ in } x + 1
\]

is well-formed: we know that \( x \) must be an \texttt{int} since it is set equal to 1, and then \( x + 1 \) is well-formed because \( x \) is an \texttt{int} and 1 is an \texttt{int}.

On the other hand,

\[
\text{let } x = 1 \text{ in if } x \text{ then 42 else 17}
\]

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

Type Environments

- We write \( \Gamma \) 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 \texttt{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 \( \Gamma \) and \( \Gamma, x : \tau \) to indicate extending \( \Gamma \) with the mapping \( x \) to \( \tau \).

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 \( \tau \) (or just “has type \( \tau \)”) in context \( \Gamma \).

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

\[
\begin{align*}
\Gamma &\vdash e : \tau & \text{for } \texttt{LLet} \\
\Gamma(x) &= \tau & \Gamma \vdash e_1 : \tau_1 & \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]
Types for variables and let, formally

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

$$
\Gamma \vdash e : \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_1 \oplus e_2 : \tau \\
\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau
$$

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

We can now typecheck as follows:

$$
\Gamma \vdash n : \text{int} \\
\Gamma \vdash \text{let } x = 1 \text{ in } x + 1 : \text{int}
$$

On the other hand:

$$
\Gamma \vdash \text{let } x = 1 \text{ in if } x \text{ then } 42 \text{ else } 17 : \text{??}
$$

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

Evaluation for let and variables

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

$$
e_1 \downarrow v_1 \quad e_2[v_1/x] \downarrow v_2 \\
\text{let } x = e_1 \text{ in } e_2 \downarrow v_2
$$

- 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

```scala
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)
  }
  ...

  case Var(x) => ...
}
```

- Note: No case for Var(x).
Alternative semantics: environments

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

$$
\sigma, e \Downarrow v
$$

$$
\frac{
\sigma_1, e_1 \Downarrow v_1, \sigma_2 \Downarrow v_2}{\sigma, e_1 + e_2 \Downarrow v_1 +N v_2}
\frac{
\sigma_1, e_1 \times e_2 \Downarrow v_1 \times_N v_2}{\sigma, e_1 \times e_2 \Downarrow v_1 \times_N v_2}
\frac{
\sigma, e_1 \Downarrow v_1, e_2 \Downarrow v_2}{\sigma, x = v_1, e_2 \Downarrow v_2}
\frac{
\sigma, let \ x = e_1 \ in \ e_2 \Downarrow v_2}{\sigma, x \Downarrow \sigma(x)}
$$

- 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