Elements of Programming Languages
Lecture 4: Variables, substitution, and scope

James Cheney
University of Edinburgh
October 2, 2023
Variables

- A variable is a symbol that can ‘stand for’ a value.
- Often written $x, y, z, \ldots$.
- Let’s extend $L_{lf}$ 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 $L_{\text{Var}}$. 
Aside: Operators, operators everywhere

- We have now considered several *binary operators*:
  
  \[ + \times \land \lor \approx \]

- as well as a unary one (¬)

- 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:

\[
\begin{align*}
  e_1 \Downarrow v_1 & \quad e_2 \Downarrow v_2 \\
  e_1 \oplus e_2 \Downarrow v_1 \oplus_A v_2
\end{align*}
\]

\[
\frac{\vdash e_1 : \tau_1 \quad \vdash e_2 : \tau_2 \quad \oplus : \tau_1 \times \tau_2 \to \tau}{\vdash e_1 \oplus e_2 : \tau}
\]

- where \( \oplus : \tau_1 \times \tau_2 \to \tau \) means that operator \( \oplus \) takes arguments \( \tau_1, \tau_2 \) and yields result of type \( \tau \)

- (e.g. \( + : \text{int} \times \text{int} \to \text{int}, \text{==} : \tau \times \tau \to \text{bool} \))
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
  $$\text{if } x == y \text{ then } x \text{ else } y \rightsquigarrow \text{if } 3 == y \text{ then } 3 \text{ else } y$$
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 $L_{\text{Var}}$, define substitution as follows:

- $\nu_0[v/x] = \nu_0$
- $x[v/x] = v$
- $y[v/x] = y \ (x \neq y)$
- $(e_1 \oplus e_2)[v/x] = e_1[v/x] \oplus e_2[v/x]$
- $(\text{if } e \text{ then } e_1 \text{ else } e_2)[v/x] = \text{if } e[v/x] \text{ then } e_1[v/x] \text{ else } e_2[v/x]$
As we all know from programming, we can *reuse* variable names:

```scala
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`):

```scala
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.
- In that case, the “same thing” they refer to is the reference cell, not the value in it.
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
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 \( L_{\text{Let}} \) to be \( L_{\text{If}} \) extended with variables and let.
- In an expression of the form \( \text{let } x = e_1 \text{ in } e_2 \), we say that \( x \) is \textit{bound} in \( e_2 \).
- Intuition: let-binding allows us to use a variable \( x \) as an abbreviation for (the value of) some other expression:

\[ \text{let } x = 1 + 2 \text{ in } 4 \times x \leadsto \text{let } x = 3 \text{ in } 4 \times x \leadsto 4 \times 3 \]
Equivalence up to consistent renaming

- We wish to consider expressions *equivalent* (written $e_1 \equiv e_2$) if they have the same binding structure.

- We can *rename* bound names to get equivalent expressions:

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

- But some renamings change the binding structure:

  \[
  \text{let } x = y + z \text{ in } x == w \not\equiv \text{let } w = y + z \text{ in } w == w
  \]

- 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”.

Free variables

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

$$FV(n) = \emptyset$$

$$FV(x) = \{x\}$$

$$FV(e_1 \oplus e_2) = FV(e_1) \cup FV(e_2)$$

$$FV(\text{if } e \text{ then } e_1 \text{ else } e_2) = FV(e) \cup FV(e_1) \cup FV(e_2)$$

$$FV(\text{let } x = e_1 \text{ in } e_2) = FV(e_1) \cup (FV(e_2) - \{x\})$$

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:

$$FV(x + y) = \{x, y\} \quad FV(\text{let } x = y \text{ in } x) = \{y\}$$

$$FV(\text{let } x = x + y \text{ in } z) = \{x, y, z\}$$
Renaming

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

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

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

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

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

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

- 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 $y \not\in FV(e_2)$. Then we can rename a let-expression as follows:

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

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

- Examples:

\[
\begin{align*}
\text{let } x &= y + z \text{ in } x &= w \\
\xRightarrow{\alpha} \quad \text{let } u &= y + z \text{ in } (x &= w) (x \leftrightarrow u) \\
&= \text{ let } u &= y + z \text{ in } x (x \leftrightarrow u) = w (x \leftrightarrow u) \\
&= \text{ let } u &= y + z \text{ in } u &= w
\end{align*}
\]

since \( u \notin FV(x = w) \).

- But

\[
\begin{align*}
\text{let } x &= y + z \text{ in } x &= w \xRightarrow{\alpha} \text{ let } w &= y + z \text{ in } w &= w
\end{align*}
\]

because \( w \) already appears in \( x = w \).
Evaluation for let and variables

- One approach: whenever we see `let x = e₁ in e₂`,
  1. evaluate `e₁` to `v₁`
  2. replace `x` with `v₁` in `e₂` and evaluate that

\[
e ⇓ v
\text{ for } L_{\text{Let}}
\]

\[
\frac{e₁ ⇓ v₁ \quad e₂[v₁/x] ⇓ v₂}{\text{let } x = e₁ \text{ in } e₂ ⇓ v₂}
\]

- 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

```java
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).
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 $e_2$. 
Types for variables and let, informally

- For example:

  ```
  let \( x = 1 \) 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,

  ```
  let \( x = 1 \) in if \( x \) then 42 else 17
  ```

  is not well-formed: we again know that \( x \) must be an \texttt{int} while checking if \( x \) 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 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 $\tau$. 
Types for variables and let, formally

- We now generalize the ideas 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:

$$
\Gamma \vdash e : \tau \quad \text{for } L_{\text{Let}} \\
\frac{\Gamma (x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}
$$
Types for variables and let, formally

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

  \[ \Gamma \vdash e : \tau \] for $L_{lf}$

  \[
  \begin{align*}
  \Gamma \vdash n : \text{int} & \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
  \quad & \implies \Gamma \vdash e_1 \oplus e_2 : \tau \\
  \Gamma \vdash b : \text{bool} & \quad \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \\
  \quad & \implies \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau
  \end{align*}
  \]

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

We can now typecheck as follows:

\[
\begin{array}{c}
\frac{x : \text{int} \vdash x : \text{int}}{\vdash x : \text{int}} \quad \frac{x : \text{int} \vdash 1 : \text{int}}{\vdash 1 : \text{int}} \\
\frac{\vdash 1 : \text{int}}{\vdash x : \text{int} \vdash x + 1 : \text{int}} \quad \frac{\vdash \text{let } x = 1 \text{ in } x + 1 : \text{int}}{\vdash \text{let } x = 1 \text{ in } x + 1 : \text{int}}
\end{array}
\]

On the other hand:

\[
\begin{array}{c}
\frac{x : \text{int} \vdash x : \text{bool}}{\vdash x : \text{int} \vdash x : \text{bool}} \quad \cdots \\
\frac{\vdash 1 : \text{int}}{\vdash x : \text{int} \vdash \text{if } x \text{ then } 42 \text{ else } 17 : ??} \quad \frac{\vdash \text{let } x = 1 \text{ in } \text{if } x \text{ then } 42 \text{ else } 17 : ??}{\vdash \text{let } x = 1 \text{ in } \text{if } x \text{ then } 42 \text{ else } 17 : ??}
\end{array}
\]

is not derivable because the judgment \(x : \text{int} \vdash x : \text{bool}\) isn't.
Summary

Today we’ve covered:

- Variables that can be substituted with values
- Scope and binding, alpha-equivalence
- Let-binding and how it affects typing and evaluation

Next time:

- Functions and function types
- Recursion