Variables

A variable is a symbol that can stand for another expression.
Often written \(x, y, z, \ldots\)
Let’s extend \(L_{\text{Arith}}\) with variables:
\[
e ::= e_1 + e_2 \mid e_1 \times e_2 \mid n \mid x \in \text{Var}
\]
Here, \(x\) is shorthand for an arbitrary variable in \(\text{Var}\), 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:

\[
x + 1 \leadsto 42 + 1
\]
Another example: if \(y\) “stands for” \(x + 1\) then

\[
x + y + 1 \leadsto x + (x + 1) + 1
\]
(Remember that we insert parentheses when necessary to disambiguate in abstract syntax expressions.)

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 \(L_{\text{Arith}}\) with variables, define substitution as follows:

\[
n[e/x] = n
x[e/x] = e
y[e/x] = y \quad (x \neq y)
(e_1 + e_2)[e/x] = e_1[e/x] + e_2[e/x]
(e_1 \times e_2)[e/x] = e_1[e/x] \times e_2[e/x]
\]
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
```

**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

```scala
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.
- \( e ::= \cdots \mid x \mid \text{let } x = e_1 \text{ in } e_2 \)
- We define \( L_{\text{Let}} \) to be \( L_{\text{If}} \) extended with variables and let.
- In an expression of the form let \( x = e_1 \) 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:
  \[
  \text{let } x = 1 + 2 \text{ in } 3 \times x \leadsto 3 \times (1 + 2)
  \]

Free variables

- The set of **free variables** of an expression is defined as:
  \[
  \begin{align*}
  FV(n) &= \emptyset \\
  FV(x) &= \{x\} \\
  FV(e_1 \oplus e_2) &= FV(e_1) \cup FV(e_2) \\
  FV(\text{if } e \text{ then } e_1 \text{ else } e_2) &= FV(e) \cup FV(e_1) \cup FV(e_2) \\
  FV(\text{let } x = e_1 \text{ in } e_2) &= FV(e_1) \cup (FV(e_2) - \{x\})
  \end{align*}
  \]
  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:**
  \[
  \begin{align*}
  FV(x+y) &= \{x, y\} & FV(\text{let } x = y \text{ in } x) &= \{y\} \\
  FV(\text{let } x = x + y \text{ in } z) &= \{x, y, z\}
  \end{align*}
  \]

Alpha-Equivalence

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

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

  \[
  \begin{align*}
  v &\equiv_\alpha v \quad x \equiv_\alpha x \quad e_1 \equiv_\alpha e_1' \quad e_2 \equiv_\alpha e_2' \\
  e_1 \oplus e_2 &\equiv_\alpha e_1' \oplus e_2' \\
  e_1 \mid x &\equiv_\alpha e_1' \mid x \quad e_2/z \equiv_\alpha e_2'[z/y] \quad z \notin FV(e_2) \cup FV(e_2')
  \end{align*}
  \]
- Structural equality except for let
- For let, we compare the \( e_1 \)'s and replace the bound names with fresh names and compare the \( e_2 \)'s.

Alpha-equivalence: examples

To illustrate, here are some examples of equivalent terms:

- \( x \equiv_\alpha x \quad \text{(let } x = y \text{ in } x) \equiv_\alpha (\text{let } z = y \text{ in } z) \)
- \( \text{(let } y = 1 \text{ in let } x = 2 \text{ in } x + y) \equiv_\alpha (\text{let } w = 1 \text{ in let } z = 2 \text{ in } z + w) \)

And here are some inequivalent terms:

- \( x \not\equiv_\alpha y \quad \text{(let } x = y \text{ in } x) \not\equiv_\alpha (\text{let } y = x \text{ in } y) \)
- \( \text{(let } y = 1 \text{ in let } x = 2 \text{ in } x + y) \not\equiv_\alpha (\text{let } y = 1 \text{ in let } y = 2 \text{ in } y + y) \)
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 \( \text{let } x = e_1 \text{ in } e_2 \), find out the type of \( e_1 \).
  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 \( \text{int} \) since it is set equal to 1, and then \( x + 1 \) is well-formed because \( x \) is an \( \text{int} \) and 1 is an \( \text{int} \).
- On the other hand,
  \( \text{let } x = 1 \text{ in if } x \text{ then } 42 \text{ else } 17 \)
  is not well-formed: we again know that \( x \) must be an \( \text{int} \) while checking \( \text{if } x \text{ then } 42 \text{ else } 17 \), but then when we check that the conditional’s test \( x \) is a \( \text{bool} \), we find that it is actually an \( \text{int} \).

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 \( \text{ListMap[Variable,Type]} \) for this.
- 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:

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

  - $\Gamma(x) = \tau$
  - $\Gamma \vdash x : \tau$
  - $\Gamma \vdash e_1 : \tau_1$
  - $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$
  - $\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2$

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:

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

On the other hand:

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

is not derivable because the judgment $x : \text{int} \vdash x : \text{bool}$ 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

$$
e \Downarrow v \text{ for } L_{\text{If}}$$

- 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 {
  ...
  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:

\[
(\text{let } x = 1 \text{ in } x + y) \equiv_\alpha (\text{let } z = 1 \text{ in } z + y)
\]

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:

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

But

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

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 \(L_{\text{Let}}\), this works as follows:

\[
(\text{let } y = e_1 \text{ in } e_2)[e/x] = \text{let } y = e_1[e/x] \text{ in } e'_2
\]

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 \(z \notin FV(e, e_1, e_2)\)

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

so that the second case applies

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

Example, revisited

Now consider the example:

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

Neither case of capture-avoiding substitution for \(L_{\text{Let}}\) applies. But we can \(\alpha\)-rename:

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

Now the second case applies:

\[
(\text{let } w = 1 \text{ in } w + y)[x/y] = \text{let } w = 1 \text{ in } w + x
\]
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:

\[
\sigma, e \Downarrow v
\]

\[
\frac{\sigma \Downarrow v_1 \sigma \Downarrow v_2}{\sigma, e_1 + e_2 \Downarrow v_1 + v_2}
\]

\[
\frac{\sigma, e_1 \Downarrow v_1 \sigma, e_2 \Downarrow v_2}{\sigma, e_1 \times e_2 \Downarrow v_1 \times v_2}
\]

\[
\frac{\sigma, e_1 \Downarrow v_1 \sigma[x = v], e_2 \Downarrow v_2}{\sigma, let x = e_1 in e_2 \Downarrow v_2}
\]

\[
\frac{\sigma, x \Downarrow \sigma(x)}{}
\]

Coursework 1 asks you to implement such an interpreter.

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