Variables

A variable is a symbol that can ‘stand for’ a value.

Often written \( x, y, z, \ldots \).

Let’s extend \( L_{\text{If}} \) with variables:

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

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

\(+ \times \wedge \vee \approx\)

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:

\[
\begin{align*}
\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{\vdash e_1 : \tau' \quad \vdash e_2 : \tau' \quad \vdash e_1 \oplus e_2 : \tau} \\
\frac{e_1 \oplus e_2 \downarrow v_1 \oplus_A v_2}{\vdash e_1 \oplus e_2 : \tau}
\end{align*}
\]

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

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:

\( x + 1 \rightarrow 42 + 1 \)

Similarly, if \( x \) “stands for” 3 then

\( \text{if } x == y \text{ then } x \text{ else } y \rightarrow \text{if } 3 == y \text{ then } 3 \text{ else } y \)
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:

\[
\begin{align*}
  v_0[v/x] &= v_0 \\
  x[v/x] &= v \\
  y[v/x] &= y \quad (x \neq y) \\
  (e_1 + e_2)[v/x] &= e_1[v/x] + e_2[v/x] \\
  (\text{if } e \text{ then } e_1 \text{ else } e_2)[v/x] &= \text{if } e[v/x] \text{ then } e_1[v/x] \text{ else } e_2[v/x]
\end{align*}
\]

Scope

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

\[
\begin{align*}
  \text{def foo(x: Int) = x + 1} \\
  \text{def bar(x: Int) = x * x}
\end{align*}
\]

The occurrences of \( x \) in \( \text{foo} \) have nothing to do with those in \( \text{bar} \).

Moreover the following code is equivalent (since \( y \) is not already in use in \( \text{foo} \) or \( \text{bar} \)):

\[
\begin{align*}
  \text{def foo(x: Int) = x + 1} \\
  \text{def bar(y: Int) = y * y}
\end{align*}
\]

Scope, Binding and Bound Variables

- Certain occurrences of variables are called binding.

- Again, consider

\[
\begin{align*}
  \text{def foo(x: Int) = x + 1} \\
  \text{def bar(y: Int) = y * y}
\end{align*}
\]

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 *bound* in \( e_2 \).

- Intuition: let-binding allows us to use a variable \( x \) as an abbreviation for some other expression:
  \[
  \text{let } x = 1 + 2 \text{ in } 3 \times x \leftrightarrow 3 \times (1 + 2)
  \]

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

- 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:
  \[
  \begin{align*}
  FV(n) &= \emptyset \\
  FV(x) &= \{x\} \\
  FV(e_1 \oplus e_2) &= FV(e_1) \cup FV(e_2) \\
  FV(\text{if } e \text{ then } e_1 \text{ else } e_2) &= FV(e) \cup FV(e_1) \cup FV(e_2) \\
  FV(\text{let } x = e_1 \text{ in } e_2) &= FV(e_1) \cup (FV(e_2) - \{x\})
  \end{align*}
  \]

- 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:
  \[
  \begin{align*}
  FV(x + y) &= \{x, y\} \\
  FV(\text{let } x = y \text{ in } x) &= \{y\} \\
  FV(\text{let } x = x + y \text{ in } z) &= \{x, y, z\}
  \end{align*}
  \]

### Renaming

- We will also use the following *swapping* operation to rename variables:
  \[
  \begin{align*}
  x(y\leftrightarrow z) &= \begin{cases} 
  y & \text{if } x = z \\
  x & \text{otherwise}
  \end{cases} \\
  v(y\leftrightarrow z) &= v \\
  (e_1 \oplus e_2)(y\leftrightarrow z) &= e_1(y\leftrightarrow z) \oplus e_2(y\leftrightarrow z) \\
  (\text{if } e \text{ then } e_1 \text{ else } e_2)(y\leftrightarrow z) &= \begin{cases} 
  e(y\leftrightarrow z) & \text{if } e(y\leftrightarrow z) \text{ then } e_1(y\leftrightarrow z) \\
  e_2(y\leftrightarrow z) & \text{else}
  \end{cases} \\
  (\text{let } x = e_1 \text{ in } e_2)(y\leftrightarrow z) &= \begin{cases} 
  \text{let } x(y\leftrightarrow z) = e_1(y\leftrightarrow z) \\
  \text{in } e_2(y\leftrightarrow z)
  \end{cases}
  \end{align*}
  \]

- Example:
  \[
  (\text{let } x = y \text{ in } x + z)(x\leftrightarrow z) = \text{let } z = y \text{ in } z + x
  \]
We can now define “consistent renaming”.

Suppose \( y \notin FV(e_2) \). Then we can rename a

let-expression as follows:

\[
\text{let } x = e_1 \text{ in } e_2 \xrightarrow{\alpha} \text{let } y = e_1 \text{ in } e_2(x \mapsto y)
\]

This is called alpha-conversion.

Two expressions are alpha-equivalent if we can convert

one to the other using alpha-conversions.

Examples:

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

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

But

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

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

Evaluation for let and variables

One approach: whenever we see \( \text{let } x = e_1 \text{ in } e_2 \),

- evaluate \( e_1 \) to \( v_1 \)
- replace \( x \) with \( v_1 \) in \( e_2 \) and evaluate that

\[
\downarrow v \text{ for } L_{\text{Let}}
\]

\[
e_1 \downarrow v_1 \quad e_2[v_1/x] \downarrow v_2
\]

\[
\text{let } x = e_1 \text{ in } e_2 \downarrow v_2
\]

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
...

class Var(x: Variable) extends Expr

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 \( \text{Var}(x) \).
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 \( \text{let } x = e_1 \text{ in } e_2 \), 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 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,

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

is not well-formed: we again know that \( x \) must be an 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 \( \Gamma \) 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.

\[
\text{hey, maybe that’s why the Lab has all that stuff about ListMaps!}
\]

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 \( \tau \).
Variables and Substitution Scope and Binding Evaluation and types

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

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

$\Gamma \vdash e : \tau$ for LLet

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \vdash e_1 : \tau_1 \\
\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]

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

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

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

- 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:

\[
\begin{align*}
\vdash 1 : \text{int} \\
\vdash x : \text{int} \vdash x : \text{int} \\
\vdash x : \text{int} \vdash 1 : \text{int} \\
\vdash \text{let } x = 1 \text{ in } x + 1 : \text{int}
\end{align*}
\]

On the other hand:

\[
\begin{align*}
\vdash 1 : \text{int} \\
\vdash x : \text{int} \vdash x : \text{bool} & \quad \text{...} \\
\vdash \text{let } x = 1 \text{ in } \text{if } x \text{ then } 42 \text{ else } 17 : ?? \\
\vdash \text{let } x = 1 \text{ in } \text{if } x \text{ then } 42 \text{ else } 17 : ??
\end{align*}
\]

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