Towards Mini-ML

Brigitte Pientka

March 9, 2007

In these notes, we will extend the language we have seen so far with variables, let-expressions. We will do this in two steps. First, we extend our inductive definition to allow variables, and let-expressions. Next, we will consider issues arising due to variables. In particular we will explain the notion of a bound and free variables, overshadowing of variables, and substitution. Finally, we are extending the operational semantics to evaluate let-expressions.

1 Syntax for variables and let-expressions

First, we will extend our inductive definition for expressions to allow for variables and let-expressions. We write ... to indicate we have all the previously defined expressions of numbers, booleans, primitive operations, and if-statements.

$$\text{Expressions } e ::= \ldots \mid x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$$

We will write x, y, z for variables. To illustrate again these new expressions, here are some examples:

Examples of well-formed expressions

- $\mathsf{let}\ z = \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ z + 123\ \mathsf{end}$
- $\mathsf{let}\ z = \mathsf{if}\ 7\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ z + \mathsf{false}\ \mathsf{end}$
- $\mathsf{let}\ x = x + 3\ \mathsf{in}\ y + 123\ \mathsf{end}$
- $\mathsf{let}\ x = x + 3\ \mathsf{in}\ x + 123\ \mathsf{end}$

The last two examples are well-formed expressions, although we may not yet understand the meaning of it. In particular, we have not clarified the status of the variables x and y occurring in the expression.

Examples of ill-formed expressions

- $\mathsf{let}\ z = \mathsf{if}\ true\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ -123\ \mathsf{end}$
- $\mathsf{let}\ x = 3\ \mathsf{in}\ x$
- $\mathsf{let}\ x = 3\ x + 2\mathsf{end}$

The first expression here is ill-formed because -123 is not syntactically allowed. The second expression is erroneous because the end is missing to indicate the end of the let-expression. The last one is rejected because the key word in is missing.

Before we can define the operational semantics for evaluating let-expressions, we first clarify issues arising from variables. The first question we must answer is: When is variable bound and when is it considered free? Intuitively, a variable is considered free, if it is not bound. We will define these two concepts next.

Free variables Fist, let us define inductively the set of free variables occurring in an expression e. We will define a function FV which takes as input an expression e and returns a set of the free variables occurring in e.

$$\begin{array}{lcl}
\mathsf{FV}(x) & = & \{x\} \\
\mathsf{FV}(e_1\ \mathsf{op}\ e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{FV}(e) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}) & = & \mathsf{FV}e_1 \cup (\mathsf{FV}(e_2)/\ \{x\})
\end{array}$$

This definition also highlights the fact that a variable x may become *bound* in the let-expression. The let-expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ introduces a binder x which binds all the occurrences of x in e_2. An expression e which has no free variables, i.e. $\mathsf{FV}(e) = \emptyset$ is called *closed*. The following example clarifies the concept of bound and free variables.

$$\mathsf{let}\ x = 5\ \mathsf{in}\ \underbrace{(\mathsf{let}\ y = \overbrace{x+3}^{\text{x is free}}\ \mathsf{in}\ \overbrace{y+x}^{\text{x, y are free}}\ \mathsf{end})}_{\text{x is free, y is bound}}\ \mathsf{end}$$

The subexpression $y + x$ contains the free occurrences of the variable x and y. The variable y then gets bound in $\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + x\ \mathsf{end}$. Hence only x remains a free. The scope of y is only in the body of the let expression. More generally, given a let-expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$, the binder x binds variables occurring in e_2 only.

To clarify let us consider the free and bound variables in the following expression:

$$\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ x = \overbrace{x+3}^{\substack{\text{x is free} \\ \text{bound by x = 5}}}\ \mathsf{in}\ \overbrace{x+x}^{\substack{\text{x is free} \\ \text{bound by x = x+3}}}\ \mathsf{end})\ \mathsf{end}$$

In other words, a free variable x gets bound by the first binder enclosing it. Another important property of bound variables is that their names do not matter. Clearly, it should not matter, if we write

$$\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + y\ \mathsf{end})\ \mathsf{end}$$

or

$$\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ x = x + 3\ \mathsf{in}\ x + x\ \mathsf{end})\ \mathsf{end}$$

Both terms denote the same expression up to renaming of the bound variables. Before we describe the evaluation of a let-expression, we will define the important operation of substituting an expression for a free variable in another expression. We will write $[e'/x]e$ for replacing all *free* occurrences of the variable x in the expression e with the expression e'. We will define this operation inductively on the expression e. This is straightforward in most cases. Applying the substitution $[e'/x]$ to a variable x clearly should return the term e'. When applying the substitution $[e'/x]$ to the expression $e_1\ \mathsf{op}\ e_2$, we need to apply $[e'/x]$ to each of its sub-expressions. Similarly, when applying the substitution $[e'/x]$ to the expression $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, we need to apply the substitution $[e'/x]$ to the sub-expression e, e_1, and e_2. This is defined as follows more formally:

$$\begin{array}{lcl}
[e'/x](x) & = & e' \\
[e'/x](e_1\ \mathsf{op}\ e_2) & = & ([e'/x]e_1\ \mathsf{op}\ [e'/x]e_2) \\
[e'/x](\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ [e'/x]e\ \mathsf{then}\ [e'/x]e_1\ \mathsf{else}\ [e'/x]e_2
\end{array}$$

The most interesting case is what should happen when we apply the substitution $[e'/x]$ to the expression $\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$.

In the first attempt, we just apply the substitution $[e'/x]$ to the sub-expressions e_1 and e_2.

$$[e'/x](\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end} = \mathsf{let}\ y = [e'/x]e_1\ \mathsf{in}\ [e'/x]e_2\ \mathsf{end} \qquad \text{Attempt 1}$$

This seems to work fine for this example:

$$\begin{array}{l}
[5/x](\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + x\ \mathsf{end}) = \\
\mathsf{let}\ y = [5/x](x + 3)\ \mathsf{in}\ [5/x](y + x)\ \mathsf{end} = \\
\mathsf{let}\ y = 5 + 3\ \mathsf{in}\ y + 5\ \mathsf{end}
\end{array}$$

However, what will happen when we try to replace the variable x with the term $y + 1$? Simple replacement would yield,

$$\begin{array}{ll}
 & [y + 1/x](\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + x\ \mathsf{end}) \\
= & \mathsf{let}\ y = [y + 1/x](x + 3)\ \mathsf{in}\ [y + 1/x](y + x)\ \mathsf{end} \\
= & \mathsf{let}\ y = (y + 1) + 3\ \mathsf{in}\ y + (y + 1)\ \mathsf{end}
\end{array}$$

But this seems wrong, since y occurred free in the term $y + 1$, but becomes bound in the result! This phenomena is called *variable capture*, and clearly needs to be avoided.

The key is the observation that the name of bound variables do not matter, and hence we can always rename the bound variable y in $(\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + x\ \mathsf{end})$ to z and obtain $(\mathsf{let}\ z = x + 3\ \mathsf{in}\ z + x\ \mathsf{end})$ which is equivalent. A change of bound variables is often called *α-conversion*. To ensure substitution works correctly for let-expressions, i.e. $[e'/x](\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$, we need to impose the restriction that y does not occur free in the expression e'. This side condition can always be achieved by renaming bound variables. We can now define more formally the last case for let-expressions:

$$[e'/x](\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}) = \mathsf{let}\ y = [e'/x]e_1\ \mathsf{in}\ [e'/x]e_2\ \mathsf{end} \quad \text{provided } x \neq y \text{ and } y \notin \mathsf{FV}(e')$$

The problem of variable capture occurs whenever we have bound variables. Once we extend the language to include functions for example we will encounter the same problem.

2 Evaluation of let-expressions

Finally, we are in a position to describe formally how a let-expression will be evaluated. Intuitively, when given an expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$, we first evaluate the expression e_1 to some value v_1. Next, we replace all free occurrences of x in the expression e_2 by the value v_1 and continue to evaluate $[v_1/x]e_2$ to some value v_2. This can be formally described by the following inference rule:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \Downarrow v}\ \text{B-LET}$$

Next, a sample derivation illustrating evaluation of a let-expression.

$$\dfrac{\dfrac{}{5 \Downarrow 5}\ \text{B-NUM} \qquad \dfrac{\dfrac{\dfrac{}{5 \Downarrow 5}\ \text{B-NUM} \quad \dfrac{}{3 \Downarrow 3}\ \text{B-NUM}}{5 + 3 \Downarrow 8}\ \text{B-OP} \qquad \dfrac{\dfrac{}{5 \Downarrow 5}\ \text{B-NUM} \quad \dfrac{}{8 \Downarrow 8}\ \text{B-NUM}}{5 + 8 \Downarrow 13}\ \text{B-OP}}{(\mathsf{let}\ y = 5 + 3\ \mathsf{in}\ 5 + y\ \mathsf{end}) \Downarrow}\ \text{B-LET}}{\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ y = x + 3\ \mathsf{in}\ x + y\ \mathsf{end})\ \mathsf{end}}\ \text{B-LET}$$

3 Functions and Function application

Next we will add functions and function application. They follow the same principles we employed when handling let-expressions. We will first introduce nameless functions as we have encountered them in SML, and recursive functions.

$$\text{Expressions}\ \ e\ \ ::=\ \ \ldots \mid \mathsf{fn}\ x \Rightarrow e\ \mid e_1\ e_2 \mid \mathsf{rec}\ f \Rightarrow e$$

Before we consider evaluation, let us define the free variables occurring in a function and substitution.

$$
\begin{array}{lcl}
\mathsf{FV}(e_1\ e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\mathsf{fn}\ x \Rightarrow e\ ) & = & \mathsf{FV}(e) / \{x\} \\
\mathsf{FV}(\mathsf{rec}\ f \Rightarrow e\ ) & = & \mathsf{FV}(e) / \{f\}
\end{array}
$$

The definition highlights that both $\mathsf{fn}\ x \Rightarrow e$ and $\mathsf{rec}\ f \Rightarrow e$ bind variables. In the first case of $\mathsf{fn}\ x \Rightarrow e$ the input variable x is bound, and in the second case of $\mathsf{rec}\ f \Rightarrow e$ the function name is bound.

Next, we extend the definition for substitution. Similar to let-expressions, we must be careful to avoid the problem of variable capture.

$$
\begin{array}{lcll}
[e'/x](e_1\ e_2) & = & [e'/x]e_1\ [e'/x]e_2 & \\
[e'/x](\mathsf{fn}\ y \Rightarrow e\ ) & = & \mathsf{fn}\ y \Rightarrow [e'/x]e & \text{provided } x \neq y \text{ and } y \notin \mathsf{FV}(e') \\
[e'/x](\mathsf{rec}\ f \Rightarrow e\ ) & = & \mathsf{rec}\ f \Rightarrow [e'/x]e & \text{provided } x \neq f \text{ and } f \notin \mathsf{FV}(e')
\end{array}
$$

Functions are considered first-class values, and hence will evaluate to themselves (see rule B-FN). Evaluation of function application $(e_1\ e_2)$ is done in three steps. First, we evaluate e_1 which will (hopefully!) yield a function $\mathsf{fn}\ x \Rightarrow e$. In addition, we evaluate e_2 to obtain some value v_2. Finally, we need to evaluate the function body e where we have replaced any occurrence of x with v_2.

$$
\frac{}{\mathsf{fn}\ x \Rightarrow e\ \Downarrow \mathsf{fn}\ x \Rightarrow e}\ \text{B-FN} \qquad \frac{e_1 \Downarrow \mathsf{fn}\ x \Rightarrow e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\mathsf{apply}\ (e_1,\ e_2) \Downarrow v}\ \text{B-APP}
$$

This allows us to evaluate functions and function application. However, sometimes we would also like to write recursive functions. This is handled by the construct $\mathsf{rec}\ f \Rightarrow e$. The intention is that we separate the name of the recursive function and the name of the arguments passed to it. Intuitively, the correspondence between the function written in SML and in our fictional language is as follows. Recall that in SML instead of writing

```
fun f(x) = if x = 0 then 1 else f(x-1)
```

we could have written

```
val rec f = (fn x => if x = 0 then 1 else f(x-1))
```

which already resembles more our formal notation. Using our syntax, we can write this function as :

$$\mathsf{rec}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(x-1)$$

How do we evaluate recursive functions? – The simplest way to model recursive evaluation is again by substitution. To evaluate $\mathsf{rec}\ f \Rightarrow e$ we will simply replace any occurrence of f in e with the actual function definition $\mathsf{rec}\ f \Rightarrow e$. This means that when we need call

the recursive function we will have the actual code present. This is a very simple model which gives a very high level description and is ideally suited for reasoning about evaluation. However, we would like to point out that this is typically not how execution of recursive functions is implemented for real.

$$\frac{[\mathsf{rec}\ f \Rightarrow e\ /f]e \Downarrow v}{\mathsf{rec}\ f \Rightarrow e\ \Downarrow v}\ \text{B-REC}$$