15-312 Foundations of Programming Languages
Recitation 3: De Bruijn Representation

Daniel Spoonhower
spoons+@cs

September 10, 2003

1 Substitution

While *named* variables are certainly useful from a programmer's point of view, we saw last week in lecture how they can create problems for those of us trying to reason about and implement a language. In particular, we saw that defining substitution in the presence of named variables is not as trivial as it might initially seem.

For the discussion that follows we'll concentrate on a subset of MinML, shown in mathematical notation below.

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \ldots$$

(Why is this a meaningful subset? Convince yourself that the remaining cases will be handled in similar ways.)

To refresh your memory, carry out each of the following substitutions.

$$\{x/y\}\lambda z.y\ z \quad \{z/y\}\lambda z.y\ z \quad \{x/y\}\lambda y.y\ z$$

Notice when we are required to rename bound variables: this is easy to do when writing on a blackboard, but it's not as easy to carry out in an implementation.

2 A Nameless Representation

Rather than refer to variables by name, we will replace them with pointers to the binders that define them. For our nameless representation, we will use the following syntax.

$$e' ::= k \mid \lambda.e' \mid e'_1\ e'_2 \mid \ldots$$

(In assignment 2, this corresponds to the version of the abstract syntax represented by the structure `DBMinML`.)

For example, the expression

$$\lambda x.\lambda y.x$$

becomes

$$\lambda.\lambda.2$$

(How would we write the identity function in this syntax?) Write the following expressions in de Bruijn form:

$$\lambda s.\lambda z.s\ (s\ z)$$
$$\lambda x.\lambda y.y\ (\lambda z.x\ z)$$

Define a (recursive) function that takes named expressions to their corresponding nameless counterparts. (Hint: define one clause for each form of expression.)

2.1 Substitution with de Bruijn Indices

Take our original example of substitution, this time in de Bruijn form. How would we carry out substitution in this case?

$$\{3/1\}\lambda.2\ 1$$

(We will chose an arbitrary index for each of the free variables.) Each time we move a variable under a function abstraction, we simply add one to its index. (We must also remember to increment the variable for which we are substituting.) We can extend this idea in a straightforward manner to simple applications:

$$\{(2\ 3)/1\}\lambda.2\ 1$$

Just as before, we add one to the variable that is being replaced (1) as it is pushed under the abstraction, along with each of the variables in the left-hand side of the substitution. Unfortunately, this is not always the case. Think about the following substitution:

$$\{\lambda.1/1\}\lambda.2\ 1$$

In this case, we should *not* modify the left-hand side of the substitution, because all of the variables that appear in it are bound *within* it. In other words, it is a *closed* expression. Any closed expressions may be substituted as is, without any shifting of variables.

While you work on assignment 2, you should convince yourself that only closed expressions are substituted for variables. You may take advantage of this fact in your implementation, but leave yourself a reminder to reconsider this invariant in future assignments!

2.2 Further Reading

- Pierce gives a more detailed treatment of de Bruijn indices (including shifting and substitution) in Chapter 6 of *Types and Programming Languages*.