CS411 Notes 6 – Copy Rule Definition

A. Demers

12 Feb 2001

As threatened, we now give a formal development of a copy rule for which we can give lazy evaluation proof rules.

1 Substitution

We start with a formal definition of substitution. We will denote by

$$[e/f]e_0$$

the result of replacing all free occurrences of the function variable f by expression e in e_0. Deciding which occurrences of f will be "free" (hence candidates for substitution) is the hard part. This definition of free variables is implicit in the definition of substitution, and affects only the cases involving expression abstractions, which are presented last.

As you would expect, the definition is by recursion on the structure of **Exp**. The rules for *IMPX* without function declarations are obvious, and are given here without comment. Parentheses are for the reader's convenience in grouping terms, and are not part of the actual syntax.

$$[e/f]n \quad = \quad n$$

$$[e/f]X \quad = \quad X$$

$$[e/f]\psi e_0 \quad = \quad \psi[e/f]e_0$$

$$[e/f]e_0\theta e_1 \quad = \quad ([e/f]e_0)\theta([e/f]e_1)$$

$$[e/f]X \leftarrow e_0 \quad = \quad X \leftarrow ([e/f]e_0)$$

$$[e/f](e_0; e_1) \quad = \quad ([e/f]e_0); ([e/f]e_1)$$

$$[e/f](\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) \quad = \quad \textbf{if } [e/f]e_0 \textbf{ then } [e/f]e_1 \textbf{ else } [e/f]e_2$$

$$[e/f](\textbf{while } e_0 \textbf{ do } e_1) \quad = \quad \textbf{while } [e/f]e_0 \textbf{ do } [e/f]e_1$$

Again, these clauses are straightforward. Now comes the hard part: the *IMPX*-1 rules involving expression abstractions.

$$[e/f]g() \quad = \quad g() \qquad \text{where } g \not\equiv f$$

$$[e/f]f() \quad = \quad e$$

$$[e/f](\textbf{let } g \sim e_0 \textbf{ in } e_1) \quad = \quad \textbf{let } g \sim ([e/f]e_0) \textbf{ in } ([e/f]e_1)$$

I lied – those rules were easy, too. The real difficulty arises when we try to define substitution into a **let** block that *reuses* some function variable f. The following (arguably broken) attempt reflects the (arguably broken) lazy evaluation rule (LX1.call.l) from above:

$$[e/f](\textbf{let } f \sim e_0 \textbf{ in } e_1) \quad = \quad \textbf{let } f \sim e_0 \textbf{ in } e_1 \qquad \text{(SUB.call.bad)}$$

One consequence of this definition is that occurrences of f in e_1 are not replaced. That seems fine – such occurrences should refer to the newly-introduced binding e_0 for f. But what about occurrences of f in e_0? Definition (SUB.call.bad) says such occurrences also are *not* replaced, leading to the problems of dynamic scoping we've discussed earlier. The alternative – which works out much better, trust me – is to substitute into e_0:

$$[e/f](\textbf{let } f \sim e_0 \textbf{ in } e_1) \quad = \quad \textbf{let } f \sim [e/f]e_0 \textbf{ in } e_1 \qquad \text{(SUB.call)}$$

Eventually, this will lead to a system in which the defining expression e_0 in an abstraction gets its meaning from the context surrounding the **let** block, in spite of the possibility that occurrences of e_0 may be substituted into deeply nested subexpressions of e_1 by the semantic rules. Of course, we have more work to do before we get to that point.

As we have already mentioned, there is a notion of "free variable" implicit in any definition of substitution. What definition of free variable is induced by the substitution definition just given? A straightforward structural recursion defines the "Free Function Variables" of an element of **Exp**, a function

$$\mathit{FFV} \;:\; \mathbf{Exp} \rightarrow 2^{\mathbf{Fname}}$$

that computes the (finite) set of free variables of an *IMPX* expression. Most of the clauses have the simple form

$$\mathit{FFV}(\ldots e_i \ldots) \quad = \quad \ldots \cup \mathit{FFV}(e_i) \cup \ldots$$

For example,

$$\mathit{FFV}(n) \quad = \quad \emptyset$$

$$\mathit{FFV}(\psi e_0) \quad = \quad \mathit{FFV}(e_0)$$

$$\mathit{FFV}(e_0; e_1) \quad = \quad \mathit{FFV}(e_0) \cup \mathit{FFV}(e_1)$$

and so forth, so we won't give them all explicitly. As usual, the interesting claus-es in the definition deal with the constructors of *IMPX*-1 that define expression abstractions themselves:

$$\mathit{FFV}(f()) \quad = \quad \{f\}$$

This clause states that a reference to function variable f is free in itself, which should not be too surprising.

$$\mathit{FFV}(\mathbf{let}\ f \sim e_0\ \mathbf{in}\ e_1) \quad = \quad \mathit{FFV}(e_0) \cup (\mathit{FFV}(e_1) - \{f\})$$

This clause reflects the fact that, using the form (SUB.call) for the definition of substitution above, any free variable of e_0, including f itself, is a candidate for substitution, and thus is a free variable of the **let** block. However, a free occurrence of f in e_1 is *not* free in the **let** block, as it is bound to the new definition $f \;=\; e_0$ being introduced by the block.

2 Capture

Recall our earlier discussions of "capture" of free variables during substitution. Informally, capture occurs when an expression e containing free variable f is substituted into the scope of a **let** block that (re-)defines f, for example

$$[g()/f]\mathbf{let}\ g \sim 1\ \mathbf{in}\ f() + f()$$

As usual, we can give a formal definition of this concept by recursion on the structure of program expressions. We define a predicate K such that

$$K(e, f, e') \qquad \Leftrightarrow \qquad [e/f]e' \quad \text{results in capture}$$

The clauses of the definition are:

$$K(e, f, \textbf{let } f \ \sim \ e_0 \textbf{ in } e_1) \ = \ K(e, f, e_0)$$

$$\begin{aligned} K(e, f, \textbf{let } g \ \sim \ e_0 \textbf{ in } e_1) \ &= \ K(e, f, e_0) \vee K(e, f, e_1) \vee \\ &\quad ((f \in \mathit{FFV}(e_1)) \wedge (g \in \mathit{FFV}(e))) \end{aligned}$$

The final disjunct of this clause is the important one – in words, f occurs free in the block body and g occurs free in the expression being substituted there.

$$K(e, f, f()) \ = \ \textbf{false}$$

Yes, I really mean **false** here. You might expect **true**, but that case is covered by the previous clause.

$$K(e, f, \ldots e_i \ldots) \ = \ \ldots \vee K(e, f, e_i) \vee \ldots$$

This is our usual catch-all clause – capture occurs in an expression if it occurs in any of its subexpressions.

3 Substitution Without Capture

We are now in a position to define a substitution operation that explicitly avoids capture. We denote by

$$[e//f]e_0$$

the *capture-avoiding* (or *safe*) substitution of e for f in e_0.

For most constructors the definition of $[\cdot//\cdot](\cdot)$ corresponds exactly with that of $[\cdot/\cdot](\cdot)$. Thus,

$$[e//f]n \ = \ n$$

$$[e//f]X \quad = \quad X$$

$$[e//f]\psi e_0 \quad = \quad \psi([e//f]e_0)$$

and so forth, until we get to the final few clauses for the *IMPX*-1 constructors that treat function references and **let** blocks. Function references are simple:

$$[e//f]g() \quad = \quad g \qquad g \not\equiv f$$

$$[e//f]f() \quad = \quad e$$

The clauses for substituting into a **let** block are sufficiently subtle that I got them wrong in lecture the first time. To motivate them, observe that capture occurs when an expression e is substituted into the body a **let** block that binds some variable f that occurs free in e. Therefore, we can avoid capture by making certain that, whenever an expression e is substituted into the body of a **let** block, the bound variable of that **let** block does not occur free in e. This is not difficult to achieve. We simply rename the bound variables of **let** blocks when we substitute into them. Specifically, consider the two blocks

$$\textbf{let } f \sim e_0 \textbf{ in } e_1 \quad \text{and} \quad \textbf{let } g \sim e_0 \textbf{ in } [g()/f]e_1$$

where there are no free occurrences of g in e_1. These blocks are not quite equivalent. The first block will capture free occurrences of f substituted into it; the second will capture free occurrences of g. Absent free occurrences of f or g, the blocks behave identically. Absent free occurrences of g, the renaming of f to g has the effect of eliminating capture of f. This near-equivalence should be intuitively clear: the particular bound variable name used in a **let** block should not matter, as long as it is used consistently and does not cause capture. The following clauses avoid capture by renaming the bound variable of any **let** block encountered during substitution, choosing a new name that cannot possibly participate in capture.

$$[e//f](\textbf{let } g \sim e_0 \textbf{ in } e_1) \quad = \quad \textbf{let } h \sim [e//f]e_0 \textbf{ in } [e//f]([h/g]e_1)$$

$$[e//f](\textbf{let } f \sim e_0 \textbf{ in } e_1) \quad = \quad \textbf{let } h \sim [e//f]e_0 \textbf{ in } [h/f]e_1$$

where h is a new variable that does not occur in the original expressions. To ensure that $[\cdot//\cdot](\cdot)$ is a single-valued function the new variable h must be chosen deterministically. This is fairly easy to arrange – for example, given a well-founded total order on **Fname**, simply let h be the least name not appearing (either free or bound) in e or e_1.

The following property is easy to prove by structural induction, but is not quite automatically true, and deserves to be noted:

Theorem: Let

$$e_0 \in \textbf{Exp} \qquad \text{and} \qquad e_1 \in \textbf{Exp}$$

Then

$$[e_0/f]e_1 \in \textbf{Exp} \qquad \text{and} \qquad [e_0//f]e_1 \in \textbf{Exp}$$

That is, syntactic correctness of expressions is preserved by either ordinary substitution or safe substitution.

4 Lexical Scope Copy Rule

Finally we have enough mechanism to define precisely a copy rule for lexical scope. Recall our informal presentation of the copy rule

$$\textbf{let } f \sim e_0 \textbf{ in } \ldots f \ldots f \ldots \quad \Rightarrow \quad \ldots \quad \ldots e_0 \ldots e_0 \ldots$$

Even with the new formal definition of safe substitution, $[\cdot//\cdot](\cdot)$, this technically does not provide a definition of copy rule substitution, since it only tells us what to do with an expression whose top-level constructor is **let**.

There are (at least) two ways we can proceed.

1. We can define a function C that maps expressions to equivalent "copy-rule-expanded" expressions. Naturally, this will be done by recursion on the structure of expressions. Such a definition is guaranteed to produce a well-defined function, because it imposes a particular order in which copy rule substitutions must be performed.
2. We can give a set of proof rules defining a relation $\mapsto^*$ that holds between expressions e_0 and e_1 whenever e_1 can be obtained from e_0 by some number of applications of copy rule expansion. With a little care, we can define the proof rules to allow copy rule substitutions to be performed on arbitrary proper subexpressions in arbitrary order. We have already done examples of copy rule substitution on proper subexpressions many times in this discussion – this just reflects the intuitive notion that one can always "substitute equals for equals." This approach raises soundness issues. For example, there is no automatic guarantee that an expression cannot be provably reducible to two different numbers.

Below we pursue both approaches.

4.1 Copy Rule as Function

Here we define the copy rule as a function C that maps each expression to an equivalent expression from which all **let** blocks have been eliminated by copy rule substitutions.

As usual, the clauses for *IMPX* constructors are straightforward.

$$C(n) \;=\; n$$

$$C(X) \;=\; X$$

$$C(\psi e_0) \;=\; \psi(C(e_0))$$

and so on, through

$$C(\textbf{while } e_0 \textbf{ do } e_1) \;=\; \textbf{while } C(e_0) \textbf{ do } C(e_1)$$

Now the interesting clauses for the *IMPX*-1 abstraction rules:

$$C(f()) \;=\; f()$$

$$C(\textbf{let } f \sim e_0 \textbf{ in } e_1) \;=\; [C(e_0)//f]C(e_1)$$

The use of capture-avoiding substitution in this clause is absolutely critical. It is the reason this copy rule definition corresponds to lexical rather than dynamic scope.

Note also the ordering imposed on substitution by this last clause. It says that we reduce the **let** block

$$\textbf{let } f \sim e_0 \textbf{ in } e_1$$

by substituting the fully-reduced version of e_0 into the fully-reduced version of e_1. Thus, reduction proceeds in some sense “bottom-up.”

4.2 Copy Rule as Proof Rules

Here we pursue the second approach discussed above for defining the copy rule. We give proof rules defining a relation $\mapsto^*$ corresponding to one or more applications of copy rule substitution.

Reflexivity

$$\frac{}{e \mapsto^* e}$$

Every expression reduces to itself by applying 0 substitutions.

Transitivity

$$\frac{e_0 \mapsto^* e_1 \qquad e_1 \mapsto^* e_2}{e_0 \mapsto^* e_2}$$

Two reduction sequences can be "pasted together" to produce a longer reduction sequence.

Substitutivity

$$\frac{e_0 \mapsto^* e_0' \qquad e_1 \mapsto^* e_1'}{[e_0//f]e_1 \mapsto^* [e_0'//f]e_1'}$$

This rule allows substitution of equals for equals. It's subtle, and deserves careful thought. I sincerely hope (and believe) I've got it right . . .

Abstraction

$$\frac{}{\textbf{let } f \sim e_0 \textbf{ in } e_1 \ \mapsto^* \ [e_0//f]e_1}$$

This rule relates **let** binding to capture-avoiding substitution.

4.3 Properties of the Definitions

If you have ever studied the lambda calculus, you will have noticed a close connection between the α and β conversion rules of the lambda calculus and our discussion of the copy rule and capture in *IMPX*. Eventually we shall discuss the lambda calculus, and we'll defer formal proofs to that part of the course. But just for the record, we state the corresponding properties of *IMPX* here without proof.

Termination The *termination* property states that there are no infinite reduction sequences; that is, there is no way for a copy rule reduction sequence to get into an infinite loop. This is also a property of the typed (but not the untyped) lambda calculus. Since our proof rules define a reflexive relation $\mapsto^*$, we have to characterize finiteness of reduction sequences in a slightly roundabout way.

We say a sequence $e_0, e_1, \ldots$ is *eventually constant* if

$$(\exists i)(\forall j > i) e_j = e_i$$

Given that definition, the termination property of the copy rule is simply

Theorem: Every countably infinite $\mapsto^*$ chain is eventually constant.

Note it is obvious from the definition of C that every *IMPX* expression has at least one reduction sequence that terminates in an irreducible expression (an expression that contains no **let** blocks). The theorem implies the stronger claim that *every* nontrivial infinite reduction sequence eventually reaches an irreducible expression; i.e., it is not possible to make an infinite sequence of "bad" choices and thereby fail indefinitely to reach an irreducible expression.

Confluence The *confluence* property is a consistency property of the proof rules:

Theorem: Suppose

$$e_0 \mapsto^* e_1 \qquad \text{and} \qquad e_0 \mapsto^* e_2$$

Then there exists e_3 such that

$$e_1 \mapsto^* e_3 \qquad \text{and} \qquad e_2 \mapsto^* e_3$$

Thus, any e reduces to at most one irreducible expression.

Combining the above theorems, we get the delightful conclusion that for a given expression e *every* reduction sequence eventually terminates in the same irreducible expression $C(e)$.

Soon we shall present a new set of lazy evaluation proof rules that implement lexical scope. To prove these rules are consistent with our newly-defined copy rule, it will suffice to show that they agree with the old semantics on the language without abstractions (i.e. the *IMPX* subset), and evaluation of an expressions e containing **let** blocks is equivalent to evaluation of the **let**-block-free expression $C(e)$.

We could defined a C function and $\mapsto^*$ relation using unsafe substitution $[\cdot/\cdot](\cdot)$ rather than capture-avoiding substitution $[\cdot//\cdot](\cdot)$. What would happen? If you experiment with a few of the examples of capture we used earlier in our discussion of dynamic scope, you will discover that *the confluence property fails*. The failure of confluence – hence the failure of an expression to have a unique equivalent irreducible form – is for me the strongest argument against dynamic scope.