CS411 Notes 7 – Lexical Scope

A. Demers

13 Feb 2001

In a sense our definition of the copy rule as a function leads directly to a semantics for programs with expression abstractions. Suppose e is an expression with no free variables. Then to show

$$\langle e, \phi_0, \sigma \rangle \ \rightarrow \ \langle n, \sigma' \rangle$$

where e is an expression in *IMPX*-1, it is sufficient to show

$$\langle C(e), \sigma \rangle \ \rightarrow \ \langle n, \sigma' \rangle$$

using the *IMPX* rules. This makes sense because C maps closed *IMPX*-1 expressions (those containing no references to unbound function names) into equivalent *IMPX* expressions, for which the old rules were sufficient.

The technique of defining a language extension by conversion rules that translate extended programs to equivalent programs in the core language is not uncommon, and we shall see it again.

However, we can give large step rules for lazy static scope by changing the structure of the environment a bit.

1 Lazy Rules for Lexical Scope

Our two previous sets of rules for *IMPX* have used environments of the form

$$\Phi \ = \ \mathbf{Fname} \rightarrow \mathbf{Fval}$$

where **Fval** was chosen to be numbers (for the eager rules) or expressions (for the lazy rules). Using simple expressions for **Fval** led to a dynamic scope semantics,

in which an invocation evaluated in the invoking environment rather than the defining environment. We have argued that the alternative of lexical scope – evaluating an invocation in the defining environment rather than the invoking environment – would be a better choice.

Suppose we try to define lexical scope explicitly, by storing in the environment not just an expression to evaluate, but also the environment in which it should be evaluated. We would like our environments to satisfy

$$\mathbf{Fval} \;=\; (\mathbf{Exp} \times \Phi) \qquad \text{(FV-lazylex)}$$

or, equivalently

$$\Phi \;=\; \mathbf{Fname} \to (\mathbf{Exp} \times \Phi) \qquad \text{(Env-lazylex)}$$

This "definition" has serious foundational problems, allowing environments that properly contain themselves. Fortunately, we can fix things without too much effort. We don't need arbitrary environments; instead, we can get by with a recursively defined set of environments that are "ranked" by their nesting depth. We treat them as if they were functions by explicitly defining a "map" or "apply" operation suggestively called **map**. At the lowest rank we have an identifiable "empty" environment:

$$\Phi_0 \;=\; \{\phi_0\}$$

Higher ranks are defined by

$$\Phi_{i+1} \;=\; \Phi_i \;\cup\; (\mathbf{Fname} \to (\mathbf{Exp} \times \Phi_i))$$

with an application function defined by

$$\mathbf{map}(\phi, f) \;=\; \begin{cases} \langle 0, \phi_0 \rangle & \phi = \phi_0 \\ \phi(f) & \text{o.w.} \end{cases}$$

The set of valid environments is just the union over all finite ranks:

$$\Phi \;=\; \bigcup \{\Phi_i \mid i \geq 0\}$$

With this definition of Φ, we present the following rules for expression abstraction and invocation. They implement lexical scope by carrying the associated environments along with expression abstractions.

Let Blocks

$$\frac{\langle e_1, \phi[\langle e_0, \phi\rangle/f], \sigma\rangle \to \langle n, \sigma'\rangle}{\langle \textbf{let } f \sim e_0 \textbf{ in } e_1, \phi, \sigma\rangle \to \langle n, \sigma'\rangle} \qquad \text{(LX1.let.ll)}$$

To evaluate a **let** block, evaluate the body e_1 using an environment in which the defined function variable f is bound to a pair consisting of the abstracted expression e_0 and the current environment ϕ.

Invocation

$$\frac{\langle e_f, \phi_f, \sigma\rangle \to \langle n, \sigma'\rangle}{\langle f(), \phi, \sigma\rangle \to \langle n, \sigma'\rangle} \qquad \text{where } \langle e_f, \phi_f\rangle = \phi(f) \qquad \text{(LX1.call.ll)}$$

To evaluate a function invocation, look up the function name f in the environment to get a pair consisting of an expression e_f and an environment ϕ_f; then evaluate e_f using environment ϕ_f and the current store.

1.1 An Example

We return to an example we discussed in connection with the dynamic scope lazy rules:

$$\textbf{let } f \sim 1 \textbf{ in let } g \sim f() \textbf{ in let } f \sim 2 \textbf{ in } g()$$

Recall this program returns 1 using eager evaluation rules, but we showed that it returns 2 using the dynamic scope lazy rules. Here we evaluate it using the new lexical scope lazy rules to produce the correct answer (i.e. 1). As before, the derivation is a simple chain of rule instances with one hypothesis each:

$$\langle \textbf{let } f \sim 1 \textbf{ in let } g \sim f() \textbf{ in let } f \sim 2 \textbf{ in } g(), \phi_0, \sigma\rangle \quad \to \quad \langle 1, \sigma\rangle$$

follows by rule (LX1.let.ll) with hypothesis

$$\langle \textbf{let } g \sim f() \textbf{ in let } f \sim 2 \textbf{ in } g(), \phi_0[\langle 1, \phi_0\rangle/f], \sigma\rangle \quad \to \quad \langle 1, \sigma\rangle$$

If we introduce the abbreviation

$$\phi_1 \quad = \quad \phi_0[\langle 1, \phi_0\rangle/f]$$

then the above relation is the conclusion of rule (LX1.let.ll) with hypothesis

$$\langle \textbf{let } f \sim 2 \textbf{ in } g(), \phi_1[\langle f(), \phi_1\rangle/g], \sigma\rangle \quad \to \quad \langle 1, \sigma\rangle$$

Now abbreviate

$$\phi_2 \quad = \quad \phi_1[\langle f(), \phi_1 \rangle / g]$$

and this is the conclusion of rule (LX1.let.ll) with hypothesis

$$\langle g(), \phi_2[\langle 2, \phi_2 \rangle / f], \sigma \rangle \quad \rightarrow \quad \langle 1, \sigma \rangle$$

This follows by rule (LX1.call.ll) with hypothesis

$$\langle f(), \phi_1, \sigma \rangle \quad \rightarrow \quad \langle 1, \sigma \rangle$$

since it is easy to verify that

$$(\phi_2[\langle 2, \phi_2 \rangle / f])(g) \quad = \quad \langle f(), \phi_1 \rangle$$

Note this is the point where the derivation using lexical lazy rules departs significantly from our previous derivation using dynamic lazy rules – this invocation of f() is to be evaluated using environment ϕ_1, which does not reflect the binding $f \sim 2$. Instead, we use rule (LX1.call.ll) with hypothesis

$$\langle 1, \phi_0, \sigma \rangle \quad \rightarrow \quad \langle 1, \sigma \rangle$$

since

$$\phi_1(f) \quad = \quad \langle 1, \phi_0 \rangle.$$

As the expression in this hypothesis is irreducible, the derivation is complete.

1.2 Soundness

This section should contain a proof that the proof rules just developed satisfy the copy rule. It is yet to be written ....

2 Recursive Procedures

In our discussion up to now we have interpreted the block

let $f \sim e_0$ **in** e_1

as a nonrecursive definition. That is, the abstracted expression e_0 is interpreted in the surrounding environment, which does not include the new binding being created for f. This intepretation is essential if we want the termination property to hold for copy rule substitution. However, it prevents us from doing recursive function declarations in the *IMPX*-1 language.

Here we modify the lazy lexical scope rules so they support recursive function declarations.

Let's replace the **let** block construct with **rec** blocks:

$$e \ ::= \ \textbf{rec}\ f \sim e_0\ \textbf{in}\ e_1$$

The interpretation is that free occurrences of f in e_0 should be bound to the newly introduced f rather than being free in the **rec** block as a whole. This interpretation enables us to define recursive functions. For example, the expression

$$\textbf{rec}\ w \sim \textbf{if}\ x > 0\ \textbf{then}\ f(); x = x - 1; w()\ \textbf{else}\ 0\ \textbf{in}\ w()$$

should behave like the loop

$$\textbf{while}\ x > 0\ \textbf{do}\ (f(); x = x - 1)$$

Proceeding boldly down the garden path, let's assume as in our previous semantics that the the environment will contain pairs $\langle e, \phi \rangle$, where e is a function body and ϕ is the environment in which it should be evaluated, In that case the rule for function invocations will be identical to rule (LX1.call.ll) above. What will the rule for **rec** blocks look like? Presumably it will have the form

$$\frac{\langle e_1, \phi[\langle e_0, \phi' \rangle / f], \sigma \rangle \rightarrow \langle n, \sigma' \rangle}{\langle \textbf{rec}\ f \sim e_0\ \textbf{in}\ e_1, \phi, \sigma \rangle \rightarrow \langle n, \sigma' \rangle}$$

and the only unanswered question is the exact form of ϕ', the environment in which the body of f (that is, e_0) should be evaluated.

What can we say about ϕ'? In the nonrecursive rule (LX1.let.ll), ϕ' is just ϕ. In the recursive case, it is intuitively clear that ϕ will not suffice: ϕ' must include a binding for f; and that binding should be the *same* one that is used in evaluating the block body e_1. This strongly suggests

$$\phi' \ = \ \phi[\langle e_0, \phi' \rangle / f]$$

and we have arrived at the same foundational problems were were trying to avoid when we defined Φ and the set of all finite-rank environments. A "continued fraction" representation of ϕ' would look something like

$$\phi' \;=\; \phi[\langle e_0, \phi[\langle e_0, \phi[\langle e_0, \ldots\rangle/f]\rangle/f]\rangle/f]$$

Where the ellipsis indicates "infinitely-deep" nesting. It should be clear no such object exists in any finite-rank environment Φ_i.

Here are two ways to get out of this difficulty. Each is clever in its own way.

2.1 A Specialized Invocation Rule

We have argued that we need an unrealizable environment ϕ' of infinite rank. But we don't *really* need it – we can instead change the invocation rule to construct adequate approximations to ϕ' "one rank at a time." This approach leads to the following pair of rules:

Rec Blocks

$$\frac{\langle e_1, \phi[\langle e_0, \phi\rangle/f], \sigma\rangle \to \langle n, \sigma'\rangle}{\langle \textbf{rec } f \sim e_0 \textbf{ in } e_1, \phi, \sigma\rangle \to \langle n, \sigma'\rangle} \qquad \text{(LX1.rec.ll)}$$

This is identical to (LX1.let.ll).

Invocation

$$\frac{\langle e_f, \phi_f[\langle e_f, \phi_f\rangle/f], \sigma\rangle \to \langle n, \sigma'\rangle}{\langle f(), \phi, \sigma\rangle \to \langle n, \sigma'\rangle} \qquad \text{where } \langle e_f, \phi_f\rangle = \phi(f) \quad \text{(LX1.callr.ll)}$$

To evaluate a recursive function invocation, look up the function name f in the environment to get a pair consisting of an expression e_f and an environment ϕ_f. then evaluate e_f in the current store, using a new environment obtained by re-introducing the definition of f into ϕ_f.

This rule has the effect of approximating the unrealizable ϕ' incrementally. Note that in the nonrecursive rule (LX1.call.ll), the environment ϕ_f appearing in the hypothesis is at strictly lower rank than the original environment ϕ. In this rule, the environment in the hypothesis is an updated version of ϕ_f, and is potentially at the same rank as ϕ.

2.2 A Syntactic Transformation

Here is a slightly different solution, much more syntactic, and similar in spirit to the **while** rule (LX.wht). We exploit the following syntactic equivalence:

$$\textbf{rec } f \sim e_0 \textbf{ in } e_1 \quad \equiv \quad \textbf{rec } f \sim (\textbf{rec } f \sim e_0 \textbf{ in } e_0) \textbf{ in } e_1$$

(Note this is an equivalence for **rec** blocks, and not for **let** blocks.) We simply build this syntactic transformation into the **rec** rule, as follows:

Rec Blocks

$$\frac{\langle e_1, \phi[\langle \textbf{rec } f \sim e_0 \textbf{ in } e_0, \phi\rangle/f], \sigma\rangle \rightarrow \langle n, \sigma'\rangle}{\langle \textbf{rec } f \sim e_0 \textbf{ in } e_1, \phi, \sigma\rangle \rightarrow \langle n, \sigma'\rangle} \qquad \text{(LX1.rec.ll2)}$$

In this rule, the environment for evaluating a recursive invocation binds f not just to its definition e_0, but to a newly-synthesized **rec** block defining f.

Invocation

$$\frac{\langle e_f, \phi_f, \sigma\rangle \rightarrow \langle n, \sigma'\rangle}{\langle f(), \phi, \sigma\rangle \rightarrow \langle n, \sigma'\rangle} \qquad \text{where } \langle e_f, \phi_f\rangle = \phi(f) \qquad \text{(LX1.rcall.ll2)}$$

This rule is identical to (LX1.call.ll).

It is a fairly easy exercise to prove the equivalence of these two approaches.