CS411 Notes 4-5 – Abstraction I

A. Demers

12 Feb 2001

1 IMPX

The original *IMP* syntax uses separate syntactic sets to enforce some limited static semantic properties – for example, some primitive type checking is performed by maintaining a distinction between **Aexp** (arithmetic expressions) and **Bexp** (Boolean expressions).

Eventually we shall specify typechecking and other static semantic checks using proof rules, similar to the rules of our structural operational semantics. Proof rules are considerably more powerful than the original technique, and very soon now we will need the extra power. As an added benefit, using proof rules for static semantics will enable us to simplify the formation rules for the language, since they will no longer be required to do *any* typechecking.

In the next couple of lectures we won't yet be doing any sophisticated typechecking; but we would still like a simpler set of formation rules. So here is a revised version of *IMP* called *IMPX* in which we eliminate the distinction between numeric and Boolean expressions (we use integers in place of Booleans, in the style of C), and we eliminate the distinction between commands and expressions (every language construct produces a value, and expressions can affect the store). The name "*IMPX*" comes from "IMPerative eXpression language."

We give the formation rules and a large-step structural operational semantics for *IMPX*. By this point you should be fairly comfortable with a succinct presentation like this ...

1.1 Syntax

The syntax of *IMPX* requires only three syntactic sets:

N the (positive and negative) integers.

Loc the set of locations of program variables.

Exp the set of expressions.

The associated notational conventions are:

N contains $n, n', n'', n_0, n_1, \ldots$

Loc contains $X, X', X_0, Y, Y', Y_0, \ldots$

Exp contains $e, e', e'', e_0, e_1, \ldots$

The formation rules for *IMPX* are:

$$\begin{aligned} e \ ::= \ & n \mid X \mid \psi e_0 \mid e_0 \theta e_1 \mid X \leftarrow e_0 \mid e_0; e_1 \\ & \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \mid \textbf{while } e_0 \textbf{ do } e_1 \end{aligned}$$

Here ψ is any unary operator (e.g. $\neg$ or unary $-$) and θ is any binary operator (e.g. $+$, $*$, $\vee$).

1.2 Large Step Semantics of *IMPX*

Here is a large step semantics for the new language.

The definition of the store is unchanged from our large step semantics for *IMP*:

$$\Sigma \ = \ \textbf{Loc} \to \textbf{N}$$

that is, the store consists of (total) functions from locations to values. Technically, it would be sufficient to use "finite" (almost-everywhere-constant) functions, as defined in the first problem set, or even (with just a little more work) finite sets of pairs $\langle X, n \rangle$.

Since *IMPX* makes no distinctions among arithmetic and Boolean expressions and commands, we need only one kind of configuration instead of three. A configuration is a pair

$$\langle e, \sigma \rangle \qquad e \in \textbf{Exp}, \sigma \in \Sigma$$

A configuration represents an expression to be evaluated in a store. A *terminal* configuration is one in which the expression part is already fully evaluated – that is, a configuration of the form $\langle n, \sigma \rangle$.

The large step *execution relation* $\rightarrow$ holds between a configuration and a terminal configuration. Relation

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

holds when evaluation of e starting in state σ terminates producing the result value $n \in \mathbf{N}$ and final state σ'.

The proof rules for *IMPX* are quite similar to those for *IMP* with the **resultis** extension (which allows expressions to affect the store, as discussed in the first problem set).

Variables

$$\frac{}{\langle X, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle} \qquad \text{(LX.vref)}$$

A variable reference evaluates to the contents of the variable's location in the store, and has no effect on the store.

Unary Operations

$$\frac{\langle e_0, \sigma \rangle \rightarrow \langle m, \sigma' \rangle}{\langle \psi e_0, \sigma \rangle \rightarrow \langle n, \sigma' \rangle} \quad \text{where } n = \psi m \qquad \text{(LX.unop)}$$

According to this rule a unary operator returns a result without affecting the store. The final store σ' results from evaluating the subexpression e_0 returning the value m; evaluation of $n = \psi(m)$ happens in σ' with no side effects.

Binary Operations

$$\frac{\langle e_0, \sigma \rangle \rightarrow \langle n_0, \sigma_0 \rangle \qquad \langle e_1, \sigma_0 \rangle \rightarrow \langle n_1, \sigma_1 \rangle}{\langle e_0 \theta e_1, \sigma \rangle \rightarrow \langle n, \sigma_1 \rangle} \quad \text{where } n = n_0 \theta n_1 \qquad \text{(LX.binop)}$$

As we argued for the (LX.unop) rule, evaluation of a binary operator application returns a result without affecting the store.

Assignments

$$\frac{\langle e, \sigma \rangle \rightarrow \langle n, \sigma' \rangle}{\langle X \leftarrow e, \sigma \rangle \rightarrow \langle n, \sigma'[n/X] \rangle} \qquad \text{(LX.asgn)}$$

An assignment updates the store at the specified location to the value of the right-hand-side expression with no further effect on the store; it also produces the computed value as its result.

Sequential Composition

$$\frac{\langle e_0, \sigma\rangle \to \langle n_0, \sigma''\rangle \qquad \langle e_1, \sigma''\rangle \to \langle n_1, \sigma'\rangle}{\langle e_0; e_1, \sigma\rangle \to \langle n_1, \sigma'\rangle} \tag{LX.seq}$$

Evaluation of the sequential composition $e_0; e_1$ can be understood as the consecutive evaluation of e_0 and e_1 where the intermediate state σ'' (the result of executing e_0) appears explicitly in the instantiated rule. The value produced by e_0 is discarded (though we require that evaluating e_0 terminates producing *some* value). The value produced by e_1 is returned.

Conditionals

$$\frac{\langle e_0, \sigma\rangle \to \langle n_0, \sigma''\rangle \qquad \langle e_1, \sigma''\rangle \to \langle n_1, \sigma'\rangle}{\langle \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \sigma\rangle \to \langle n_1, \sigma'\rangle} \qquad \text{where } n_0 \neq 0 \tag{LX.ift}$$

Evaluation of a conditional whose test evaluates "true" (i.e., nonzero) evaluates the **then** clause, e_1.

$$\frac{\langle e_0, \sigma\rangle \to \langle n_0, \sigma''\rangle \qquad \langle e_2, \sigma''\rangle \to \langle n_2, \sigma'\rangle}{\langle \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \sigma\rangle \to \langle n_2, \sigma'\rangle} \qquad \text{where } n_0 = 0 \tag{LX.iff}$$

Evaluation of a conditional whose test evaluates "false" (i.e., zero) evaluates the **else** clause, e_2.

Loops

$$\frac{\langle e_0, \sigma\rangle \to \langle 0, \sigma'\rangle}{\langle \textbf{while } e_0 \textbf{ do } e_1, \sigma\rangle \to \langle 0, \sigma'\rangle} \tag{LX.whf}$$

A loop whose condition evaluates "false" terminates immediately with no further effect on the store, producing the value 0.

$$\frac{\langle e_0, \sigma\rangle \to \langle n_0, \sigma''\rangle \qquad \langle e_1;\ \textbf{while } e_0 \textbf{ do } e_1, \sigma''\rangle \to \langle n, \sigma'\rangle}{\langle \textbf{while } e_0 \textbf{ do } e_1, \sigma\rangle \to \langle n, \sigma'\rangle} \tag{LX.wht}$$

where $n_0 \neq 0$. Execution of a loop whose condition evaluates "true" is equivalent to "unrolling" the loop once – that is, executing the loop body e_1 and then *re-executing* the entire loop. Note the result ultimately produced by a terminating loop will necessarily be 0 by (LX.whf).

2 Abstraction and the Copy Rule

We are going to introduce a number of "principles" of language design, and illustrate them using variants of *IMPX* and (usually) large step semantics. Probably the most important of these principles is Abstraction. The version below is quoted from the Schmidt text, where it is attributed to Peter Landin in an early paper on Algol60.

Principle of Abstraction: The phrases of any semantically meaningful syntactic class may be named.

Stated thus, the principle is vague in at least a couple of ways.

First, what is the meaning of "semantically meaningful syntactic class" in the Principle? This is not an easy question to answer. Indeed, given an awkward syntactic definition I could probably manage to make every syntactic set semantically meaningful by giving a similarly awkward semantics. However, an example may help clarify the intent. The constructors for *IMPX* include

$$e \quad ::= \quad e_0 \theta e_1 \quad | \quad \ldots$$

to generate expressions over binary operators. We might instead have described the binary expressions by

$$e \quad ::= \quad e_0 d$$

where d is in a new syntactic set D including the rules

$$d \quad ::= \quad \theta e_0 d_0 \quad | \quad \langle \text{empty} \rangle$$

This new definition does indeed describe the same set of concrete expressions (though not exactly the same abstract syntax trees).

An element of D represents the "tail" of a compound binary expression, something like

$$d \quad = \quad +3-X$$

Is this a "semantically meaningful" phrase or not? We could certainly devise a *denotational* semantics that gave meaning to an element of D as a function from values and states to values and states, for example

$$[\![+3-X]\!](\langle m, \sigma \rangle) \quad = \quad \langle n, \sigma \rangle \qquad \text{where } n = m + 3 - \sigma(X)$$

But we won't be discussing denotational semantics for a while yet. The configurations and evaluation relation for the corresponding operational semantics would be rather more complex than the ones we have been using, and it is hard to see what it would mean to bind a name to such an ill-formed subexpression. Probably it would not be wise to apply the Abstraction Principle to phrases of *D*.

The second imprecision in our statement of the Abstraction Principle is the meaning of "may be named." Obviously just *naming* a phrase is uninteresting without a way subsequently to *refer* to the phrase by name. Referring to a name is generally described by some form of syntactic substitution. In the same spirit of precision embodied by the Abstraction Principle itself, we propose:

Copy Rule: A reference to a defined name may be syntactically replaced by a copy of the phrase bound to that name, yielding an equivalent program.

More symbolically, though not much more precisely,

$$\textbf{define } f \textbf{ by } \alpha \textbf{ in } \ldots f \ldots f \ldots \quad \equiv \quad \ldots \ \ldots \alpha \ldots \alpha \ldots$$

The term "copy rule" also dates back at least as far as Algol60.

Having developed a bit of intuition about the Abstraction Principle's notion of "naming semantically meaningful phrases," let's try to apply that intuition to *IMPX*. There are three syntactic classes in *IMPX*: **N**, **Loc**, and **Exp**. Intuitively each of them is "semantically meaningful," and the result of abstracting with respect to any of them is a more-or-less familiar concept from programming languages:

Numbers: A named number is just a constant declaration, as in Java, C++ and a host of other languages.

Locations: What is a named location in *IMPX*? Roughly, it is a name for a variable name – an *alias*. Few programming languages support alias as an explicit construct (it is similar, but not identical, to pointers). However, the notion exists in the somewhat disguised form of call-by-reference parameters in many languages, for example Pascal and C++.

Expressions: An expression abstraction – a named expression – corresponds to a 0-argument function or procedure in a conventional imperative programming language.

Because numbers and locations are atomic syntactic elements, they cannot contain references to newly-abstracted names, and it is comparatively easy to define number and location abstractions that obey a sensible copy rule. The structure of expression abstractions is much richer – an expression can contain references to previously-defined abstractions, or (if we're not careful) even to not-yet-defined abstractions. Such "free variables" and "capture" are where most of the interesting behavior arises, so we'll concentrate on expression abstraction below.

3 IMPX with Expression Abstraction

In this section we extend *IMPX* by expression abstractions, which we shall often just call "functions" even though they have no parameters. Creatively, we call the resulting language *IMPX*-1.

We want to add expression abstraction to *IMPX*, without adding any static semantics checking proof rules. A little thought should convince you that we will need to add a new syntactic set of names:

Fname: an infinite set of *function names*

with the convention

Fname contains $f, f', f_i, \ldots, g, g', g_i, \ldots$

Here (and whenever we define syntax by formation rules), we follow the convention that all the ground syntactic sets are disjoint. Specifically, we assume

$$\mathbf{Loc} \cap \mathbf{Fname} = \emptyset$$

so any given name may be either a function name or a location but not both.

With these new syntactic sets, the extension of *IMPX* to *IMPX*-1 requires only two new constructors. The first new constructor binds a name to an expression:

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

The second new constructor refers to a name:

$$e \quad ::= \quad f()$$

The parentheses in function reference are not strictly necessary, but they make program examples a bit easier to read.

We illustrate the intended meaning by a few examples. Keep in mind that our goal is for *IMPX*-1 programs to obey some sensible form of copy rule.

$$\textbf{let } f \sim 17 \textbf{ in } f() + 3 \quad \text{returns} \quad 20$$

This example is straightforward. Since evaluating the expression 17 always terminates producing the same result valus with no effect on the store, the behavior is indistinguishable from a constant declaration.

$$\textbf{let } f \sim 2 * X \textbf{ in } (X \leftarrow 1) + f() + (X \leftarrow 2) + f() \quad \text{returns} \quad 9$$

In this example the final store contains 2 at location X.

$$\textbf{let } f \sim \textbf{while } 1 \textbf{ do } 0 \textbf{ in } f() \quad \text{diverges}$$

The **while** loop inside f fails to terminate.

These intended meanings reflect some rather serious implicit assumptions. Next we'll develop a large step semantics for *IMPX*-1, and in the process make some of these assumptions (and their consequences) explicit.

4 Rules for Abstractions – Preliminaries

When we defined *IMPX*-1 by adding abstractions to *IMPX*, we added a new syntactic set **Fname** of names for abstracted expressions. This will force us to change the structure of the operational semantics. We need a way to carry around the definitions for the names that have been introduced in a program. For this we introduce function *environments*. An environment $\phi \in \Phi$ will map names to their values, that is

$$\Phi \quad = \quad \textbf{Fname} \rightarrow \textbf{Fval}$$

We haven't yet defined the set **Fval** of function values, but we'll explore several possibilities below.

The semantic rules for *IMPX*-1 will describe evaluation of an expression with a given environment ϕ and a given initial store σ; they will enable us to determine

the result value and the final store. Thus, the evaluation relation $\rightarrow$ will be a 5-place relation, with the interpretation that

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

holds exactly if e, when evaluated in environment ϕ and initial store σ, can terminate returning value n with final store σ'.

Augmenting the semantic rules to handle environments is quite straightforward for all the constructors of *IMPX* – that is, the constructors that do not involve defining or referring to abstractions. It simply involves carrying an environment unchanged between hypotheses and conclusion. A few examples of constructing an *IMPX*-1 rule from the corresponding *IMPX* rule should make this clear.

The rule (LX.vref) is

$$\frac{}{\langle X, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle}$$

Since there are no hypotheses, the rule simply becomes

$$\frac{}{\langle X, \phi, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle} \qquad \text{(LX1.vref)}$$

When one or more hypotheses are involved, the environment is duplicated without change. For example, rule (LX.binop) is

$$\frac{\langle e_0, \sigma \rangle \rightarrow \langle n_0, \sigma_0 \rangle \qquad \langle e_1, \sigma_0 \rangle \rightarrow \langle n_1, \sigma_1 \rangle}{\langle e_0 \theta e_1, \sigma \rangle \rightarrow \langle n, \sigma_1 \rangle} \quad n = n_0 \theta n_1$$

When environments are propagated to the hypotheses, the rule becomes

$$\frac{\langle e_0, \phi, \sigma \rangle \rightarrow \langle n_0, \sigma_0 \rangle \qquad \langle e_1, \phi, \sigma_0 \rangle \rightarrow \langle n_1, \sigma_1 \rangle}{\langle e_0 \theta e_1, \phi, \sigma \rangle \rightarrow \langle n, \sigma_1 \rangle} \quad n = n_0 \theta n_1 \quad \text{(LX1.binop)}$$

This pattern continues absolutely unchanged through the remaining *IMPX* rules.

Finally we reach the interesting rules – the ones that describe abstractions in *IMPX*-1.

It should be clear that the rule for **let** blocks will involve inserting something into the environment; and the rule for referring to an abstraction ("function call") will involve looking up the referenced name in the enviroment and using the result somehow. Thus, we can no longer put off defining the set **Fval** of function values, since these are what the environment contains.

What should we use as the "value" of an expression abstraction? On the face of it, there appear to be two equally sensible approaches.

One approach is to note that the value of an expression is a number. We let

$$\mathbf{Fval} \;=\; \mathbf{N} \qquad \text{(FV-eager)}$$

To evaluate the definition of an expression abstraction, we evaluate the expression to produce a number, which we insert into the environment. References to the defined name are evaluated by looking up the number in the environment and returning it. This approach is usually called *eager evaluation*, because the abstracted expression is evaluated once, as early as possible.

An alternative approach is to keep unevaluated expressions in the environment. We let

$$\mathbf{Fval} \;=\; \mathbf{Exp} \qquad \text{(FV-lazy)}$$

To evaluate the definition of an expression abstraction, we insert the expression itself directly into the environment. References to the defined name are evaluated by looking up the expression in the environment and evaluating it at that point to produce a number. This approach is usually called *lazy evaluation*, because the abstracted expression is not evaluated until its value is needed.

Eager and lazy evaluation have very different properties. As we shall see, neither of them obeys a reasonable copy rule.

5 Eager Rules

Here are rules for eager evaluation of expression abstractions. These rules use the environment definition (FV-eager) from above.

Let blocks

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

To evaluate a **let** block, first evaluate the bound expression e_0 returning some integer m, then evaluate the body e_1 in the resulting store using an environment in which the defined function variable f is bound to m.

Invocation

$$\frac{}{\langle f(), \phi, \sigma\rangle \;\rightarrow\; \langle \phi(f), \sigma\rangle} \qquad \text{(LX1.call.e)}$$

To evaluate a function invocation, look up the function name in the environment and return the value found, with no effect on the store.

These rules have some appeal. They are certainly simple to understand. However, even without a formal definition of the copy rule, we can argue that the rules *do not* satisfy any reasonably-defined copy rule. There are (at least) three different ways a copy rule can fail under eager evaluation.

Effects. Because *IMPX* has constructs that modify a shared store, an abstracted expression that depends on the store can violate the copy rule. For example, the program

$$X \leftarrow 1;\ \textbf{let}\ f \sim X\ \textbf{in}\ f() + (X \leftarrow 2) + f()$$

returns 4 under eager evaluation. By the copy rule, we would expect it to be equivalent to the transformed program

$$X \leftarrow 1; \ldots X + (X \leftarrow 2) + X$$

But this program returns 5 rather than 4, because the second evaluation of program variable X sees the updated value 2 rather than the initial value 1. It is easy to construct examples in which the relevant store modifications happen in the abstracted expression itself, such as

$$X \leftarrow 2;\ \textbf{let}\ f \sim X \leftarrow X * 3\ \textbf{in}\ f() + f() + f()$$

which returns 18 under eager evaluation and 78 under the copy rule.

Nontermination. Because *IMPX* has looping expressions, a program might abstract an expression containing a nonterminating loop. The program

$$\textbf{let}\ f \sim\ (\textbf{while}\ 1\ \textbf{do}\ 17)\ \textbf{in}\ 2$$

diverges under eager evaluation attempting to evaluate the **while** expression. The copy rule substitutes the abstraction into the program body *0 times* (since there are no invocations of f in the program body); resulting in the program

$$2$$

which clearly terminates.

Capture of Free Variables. This is the most subtle and interesting way to violate the copy rule. An expression being abstracted may itself contain references to abstraction names. How should these references be interpreted? Our evaluation rules interpret them according to the environment ϕ in effect at the point of evaluation. Because the copy rule has the effect of copying a expression, potentially to many different points of a program, it may have the effect of changing the environment in which evaluation takes place. For example, under our eager evaluation semantics the program

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

returns 1, reflecting the binding of f at the point where g is defined.

What does the copy rule say about this program? We might substitute the definition $f \sim 1$ into the definition of g to get

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

leading to a program that returns 1, just as the eager semantics says. However, it is at least plausible that we could substitute the definition $g \sim f()$ for the use of g in the inner block, to get

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

leading to a program that returns 2 rather than 1. Which is the "correct" interpretation?

An similarly problematic case arises whenever a **let** block redefines a function name that is used in the abstracted expression. For example, what is the meaning of

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

Using the eager evaluation rules, this program returns 2 – the "outer" definition of f yields a function that returns 1, the "inner" definition of f redefines f in terms of its previous definition (verify this from the rules), yielding a function that returns 2, and the latter function is invoked within the body of the program. What does the copy rule say?

These examples are problematic partly because they illustrate the imprecision of our definition of the copy rule (or really in our *lack* of a definition up to this point). Let's rewrite the last example with a subscript to identify each occurrence of f:

$$\textbf{let } f_0 \sim 1 \textbf{ in let } f_1 \sim f_2() + 1 \textbf{ in } f_3()$$

Consider applying the copy rule to this example. Naively, the copy rule says we should substitute 1 (the defining expression in the outermost **let** block) for occurrences of f in the block body.

1. Should we substitute for f_1? Certainly not! f_1 is a *binding occurrence*, a definition or "declaration" of the name f; substituting an expression for f_1 doesn't make sense (and doesn't even yield a syntactically correct program).

2. Should we substitute for f_3? Again, we should not. The invocation f_3 in in the body of the inner **let** block; it should refer to the definition f_1. Substituting the defining expression from definition f_0 would be wrong.

3. Should we substitute for f_2? Gosh. Think about this: if we *don't* substitute for f_2, that will have the effect of defining f_1 in terms of itself. If we *do* substitute for f_2, the effect will be to define f_1 as $1 + 1$, which is probably closer to the programmer's intent, and certainly closer to the meaning given by the eager evaluation rules. We may have to revisit this, however, when we consider recursive function definitions.

4. After substituting 1 for f everywhere we choose to do so, do we remove the **let** block introducing f_0 or do we keep it? More generally, does the copy rule say

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

   or

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

   These are definitely *not* equivalent when e_0 may contain occurrences of f.

We have raised four questions on interpreting the copy rule. We argued (convincingly, I hope) that the answers to the first two should be "no." For the last two, we answered "maybe." This really does make a difference. You should convince yourself that our second example program

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

returns the result

- 2 if eager evalutation is used,
- 2 under the $\langle \textbf{no}, \textbf{no}, \textbf{yes}, \textbf{yes} \rangle$ interpretation of the copy rule,

- divergence under the $\langle \mathbf{no},\ \mathbf{no},\ \mathbf{no},\ \mathbf{no} \rangle$ interpretation of the copy rule.

Even worse, using the $\langle \mathbf{no},\ \mathbf{no},\ \mathbf{no},\ \mathbf{no} \rangle$ copy rule, our first example program

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

returns

- 1 if we substitute for the outer definition first, and
- 2 if we substitute for the inner definition first.

So the result is not even well-defined!

The conclusion to draw from all this: if we want to say something rigorous relating our semantic rules and the copy rule, we will need a precise formal definition of the copy rule. We'll get there eventually, but the definition will ultimately be motivated by some observations on the behavior of lazy evaluation rules; so we present some lazy rules first.

6 Lazy Rules – First Attempt

The intuition behind "lazy evaluation" is to avoid evaluating an expression abstraction until it becomes absolutely necessary to do so. To achieve this, instead of an environment of numeric values we use the definition (FV-lazy) from above:

$$\mathbf{Fval} \ = \ \mathbf{Exp} \tag{FV-lazy}$$

That is, we keep unevaluated expressions (elements of **Exp**) in the environment.

The semantic rules for abstractions now become

Let blocks

$$\frac{\langle e_1, \phi[e_0/f], \sigma \rangle \rightarrow \langle n, \sigma' \rangle}{\langle \textbf{let } f \sim e_0 \textbf{ in } e_1, \phi, \sigma \rangle \ \rightarrow \ \langle n, \sigma' \rangle} \tag{LX1.let.l}$$

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

Invocation

$$\frac{\langle \phi(f), \phi, \sigma \rangle \;\rightarrow\; \langle n, \sigma' \rangle}{\langle f(), \phi, \sigma \rangle \;\rightarrow\; \langle n, \sigma' \rangle} \qquad \text{(LX1.call.l)}$$

To evaluate a function invocation, look up the function name in the environment to get an expression, then evaluate that expression using the current environment and store.

This definition, like the eager one, has some intuitive appeal. Moreover, it avoids the first two problem areas we discussed in connection with eager evaluation and the copy rule. Specifically, we gave an example that violated the copy rule due to effects on the store, and another example that violated the copy rule because of a nonterminating loop. You should verify that neither of these examples causes a problem when the lazy rules are used. Using the lazy rules, these example programs behave identically before and after expansion by the copy rule.

As before, the interesting behaviour happens when an expression being abstracted may itself contain references to (its own or other) abstraction names.

Recall the first example we used to show that eager evaluation does not obey the copy rule:

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

Recall this program returns 1 using eager evaluation rules, but returns 2 when transformed by the copy rule. This program also returns 2 using the lazy evaluation rules. The derivation is a simple chain of rule instances with one hypothesis each:

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

by rule (LX1.let.l) with hypothesis

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

by rule (LX1.let.l) with hypothesis

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

by rule (LX1.let.l) with hypothesis

$$\langle g(), \phi_0[1/f][f()/g][2/f], \sigma \rangle \;\rightarrow\; \langle 2, \sigma \rangle$$

by rule (LX1.call.l) with hypothesis

$$\langle f(), \phi_0[1/f][f()/g][2/f], \sigma\rangle \quad \rightarrow \quad \langle 2, \sigma\rangle$$

by rule (LX1.call.l) with hypothesis

$$\langle 2, \phi_0[1/f][f()/g][2/f], \sigma\rangle \quad \rightarrow \quad \langle 2, \sigma\rangle$$

which is irreducible.

A slightly more involved example (presented without derivation):

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

Using lazy evaluation rules this evaluation returns 3. The first evaluation of $g()$ returns 1, computed using the outer definition $f \sim 1$; the second evaluation of $g()$ returns 2, computed using the inner definition $f \sim 1$. The same result could be produced using the $\langle \textbf{no}, \textbf{no}, \textbf{no}, \textbf{no}\rangle$ interpretation of the copy rule. In fact, once we have suitably formalized the definition of the copy rule, it will be possible to show (by a nontrivial induction on derivations) that our lazy evaluation rules *always* obey the $\langle \textbf{no}, \textbf{no}, \textbf{no}, \textbf{no}\rangle$ copy rule.

One can see from the above examples (or by directly examining proof rule LX1.call.l) that a function invocation is evaluated using the environment in effect at the point of invocation, *not* the environment in effect at the point of definition. Thus, two different invocations of the same function may be evaluated using completely different environments, thus may have completely different meanings. Arguably this is not very desirable behavior, but it is simple to implement (inefficiently, at least), and it has been used in real systems (LISP, APL).

The property that the dynamic execution path leading to a function invocation determines the environment used to evaluate it has inspired the term "dynamic scope" to describe this name lookup strategy. Contrast this with "static scope" (or often "lexical scope"), in which the evaluation environment is determined by the definition of the function.

We are about to develop a lazy semantics for lexical scope. At this point we can no longer defer giving precise definitions for substitution and the copy rule, since we're going to need to change them in rather subtle ways.

But here is one final example to illustrate the badness of dynamic scope. We return to the previous example,

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

Consider the innermost **let** block subexpression,

let $f \sim 2$ **in** $g()$

Suppose we rename f to h in this subexpression, obtaining

let $h \sim 2$ **in** $g()$

Is the new subexpression equivalent to the old? Certainly not. In the context of the entire example, we will eventually substitute the definition of g – that is, $f()$ – in place of the invocation $g()$. The two subexpressions behave quite differently. In the first subexpression, the substituted invocation of f is captured by the immediately enclosing definition $f \sim 2$. In the second subexpression, the substituted invocation of f is not affected by the enclosing definition $h \sim 2$, and is resolved instead to the outermost definition $f \sim 1$.

Would we *want* the two subexpressions to be equivalent? Probably yes. We would prefer that an apparently arbitrary choice for the name of a local function definition should not affect the external meaning of the program, by affecting its behavior with respect to capture.

For now we could define this problem away using static semantics: we could require that all names used in a program be distinct. Clearly this approach would not scale to a real language ...