An Adequate Operational Semantics of Sharing in Lazy Evaluation *

S. Purushothaman and Jill Seaman
Dept of Computer Science
The Pennsylvania State University
University Park, PA 16802
e-mail: purush@cs.psu.edu and jseaman@cs.psu.edu

Abstract

We present LAZY-PCF+SHAR, an extension of **PCF**, that deals with lazy evaluation and explicit substitutions to model the sharing engendered by the lazy evaluation strategy. We present a natural operational semantics for LAZY-PCF+SHAR and show that it is equivalent to the standard fixed-point semantics. Sharing is modeled by explicit substitutions, which require a great deal of careful attention in the proof.

1 Introduction

In this paper we develop an operational semantics for an extension of PCF, called LAZY-PCF+SHAR, that provides a formalism for dealing with the sharing involved in lazy evaluation. The language is different from PCF in that explicit substitutions are made part of the language. The central part of the paper is a soundness theorem and an adequacy theorem which show that the operational semantics developed is equivalent to standard fixed-point semantics, as found in the literature.

The work presented here is aimed at providing the basis (and the tools necessary) for developing analyses of sharing in lazy functional languages. Compile-time analysis of sharing is fundamental to a number of other compile-time analyses such as garbage collection and order of evaluation (and its use in parallelization) [3,6,4]. Hence there is a need for a study of sharing.

In [4] we showed how a *compositional* compile-time analysis for evaluation order and aggregate update of a first-order lazy language can be developed from an operational semantics. The technique is to define predicates based on the operational semantics and to extend these predicates to the denotational semantics based on the equivalence of the

*Supported in part by NSF grants CDA-89-14587 and CCR-90-04121

two semantics. Then compositional analysis can be done using abstract interpretation with the denotational semantics. The advantage was that we were able to avoid development of complicated continuation based semantics and the development of abstract interpretation to deal with such continuations. Of course, the entire development hinged on the equivalence theorem between the operational and the fixed-point semantics. The work reported here was started as the basis for extending the ideas of [4] to higher-order lazy evaluation. The equivalence shown here should also be of independent interest in the study of lazy evaluation and in the study of establishing equivalence between semantic definitions.

The characteristics of lazy evaluation have typically been identified from the theoretical perspective as the use of weak-head normal forms in λ-calculus, and from the implementation perspective as an evaluation strategy that improves on the space-and-time requirements of call-by-name evaluation mechanism by explicit sharing of actual parameters in function calls.

Abramsky has formalized lazy evaluation in an untyped λ-calculus as the reduction of the outermost function to weak-head normal form before applying to an unevaluated argument [2]. Howard and Mitchell have similarly considered a lazy version of PCF with algebraic datatypes in [7]. The languages (and their semantics) considered in both of these papers do not deal explicitly with the sharing engendered in lazy evaluation.

Recently, there has been lot of research activity on explicit substitutions [1,5,8]. In the first two papers the reduction system $\lambda\sigma$ is considered, while in [8] a calculus, weak-$\lambda\sigma$, with weak-head normal forms has been considered. Our work is more closely related to the work of [8] in that we consider weak-head normal forms as the normal form for λ-abstractions and use explicit substitutions. But each of these papers studies the reduction system with emphasis on optimality of reduction strategies, while we fix the reduction strategy by the operational semantics and are more interested in the relation between the operational (i.e., the reduction system) and the fixed-point approach to semantics. As compared to all of these papers our semantics is defined over a typed language.

In the next section we introduce the language LAZY-PCF+SHAR which includes explicit substitutions as components of expressions. In Section 3 we present the soundness theorem and in Section 4 we prove the adequacy theorem. Both of these theorems, especially the adequacy theorem, are complicated, compared to Plotkin's original proof of equivalence between operational and fixed-point semantics of PCF [10]. The main difficulty in extending Plotkin's proof are due to (a) the presence of explicit substitutions, and (b) the assumption of a fixed evaluation strategy, i.e., lazy evaluation strategy. For example, the soundness theorem, whose proof is very simple in Plotkin's paper, depends on properties of a relationship between the environments that arise in the operational semantics and those of the fixed-point semantics. The adequacy proof, though similar to Plotkin's proof, is complicated because of the presence of the explicit substitutions, as opposed to syntactic substitutions. The proof sketches for all the important theorems have been presented. A more thorough presentation of the proofs is contained in [11].

Types:

t = nat | bool | $t_1 \rightarrow t_2$

Expressions:

e = 0 | *true* | *false* |
$\lambda x{:}t.e$ | $e_1\ e_2$
| *if*(e_1, e_2, e_3) | x |
succ(e) | *pred*(e) |
iszero(e) | $\mu x{:}t.e$ |
$\langle e, [x{:}t \mapsto e_1] \rangle$

$FV(x) = \{x\}$
$FV(0) = FV(true) = FV(false) = \{\ \}$
$FV(succ(e)) = FV(pred(e)) = FV(e)$
$FV(iszero(e)) = FV(e)$
$FV(\lambda x{:}t.e) = FV(\mu x{:}t.e) = FV(e) - \{x\}$
$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$
$FV(\langle e, [x{:}t \mapsto e_1] \rangle) = FV((\lambda x{:}t.e)\, e_1)$
$FV(if(e_1, e_2, e_3)) = FV(e_1) \cup FV(e_2) \cup FV(e_3)$

Figure 1: Syntax of LAZY-PCF+SHAR

2 The Language and its Semantics

The language that we use, LAZY-PCF+SHAR, is a lazy version of **PCF** extended to include explicit substitutions in order to capture the sharing involved in parameter passing. The syntax, along with the rules for free variables, is as given in Figure 1. As usual, an expression e such that $FV(e) = \emptyset$ is called closed. The only syntactic difference between PCF and our language is the explicit use of the substitution $[x : t \mapsto e_1]$ in the closure $\langle e, [x{:}t \mapsto e_1] \rangle$. In such a closure, the substitution is a binding for the free variable x in e. The purpose of these closures is to model function application according to the rules of lazy evaluation, which require that the argument is not evaluated until needed, and then evaluated only once (it is not reevaluated if it is needed more than once). The substitution in a closure provides storage for the initially unevaluated argument which can be updated if and when it is evaluated. This explains why $FV(\langle e, [x{:}t \mapsto e_1] \rangle)$ is defined in terms of application. Closures could also be used to model the *let* expression, as in $\langle e, [x{:}t \mapsto e_1] \rangle$ for "let $x{:}t = e_1$ in e".

Every valid expression has a unique type. The type t of an expression e is constructed with respect to a type environment, which maps the free variables of e to types. Type environments are denoted as a list $H = [x_1 : t_1, \ldots, x_n : t_n]$, where the variables x_i are unique. As is customary we will use the notation $H[s/x]$ to denote a perturbed environment which respects H on all variables other than x, and binds x to type s. The type judgement rules are provided in Figure 2. An expression e has type t in type environment H if $H \vdash e : t$ can be justified by an inference built up from the type rules. We refer to the types **nat** and **bool** as *basic* types.

2.1 The Operational Semantics

The main task of the operational semantics is to model function application according to the lazy evaluation rules. As described above, closures provide the means to model the storing of the argument, but the semantics will be responsible for modeling the evaluation and updating of the arguments appropriately. Furthermore, since a given expression

$$\text{C0:}\quad \vdash 0 : \mathbf{nat} \qquad \text{CS:}\quad \frac{H \vdash e : \mathbf{nat}}{H \vdash succ(e) : \mathbf{nat}}$$

$$\text{CT:}\quad \vdash true : \mathbf{bool} \qquad \text{CP:}\quad \frac{H \vdash e : \mathbf{nat}}{H \vdash pred(e) : \mathbf{nat}}$$

$$\text{CF:}\quad \vdash false : \mathbf{bool} \qquad \text{CZ:}\quad \frac{H \vdash e : \mathbf{nat}}{H \vdash iszero(e) : \mathbf{bool}}$$

$$\text{Var:}\quad H[t/x] \vdash x : t$$

$$\text{Abs:}\quad \frac{H[s/x] \vdash e : t}{H \vdash \lambda x{:}s.e : s \to t} \qquad \text{App:}\quad \frac{H \vdash e : s \to t \quad H \vdash e_1 : s}{H \vdash e\, e_1 : t}$$

$$\text{Cond:}\quad \frac{H \vdash e_1 : \mathbf{bool} \quad H \vdash e_2 : t \quad H \vdash e_3 : t}{H \vdash if(e_1, e_2, e_3) : t}$$

$$\text{Rec:}\quad \frac{H[t/x] \vdash e : t}{H \vdash \mu x{:}t.e : t} \qquad \text{CL:}\quad \frac{H \vdash e_1 : s \quad H[s/x] \vdash e : t}{H \vdash \langle e, [x{:}s \mapsto e_1] \rangle : t}$$

Figure 2: Type Rules

may have several levels or nestings of closures, binding each of the free variables of that expression, the operational semantics must be able to maintain several bindings at once. In order to evaluate an expression, the semantics collects these bindings into a list, called the *environment* of the expression. An environment will be listed as follows:

$$[x_1{:}t_1 \mapsto e_1, \ldots, x_n{:}t_n \mapsto e_n]$$

and must have the special properties that each x_i is unique and for each expression e_i, $FV(e_i) \in \{x_{i+1}, \ldots, x_n\}$. An expression is paired with an environment in a *configuration*, as in the following: $\ll e, [x_1{:}t_1 \mapsto e_1, \ldots, x_n{:}t_n \mapsto e_n] \gg$ with the property that $FV(e) \in \{x_1, \ldots, x_n\}$. Note that configurations are more general than closures, since their second element is a *list* of bindings.

The operational semantics defines a relation between configurations which models the evaluation of an expression within an environment. They are actually described as a *natural* semantics, which is a style of describing semantics where one expression (or configuration, in this case) "reduces" to another if an inference or proof tree can be demonstrated using the rules and axioms of the semantics to justify the reduction. In fact this is the only way one expression reduces to another—there is no notion of transitive closure or many-step reductions in natural semantics. Therefore, an expression reduces directly to its normal form, and the intermediate steps of the evaluation can be found in each level of the inference. Consequently, in a natural semantics a normal form is an expression that evaluates to itself. Since there are no many-step reductions, this does not cause the problem of infinite reductions as is found in the traditional rewriting systems.

The operational semantics for LAZY-PCF+SHAR are shown in Figure 3. In this description, note that $A_1 \cdot A_2$ is used to denote concatenation of environments, where

the binding $[x:t \mapsto e]$ of a closure is considered a one-element environment.

The first four rules, C0, CT, CF, and L show that 0, *true*, *false*, and expressions of the type $\lambda x:t.e$ are normal forms (they reduce to themselves in any environment). The first three are as expected, but λ-abstractions are also normal forms. This prevents the body of a function from being evaluated until it is applied, which is part of the evaluation strategy dictated by lazy evaluation. The next five rules simply carry out the evaluation of the primary functions *pred*, *succ*, and *iszero*.

The rules Var1 and Var2 manipulate the environment whenever the evaluation calls for a variable access. The rule Var1 does two things. First it evaluates the expression bound to the variable, and then it updates the environment to bind that variable to the new normal form. The rule Var2 is used when the variable being looked up in the environment is not the leftmost binding in the environment. It searches for the binding of that variable in the tail of the environment.

The rule Appl carries out application of a function to an argument by first evaluating the function, e_1, to a functional normal form, N. Then the function Ap, also defined in Figure 3, is used to create the appropriate closure of the body of N with the argument e_2. The new closure created by the Ap function is evaluated in the updated environment A' to find the normal form of the original application.

As an example of how the Ap function works, consider the evaluations of the expressions f_1 and f_2 defined as follows: $f \equiv \lambda x:s.\lambda y:t.p\ x\ y$, $f_1 \equiv (f\ 4)$ and $f_2 \equiv (f_1\ 5)$ in an environment P that contains a binding for the function p. From the Appl rule, we can infer that $\ll f_1, P \gg \rightarrow \ll \langle \lambda y:t.p\ n\ y, [n:s \mapsto 4] \rangle, P \gg$ (since f evaluates to itself in P by rule L and $Ap(f,4) \equiv \langle \lambda y:t.p\ n\ y, [n:s \mapsto 4] \rangle$, which also evaluates to itself in P by rules CL and L). In the evaluation of f_2 in P, the closure $\langle \lambda y:t.p\ n\ y, [n:s \mapsto 4] \rangle$ will be constructed and then applied to 5. Clearly, the operational semantics should be able to deal with the application of a closure to an argument, which the function Ap is defined to do. It recursively searches inside the closure for a λ-expression to discover the variable to which the argument is to be bound. Ap also renames the variable found in the body of the λ-expression and binds the argument to this new variable at the outermost level. The new variable is used in order to maintain the property that all environments will always contain unique variable names. This simulates the creation of a new location in storage. In this example $Ap(\langle \lambda y:t.p\ n\ y, [n:s \mapsto 4] \rangle, 5)$ yields $\langle \langle p\ n\ m, [n:s \mapsto 4] \rangle, [m:t \mapsto 5] \rangle$, which is then evaluated in P to get the final result.

IfTrue and IfFalse operate symmetrically. First the boolean expression e_1 is evaluated to *true* or *false*, and then either e_2 or e_3 is evaluated in the updated environment to find the appropriate result. The rule Rec evaluates the recursive operator μ by creating a closure with the body of the μ expression and a binding of the bound variable to the entire μ expression. Actually, a new variable is used in the binding (and appropriately substituted into the body) as described in the Appl rule. The binding of the body of the μ-expression with the μ-expression itself is in effect one unfolding of the recursive expression. Whenever the bound variable is encountered in the body, this unfolding will occur again.

$$\text{C0:}\quad \ll 0, A\gg \to \ll 0, A\gg \qquad \text{P0:}\quad \frac{\ll e, A\gg \to \ll 0, A'\gg}{\ll pred(e), A\gg \to \ll 0, A'\gg}$$

$$\text{CT:}\quad \ll true, A\gg \to \ll true, A\gg \qquad \text{P:}\quad \frac{\ll e, A\gg \to \ll succ(e_1), A'\gg}{\ll pred(e), A\gg \to \ll e_1, A'\gg}$$

$$\text{CF:}\quad \ll false, A\gg \to \ll false, A\gg \qquad \text{ZT:}\quad \frac{\ll e, A\gg \to \ll 0, A'\gg}{\ll iszero(e), A\gg \to \ll true, A'\gg}$$

$$\text{L:}\quad \ll \lambda x{:}t.e, A\gg \to \ll \lambda x{:}t.e, A\gg \qquad \text{ZF:}\quad \frac{\ll e, A\gg \to \ll succ(e_1), A'\gg}{\ll iszero(e), A\gg \to \ll false, A'\gg}$$

$$\text{S:}\quad \frac{\ll e, A\gg \to \ll e_1, A'\gg}{\ll succ(e), A\gg \to \ll succ(e_1), A'\gg}$$

$$\text{Var1:}\quad \frac{\ll e, A\gg \to \ll N, A'\gg}{\ll x, [x{:}t \mapsto e]\cdot A\gg \to \ll N, [x{:}t \mapsto N]\cdot A'\gg}$$

$$\text{Var2:}\quad \frac{\ll y, A\gg \to \ll N, A'\gg}{\ll y, [x{:}t \mapsto e]\cdot A\gg \to \ll N, [x{:}t \mapsto e]\cdot A'\gg}\ y \not\equiv x$$

$$\text{Appl:}\quad \frac{\ll e_1, A\gg \to \ll N, A'\gg \quad \ll Ap(N, e_2), A'\gg \to \ll N', A''\gg}{\ll e_1\ e_2, A\gg \to \ll N', A''\gg}$$

$$\text{IfTrue:}\quad \frac{\ll e_1, A\gg \to \ll true, A'\gg \quad \ll e_2, A'\gg \to \ll N, A''\gg}{\ll if(e_1, e_2, e_3), A\gg \to \ll N, A''\gg}$$

$$\text{IfFalse:}\quad \frac{\ll e_1, A\gg \to \ll false, A'\gg \quad \ll e_3, A'\gg \to \ll N, A''\gg}{\ll if(e_1, e_2, e_3), A\gg \to \ll N, A''\gg}$$

$$\text{Rec:}\quad \frac{\ll \langle e[nx/x], [nx{:}s \mapsto \mu x{:}t.e]\rangle, A\gg \to \ll N, A'\gg}{\ll \mu x{:}t.e, A\gg \to \ll N, A'\gg}$$

$$\text{CL:}\quad \frac{\ll e, [x{:}t \mapsto e_1]\cdot B\gg \to \ll N, [x{:}t \mapsto e_1']\cdot B'\gg,\ N \text{ is NOT basic}}{\ll \langle e, [x{:}t \mapsto e_1]\rangle, B\gg \to \ll \langle N, [x{:}t \mapsto e_1']\rangle, B'\gg}$$

$$\text{CL}'\text{:}\quad \frac{\ll e, [x{:}t \mapsto e_1]\cdot B\gg \to \ll c, [x{:}t \mapsto e_1']\cdot B'\gg,\ c \text{ is basic type}}{\ll \langle e, [x{:}t \mapsto e_1]\rangle, B\gg \to \ll c, B'\gg}$$

$$\begin{array}{lcl} Ap(\lambda x{:}t.e_0, e) & = & \langle e_0[nx/x], [nx{:}t \mapsto e]\rangle \\ Ap(\langle N, [x{:}t \mapsto e_1]\rangle, e) & = & \langle \langle K, [x{:}t \mapsto e_1]\rangle, [n{:}s \mapsto e]\rangle \\ & & \text{where } Ap(N, e) = \langle K, [n{:}s \mapsto e]\rangle \end{array}$$

Note: nx denotes a new variable.

Figure 3: The Operational Semantics of LAZY-PCF+SHAR

The rules CL and CL′ evaluate closures. They do so by evaluating the expression inside the closure in an environment formed by concatenating its original closure binding with the enclosing environment. Their difference is in whether or not the resulting normal form is a closure or not. If the expression evaluates to a normal form of basic type (**nat** or **bool**) then CL′ is used and the binding is no longer included in the normal form. In this way, every constant of basic type has only one normal form so that a constant contained in a closure with a binding is not a normal form. On the other hand, if the resulting normal form is not basic type, then it is a function type and we must include the bindings in the normal form because the function body may contain a free occurrence of that variable.

There are some special properties of the operational semantics that should be noted. Recall that all of the variables bound by an environment must be unique. A special property of the semantic rules is that if all of the variables in an initial configuration that are bound either in a closure or in the environment are distinct from each other (we call such a configuration *strict*), then every environment in the inference for the reduction of that configuration will also define unique variables. This property, is captured in the following lemma:

Lemma 2.1 *Let* $\ll e, A \gg \rightarrow \ll N, A' \gg$ *be a reduction. Then if* $\ll e, A \gg$ *is strict then all environments that arise in the proof tree of* $\ll e, A \gg \rightarrow \ll N, A' \gg$ *will not contain more than one binding of a given variable.*

In working with the operational semantics, it will be beneficial to know what the structure of the normal forms is. We will consider an expression e to be a normal form if it reduces to itself in any environment ($\ll e, A \gg \rightarrow \ll e, A \gg$). The normal forms are the set NF, described as follows:

$$\begin{aligned} NF &= 0 \mid \mathit{true} \mid \mathit{false} \mid \mathit{succ}^n(0) \mid F \\ F &= \lambda x{:}t.e \mid \langle F, [x{:}t \mapsto e_1] \rangle \end{aligned}$$

It is easy to see that 0, *true*, *false*, and $\lambda x:t.e$ are normal forms. $\mathit{succ}^n(0)$ denotes n applications of *succ* to 0, and represents the natural number n. It is easy to show that $\mathit{succ}^n(0)$ is a normal form by induction on n using the rule for *succ*. The subset F of NF describes lambda expressions nested in zero or more closures. Those nested in closures are lambda expressions having free variables which are bound in the closures. These expressions can be shown to be normal forms by structural induction on the set F using the rules L and CL. All other well-typed expressions that reduce to some expression according to the operational semantic rules will reduce to an expression in NF. This can be shown by induction on the height of the inference tree justifying the reduction, and can be seen easily by examining the operational semantics rules.

2.2 The Fixed-Point Semantics

In this section we provide the standard fixed-point semantics for LAZY-PCF+SHAR. The biggest difference between the operational semantics and the fixed-point semantics is the

$$E[\![0]\!]\sigma = 0 \qquad E[\![pred(e)]\!]\sigma = \begin{cases} 0 & \text{if } E[\![e]\!]\sigma = 0 \\ \bot & \text{if } E[\![e]\!]\sigma = \bot \\ E[\![e]\!]\sigma - 1 & \text{otherwise} \end{cases}$$

$$E[\![true]\!]\sigma = true \qquad E[\![succ(e)]\!]\sigma = \begin{cases} E[\![e]\!]\sigma + 1 & \text{if } E[\![e]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$E[\![false]\!]\sigma = false$$

$$E[\![x]\!]\sigma = \sigma(x) \qquad E[\![iszero(e)]\!]\sigma = \begin{cases} true & \text{if } E[\![e]\!]\sigma = 0 \\ \bot & \text{if } E[\![e]\!]\sigma = \bot \\ false & \text{otherwise} \end{cases}$$

$$E[\![\lambda x\!:\!t.e]\!]\sigma = \lambda v.E[\![e]\!]\sigma[v/x] \qquad E[\![e_1\, e_2]\!]\sigma = (E[\![e_1]\!]\sigma)\,(E[\![e_2]\!]\sigma)$$

$$E[\![\mu x\!:\!t.e]\!]\sigma = fix(\lambda d.E[\![e]\!]\sigma[d/x]) \qquad E[\![\langle e, [x\!:\!t \mapsto e_1]\rangle]\!]\sigma = E[\![(\lambda x\!:\!t.e)\, e_1]\!]\sigma$$

$$E[\![if(e_1, e_2, e_3)]\!]\sigma = \begin{cases} E[\![e_2]\!]\sigma & \text{if } E[\![e_1]\!]\sigma = true \\ E[\![e_3]\!]\sigma & \text{if } E[\![e_1]\!]\sigma = false \\ \bot & \text{if } E[\![e_1]\!]\sigma = \bot \end{cases}$$

Figure 4: Fixed-point semantics of LAZY-PCF+SHAR

type of environments used. While the environment A used in a configuration $\ll e, A \gg$ binds arbitrary expressions to variables, the environments used here bind variables to denotable values of natural numbers, booleans, and functions over them. More concretely, we have:

$$\begin{aligned} D &= N_\bot + B_\bot + (D \to D) \\ FEnv &= Var \to D \end{aligned}$$

where N denotes the set of natural numbers and B the set of Boolean values.

The fixed-point semantics provided by the function $E : exp \to FEnv \to D$ is presented in Figure 4. What is notable in this semantics is the absolute lack of details regarding sharing and memory management. Note how the semantics of closures is defined in terms of λ-expressions and applications. This corresponds to the observation that the difference between call-by-name and lazy evaluation is sharing of arguments.

2.3 Properties of *Ap*

In the fixed-point semantics, application is carried out by evaluating the function body in an environment that contains the fixed-point value of the argument. In the operational semantics, the function *Ap* creates a closure containing the argument so that it is not evaluated until it is needed. It is essential that this function be consistent with the fixed-point semantics, as the following lemma states.

Lemma 2.2 *Let M be an expression of non-basic type (i.e., function type) in normal form.*

$$\forall\sigma\forall a E[\![Ap(M,a)]\!]\sigma = E[\![Ma]\!]\sigma$$

Proof By induction on the structure of M. ■

The next lemma characterizes the normal forms of a string of applications as being either a normal form of basic type or a nesting of closures containing a binding corresponding to each of the arguments. This characterization will be useful in later proofs.

Lemma 2.3 *$\ll e\, e_1 \ldots e_n, A\gg \rightarrow \ll M, A'\gg$ where either M is of basic type or $M \equiv \langle\langle N, [x_1 \mapsto e_1']\rangle, \ldots [x_n \mapsto e_n']\rangle$ for some normal form N.*

Proof The proof is by induction on n and uses the Appl, CL, and CL′ rules. ■

3 The Soundness Theorem

The two parts of the equivalence theorem are the soundness and adequacy theorems. In this section we will deal with the soundness theorem which shows that the fixed-point semantics respects the operational semantics. More formally,

Theorem 3.1 (Soundness) *If e is a closed expression such that $\ll e, []\gg \rightarrow \ll N, []\gg$ then $E[\![e]\!]\perp = E[\![N]\!]\perp$.*

Here and throughout the rest of the paper we will use $\perp$ to signify the fixed-point environment that maps all variables to $\perp$. We will also use the term *inference induction* to denote induction on the height of the proof tree engendered by the operational semantics rules.

In showing that expressions yield equal values we will want to use inference induction on the reduction $\ll e, []\gg \rightarrow \ll N, []\gg$, but in doing so, environments will inevitably arise in the proof trees. Thus we will need a more general statement of the theorem that includes environments, and we will need a way of relating operational environments to fixed-point environments. We will do the latter through the definition of the function ρ. This function maps an operational environment A to a corresponding fixed-point environment $\rho(A)$. More specifically, $\rho(A)$ is a fixed-point environment which binds each variable x of A to the fixed-point value of the expression e, where x is bound to e in A. Formally, we have the following definition:

Definition 3.1 $\rho(A)$:

$$\begin{array}{lcl} \rho([\,]) & = & \perp \\ \rho([x\!:\!s \mapsto e_1]\cdot A_1) & = & \rho(A_1)[E[\![e_1]\!]\rho(A_1)/x] \end{array}$$

■

A consequence of such a definition is the following lemma. It shows that the fixed-point semantics can simulate the closure rules of the operational semantics and is necessary in the closure case of the inference induction proof.

Lemma 3.1 $E[\![\langle e, [x : s \mapsto e_1]\rangle]\!]\rho(A) = E[\![e]\!]\rho([x : s \mapsto e_1]\cdot A)$.

Proof The proof simply uses the rules of the fixed-point semantics and the definition of $\rho([x : s \mapsto e_1]\cdot A)$. ∎

With this definition and lemma, we are now ready to state and prove the more general statement of the Soundness Theorem (Theorem 3.1). The proof depends upon the fact that reduction preserves the fixed-point interpretation of environments (part (a)). The Soundness Theorem is a direct corollary of this theorem (the case where $A = [\,]$).

Theorem 3.2 (Generalized Soundness Theorem)

$$\ll e, A \gg \rightarrow \ll N, A' \gg \Rightarrow \begin{array}{l} (a)\rho(A) = \rho(A') \text{ and} \\ (b)E[\![e]\!]\rho(A) = E[\![N]\!]\rho(A') \end{array}$$

Proof *Sketch:* The proof is by inference induction. The harder cases are Var1, Var2, Appl, and CL, of which three are shown below (Var2 is similar to Var1). The type annotations are left out for readability. Let $A = [x \mapsto e']\cdot A_1$ for Var1 and Var2.

1. Var1: $\ll x, [x \mapsto e']\cdot A_1 \gg \rightarrow \ll N, [x \mapsto N]\cdot A_1' \gg$ because $\ll e', A_1 \gg \rightarrow \ll N, A_1' \gg$.
   $\rho(A) = \rho(A_1)[E[\![e']\!]\rho(A_1)/x]$ by definition. By induction, $\rho(A_1) = \rho(A_1')$ and $E[\![e']\!]\rho(A_1) = E[\![N]\!]\rho(A_1')$ so $\rho(A) = \rho(A_1')[E[\![N]\!]\rho(A_1')/x]$ which is $\rho(A')$ by definition.

   For part (b), $E[\![x]\!]\rho(A) = \rho(A)(x) = E[\![e']\!]\rho(A_1)$ by definition of $\rho(A)$. But this equals $E[\![N]\!]\rho(A_1')$ by induction. Now since $x \notin FV(N)$, this is equal to $E[\![N]\!]\rho(A')$.

2. Appl: $\ll e_1\ e_2, A \gg \rightarrow \ll N, A'' \gg$ because $(I) \ll e_1, A \gg \rightarrow \ll M, A' \gg$ and $(II) \ll Ap(M, e_2), A' \gg \rightarrow \ll N, A'' \gg$.

   For (a), $\rho(A) = \rho(A')$ and $\rho(A') = \rho(A'')$ by induction. Thus $\rho(A) = \rho(A'')$.

   For part(b), by induction $E[\![e_1]\!]\rho(A) = E[\![M]\!]\rho(A')$ and $E[\![Ap(M, e_2)]\!]\rho(A') = E[\![N]\!]\rho(A'')$. By lemma 2.2, this is equivalent to $E[\![M\ e_2]\!]\rho(A') = E[\![N]\!]\rho(A'')$. Using the fixed-point rules and induction on (I), this equals $E[\![e_1]\!]\rho(A)\,E[\![e_2]\!]\rho(A')$. Since $\rho(A) = \rho(A')$ (also by induction), the environments of both terms are the same, so they can be combined to get $E[\![e_1\ e_2]\!]\rho(A)$.

3. CL: $\ll \langle e, [x \mapsto e_a]\rangle, A \gg \rightarrow \ll \langle N, [x \mapsto e_a']\rangle, A' \gg$ because $\ll e, [x \mapsto e_a]\cdot A \gg \rightarrow \ll e, [x \mapsto e_a']\cdot A' \gg$.

   For part (a), $\rho([x \mapsto e_a]\cdot A) = \rho([x \mapsto e_a']\cdot A')$ by induction. It is easy to see that this implies that $\rho(A) = \rho(A')$.

   For part(b), $E[\![\langle e, [x \mapsto e_a]\rangle]\!]\rho(A) = E[\![e]\!]\rho([x \mapsto a]\cdot A)$ by lemma 3.1. This equals $E[\![N]\!]\rho([x \mapsto e_a']\cdot A')$, which, by lemma 3.1, equals $E[\![\langle N, [x \mapsto e_a']\rangle]\!]\rho(A')$.

∎

4 Adequacy Theorem

The adequacy theorem establishes that the operational semantics respects the fixed-point semantics. We will treat values of the base type (**bool** and **nat**) as the "observables" in the following.

Theorem 4.1 (Adequacy) *If e is a closed expression of basic type and c is an expression of basic type in normal form then* $E[\![e]\!]\perp = E[\![c]\!]\perp$ *implies* $\ll e, [\,]\gg \rightarrow \ll c, [\,]\gg$.

The proof of this theorem will be by structural induction on e, and thus will require induction on expressions that are not of basic type. Therefore we (again) must prove something stronger than Theorem 4.1 that is able to deal with functional types and non-closed expressions. Specifically, we will prove that all expressions are *computable*, a characterization that extends the notion of adequacy to treat higher types and environments. Our definition of computability is based on the definition used by Plotkin [10], but is revised to suit our semantics better. It differs from Plotkin's in that it does not treat closed expressions separately from those with free variables and it uses environments in the place of syntactic substitutions to close expressions.

Definition 4.1 Computability of Expressions

If $x_1 : s_1, \ldots, x_n : s_n \vdash e : s$, and $A = [x_1 : s_1 \mapsto e_1, \ldots, x_n : s_n \mapsto e_n]$ for any computable e_i satisfying $x_{i+1} : s_{i+1}, \ldots, x_n : s_n \vdash e_i : s_i$ then e is computable if one of the following is met:

1. s is a basic type and $E[\![e]\!]\rho(A) = E[\![c]\!]\rho(A) \Rightarrow \ll e, A\gg \rightarrow \ll c, A'\gg$,
2. $s \equiv t_1 \rightarrow t_2$ and $e\,e'$ is computable for all computable e' satisfying $x_1 : s_1, \ldots, x_n : s_n \vdash e' : t_1$

■

If closed terms of basic type are computable, then for any environment A that meets the criteria listed above we know $E[\![e]\!]\rho(A) = E[\![c]\!]\rho(A) \Rightarrow \ll e, A\gg \rightarrow \ll c, A'\gg$. Since the environment $[\,]$ meets the criteria when e is closed, the implication is also true when $A = [\,]$, which is precisely the adequacy theorem.

Plotkin's definition uses what we will refer to as syntactic substitutions to close expressions and then determine their computability. These syntactic substitutions are the common, syntactically defined substitutions mapping variables to expressions as are used in lambda calculus, usually denoted as $e[e_1/x]$ or $e[x := e_1]$. This definition works well in Plotkin's proof that all terms are computable because his operational (as well as fixed-point) semantics defines application in terms of syntactic substitutions. Our semantics, on the other hand, defines application in terms of closures, which become part of the environment in the process of evaluation. Thus we define computability using environments to close expressions.

Plotkin's proof that all terms are computable is by structural induction on expressions, and depends on inherent properties of syntactic substitutions, such as the fact that

they distribute over application ($(e_1\ e_2)[a/x] = e_1[a/x]\ e_2[a/x]$). Thus, a significant part of our proof amounts to showing that semantically environments have these same properties that syntactic substitutions have. We begin with several lemmas that establish these properties. The first states that the order of certain bindings may be changed under the right conditions. It imitates the following property of syntactic substitutions: $e_0[e/x][a/y] = e_0[a/y][e/x]$ if $y \notin FV(e)$.

In the statements and proofs of the following lemmas and theorems, the type annotations will be left out to make the expressions more readable.

Lemma 4.1 *If $y \notin FV(e)$ then*

$$\ll e_0, B\cdot[x\mapsto e]\cdot[y\mapsto a]\cdot A\gg \rightarrow \ll N, B'\cdot[x\mapsto e']\cdot[y\mapsto a']\cdot A'\gg$$
$$\Rightarrow \ll e_0, B\cdot[y\mapsto a]\cdot[x\mapsto e]\cdot A\gg \rightarrow \ll N, B'\cdot[y\mapsto a']\cdot[x\mapsto e']\cdot A'\gg$$

Proof The proof of this lemma is fairly simple by inference induction. The only interesting cases are Var1 and Var2 when $B = [\,]$. ■

The next lemma shows that a binding may be moved into an arbitrary level of nestings of closures. It is an extension of the previous lemma.

Lemma 4.2 *If $x \notin \cup_i FV(e_i)$ then*

$$\ll\langle\langle e, [x_1\mapsto e_1]\rangle, \ldots [x_n\mapsto e_n]\rangle, [x\mapsto a]\cdot A\gg \rightarrow \ll M, [x\mapsto a']\cdot A'\gg$$
$$\Rightarrow \ll\langle\langle\langle e, [x\mapsto a]\rangle, [x_1\mapsto e_1]\rangle, \ldots [x_n\mapsto e_n]\rangle, A\gg \rightarrow \ll M', A'\gg$$

where

1. $M \equiv M'$, *if M is of basic type OR*

2. $M \equiv \langle\langle N, [x_1\mapsto e'_1]\rangle, \ldots [x_n\mapsto e'_n]\rangle$ *and*
   $M' \equiv \langle\langle\langle N, [x\mapsto a']\rangle, [x_1\mapsto e'_1]\rangle, \ldots [x_n\mapsto e'_n]\rangle$
   if M is NOT of basic type for some normal form N.

Proof The proof is by induction on n, using CL, CL′, and lemma 4.1. ■

The next lemma shows that bindings in environments can be distributed to the subexpressions of an application. It is restricted to bindings of variables that are not free in arguments of the application, and thus the substitution is not applied to the arguments. This lemma imitates the syntactic substitution rule $(e\,e_1)[a/x] = e[a/x]\,e_1$ if $x \notin FV(e_1)$. An additional property characterized by this lemma is that the closure $\langle e, [x\mapsto a]\rangle$ can be replaced by the corresponding lambda expression and application $(\lambda x.e)\ a$ with the same results.

Lemma 4.3 *If* $x \notin \cup_i FV(e_i)$ *then*

$$\textit{I.} \quad \ll e\, e_1 \ldots e_n, [x \mapsto a] \cdot A \gg \rightarrow \ll M, [x \mapsto a'] \cdot A' \gg \Rightarrow \ll \langle e, [x \mapsto a] \rangle\, e_1 \ldots e_n, A \gg \rightarrow \ll M', A' \gg \textit{ and}$$

$$\textit{II.} \quad \ll e\, e_1 \ldots e_n, [x \mapsto a] \cdot A \gg \rightarrow \ll M, [x \mapsto a'] \cdot A' \gg \Rightarrow \ll (\lambda x.e)\, a\, e_1 \ldots e_n, A \gg \rightarrow \ll M', A' \gg.$$

where

1. $M \equiv M'$, *if* M *is of basic type OR*

2. $M \equiv \langle \langle N, [x_1 \mapsto e_1'] \rangle, \ldots [x_n \mapsto e_n'] \rangle$ *and* $M' \equiv \langle \langle \langle N, [x \mapsto a'] \rangle, [x_1 \mapsto e_1'] \rangle, \ldots [x_n \mapsto e_n'] \rangle$ *if* M *is NOT of basic type.*

Proof By induction on n, using CL, CL', Appl, and lemmas 2.3 and 4.2. ■

Theorem 4.2 *All expressions are computable.*

The proof is similar to Plotkin's original proof, and is carried out by structural induction on e. As in Plotkin's proof, the hardest case is $\mu x.e$, which relies on a notion of syntactic approximation and unwindings of the μ-expression. For this proof, the unwindings $\mu^n x.e$ are defined as follows:

$$\begin{aligned} \mu^0 x.e &\equiv \Omega_s \equiv \mu x.x \\ \mu^{n+1} x.e &\equiv \langle e[nx/x], [nx \mapsto \mu^n x.e] \rangle \end{aligned}$$

In order for the proof to work correctly, these syntactic approximations must satisfy the following lemma:

Lemma 4.4 $E[\![\mu x.e]\!]\perp = \sqcup_i E[\![\mu^i x.e]\!]\perp$

Proof From the fixed-point rules, $E[\![\mu x.e]\!]\perp = fix(\lambda d.E[\![e]\!]\perp[d/x])$. From fixed-point theory, we know that this is equal to $\sqcup_i d_i$ where $d_0 = \perp$ and $d_{n+1} = E[\![e]\!]\perp[d_n/x]$. To complete the proof we need only show that $d_n = E[\![\mu^n x.e]\!]\perp$ which is easy using induction on n and the fixed-point rules. ■

The syntactic approximation relation $\leq$, given in Figure 5, allows one to relate the syntactic notion of unwindings of μ expressions to the semantic use of Kleene sequences in providing the semantics of μ-expressions. The significant intermediate step in completing the adequacy proof for the case of μ is the following theorem which establishes that the operational semantics (i.e., $\rightarrow$) is monotonic with respect to the relation $\leq$. It will allow us to relate $\mu x.e$ to $\mu^k x.e$ in the operational semantics.

$$H \vdash 0 \leq 0 : \mathbf{nat}$$

$$H \vdash \mathit{true} \leq \mathit{true} : \mathbf{bool}$$

$$H \vdash \mathit{false} \leq \mathit{false} : \mathbf{bool}$$

$$H[s/x] \vdash x \leq x : s$$

$$\frac{H \vdash e \leq e' : \mathbf{nat}}{H \vdash \mathit{succ}(e) \leq \mathit{succ}(e') : \mathbf{nat}}$$

$$\frac{H \vdash e \leq e' : \mathbf{nat}}{H \vdash \mathit{pred}(e) \leq \mathit{pred}(e') : \mathbf{nat}}$$

$$\frac{H \vdash e \leq e' : \mathbf{nat}}{H \vdash \mathit{iszero}(e) \leq \mathit{iszero}(e') : \mathbf{bool}}$$

$$\frac{H, x:s \vdash e \leq e' : t}{H \vdash \lambda x:s.e \leq \lambda x:s.e' : s \to t} \qquad \frac{H \vdash e_1 \leq e_1' : s \to t \quad H \vdash e_2 \leq e_2' : s}{H \vdash e_1\, e_2 \leq e_1'\, e_2' : t}$$

$$\frac{H \vdash e_1 \leq e_1' : \mathbf{bool} \quad H \vdash e_2 \leq e_2' : t \quad H \vdash e_3 \leq e_3' : t}{H \vdash \mathit{if}(e_1, e_2, e_3) \leq \mathit{if}(e_1', e_2', e_3') : t}$$

$$\frac{H \vdash e_a \leq e_a' : s \quad H[s/x] \vdash e \leq e' : t}{H \vdash \langle e, [x:s \mapsto e_a] \rangle \leq \langle e', [x:s \mapsto e_a'] \rangle : t} \qquad \frac{H[t/x] \vdash e \leq e' : t}{H \vdash \mu x:t.e \leq \mu x:t.e' : t}$$

$$H \vdash \Omega_s \leq e : s \qquad \frac{H[s/x] \vdash e \leq e' : s}{H \vdash \mu^n x:s.e \leq \mu x:s.e'}$$

Figure 5: Ordering Rules

Theorem 4.3 *If* $H \vdash e \leq e' : s$ *then*

$$\ll e, A \gg \to \ll N, A' \gg \;\Rightarrow\; \ll e', A \gg \to \ll N', A'' \gg$$

where $H \vdash N \leq N' : s$

Though Plotkin's proof uses a similar theorem, our proof of this theorem differs significantly from Plotkin's proof as we again have to deal with explicit substitutions. In order to prove this inductively, a relation on environments must be established, and a more general statement of the theorem must be made, in a manner similar to the soundness theorem. The relation on environments is simply an extension of $\leq$ (defined on expressions) to environments.

Definition 4.2 $\leq$ extended to environments is as follows: $H \vdash A \leq B$ if either:

1. $A = B = [\,]$.
2. $A = [x:s \mapsto e_a] \cdot A_1$, $B = [x:s \mapsto e_b] \cdot B_1$ and
   (a) $H \vdash A_1 \leq B_1$, and
   (b) $H \vdash e_a \leq e_b : s$

■

Now we can state the generalized version of theorem 4.3 (theorem 4.3 is the case where $S = A$).

Theorem 4.4 *If* $H \vdash e \leq e' : s, H \vdash A \leq S$ *then*

$$\ll e, A \gg \rightarrow \ll N, A' \gg \Rightarrow \ll e', S \gg \rightarrow \ll N', S' \gg$$

where $H \vdash N \leq N' : s$ *and* $H \vdash A' \leq S'$

Proof The proof of this theorem is by inference induction on the operational semantic rules, but the proof proceeds on a case by case consideration for $H \vdash e \leq e' : s$. Most of the cases are simple induction, but the more interesting cases are $H \vdash e_1\ e_2 \leq e_1'\ e_2' : s$ and $H \vdash \mu^n x.e \leq \mu x.e' : s$. The application case is basic once it is shown that $H \vdash Ap(M, e_2) \leq Ap(M', e_2')$ where $\ll e_1, A \gg \rightarrow \ll M, A' \gg$. The recursive case is done by induction on n and depends on the rules CL, CL', and Rec. ■

Now we are ready to prove theorem 4.2, that all terms are computable.

Proof The proof is by structural induction on the term e. Some representative cases will be shown here.

$e \equiv x$ Then $H[s/x] \vdash x : s$ and $A = A_1 \cdot [x \mapsto e_x] \cdot A_2$. Only the case where s is basic is considered here. Then $E[\![x]\!]\rho(A) = \rho(A)(x) = E[\![e_x]\!]\rho(A_2)$. Since e_x is computable and of type s, by definition $E[\![e_x]\!]\rho(A_2) = E[\![c]\!]\rho(A_2) \Rightarrow \ll e_x, A_2 \gg \rightarrow \ll c, A_2' \gg$. By Var1, we get from this $\ll x, [x \mapsto e_x] \cdot A_2 \gg \rightarrow \ll c, [x \mapsto c] \cdot A_2' \gg$. Now we can get the final result by induction on the structure of A_1. If it is empty, we already have the result. If $A_1 = [y \mapsto e_y] \cdot A_1'$ then by induction on A_1' we get $\ll x, A_1' \cdot [x \mapsto e_x] \cdot A_2 \gg \rightarrow \ll c, A_1' \cdot [x \mapsto c] \cdot A_2' \gg$. Then from this by Var2 we get $\ll x, [y \mapsto e_y] \cdot A_1' \cdot [x \mapsto e_x] \cdot A_2 \gg \rightarrow \ll c, [y \mapsto e_y] \cdot A_1' \cdot [x \mapsto c] \cdot A_2' \gg$.

$e \equiv \lambda x.a$ Then $H \vdash \lambda x : s.a : s \rightarrow t$. For this expression, we need consider only the non basic case, so we start with $E[\![(\lambda x.a)\ e_1\ \ldots\ e_n]\!]\rho(A) = E[\![c]\!]\rho(A)$. Using the fixed point rules, we can show $E[\![(\lambda x.a)\ e_1]\!]\rho(A) = E[\![a]\!]\rho(A)[E[\![e_1]\!]\rho(A)/x]$. We can then rename x so that it does not interfere with any x occurring free in any other e_i, and then we have the following: $E[\![a[nx/x]e_2 \ldots e_n]\!]\rho(A)[E[\![e_1]\!]\rho(A)/nx] = E[\![c]\!]\rho(A)$. By induction, a is computable, so we can claim the following reduction: $\ll a[nx/x]\ e_2 \ldots e_n, [nx \mapsto e_1] \cdot A \gg \rightarrow \ll c, [nx \mapsto e_1'] \cdot A' \gg$. But according to lemma 4.3 this implies $\ll (\lambda x.a)\ e_1\ \ldots\ e_n, A \gg \rightarrow \ll c, A' \gg$.

$e \equiv \mu x : s.a$ For this case, Plotkin's proof is imitated.

First we show $\mu^k x : s.a$ is computable for all k. For $k = 0$, $E[\![\mu^0 x.a]\!]\rho(A) = \perp$ which implies $E[\![\mu^0 x.a]\!]\rho(A) \neq E[\![c]\!]\rho(A)$ for any basic c. By theorem 3.1, then, $\ll \mu^0 x.a, A \gg \not\rightarrow \ll c, A' \gg$, so the case is vacuous. For $k > 0$, $\mu^k x.a \equiv \langle a, [x \mapsto \mu^{k-1} x.a] \rangle$. By structural induction, a is computable, and by induction on k, $\mu^{k-1} x.a$ is computable, and since closures are computable when their subparts are (from the closure case), $\mu^k x.a$ is computable.

Now for $\mu x.a$: By lemma 4.4, $E[\![\mu x.a]\!]\rho(A) = E[\![\mu^k x.a]\!]\rho(A)$ for some k. Since $\mu^k x.a$ is computable, this implies $\ll \mu^k x.a\, e_1 \ldots e_n, A \gg \rightarrow \ll c, A' \gg$. Since $\mu^k x.a \leq \mu x.a$, by theorem 4.3 we have $\ll \mu x.a\, e_1 \ldots e_n, A \gg \rightarrow \ll c', A'' \gg$ where $c \leq c'$. Since c and c' are normal forms of basic type, it must be that $c \equiv c'$.

■

5 Conclusion

In this paper we have proposed a natural operational semantics for lazy evaluation which takes into account the sharing involved in evaluation of actual parameters. We have also shown it equivalent to the standard fixed-point semantics. The proof was fairly complicated due to the presence of semantically defined explicit substitutions. We can now hope to base our compile-time sharing analyses of higher-order lazy languages on operational semantics and show the semantic soundness of such schemes easily.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit Substituition. In *Proc of XVII ACM Symposium on Principles of Programming Languages*, ACM, Jan 1990.

[2] S. Abramsky. *The Lazy Lambda Calculus*, pages 65–116. Addison-Wesley, 1990.

[3] A. Bloss. *Path Analysis and the Optimization of Non-Strict Functional Languages.* PhD thesis, Yale University, 1989.

[4] M. Draghicescu and S. Purushothaman. Compositional Analysis of Evaluation Order and its Application. In *Proc of 1990 ACM Symposium on LISP and Functional Programming*, pages 242–250, 6 1990.

[5] J. Field. On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis. In *Proc of XVII ACM Symposium on Principles of Programming Languages*, ACM, Jan 1990.

[6] J. Guzman and P. Hudak. Single-Threaded Polymorphic Lambda Calculus. In *V Annual IEEE Symposium on Logic in Computer Science*, 1990.

[7] B. Howard and J. Mitchell. Operational and Axiomatic Semantics of PCF. In *Proc of 1990 ACM Symposium on LISP and Functional Programming*, pages 298–306, 6 1990.

[8] L. Maranget. Optimal Derivations in Weak Lambda-Calculi and in Orthogonal Term Rewriting Systems. In *Proc of XVIII ACM Symposium on Principles of Programming Languages*, pages 255–269, ACM, Jan 1991.

[9] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[10] G. Plotkin. LCF considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

[11] S. Purushothaman and Jill Seaman. *An Adequate Operational Semantics of Sharing in Lazy Evaluation.* Tech Report PSU-CS-91-18, Penn State Univ., July 1991.