An operational semantics of sharing in lazy evaluation

Jill Seaman a,*, S. Purushothaman Iyer b

a Center for Advanced Computer Studies, University of Southwestern Louisiana, PO Box 44330, Lafayette, LA 70504-4330, USA
b Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA

Received January 1992; revised March 1996
Communicated by G. Berry

Abstract

From a theoretical point of view, lazy evaluation corresponds to the call-by-name evaluation method, which substitutes arguments for parameters before evaluating them and never evaluates under a lambda. From an implementation perspective, lazy evaluation is often equated with the call-by-need method, which is similar to call-by-name except that arguments are shared. When an argument’s value is required, it is evaluated and its result is stored and used for any other reference to it. The theoretical version of lazy evaluation, or call-by-name, is easily formalized with the reduction rules of lambda calculus. However, it has proven rather difficult to formalize the rules of lazy evaluation with sharing, or call-by-need, in such a way that it both captures sharing and is useful for reasoning.

Many optimizations are based on analyses of program behavior which are dependent on whether arguments are implemented via sharing or not. Thus, it is important to have such a model of lazy evaluation in order to develop correct analyses.

In this paper, an operational semantics of PCF is presented which captures the sharing inherent in the call-by-need implementation of lazy evaluation in a form that is suitable for reasoning. The semantics uses explicit substitutions to implement sharing. The link between the theoretical and implementation versions of lazy evaluation is made by showing the correctness of the call-by-need semantics with respect to a standard call-by-name semantics for PCF.

1. Introduction

From a theoretical perspective, functional languages are nice because they are easy to reason about, especially within the framework of call-by-name or call by value evaluation. However, implementing a functional language strictly according to call-by-name causes problems of efficiency due to the fact that arguments that are referred to more than once are copied and possibly re-evaluated each time they are needed. However, due to the referential transparency in functional languages, this value will always be
the same. This unnecessary re-evaluation is usually avoided in practice by sharing the argument among each of its references so that there is only one copy of the argument at any one point in time. When the value of the argument is first needed, the argument is evaluated and the original copy of the argument is replaced by its value. This value is the one used for later references to the argument. So sharing can be characterized by a lack of duplication of the argument and by updating the original copy of the argument when it is evaluated. This method of evaluation, usually referred to as call-by-need, provides the same resulting values as call-by-name, but has different behavior due to the reduction of unnecessary re-evaluation.

Additional improvements in efficiency may be made in the usual way by analyzing the behavior of given programs and performing certain program transformations which improve the behavior of the program without affecting its results. Sharing behavior is fundamental to a number of compile-time analyses such as garbage collection, order of evaluation, and update-in-place [3, 5, 9]. By basing the analysis on an operational model of lazy evaluation which captures sharing, the analysis can be proven correct with respect to the model. A demonstration of correctness of analyses methods has been absent from much of the current literature. Many of these analyses are based on an instrumented denotational semantics. However, a denotational semantics is not suitable as a basis for a proof of correctness for an analysis because the sharing behavior is not present in the model. A more suitable basis could be an abstract machine which implements lazy evaluation, such as the G-machine [10], the Three Instruction Machine [6], and the Krivine Machine [4]. Although these machines do capture sharing, they generally are not appropriate as a basis for correctness proofs because they are not well-suited to reasoning.

The semantics presented in this paper have in fact been used as the basis of an analysis called reduction to variables [17] which indicates whether the result of an evaluation is referred to by some variable that could be used later in the program. An instrumented version of the semantics is used to develop the analysis and to demonstrate its correctness.

It should be noted that suitability for reasoning is one of the main goals for our model. The call-by-need evaluation behavior can be captured using graph reduction, which uses graphical representation of terms and reduction rules over these graphs. Our model has the advantage of being independent from such graph formalizations. It should also be noted that another goal is to model the call-by-need strategy corresponding to the sharing of arguments and a weak evaluation strategy (no evaluating under a lambda). Thus there is no sharing of lambda bodies and our model is not optimal in the sense of [7, 12, 14]. We seek to reflect current implementation rather than model optimal evaluation.

The remainder of the paper begins with an introduction of the syntax and semantics of the language. The syntax is the same as PCF, but the semantics implement call-by-need. In Section 3 we present some properties of the lazy evaluation semantics, such as subject reduction. In Section 4 we prove the computational correctness theorem which demonstrates the equivalence of the lazy evaluation semantics and
the call-by-name semantics. This is followed by a discussion of related work and a conclusion.

2. The syntax and semantics of LAZY-PCF+SHAR

In this section an operational semantics is developed for the toy functional language PCF [8, 15] which formalizes the evaluation of terms of the language according to the call-by-need evaluation order. Although the syntax is the same as PCF, the language along with its call-by-need semantics will be referred to as LAZY-PCF+SHAR. We choose PCF for several reasons. First, a call-by-name semantics and adequacy results for a denotational semantics are already given for PCF in [8]. Second, since PCF has primitives and constants our results are shown to hold for more than just the basic lambda calculus. Additionally, we can state certain theorems more simply by stating them in terms of expressions of ground type. Finally, the results that hold for call-by-name semantics of PCF can be directly applied to LAZY-PCF+SHAR once their equivalence is shown. Thus we include types in our syntax in order to maintain consistency with Gunter's PCF, but they do not contribute significantly to the main result, which is the equivalence of the call-by-name and call-by-need semantics of PCF.

2.1. The syntax of LAZY-PCF+SHAR

The syntax of LAZY-PCF+SHAR is presented in Fig. 1. Types consist of natural numbers, boolean values, and function types over these base types. Expressions consist of the constants 0, true, and false, the primitive operators succ, pred, iszero, and if applied to the correct number of arguments, variables, lambda abstractions (function

<table>
<thead>
<tr>
<th>Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expressions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Syntax of LAZY-PCF+SHAR.
Every valid expression has a unique type which is assigned according to the type judgment rules which are shown in Fig. 2. These rules are the typing rules for PCF given in [S]. The type of an expression is constructed with respect to a type context, which is a mapping of variables to types. As is customary the notation $H[s/x]$ denotes a perturbed mapping which respects $H$ on all variables other than $x$, and maps $x$ to type $s$. An expression $e$ has type $t$ in type environment $H$ if $H[t/x] \vdash e : t$ can be justified by an inference built up from the type rules. The types nat and bool are referred to as ground types.

2.2. The formalization of lazy evaluation with sharing

In order to formally describe lazy evaluation with sharing, more mechanism is required than the customary definition of substitution and rewrite rules which are usually used to define the semantics of lambda calculus. The definition of substitution will be replaced with explicit substitutions, and the rewrite rules will be replaced by natural semantics.

2.2.1. Substitution

Before considering why the usual definition of substitution is not enough to describe lazy evaluation, the definition itself will be given here along with some related definitions and conventions. Two of the most basic concepts relevant to the definition of semantics are the free and bound variables of a PCF term. These sets are defined in Figs. 3 and 4, respectively. A bound variable is one that occurs immediately after a $\lambda$ or a $\mu$ in an expression such as the $x$ in $\lambda x : t.e$. This occurrence is referred to as the binding occurrence. A variable $x$ is free in an expression if it has a non-binding occurrence in the expression which is not within the body $e$ of an abstraction $\lambda x : t.e$ or $\mu x : t.e$.
Let \( e[z/x] \) denote the term \( e \) with all of the free occurrences of \( x \) in expression \( e \) replaced by \( z \). Then the abstraction \( \lambda x : t.e \) can be transformed to the term \( \lambda z : t(e[z/x]) \) where \( z \) does not occur in \( e \). This renaming of bound variables transformation is referred to as **alpha conversion**. The same type of transformation can be done on mu abstractions as well. As in the previous literature, terms that are alpha-convertible (by applying alpha conversion to subterms) will be considered equivalent.

In order to avoid including special conditions regarding the names of bound variables as well as to be consistent with previous literature [2, 8], the standard convention concerning the names of bound variables will be assumed here. Specifically, it will be assumed in any discussion of PCF terms that the bound variable names will be distinct from the free variable names.
\[ c[e/x] = c \quad \text{for } c \in \{0, \text{true}, \text{false}\} \]
\[ p(e_1)[e/x] = p(e_1[e/x]) \quad \text{for } p \in \{\text{pred, succ, iszero}\} \]
\[ \text{if}(e_1, e_2, e_3)[e/x] = \text{if}(e_1[e/x], e_2[e/x], e_3[e/x]) \]
\[ x[e/x] = e \]
\[ y[e/x] = y \quad \text{if } y \neq x \]
\[ (\lambda x: t. e_1)[e/x] = \lambda x: t. e_1 \]
\[ (\lambda y: t. e_1)[e/x] = \lambda y: t. (e_1[e/x]) \quad \text{if } y \neq x \]
\[ (\mu x: t. e_1)[e/x] = \mu x: t. e_1 \]
\[ (\mu y: t. e_1)[e/x] = \mu y: t. (e_1[e/x]) \quad \text{if } y \neq x \]
\[ (e_1 e_2)[e/x] = (e_1[e/x] e_2[e/x]) \]

Fig. 5. Substitution.

At this point the definition of substitution can be given, which appears in Fig. 5. The notation \( e'[e/x] \) represents the term resulting from replacing the free occurrences of \( x \) in \( e' \) with \( e \). This definition is used to describe both call-by-value and call-by-name semantics in a very simple way. The previous convention is already assumed in this definition so that in \( \lambda \) and \( \mu \) abstractions, it can safely be assumed that the variable bound by the \( \lambda \) or \( \mu \) does not occur free in \( e \), the expression being substituted.

2.2.2. The need for explicit substitutions

In order to determine why a model of lazy evaluation has been difficult to define (without graph reduction), the models of call-by-value and call-by-name evaluation will be briefly considered. Since the only significant difference between the methods occurs when arguments are evaluated, it will suffice to consider how the evaluation of function application is described.

The call-by-name and call-by-value methods can be explained verbally in terms of substitution as follows. In order to evaluate the term \( ((\lambda x: t. e) e') \) by call-by-name order, we substitute the term \( e' \) for \( x \) in \( e \) and evaluate. In order to evaluate the term \( ((\lambda x: t. e) e') \) by call-by-value order, we first evaluate \( e' \) to \( v' \), then substitute \( v' \) for \( x \) in \( e \) and evaluate.

The formal semantics of application will be described as an inference rule. In order to conclude what is below the line, the premise above the line must be true. If evaluating a term \( e \) results in a term \( v \), this is denoted as \( e \Downarrow v \). Then the rules for the evaluation of application can be formally described as follows:

\[
\begin{align*}
\text{call-by-name:} & \quad \frac{e'[e/x] \Downarrow v}{((\lambda x: t. e') \Downarrow v} \\
\text{call-by-value:} & \quad \frac{e' \Downarrow v' \land e[v'/x] \Downarrow v}{((\lambda x: t. e') \Downarrow v}
\end{align*}
\]
It is quite clear that this definition of substitution simplifies the formalization of these evaluation orders. However, it is not at all clear how this definition of substitution could be used to describe call-by-need evaluation. The original argument, $e'$, should be substituted for the occurrence of $x$ which will be accessed first, and the result of evaluating $e'$ should be substituted for any remaining occurrences. Unfortunately, it is not known which occurrence of $x$ will be evaluated first until the program is run and the argument $e'$ should be evaluated only if it is needed. The problem with the substitution definition lies in the fact that it abstracts away the details of the actual process of substituting an expression for a variable in another expression. In order to implement lazy evaluation the semantics needs to be able to control the substitution process so that the substitution and evaluation of arguments can occur while the function body is being evaluated and not before.

The idea of incorporating rules into the semantics which directly carry out substitution, commonly referred to as explicit substitutions, has been previously explored [1, 7]. In these papers, explicit substitutions are used to define systems of rewrite rules for the lambda calculus. Though these rules incorporate explicit rules to carry out substitution they do not (inherently) capture sharing. In spite of this, it is the mechanism of explicit substitution which provides for a relatively simple formalization of lazy evaluation.

An example of a system using explicit substitutions is the $\lambda\sigma$-calculus with names, which appears in [1]. This calculus evaluates lambda terms which may include unevaluated substitutions. Thus the syntax includes terms and substitutions:

Terms $\ a ::= x \mid ab \mid \lambda x.a \mid a[s]$

Substitutions $s ::= \text{id} \mid (a/x) \cdot s \mid s \circ t$

where $\text{id}$ is the identity substitution, mapping $x$ to $x$ for all variables $x$. The $\cdot$ operator corresponds to the cons operator for lists, and the $\circ$ operator corresponds to the append function for lists. Some of the rewrite rules for the $\lambda\sigma$-calculus with names include the following:

Beta $(\lambda x.a)b \rightarrow a[(b/x) \cdot \text{id}]$

Var1 $x[(a/x) \cdot s] \rightarrow a$

Var2 $x[(a/y) \cdot s] \rightarrow x[s] \quad (x \neq y)$

Var3 $x[\text{id}] \rightarrow x$

Abs $(\lambda x.a)[s] \rightarrow \lambda y.(a[(y/x) \cdot s])$
(y occurs in neither $a$ nor $s$)

App $(ab)[s] \rightarrow (a[s])(b[s])$

The Beta rule is used to implement function application by creating the explicit substitution $[(b/x) \cdot \text{id}]$ for function body $a$. The purpose of the remaining rules is to carry
out the substitution process. Though these rules incorporate explicit substitution, a semantics of lazy evaluation with sharing cannot be formed by simply designating an evaluation strategy because these rules do not capture sharing. This is demonstrated by the fact that in the App rule, the substitution \( s \) is copied, which causes arguments to be duplicated and destroys sharing. Also, in the Var rules a substitution is not preserved and updated to store the new value of the argument, which is required for sharing. Thus these rules do incorporate the substitution process into the semantic rules, but they do not inherently capture sharing.

2.3. The operational semantics rules

Though the call-by-need operational semantics has some similarity to the \( \lambda \sigma \)-calculus, they differ in several ways in order to capture sharing. First, the syntax of the terms of LAZY-PCF+SHAR does not allow substitutions to occur within an expression. Instead, a term is evaluated with respect to a single substitution at the outermost level, called the operational semantics environment. This environment corresponds to an explicit substitution in that it is a list of variables bound to expressions. Another difference is the fact that in function application, environments are not duplicated and distributed to subexpressions as they are in rule App of the \( \lambda \sigma \)-calculus, which destroys sharing. A third difference is that the expression that a variable points to in the environment may be replaced by the value that it evaluates to. This allows the original copy of an argument to be replaced by its evaluated value. In order for this value to be used later, environments are not eliminated upon reaching a value as in the \( \lambda \sigma \)-calculus, but are maintained throughout the evaluation. Thus the result of the evaluation of an expression and its environment will be a value with a possibly updated environment.

An evaluation can be thought of as a relation between expression–environment pairs. This coupling of an expression with an environment is referred to as a configuration and is denoted as \( \langle e, A \rangle \) for an expression \( e \) and an environment \( A \). The structure of an environment is simply a list of bindings of typed variables to expressions, and can be formally described as follows:

\[
A ::= [ ] | [x : t \mapsto e]A
\]

Generally in an environment containing more than one binding, the bindings will be separated by commas instead of square brackets. The notation \( A[x : t \mapsto e]A' \) will be used to denote the environment which results from appending \( A \) to the environment \( [x : t \mapsto e]A' \), using the normal list definition for append. We will use \( \text{Dom}(A) \) to refer to the list of variables having bindings in \( A \).

The operational semantics of LAZY-PCF+SHAR is defined as a natural semantics, which defines the evaluation relation between a program and its final value in terms of inferences and axioms. There is no sense of a sequence of intermediate steps in the evaluation. Since an expression evaluates directly to its final value, this style of semantics is often referred to as “one step” or “big step” semantics. Proofs of theorems about the evaluation relation defined with these semantics can be carried out by in-
duction on the height of the proof justifying the evaluation relation. Natural semantics were explored by Kahn [11].

The operational semantics rules for LAZY-PCF+SHAR are shown in Fig. 6. The semantics is defined over configurations subscripted with a list of variables denoted using $Z$. This list serves as a supply of fresh names to be used in renaming bound variables during function application and recursion to guarantee that there is no name capture. In order for this to be accomplished, certain conditions are required of all the configurations in an evaluation. Specifically, the variable names bound in the environment, along with the variable names in the name supply list, must all be distinct from one another. This property is verified in the next section. Note that the colon ($:)$ is used as the cons operator in the name supply lists.

Fig. 6. Operational semantics of LAZY-PCF+SHAR.
The rules for application and recursion are defined in terms of a substitution \( e[z/x] \), and it may appear that the old definition of substitution has not in fact been eliminated. However, this is simply a renaming of a variable and is independent of the substitution and/or evaluation of the arguments.

The first four rules, \{0\}, \{T\}, \{F\}, and \{L\}, show that 0, true, false, and expressions of the form \( \lambda x : t.e \) evaluate to themselves in any environment. The first three are as expected, but lambda abstractions also evaluate directly to themselves. 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 result of evaluating a primitive function depends on the result of the evaluation of the argument of the primitive. This is contrary to the evaluation of function application where the argument is not evaluated until it is encountered during the evaluation of the function body. Thus the implementation of the application of primitive functions must be defined independently of the implementation of general function application. It is notable that in the premise of each of the rules, the environment \( A \) may change to \( A' \). This reflects the fact that the environment \( A \) may be updated in the evaluation of \( e \). A seeming anomaly is the \{P0\} rule, which states that if \( e \) evaluates to 0 then \( pred(e) \) evaluates to 0. This rule is included so that every term that has a type according to the type judgment rules will also have an evaluation. In general, applying pred to a term equivalent to 0 can be considered a programming error which could be prevented by the use of the conditional.

The \{Var\} rule handles the case when evaluation calls for a variable access. This rule makes the greatest contribution to the implementation of the call-by-need strategy. In order to determine the result of evaluating a variable in an environment containing a binding for that variable, the expression bound to that variable must be evaluated with respect to the remainder of the environment (to the right of the binding). In the process, this remainder of the environment \( (A) \) may be updated to a new environment \( (A') \). Then the result of the evaluation of the variable is the resulting value obtained in the previous evaluation \( (e) \), paired with the environment consisting of the prefix of the original environment \( (A_0) \), the original variable bound to the new value, followed by the updated environment \( (A') \). This process captures sharing by storing the evaluated value of the original expression in the environment. Arguments are stored in the environment until they are needed, at which point they are evaluated by the \{Var\} rule, and the environment is updated to contain the resulting value. The \{Var\} rule is also used for later evaluations of the variable, but in this case the premise to the rule would evaluate an already evaluated value, which evaluates to itself, avoiding the reevaluation of the original expression.

The rule \{Appl\} carries out application of a function to an argument by first evaluating the function, \( e_1 \), to a functional value, \( \lambda x : s.e \) with updated environment \( A' \). Then, the body of the function, with the parameter renamed to avoid variable name clashes, is evaluated in the environment created by adding the binding of the new variable to the argument \( e_2 \) to the updated environment \( A' \). In this way the \{Appl\} rule stores one
copy of the argument in the environment to be possibly accessed and updated later in
the evaluation of the function body.

{If True} and {If False} operate symmetrically. First the boolean expression \( e_1 \)
is evaluated to \texttt{true} or \texttt{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 \( \mu \) by evaluating the body of the mu
expression (with the parameter renamed) in an environment created by adding the new
variable bound to the the entire original mu expression to the original environment.
The binding of the body of the mu expression with the mu 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.

2.4. Valid configurations

As stated previously, configurations must have certain properties to insure proper
behavior during evaluation. Two properties are necessary for maintaining such behavior.
First, configurations should have no free variables. This is maintained by ensuring that
the free variables of each expression are bound in the environment to the right of that
expression. This must hold for the expressions within the environment as well as the
expression that is the first member of the configuration pair.

Secondly, name capture must be prevented during application and recursion. This can
be guaranteed if the variables bound in each environment are unique. This property
is maintained by the list of fresh variable names \( Z \). In order to maintain uniqueness
of variable names, it must be true of every configuration that the variables in the
environment and the name supply are distinct from each other. In other words,
\( \text{distinct}(\text{Dom}(A)@Z) \) where @ represents the append operation and distinct is true if
every element of the list is distinct.

These two properties will be captured in the definition of a \texttt{Valid} (\texttt{Z}-subscripted)
configuration. This definition depends on the definition of a valid environment, which
captures the first property (no free variables) for an environment\(^1\).

\textbf{Definition 2.1} (Valid environment),

\[
\text{Valid}(\emptyset) \\
\text{Valid}([x:t\rightarrow e]A) \quad \text{if } \text{FV}(e) \subseteq \text{Dom}(A) \\
\text{and } \text{Valid}(A)
\]

Now a valid configuration, which also incorporates the second property, distinctness,
can be defined as follows.

\(^1\)In the following definitions and theorems we use the set operations \( \in \) and \( \subseteq \) on lists. Technically, this is
an abuse of notation, and the lists in these operations should be interpreted as sets (by removing duplicates
and disregarding order).
Definition 2.2 (Valid configuration).

\[ \text{Valid}(\langle e, A \rangle_Z) \text{ if } \]
\[ \begin{align*}
\text{(a)} & \quad \text{Valid}(A) \\
\text{(b)} & \quad FV(e) \subseteq \text{Dom}(A) \\
\text{(c)} & \quad \text{distinct(Dom}(A)@Z) 
\end{align*} \]

Given this definition we now need to show that the operational semantics propagate this property (all configurations in an evaluation are valid if the initial one is).

Theorem 2.3. If \( \text{Valid}(\langle e, A \rangle_Z) \) and \( \langle e, A \rangle_Z \downarrow \langle v, A' \rangle_{Z'} \) then \( \text{Valid}(\langle v, A' \rangle_{Z'}) \).

The proof of this theorem is by induction on the height of the reduction \( \langle e, A \rangle_Z \downarrow \langle v, A' \rangle_{Z'} \), but it depends on two additional facts.

Lemma 2.4. If \( \langle e, A \rangle_Z \mid \langle v, A' \rangle_{Z'} \) then
\[ \begin{align*}
\text{(1)} & \quad \text{Dom}(A) \subseteq \text{Dom}(A') \\
\text{(2)} & \quad \forall x : (x \in \text{Dom}(A)@Z \iff x \in \text{Dom}(A')@Z') 
\end{align*} \]

The first part of this lemma states that during evaluation variable bindings are never dropped from an environment (though they may be added). The second part states that during evaluation the set of variables occurring as variables bound in the environment or in the fresh name supply stays constant. No new variables are introduced or dropped from this combined set of variables. These properties hold because bindings are only added to the environment and the variable name for the new binding is taken from the name supply. The lemma is easily verified by induction on the height of the evaluation.

Now Theorem 2.3 can be verified. The cases for Lambda abstractions and constants are trivial, and for the primitive operations are trivial applications of the inductive hypothesis. The case for the \{Var\} rule begins with an application of the inductive hypothesis giving \( \text{Valid}(\langle v, A' \rangle_{Z'}) \). From this we must verify that \( \text{Valid}(\langle v, A_0[x : t \mapsto e]A' \rangle_{Z'}) \), which we do by verifying the three parts of the definition: (a) \( \text{Valid}(A_0[x : t \mapsto v]A') \). This follows first from \( \text{Valid}(A') \) (from the inductive result); which leads to \( \text{Valid}([x : t \mapsto v]A') \) because \( FV(v) \subseteq \text{Dom}(A') \) (inductive result again). The validity of the entire environment then follows from the validity of the initial environment \( (\text{Valid}(A_0[x : t \mapsto e]A)) \) and the fact that \( \text{Dom}(A) \subseteq \text{Dom}(A') \) (by Lemma 2.4.1) and can be shown by induction on the length of \( A_0 \); (b) \( FV(v) \subseteq \text{Dom}(A_0[x : t \mapsto v]A') \) which follows from the inductive result \( FV(v) \subseteq \text{Dom}(A') \); and (c) \( \text{distinct(Dom}(A_0[x : t \mapsto v]A')@Z') \) which follows from \( \text{distinct(Dom}(A_0[x : t \mapsto e]A)@Z) \) and \( \forall x : (x \in \text{Dom}(A)@Z \iff x \in \text{Dom}(A')@Z') \) (Lemma 2.4.2). The case for \{Appl\} is a result of induction, but requires a verification that the inductive result \( \text{Valid}(\langle \lambda x : s \cdot e, A' \rangle_{Z,Z'}) \) implies \( \text{Valid}(\langle e[z/x], [z : s \mapsto e_2]A' \rangle_{Z,Z'}) \). Again we verify the three parts of the definition: (a) \( \text{Valid}([z : s \mapsto e_2]A') \) follows from \( \text{Valid}(A') \) (inductive result on first premise) and \( FV(e_2) \subseteq \text{Dom}(A') \) (given) and \( \text{Dom}(A) \subseteq \text{Dom}(A') \) (Lemma 2.4.1); (b) \( FV(e_2[z/x]) \subseteq \text{Dom}(\langle [z : s \mapsto e_2]A' \rangle) \). This follows from the inductive result \( FV(\lambda x : s \cdot e) \subseteq \text{Dom}(A') \); (c) \( \text{distinct(Dom}(\langle [z : s \mapsto e_2]A' \rangle@Z) \).
This follows from the inductive result \( \text{distinct}(\text{Dom}(A'))@Z' \). The only remaining case is \{\text{Rec}\} which is very similar to \{\text{Appl}\}.

Since we are concerned with the evaluation of only valid configurations we will assume in the remainder of the paper that all environment bound variables are distinct and that all configurations are closed. The previous theorem makes this possible. We will take advantage of the fact that environments bind only unique variables by using the notation \( A(x) \) to refer to the expression bound to \( x \) in \( A \). The name supply subscript \( Z \) will be dropped from further discussion where it is not relevant, and we will use the notation \( \text{Valid}(\langle e, A \rangle) \) to indicate that the configuration is closed and that the variables bound in \( A \) are distinct from each other.

2.5. An example evaluation

In this section, an example is presented which demonstrates how lazy evaluation with sharing is carried out by the operational semantics rules. The proof of the evaluation is laid out in a vertical fashion, as opposed to the standard horizontal layout of inferences. The proof that configuration \( c1 \) evaluates to \( c2 \) would be laid out as:

\[
\begin{align*}
\text{c1} & \\
& \text{[sub-proof-1} \\
& \text{sub-proof-2} \\
\text{c2}
\end{align*}
\]

where sub-proof-2 only exists in the cases of the conditional and application. The evaluation of the expression \((\lambda x. \text{if}(\text{iszero}(x), 5, \text{succ}(\text{pred}(\text{succ}(0))))))\) is evaluated in environment \( [y \mapsto \text{3}] \) (with name supply list \([z]\)) in Fig. 7. In this example and those of the next section the types have been dropped for brevity. In the figure, \text{3} is an abbreviation of the expression \(\text{succ}(\text{succ}(\text{succ}(\text{0})))\). This example demonstrates how the argument \(\text{pred}(\text{succ}(\text{0}))\) is shared among two references.

In the example, the * marks the first access of the variable \( z \), and the evaluation of \(\text{pred}(\text{succ}(\text{0}))\). The result of this evaluation, \text{0}, is stored in the environment, so that when \( z \) is accessed again \text{0} is evaluated to itself, as opposed to reevaluating the original argument. This evaluation of \( z \) is marked by a \( \uparrow \).

2.6. Recursion

In order to better understand how recursion is modeled by \textsc{Lazy-PCF+Shar}, we consider a few examples of the evaluation of recursive functions. In the first example a recursive function that adds five to its argument is applied to \text{2}. We will use the following abbreviations in the discussion:

\[
\begin{align*}
\text{plus5} &= \mu f. \lambda x. \text{if}(\text{iszero}(x), 5, \text{succ}(f(\text{pred}(x)))) \\
\text{plus5val}(z) &= \lambda x. \text{if}(\text{iszero}(x), 5, \text{succ}(z(\text{pred}(x))))
\end{align*}
\]
Using the operational semantics of LAZY-PCF+SHAR to evaluate the application of \texttt{plus5} to \texttt{2} we get the following result:

\[
\langle \texttt{plus5 2, } \rangle_{[1, m_1, z_1, m_2, z_2, m_3, z_3]} \downarrow
\langle 7, [m_3 \mapsto 0, m_2 \mapsto 1, m_1 \mapsto 2, z_1 \mapsto \texttt{plus5val(z_2)},
\begin{align*}
z_2 &\mapsto \texttt{plus5val(z_3)},
z_3 &\mapsto \texttt{plus5} \rangle_{[1]}
\]
\]

During each recursive call, a new binding for \texttt{plus5} (using a \texttt{z}_i variable in this example) is added to the environment by the \{Rec\} rule. The previous binding is evaluated and used once, but future recursive calls are made to the original \mu-expression (\texttt{plus5}). Thus the amount of space used by the evaluation of a recursive function is proportional to the depth of recursion for each function call.

We next consider an example where the body of the recursive function is not a value:

\[
\texttt{plus5'} = \mu f.(\texttt{plus 5})
\]
\[
\texttt{plus} = \lambda w.\lambda x.\texttt{if(iszero}(x), w, \texttt{succ}(f(\texttt{pred}(x))))
\]
\[
\texttt{plus5'val(w,z)} = \lambda x.\texttt{if(iszero}(x), w, \texttt{succ}(z \texttt{pred}(x))))
\]

The result of evaluating \texttt{plus5'} \texttt{2} appears similar to the previous example:

\[
\langle \texttt{plus5' 2, } \rangle_{[z_1, w_1, m_1, z_2, w_2, m_2, z_3, w_3, m_3]} \downarrow
\langle 7, [m_3 \mapsto 0, m_2 \mapsto 1, m_1 \mapsto 2,
\begin{align*}
w_1 &\mapsto 5, z_1 \mapsto \texttt{plus5'val(w_2,z_2)},
w_2 &\mapsto 5, z_2 \mapsto \texttt{plus5'val(w_3,z_3)},
w_3 &\mapsto 5, z_3 \mapsto \texttt{plus5'} \rangle_{[1]}
\]
\]

Fig. 7. An example evaluation.
As in the previous example, during each recursive call the body of the original \(\mu\)-expression \(\text{plus5}'\) is evaluated, and a new binding for \(\text{plus5}'\) is made. However, since the body is not a value in this example, the application of \(\text{plus}\) to \(5\) is evaluated again during each call to the recursive function.

As a final example of an evaluation of recursive functions, let us consider the first example above, but this time we will use the \(Y\) combinator to perform the recursion. Since our evaluation is independent of types we can use the \(Y\) combinator \(\lambda f. (\lambda x f(xx))(\lambda x f(xx))\) to evaluate the following function:

\[
\text{plus5f} = \lambda f. \lambda x. \text{if}(\text{iszero}(x), 5, \text{succ}(f(\text{pred}(x))))
\]

We then get the following as the evaluation of \((Y\text{plus5f})2\):

\[
\begin{align*}
\langle (Y\text{plus5f})2, [] \rangle & \Downarrow \langle 7, [m_3 \mapsto 0, m_2 \mapsto 1, m_1 \mapsto 2, \\
z_1 \mapsto \text{plus5val}(z_2), z_2 \mapsto \text{plus5val}(z_3), z_3 \mapsto (x_3, x_3), \\
x_3 \mapsto x_2, x_2 \mapsto \lambda x. f_1(x x), x_1 \mapsto \lambda x. f_1(x x), \\
f_1 \mapsto \text{plus5f} \rangle
\end{align*}
\]

This result is very similar to the \(\mu\) version. The only difference is the addition of the extra variables (the \(x, s\)) used to evaluate the \(Y\) combinator. Thus the only advantage to using the \(\mu\)-abstractions over the \(Y\) combinator is the absence of a small amount of overhead (constant in time and space) required to evaluate the combinator.

From the above examples we can see that in modeling recursion, \(\text{LAZY-PCF+SHAR}\) uses space proportional to the number of recursive calls made, and if the body of a \(\mu\)-abstraction is not a value, the body is re-evaluated during each recursive call. Unfortunately this is not a good reflection of the normal implementation of recursion, which is done using cyclic bindings. Any recursive reference in a closure points to itself and an evaluation of that body causes an update to the closure. Thus the space used is constant (the size of the function body) and the body is evaluated only once.

There are two reasonable modifications that may be made to the semantics in order to better model recursion. The first is to allow cyclic references in the environment. This would allow the following change to the \{\text{Rec}\} rule:

\[
\text{Rec} \quad \langle z, [z \mapsto e[z/x]]A \rangle_Z \Downarrow \langle v, A' \rangle_Z, \\
\langle \mu x. e, A \rangle_Z \Downarrow \langle v, A' \rangle_Z.
\]

A change would then be necessary in the \{\text{Var}\} rule to evaluate the environment-bound expression with respect to the entire environment (as opposed to just the tail of the environment). The introduction of self-references by the \{\text{Rec}\} rule leads to the possibility of mutually recursive references in the environment (when the body of the \(\mu\)-expression is an application and the recursive variable occurs in the argument). Though this solution reflects the implementation well, it is very difficult to show its correctness. The proof of correctness of \(\text{LAZY-PCF+SHAR}\) with respect to a call-by-name semantics depends heavily on the fact that environments are ordered (validity of environments). It is not at all obvious how to modify the definitions and proof to allow for cyclic environments.
Another possible solution is to implement recursion using cycles but to restrict the bodies of $\mu$-abstractions to be values. With this restriction cyclic references will occur in the environment, but they will be only self-referential – no mutually recursive bindings will occur. This allows us to redefine the $\{\text{Rec}\}$ rule as follows:

$$\{\text{Rec}\} \ (\mu x.v.A)_{z:Z} \Downarrow (v[z/x],[z \mapsto v[z/x]]A)_{Z}$$

We have subscripted the binding variable $z$ with $r$ to indicate that it is a potentially cyclic binding. Then we can use the original $\{\text{Var}\}$ rule to evaluate non-cyclic bindings, and we use the following rule for cyclic bindings:

$$\{\text{Var}_r\} \ (x, A[0][x \mapsto v]A)_{Z} \Downarrow (v, A[0][x \mapsto v]A)_{Z}$$

This additional rule is not absolutely necessary, because using the original $\{\text{Var}\}$ rule to evaluate the cyclic value $v$ would yield the same result. However, the configuration in the premise to the $\{\text{Var}\}$ rule for a cyclic binding would not be valid because one of its free variables, $x$, would not be found in the tail of the environment, $A$. We still require that the validity property be propagated by the semantic rules, because it is required by the proof of correctness. However, the definition of valid environments must be modified to account for cyclic bindings. More specifically, for an environment $[x \mapsto e]A$ to be valid, each free variable in $e$ not equal to $x$ must be found in $A$. The previous definition did not allow $x$ to be free in $e$. This modification, along with the $\{\text{Var}_r\}$ rule, is sufficient for Theorem 2.3 to hold for the modified semantics. Similar modifications can be made in the correctness proof so that it holds for the modified semantics as well. Though this model of recursion is semantically sound and a good model of the implementation, it is fairly restrictive. Expressions such as $\mu f.(\text{plus } 5)$, used above, would not be allowed.

3. Properties of the LAZY-PCF+SHAR semantics

In order to demonstrate the suitability of the call-by-need semantics for reasoning, some properties of the semantics will be presented in this section. The fact that the $\Downarrow$ relation is a partial function will be presented first, followed by an investigation of the expressions which may occur as results of LAZY-PCF+SHAR evaluation and their properties. The section concludes with a proof of the subject reduction property for LAZY-PCF+SHAR.

3.1. $\Downarrow$ is a partial function

One nice property of many semantics is that each term evaluates to a unique result. In LAZY-PCF+SHAR this is equivalent to saying that each configuration evaluates to a unique resulting configuration. This is stated formally in the following lemma. The
presence of the name supply, \( Z \), ensures that lambda bound variables always renamed with the same name in both evaluations.

**Lemma 3.1.** If \( \langle e, A \rangle_Z \downarrow \langle e', A' \rangle_{Z'} \) and \( \langle e, A \rangle_Z \downarrow \langle e'', A'' \rangle_{Z''} \) then \( e' = e'' \), \( A' = A'' \) and \( Z' = Z'' \).

This theorem follows from the fact that all of the rules are deterministic. It may also be proved by induction on the height of the inference justifying the evaluation \( \langle e, A \rangle \downarrow \langle e', A' \rangle \).

### 3.2. Values and their properties

The term *value* is used to refer to a term which may be the result of a complete evaluation. In LAZY-PCF+SHAR, however, the result of an evaluation is a configuration containing an expression and an environment. We consider LAZY-PCF+SHAR values to be the subset of expressions which may occur as the first element of the resulting configuration of an evaluation. As is the case for many evaluation models, the set of all possible values of LAZY-PCF+SHAR can be characterized.

**Definition 3.2 (Values).** \( v \) is a *value* if for some \( e, A, \) and \( A' \) \( \langle e, A \rangle \downarrow \langle v, A' \rangle \)

We define a subset of expressions, \( VAL \), which serves as a syntactic description of values.

**Definition 3.3.**

\[
\begin{align*}
S & ::= 0 \mid \text{succ}(S) \\
VAL & ::= \text{true} \mid \text{false} \mid S \mid \lambda x : t . e
\end{align*}
\]

The following two lemmas demonstrate that the set \( VAL \) completely describes the values of LAZY-PCF+SHAR.

**Lemma 3.4.** If \( \langle e, A \rangle \downarrow \langle v, A' \rangle \) then \( v \in VAL \)

**Proof.** By induction on the height of the inference justifying \( \langle e, A \rangle \downarrow \langle v, A' \rangle \).

In proving that the set \( VAL \) contains only values, a stronger statement than necessary, namely that values evaluate to themselves without updating the environment, will be proven. The stronger statement is shown here because it is useful in later proofs.

**Lemma 3.5.** If \( v \in VAL \) then \( \langle v, A \rangle \downarrow \langle v, A \rangle \)

**Proof.** By induction on the structure of the elements of \( VAL \).

Thus the set \( VAL \) describes the set of values of LAZY-PCF+SHAR. We use the letter \( v \) to denote elements of this set.
Another useful lemma regarding values states that if \( x \) is bound to a value in an environment \( A \) and the configuration \( \langle e, A \rangle \) evaluates to \( \langle e', A' \rangle \), then \( x \) will be bound to the same value in \( A' \).

**Lemma 3.6.** If \( \langle e, A \rangle \downarrow \langle v, A' \rangle \) and \( A(x) \in VAL \) then \( A'(x) = A(x) \).

**Proof.** By induction on the height of the inference justifying \( \langle e, A \rangle \downarrow \langle v, A' \rangle \) using Lemma 3.5.

Thus the operational semantics rules propagate the expression that a variable is bound to in an environment if it is a value.

### 3.3. Subject reduction

In this section we show that Lazy-PCF+SHAR has the property of subject reduction, which is also commonly referred to as type soundness. Subject reduction is the property that evaluation preserves types. In other words, the type of an expression will be the same as the type of its value. In this section we prove the semantics has this property directly from the definition of the semantics and types for Lazy-PCF+SHAR. This result could be obtained from the proof of equivalence between Lazy-PCF+SHAR and call-by-name (Theorem 4.2) and the subject reduction result for call-by-name. However, the direct proof serves to help demonstrate the suitability of the Lazy-PCF+SHAR semantics for reasoning.

Before stating the theorem, some definitions and lemmas are required. The first two definitions are related to type contexts, which are partial mappings from variables to types, as stated previously. The first definition extracts a type context from an operational semantics environment.

**Definition 3.7 (Type context of an environment).**

\[
\text{Context}([ ]) = \bot \\
\text{Context}([x:t\mapsto e]A) = \text{Context}(A)[t/x]
\]

where \( \bot \) is the mapping that is undefined for each variable.

The next definition describes the notion of one type context extending another. This concept is the same as that of one partial mapping extending another. A context \( H' \) extends another context \( H \) if for all variables \( x \) for which \( H \) is defined, \( H'(x) = H(x) \). We will use \( \text{Dom}(H) \) to denote the domain of a context \( H \).

**Definition 3.8.**

1. \( H \) extends \( H \).
2. If \( H' \) extends \( H \) and \( x \notin \text{Dom}(H') \) then \( H'[t/x] \) extends \( H \).
A simple lemma combining both of these definitions states that if one configuration, \( \langle e, A \rangle \), evaluates to another, \( \langle e', A' \rangle \), then the type context of the second environment extends the type context of the first.

**Lemma 3.9.** If \( \langle e, A \rangle \downarrow \langle e', A' \rangle \), then \( \text{Context}(A') \) extends \( \text{Context}(A) \).

**Proof.** By induction on the height of the inference justifying \( \langle e, A \rangle \downarrow \langle e', A' \rangle \).

An environment will be considered to be well-typed if each expression \( e \) found in a binding \( [x : t \mapsto e] \) has type \( t \) when typed with respect to the type context of the remainder of the environment (the bindings that occur to the right of the original binding). This is formalized as follows.

**Definition 3.10 (Well-typed environments).**
1. \( [\ ] \) is a well-typed environment.
2. If \( \mathcal{A} \) is a well-typed environment and \( \text{Context}(\mathcal{A}) \vdash e : t \), then \( [x : t \mapsto e] \mathcal{A} \) is a well-typed environment.

The definition of well-typed environments can be extended to configurations by requiring that the environment of a configuration is well-typed and that the expression of the configuration has some type in the type context of the environment.

**Definition 3.11 (Well-typed configuration).** If \( \mathcal{A} \) is a well-typed environment and for some \( t \), \( \text{Context}(\mathcal{A}) \vdash e : t \), then \( \langle e, \mathcal{A} \rangle \) is a well-typed configuration.

Now the subject reduction theorem can be stated and proved. It is proved by induction on the height of the evaluation, and in order for the induction to always apply, we simultaneously prove that the final environment must be well-typed. However, this additional result demonstrates that the evaluation preserves well-typed configurations.

**Theorem 3.12 (Subject Reduction).** If \( \text{Context}(\mathcal{A}) \vdash e : t \), \( \mathcal{A} \) is well-typed, and \( \langle e, \mathcal{A} \rangle \downarrow \langle e', \mathcal{A}' \rangle \), then \( \text{Context}(\mathcal{A}') \vdash e' : t \) and \( \mathcal{A}' \) is well-typed.

**Proof.** By induction on the height of the inference justifying \( \langle e, \mathcal{A} \rangle \downarrow \langle e', \mathcal{A}' \rangle \). In the \{Appl\} and \{If\} cases, Lemma 3.9 is required.

This theorem has been proved with Coq, an interactive theorem prover, for a previous version of the semantics of LAZY-PCF+SHAR in [18]. The earlier version of the semantics included a form of a closure, making the proof more difficult because of the way the rules for evaluating closures and application are defined. This version of the semantics is also the one introduced in [16].
4. Computational correctness of the operational semantics

In this section, we show that the operational semantics are computationally correct in the sense that the values computed by the semantics are correct. This can be done by proving that the semantics is equivalent (with respect to the values calculated) to some accepted or standard semantics, whether operational or denotational. In this case we will show that the operational semantics of LAZY-PCF+SHAR is equivalent to the call-by-name operational semantics of PCF found in [8]. The soundness and adequacy of this call-by-name semantics with respect to a standard denotational semantics are already shown in [8]. Thus once the equivalence of LAZY-PCF+SHAR and the call-by-name semantics is shown, the soundness and adequacy of LAZY-PCF+SHAR with respect to the standard denotational semantics follows automatically.

The main result of this section is the following theorem, which states that the LAZY-PCF+SHAR semantics and the call-by-name semantics yield the same values for expressions of ground type. The call-by-name semantics is a relation over terms (as opposed to configurations) and is denoted by $\Downarrow$ (the call-by-name semantics are defined in Fig. 9 and explained in Section 4.4).

**Theorem 4.1.** If $\Downarrow \vdash e : \text{nat}$ or $\Downarrow \vdash e : \text{bool}$ then

$$\exists A : \langle e, [\ ] \rangle \Downarrow \langle v,A \rangle \iff e \Downarrow v.$$  

Stating such a theorem for values of functional types introduces complications. Values of functional types in both semantics are expressed as lambda abstractions which, in LAZY-PCF+SHAR, may involve free variables defined in the environment $A$. Thus relating the functional values of LAZY-PCF+SHAR to the functional values of the call-by-name semantics would require an incorporation of the bindings in the environment into the LAZY-PCF+SHAR functional value. However, as the next section shows, simply substituting the bindings in $A$ into the LAZY-PCF+SHAR functional value is not sufficient to achieve syntactic equality with the functional value derived by the call-by-name semantics. Thus Theorem 4.1 is stated for values of ground type $(true, false, \text{succ}^n(0))$, which do not contain free variables.

The proof of Theorem 4.1 would normally involve induction on the heights of the inferences justifying the evaluations (both of which are defined by natural semantics). This process introduces expressions of functional types and environments that are not empty, which do not conform to the statement of the theorem. Thus, a standard proof approach requires a more general statement of the theorem, which allows general configurations and values of any type. In order to allow general configurations, the statement of the theorem requires the definition of some relationship between configurations (used in LAZY-PCF+SHAR) and expressions (used in the call-by-name semantics) which will satisfy the theorem. We will use $\Rightarrow$ to denote this unspecified relationship. Now the more general equivalence theorem can be stated as follows.
Theorem 4.2. If \( (e, A) \rightarrow e' \) then
\[
\exists v, A' : (e, A) \rightarrow (v, A') \iff \exists v : e' \Downarrow v'
\]
where \( (v, A') \rightarrow v' \).

The remainder of this section will be a proof of this theorem, which will include an appropriate definition for \( \rightarrow \).

Because the theorems and proofs of this section do not depend on the type annotations in the abstractions and environments they will be dropped from the syntax.

4.1. The need for an intermediate semantics

The proof of Theorem 4.2 depends on an appropriate definition of the relation \( \rightarrow \). A reasonable first attempt at this definition would be the suggestion of the previous section, which would relate configurations to expressions by substituting sequentially, from left to right, the bindings of the environment into the configuration's expression. This process is formally defined as a function from expressions and environments to expressions as follows.

Definition 4.3 (\( \{ \} \) Translation).
\[
e\{[\] \} = e, \quad e\{[x \mapsto e']A\} = e[e'/x]{A}
\]

Given this definition, we would define the \( \rightarrow \) relation as follows: \( (e, A) \rightarrow e\{A\} \). Unfortunately, as stated previously, this definition does not satisfy Theorem 4.2. A counterexample disproving the theorem is the evaluation of \( \langle \text{if(iszero}(x), \lambda y.x, \lambda z.3), [x \mapsto \text{pred}(1)] \rangle \), where \( n \) abbreviates \( \text{suc}^n(0) \). This configuration evaluates to \( \langle \lambda y.x, [x \mapsto 0] \rangle \) in \textsc{lazy-PCF+Shar}. By the \( \{ \} \) translation the original configuration is related to \( \text{if(iszero(pred}(1)), \lambda y.\text{pred}(1), \lambda z.3 \rangle \). This term evaluates to \( \lambda y.\text{pred}(1) \) according to the call-by-name semantics (Fig. 9). This value is not equal to the \( \{ \} \) translation of \( \langle \lambda y.x, [x \mapsto 0] \rangle \), which is \( \lambda y.0 \).

The problem is that due to the sharing of arguments in \textsc{lazy-PCF+Shar}, all “occurrences” of an argument are evaluated at the same time. In call-by-name evaluation, unnneed or unencountered occurrences of an argument remain unevaluated. Since in \textsc{lazy-PCF+Shar} the original value of the argument is forgotten once it is evaluated, it is difficult to relate evaluated configurations of functional type to the corresponding expression evaluated by the call-by-name semantics.

One way to overcome this problem is to modify the call-by-need semantics to remember the original value of a variable, even after it has been updated by the \{Var\} rule. Then the original values of the variables can be substituted into the value of an evaluation to get the correct call-by-name PCF expression. This modification of the lazy evaluation semantics provides the basis for an intermediate semantics which will be used to bridge the gap between the call-by-need semantics and the call-by-name semantics, and aid in the definition of \( \rightarrow \).
The remainder of Section 4 will begin with a definition of the intermediate semantics. This will be followed by the proof of a theorem relating \textsc{lazy-PCF} $\mid$ \textsc{SHAR} to the intermediate semantics. Then another theorem will be presented which relates the intermediate semantics to the call-by-name semantics. The last subsection will use these two theorems to define the $\equiv$ relation and justify Theorem 4.2 by transitivity.

4.2. The intermediate semantics

As implied above, the intermediate semantics is actually an implementation of call-by-need. The semantics rules are shown in Fig. 8. The intermediate semantics evaluates configurations denoted $\langle\langle e, D \rangle\rangle_Z$, where $D$ is the intermediate semantics environment. The structure of this environment can be formally described as:

$$D ::= [] \mid [x: \langle e_1, e_2 \rangle]D$$

The intermediate semantics environment maps each variable to a pair of expressions. In the intermediate semantics, the first expression of the pair corresponds exactly to the expression in the semantics of \textsc{lazy-PCF+SHAR} and the second expression is used to remember the original value bound to a variable. The second expression in the pair is never changed by the semantic rules; its purpose is to maintain the original expression for each variable so that call-by-need values may be compared to call-by-name values. This can be seen in the \{iVar\}, \{iAppl\}, and \{iRec\} rules. In the \{iVar\} rule, only the first element of the pair bound to $x$ is evaluated. The second element remains unchanged. In the \{iAppl\} rule two copies of the argument, $e_2$, are placed in the environment, the first to be evaluated and updated if necessary, and the second to hold the original argument. Similarly in the \{iRec\} rule two copies of the mu expression are placed in the environment.

As in the \textsc{lazy-PCF+SHAR} rules, $Z$ is used to ensure unique names in the environment. The concepts of valid environments and configurations (Definitions 2.1 and 2.2) can be extended to the intermediate semantics by updating the definition of valid environments to ensure that both expressions bound to a variable have their free variables defined in the tail of the environment:

\textbf{Definition 4.4 (Valid Environment [Instrumented Semantics]).}

$$\text{Valid}([])$$

$$\text{Valid}([x:t: \langle e_1, e_2 \rangle]D) \text{ if } FV(e_1) \cup FV(e_2) \subseteq \text{Dom}(D) \text{ and } \text{Valid}(D)$$

If we use this definition of valid environments to define valid configurations (Definition 2.2) for the instrumented semantics then the theorem that evaluation pre-
serves validity of configurations (Theorem 2.3) holds for the instrumented semantics in the same way. As with the original semantics we will drop the name supply subscript Z from further discussion and assume that variable names in environments are always unique.

4.3. The equivalence of LAZY-PCF+SHAR and the intermediate semantics

In order to state the theorems and definitions that follow, we first define \( Fst(D) \) (analogously, \( Snr(D) \)) to be the environment that maps the variables defined in \( D \) to only the first (respectively, second) expression of each pair of expressions. This gives us the LAZY-PCF+SHAR environment corresponding to \( D \) by stripping out the second (respectively first) expressions of each pair.

\[
\begin{align*}
\{i0\} & \quad \langle 0, D \rangle_Z \downarrow \langle 0, D \rangle_Z & \{iT\} & \quad \langle \text{true}, D \rangle_Z \downarrow \langle \text{true}, D \rangle_Z \\
\{iL\} & \quad \langle \lambda x.e, D \rangle_Z \downarrow \langle \lambda x.e, D \rangle_Z & \{iF\} & \quad \langle \text{false}, D \rangle_Z \downarrow \langle \text{false}, D \rangle_Z \\
\{iP0\} & \quad \langle e, D \rangle_Z \downarrow \langle 0, D' \rangle_Z' & \{iZT\} & \quad \langle e, D \rangle_Z \downarrow \langle 0, D' \rangle_Z' \\
\{iP\} & \quad \langle \text{pred}(e), D \rangle_Z \downarrow \langle 0, D' \rangle_Z' & \{iZF\} & \quad \langle \text{iszero}(e), D \rangle_Z \downarrow \langle \text{true}, D' \rangle_Z' \\
\{iS\} & \quad \langle e, D \rangle_Z \downarrow \langle v, D' \rangle_Z' & \{iVar\} & \quad \langle x, D_0[z \mapsto (e_1, e_2)]D \rangle_Z \downarrow \langle v, D_0[z \mapsto (e_1, e_2)]D' \rangle_Z' \\
\{iAppl\} & \quad \langle e_1, D \rangle_Z \downarrow \langle \lambda x.e, D' \rangle_Z' \langle e[z/x], z \mapsto (e_2, e_3) \rangle D \rangle_Z \downarrow \langle v, D'' \rangle_Z'' & \{iifTrue\} & \quad \langle \text{if}(e_1, e_2, e_3), D \rangle_Z \downarrow \langle v, D'' \rangle_Z'' \\
\{iifFalse\} & \quad \langle e_1, D \rangle_Z \downarrow \langle \text{false}, D' \rangle_Z' \langle e_3, D' \rangle_Z' \downarrow \langle v, D'' \rangle_Z'' & \{iRec\} & \quad \langle \mu x.e, D \rangle_Z \downarrow \langle \mu x.e, D \rangle_Z \downarrow \langle v, D' \rangle_Z' \\
\end{align*}
\]

Fig. 8. Intermediate semantics.
**Definition 4.5** \((Fst(D), \text{Snd}(D))\).

\[
\begin{align*}
Fst([ ]) &= [ ] \\
Fst([x \mapsto (e_1, e_2)]D) &= [x \mapsto e_1]Fst(D)
\end{align*}
\]

\[
\begin{align*}
\text{Snd}([ ]) &= [ ] \\
\text{Snd}([x \mapsto (e_1, e_2)]D) &= [x \mapsto e_2]\text{Snd}(D)
\end{align*}
\]

The next theorem states the equivalence of \textsc{lazy-pcf+shar} and the intermediate semantics.

**Theorem 4.6.** If \(A = Fst(D)\) then

\[
\exists A' : (e, A) \downarrow (v, A') \iff \exists D' : \langle e, D \rangle \downarrow \langle v, D' \rangle
\]

where \(A' = Fst(D')\).

**Proof.** The proof is trivial because the \textsc{lazy-pcf+shar} semantics can be obtained from the intermediate semantics by simply "erasing" the second expression of each pair of expressions in each binding in each environment. The second expressions of the intermediate semantics do not further constrain the evaluation in any way. More formally, this theorem may be proved by induction on the height of the reductions.

### 4.4. The call-by-name semantics

Now the computational correctness of \textsc{lazy-pcf+shar} can be verified by showing the equivalence of the intermediate semantics and the call-by-name semantics. The call-by-name semantics which will be used here is the natural semantics for call-by-name evaluation presented by Gunter in his book [8]. This semantics defines a relation between terms of PCF without the use of environments. The semantics is shown in Fig. 9.

Most of the rules are analogous to the semantics of \textsc{lazy-pcf+shar}. The significant difference, as expected, is in the application rule. Application is implemented by evaluating the first expression to a lambda expression, and then using substitution to replace the formal parameter with the argument of the function before evaluating. A similar distinction can be seen in the rule for evaluating recursive functions.

The set of values for call-by-name evaluation of PCF terms is the same as those of \textsc{lazy-pcf+shar}, so \(v\) is again used to denote them.

### 4.5. The equivalence of the intermediate semantics and the call-by-name semantics

The discussion of the theorems equating the intermediate semantics and the call-by-name semantics is preceded by a few necessary definitions and lemmas. First, a relation between intermediate semantics configurations and PCF expressions, similar to the \(\Rightarrow\)
relation, is required, along with some related lemmas. Second, a relation between the
pairs in the intermediate semantics environment, formalizing the idea that the second
element should be the "original" argument bound to the variable, is required.

The relationship between intermediate semantics configurations and PCF terms will
not quite suffice as the definition for \( \Rightarrow \), which must be defined between Lazy-PCF+
SHAR configurations and PCF terms. For this definition we will use a translation from
intermediate semantics configurations to PCF expressions by sequentially substituting
the second expression of each binding for the bound variable. It is defined in terms of
the \{\} translation (Definition 4.3) used in the first proof attempt.

**Definition 4.7 ({} Translation).** \( e[\{D\}] = e[Snd(D)] \)

Given this translation, some lemmas concerning it will be needed. The first lemma
is an extension of the Substitution Lemma \([2, 2.1.16]\), to apply to all PCF terms.

**Lemma 4.8.** \( e[e_1/x][e_2/y] = e[e_2/y][e_1[e_2/y]/x] \)

**Proof.** By structural induction on the terms of PCF.

The second lemma is required in the application case of the proof of the equivalence
theorem. It is used to verify the fact that placing an argument in the environment and
evaluating it later, when needed, is equivalent to the immediate substitution of the
argument.

**Lemma 4.9.** \( e[\{x \mapsto (e_1, e_2)\}D] = e[D][e_2\{D\}/x] \).
Proof. This lemma is proven by induction on the length of \( D \). The case for \( |D| > 0 \) uses the previous Lemma 4.8.

The third lemma states that the translation is preserved for an arbitrary expression when the environment is updated via evaluation. It is used in many cases of the proof of the equivalence theorem.

**Lemma 4.10.** If \( \langle e_1, D \rangle \downarrow \langle v, D' \rangle \) then for any \( e \) such that \( FV(e) \subseteq Dom(D) \), \( e[D] = e[D'] \).

This lemma is proved with the help of an additional definition and two more lemmas. This approach should give some intuition regarding the lemma. The definition is a relation on single-expression environments, as used in the original \textsc{LAZY-PCF+SHAR} semantics. It holds between two environments when all of the bindings of the first are included in the second and the relative order of the bindings is preserved.

**Definition 4.11.**

\[
\begin{align*}
\{ & \} \preceq \{ & \\
A \preceq B & \text{ and } x \notin \text{Dom}(B) \quad \Rightarrow \quad A \preceq [x \mapsto e]B \\
A \preceq B & \text{ and } x \notin \text{Dom}(A) \cup \text{Dom}(B) \quad \Rightarrow \quad [x \mapsto e]A \preceq [x \mapsto e]B 
\end{align*}
\]

The first lemma states that the intermediate semantics propagates this relation for the second element of the pairs in the environments. In other words, evaluation does not change the second element of pairs. It may add bindings and it may alter the first element of the pair, but it does not alter the second expressions. The lemma is easily verified by induction on the height of the evaluation.

**Lemma 4.12.** If \( \langle e_1, D \rangle \downarrow \langle v, D' \rangle \) then \( \text{Snd}(D) \preceq \text{Snd}(D') \).

The next lemma states that the \( \preceq \) relationship on environments preserves the \{ \} translation for a given expression.

**Lemma 4.13.** If \( A \preceq B \), \( \text{Valid}(A) \), and \( \text{Valid}(B) \), then for any \( e \) such that \( FV(e) \subseteq \text{Dom}(A) \), \( e\{A\} = e\{B\} \).

**Proof.** This lemma can be proved by induction on Definition 4.11. The first case is trivial. In the second case, we show that \( e\{A\} = e\{[x \mapsto e_1]B\} \) given \( A \preceq B \). Since \( x \) is not in \( \text{Dom}(A) \) it cannot be in \( FV(e) \) so \( e\{[x \mapsto e_1]B\} = e\{B\} \), and this equals \( e\{A\} \) by induction. In the third case we show that \( e\{[x \mapsto e_1]A\} = e\{[x \mapsto e_1]B\} \) given \( A \preceq B \). This reduces to \( e\{e_1/x\}\{A\} = e\{e_1/x\}\{B\} \) which is true by induction.

**Proof of Lemma 4.10.** From Lemma 4.12 we know \( \text{Snd}(D) \preceq \text{Snd}(D') \), which by Lemma 4.13 gives us \( e\{\text{Snd}(D)\} = e\{\text{Snd}(D')\} \). By definition this gives \( e[D] = e[D'] \).
Table 1
Structure of \( e(D) \), given the structure of \( e \) and \( \text{Valid}(\langle e,D \rangle) \)

<table>
<thead>
<tr>
<th>Structure of ( e(D) )</th>
<th>Structure of ( \langle e,D \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 {D} = 0 )</td>
<td>( \text{iszero}(e){D} = \text{iszero}(e){D} )</td>
</tr>
<tr>
<td>( \text{true}{D} = \text{true} )</td>
<td>( (\lambda x.e){D} = \lambda x(e){D} )</td>
</tr>
<tr>
<td>( \text{false}{D} = \text{false} )</td>
<td>( (\mu x.e){D} = \mu x(e){D} )</td>
</tr>
<tr>
<td>( \text{pred}(e){D} = \text{pred}(e){D} )</td>
<td>( (e_1, e_2){D} = (e_1){D} {e_2}{D} )</td>
</tr>
<tr>
<td>( \text{succ}(e){D} = \text{succ}(e){D} )</td>
<td>( \text{if}(e_1,e_2,e_3){D} = \text{if}(e_1){D}, e_2{D}, e_3{D} )</td>
</tr>
</tbody>
</table>

\[ x\{D\} = e_2\{D_2\} \text{ where } D = D_1[x \mapsto (e_1,e_2)]D_2 \text{ for some } D_1,D_2,e_1,e_2 \]

Table 2
Structure of \( e \), given the structure of \( e(D) \)

<table>
<thead>
<tr>
<th>Structure of ( e(D) )</th>
<th>Structure of ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e{D} = 0 )</td>
<td>( e ) is a variable or ( e = 0 ).</td>
</tr>
<tr>
<td>( e{D} = \text{true} )</td>
<td>( e ) is a variable or ( e = \text{true} ).</td>
</tr>
<tr>
<td>( e{D} = \text{false} )</td>
<td>( e ) is a variable or ( e = \text{false} ).</td>
</tr>
<tr>
<td>( e{D} = \text{pred}(e) )</td>
<td>( e ) is a variable or ( e = \text{pred}(e') ) for some ( e' ).</td>
</tr>
<tr>
<td>( e{D} = \text{succ}(e) )</td>
<td>( e ) is a variable or ( e = \text{succ}(e) ) for some ( e' ).</td>
</tr>
<tr>
<td>( e{D} = \text{iszero}(e) )</td>
<td>( e ) is a variable or ( e = \text{iszero}(e) ) for some ( e' ).</td>
</tr>
<tr>
<td>( e{D} = \lambda x.e )</td>
<td>( e ) is a variable or ( e = \lambda x.e' ) for some ( e' ).</td>
</tr>
<tr>
<td>( e{D} = \mu x.e )</td>
<td>( e ) is a variable or ( e = \mu x.e' ) for some ( e' ).</td>
</tr>
<tr>
<td>( e{D} = (e_1, e_2) )</td>
<td>( e ) is a variable or ( e = (e', e') ) for some ( e' ) and ( e' ).</td>
</tr>
<tr>
<td>( e{D} = \text{if}(e_1,e_2,e_3) )</td>
<td>( e ) is a variable or ( e = \text{if}(e', e', e') ) for some ( e' ), ( e' ), and ( e' ).</td>
</tr>
</tbody>
</table>

The proof of the equivalence theorem will be broken down into cases based on syntactic structure. Because of this, it will be convenient to be able to determine what can be concluded about (a) the structure of \( e(D) \) when the structure of \( e \) is known and (b) the structure of \( e \) when the structure of \( e(D) \) is known. This information is presented in Tables 1 and 2, respectively. In the first table, \( \text{Valid}(\langle e,D \rangle) \) is assumed. Each of the equations in Table 1 can be proved by induction on the length of \( D \). The implications in Table 2 can be proved by induction on the syntactic structure of \( e \).

The proof of the equivalence theorem will be simplified by taking a closer look at the results in Table 2. Consider the case (for all expressions) when \( e \) is a variable, say \( y \). Then, according to Table 1, \( e\{D\} = y\{D\} = e_2\{D_2\} \) where \( D = D_1[y \mapsto (e_1,e_2)]D_2 \). If \( e_2 \) is a variable we can apply the equation from the table again. We can do this repeatedly to get the following sequence:

\[ a_0\{D_0\} = a_1\{D_1\} = a_2\{D_2\} = \cdots = a_n\{D_n\} \]

where \( e = a_0, D = D_0 \) and each of the \( a_i \)'s is a variable except for the last one \( (a_n) \). This sequence is finite because each \( D_i \) (except \( D_n \)) equals \( D_i'[a_i \mapsto (a_{i+1}', a_{i+1}')]D_{i+1} \) for some \( D_i' \) and \( a_{i+1}' \). Thus each \( D_{i+1} \) is a proper suffix of \( D_i \) and the environment is always decreasing in size. We will refer to this sequence as the chain of indirection for a configuration, and we will use induction on the length of the chain as a proof method.
The final element necessary for the following equivalence theorem is a property that captures the intended purpose of the intermediate semantics. The goal for the intermediate semantics was to add an expression to the environment that stores the \textit{original} expression bound to each variable. The following property captures this by stating that either the two expressions bound to a variable must be the same (before evaluation) or the first expression should be the \textit{value} of the second (after evaluation).

\textbf{Definition 4.14.}

1. $P_\emptyset(\emptyset)$

2. If $P_\emptyset(D)$ and \{ $e_1 = e_2$ or $e_2 \{D\} \downarrow e_1 \{D\}$ and $e_1 \in \text{VAL}$ \} then $P_\emptyset([x \mapsto (e_1, e_2)]D)$

Given these definitions and lemmas, the equivalence theorem between the intermediate semantics and the call-by-name semantics can finally be stated and proved. As usual, we will assume (implicitly) that all configurations are valid. We also assume that the above property holds for the initial environment ($P_\emptyset(D)$). A by-product of the theorem is the fact that evaluation preserves this property. This fact is required in order to prove the theorem inductively.

\textbf{Theorem 4.15.} If $P_\emptyset(D)$ then

$\exists v, D': \langle \langle e, D \rangle \rangle \downarrow_i \langle \langle v, D' \rangle \rangle \iff \exists v': e \{D\} \downarrow v'$

where $v' = v \{D'\}$ and $P_\emptyset(D')$.

\textbf{Proof.} The proof in each direction is by induction on the height of the appropriate evaluation. Facts that follow from Tables 1 and 2 and from the validity of configurations may be used in the proof without being noted as such. We begin with the forward direction ($\Rightarrow$).

$\Rightarrow$: The proof is by induction on the height of the inference of $\langle \langle e, D \rangle \rangle \downarrow_i \langle \langle v, D' \rangle \rangle$. The \{iL\}, \{iVar\} and \{iAppl\} cases are shown here.

\{iL\} $\langle \langle \lambda x.e, D \rangle \rangle \downarrow_i \langle \langle \lambda x.e, D \rangle \rangle$ and $P_\emptyset(D)$ are given. $(\lambda x.e)\{D\} = \lambda x.(e\{D\}) \downarrow \lambda x.(e\{D\})$ by the lambda rule in the call-by-name semantics.

\{iVar\} $\langle \langle x, D_0[x \mapsto (e_1, e_2)]D \rangle \rangle \downarrow_i \langle \langle v, D_0[x \mapsto (v, e_2)]D' \rangle \rangle$ and $P_\emptyset(D_0[x \mapsto (e_1, e_2)]D)$ are given. By definition this implies $P_\emptyset(D)$, which allows us to apply the theorem inductively to the premise to get $(\downarrow_i)e_1 \{D\} \downarrow v \{D'\}$ and $P_\emptyset(D')$. Our goal is to show $x \{D_0[x \mapsto (e_1, e_2)]D\} \downarrow v \{D_0[x \mapsto (v, e_2)]D'\}$, but this can be simplified because, by validity of configurations, $x \{D_0[x \mapsto (e_1, e_2)]D\} = e_2 \{D\}$ and $v \{D_0[x \mapsto (v, e_2)]D'\} = v \{D'\}$. Thus in order to prove the desired result we need to show $e_2 \{D\} \downarrow v \{D'\}$. We do so by considering the definition of the given statement $P_\emptyset(D_0[x \mapsto (e_1, e_2)]D)$, which gives two cases.

1. $e_1 = e_2$. Substituting this into evaluation ($\downarrow_i$) gives $e_2 \{D\} \downarrow v \{D'\}$.

2. $e_2 \{D\} \downarrow e_1 \{D\}$ and $e_1 \in \text{VAL}$. The second statement implies by Lemmas 3.5 and 3.1 with the premise of the \{iVar\} rule that $e_1 = v$ and $D = D'$. Substituting these into the first statement gives $e_2 \{D\} \downarrow v \{D'\}$. 

We still must show that $\mathcal{P}_\downarrow(D_0[x \mapsto (v, e_2)]D')$. We do this by induction on $D_0$.

1. $D_0 = [\_]$. Then the goal is $\mathcal{P}_\downarrow((x \mapsto (v, e_2)]D')$. This results from the evaluation $e_2[D] \Downarrow v[D']$ (demonstrated above) and the fact that $e_2[D] = e_2[D']$, (Lemma 4.10 with evaluation (t)). So we now have $e_2[D'] \Downarrow v[D']$, which by definition (with $v \in VAL$ and $\mathcal{P}_\downarrow(D')$) gives $\mathcal{P}_\downarrow((x \mapsto (v, e_2)]D')$.

2. $D_0 = [y \mapsto (a_1, a_2)]D'$. Then by induction on the length of $D_0$ we have $\mathcal{P}_\downarrow(D'_0[x \mapsto (v, e_2)]D')$. Using the equation for $D_0$ we rewrite the hypothesis $\mathcal{P}_\downarrow(D_0[x \mapsto (e_1, e_2)]D)$ as $\mathcal{P}_\downarrow([y \mapsto (a_1, a_2)]D'_0[x \mapsto (e_1, e_2)])$. Now by definition there are two cases:

(a) $a_1 = a_2$. Then by definition (with the inductive result) we get $\mathcal{P}_\downarrow([y \mapsto (a_1, a_2)]D'_0[x \mapsto (v, e_2)]D')$.

(b) $a_1 \neq a_2$. Then by definition (and induction) $a_2[D'_0[x \mapsto (e_1, e_2)]D] = a_2[D'_0[e_2/x][D]$. By Lemma 4.10 with evaluation (t) this is equal to $a_2[D'_0[e_2/x][D']$, which is equal to $a_2[D'_0[x \mapsto (v, e_2)]D')$. The same transformations can be made to get $a_1[D'_0[x \mapsto (e_1, e_2)]D] = a_1[D'_0[x \mapsto (v, e_2)]D']$. Substituting these into the evaluation gives $a_2[D'_0[x \mapsto (v, e_2)]D'] \Downarrow a_1[D'_0[x \mapsto (v, e_2)]D']$ which, along with the inductive result, gives $\mathcal{P}_\downarrow([y \mapsto (a_1, a_2)]D'_0[x \mapsto (v, e_2)]D')$ by definition.

{iApp} $\langle(e_1 e_2), D\rangle \Downarrow \langle v, D'' \rangle$ and $\mathcal{P}_\downarrow(D)$ are given. By induction on the height of the first premise to the \{iApp\} rule we have $e_1[D] \Downarrow (\lambda x.e)[D'] = \lambda x(e[D'])$, and $\mathcal{P}_\downarrow(D')$. In order to claim the induction result for the second premise, we must show $\mathcal{P}_\downarrow([z \mapsto (e_2, e_2)]D')$, but this follows from the previous result by definition. Thus by induction we have: $e[z/x][[z \mapsto (e_2, e_2)]D'] \Downarrow v[D''][D/z]$, and $\mathcal{P}_\downarrow(D'')$. Now if we can show that $e[z/x][[z \mapsto (e_2, e_2)]D'] = e[D'][e_2[D'/x]]$, then our two call-by-name reductions would yield, via the call-by-name application rule, the desired result: $(e_1[D] e_2[D]) = (e_1 e_2)[D] \Downarrow v[D''].$ By definition (and since $z \notin FV(e)$) $e[z/x][[z \mapsto (e_2, e_2)]D'] = e[e_2/x][D']$. By Lemma 4.9 this equals $e[D'][e_2[D'/x]]$. By Lemma 4.10, $e_2[D'] = e[D']$ (using the evaluation of the first premise to the rule). Substituting into the previous result we get $e[D'][e_2[D'/x]]$, to complete the verification of the equation. The secondary result, $\mathcal{P}_\downarrow(D'')$, was a result of the induction on the second premise.

$\Leftarrow$: The proof in this direction is by induction on the height of the inference of $e[D] \Downarrow v'$. The fact that the instrumented semantics propagates the $\mathcal{P}_\downarrow$ property has already been shown in the proof of the other direction, and this result is not shown again here. Each case will contain within it a proof by induction on the length of the chain of indirection. The lambda abstraction and application cases are shown here.

**Lambda** $e[D] = \lambda x.e_1 \Downarrow \lambda x.e_1 = v'$ and $\mathcal{P}_\downarrow(D)$ are given. We proceed by using induction on length of the chain of indirection for $\langle(e, D)\rangle$:

$$a_0[D_0] = a_1[D_1] = a_2[D_2] = \ldots = a_n[D_n]$$

($e = a_0$, $D = D_0$ and $a_i$ is a variable if and only if $i \neq n$).
1. \( n = 0 \). Then we have \( e_1(D) = \lambda x. e_1 \) and \( e = a_0 = a_n \) and \( e \) is not a variable. According to Table 2, since \( e \) is not a variable, \( e = \lambda x. e_1 \) for some \( e_1 \). By the {I\!L} rule, \( \langle \lambda x. e_1(D) \rangle \downarrow \langle \lambda x. e_1(D) \rangle \), which is enough to complete the proof for this case.

2. \( n > 0 \). Then \( e = a_0 \) is a variable (say \( y \)) and \( e_1(D) = a_1(D_1) \) where \( D = D_0[y \mapsto (a_1', a_1)]D_1 \) for some \( D_0 \) and \( a_1' \). The given statement \( \mathcal{P}_v(D) \) implies \( \mathcal{P}_v([y \mapsto (a_1', a_1)]D_1) \), and by definition this gives us two cases:

(a) \( a_1' = a_1 \). Since \( a_1 \{D_1 \} \) has a chain of indirection of length \( n - 1 \), by induction we can claim \( \langle a_1, D_1 \rangle \downarrow \langle v, D_1 \rangle \) where \( v \{D_1 \} = v' \) (the value of the original call-by-name evaluation). Since \( a_1 = a_1' \) this can be rewritten as: \( \langle a_1, D_1 \rangle \downarrow \langle v, D_1 \rangle \). According to the {I\!Var} rule we can from this conclude \( \langle v, D_0[y \mapsto (a_1', a_1)]D_1 \rangle \downarrow \langle v, D_0[y \mapsto (u, a_1)]D_1 \rangle \) where \( v \{D_0[y \mapsto (u, a_1)]D_1 \} = v \{D_1 \} \) (validity of configurations) which equals \( v' \).

(b) \( a_1 \{D_1 \} \downarrow a_1' \{D_1 \} \) and \( a_1' \in VAL \). Since \( a_1' \in VAL \), by Lemma 3.5 we can claim \( \langle a_1', D_1 \rangle \downarrow \langle a_1', D_1 \rangle \). Then by {I\!Var}, \( \langle y, D_0[y \mapsto (a_1', a_1)]D_1 \rangle \downarrow \langle a_1', D_0[y \mapsto (a_1', a_1)]D_1 \rangle \). We must now show that \( a_1' \{D_0[y \mapsto (a_1', a_1)]D_1 \} \) is equal to the original value \( v' \). This follows from the fact that we are given \( u \{D_1 \} = e \{D_1 \} \downarrow v' \) and for this case we are given \( a_1 \{D_1 \} \downarrow a_1' \{D_1 \} \). Since the \( \downarrow \) evaluation is a partial function we then have \( a_1' \{D_1 \} = v' \).

**Application** \( e_2(D) = (e_1, e_2) \downarrow v' \) and \( \mathcal{P}_v(D) \) are given. Again we proceed by induction on the length of the chain of indirection.

1. \( n = 0 \). Then we have \( e_1(D) = (e_1, e_2) \) and \( e = a_0 = a_n \) and \( e \) is not a variable. According to Table 2, since \( e \) is not a variable, \( e = (e_1', e_2') \) for some \( e_1' \) and \( e_2' \). By substitution we can then claim \( e_1 = e_1' \{D_1 \} \) and \( e_2 = e_2' \{D_1 \} \). Then the premises to the call-by-name evaluation must be: \( e_1' \{D_2 \} \downarrow \lambda x. a \) and \( a(e_2' \{D_2 \})/x \downarrow v' \). By the induction hypothesis, the first evaluation yields \( \langle e_1', D_2 \rangle \downarrow \langle \lambda x. a, D_2 \rangle \) where \( v'' \{D_2 \} = \lambda x. a \). Since \( v'' \in VAL \) it cannot equal a variable, and thus \( v'' = \lambda x. a' \) for some \( a' \). We can then rewrite the evaluation of \( e_1' \) as: \( \langle e_1', D_2 \rangle \downarrow \langle \lambda x. a', D_2 \rangle \). If we can rewrite the term \( a(e_2' \{D_2 \})/x \) as \( a'(z/x)[z \mapsto (e_2', e_2')] \{D_2 \} \) then the second premise could be rewritten as \( a'(z/x)[z \mapsto (e_2', e_2')] \{D_2 \} \downarrow v' \) and the induction hypothesis applied to yield: \( \langle a'(z/x)[z \mapsto (e_2', e_2')] \{D_2 \} \rangle \downarrow \langle v, D' \rangle \) and \( v' \downarrow \langle v, D' \rangle \). According to {I\!Appl}, we could conclude from the last two intermediate semantics evaluations the desired result: \( \langle (e_1', e_2'), D_2 \rangle \downarrow \langle v, D' \rangle \). Now we show that \( a(e_2' \{D_2 \})/x \) does in fact equal \( a'(z/x)[z \mapsto (e_2', e_2')] \{D_2 \} \). Stated above are \( v'' = \lambda x. a' \) and \( v'' \{D'' \} = \lambda x. a \) which, by substituting for \( v'' \), also equals \( \lambda x. a'[D'' \{x \}] = \lambda x. (a'[D'' \{x \}] \{e_2' \{D_2 \} \}) \). Thus \( a = a'[D'' \{x \}] \) and \( a(e_2' \{D_2 \})/x \) can be rewritten as \( a'[D'' \{e_2' \{D_2 \} \}] \). By Lemma 4.10 and the intermediate semantics evaluation of \( e_1' \) we can claim \( e_1' \{D_2 \} = e_1' \{D'' \} \). Substituting this into the previous term yields \( a'[D'' \{e_2' \{D'' \} \}] \), which can be rewritten as \( a'[x \mapsto (e_2', e_2')] \{D'' \} \) and finally \( a'[z/x][z \mapsto (e_2', e_2')] \{D'' \} \) where \( z \) is a new variable. This establishes our claim and completes this case.

2. \( n > 0 \). This case is exactly the same as in the case for Lambda.
4.6. The equivalence theorem

In this section we complete the proof of Theorem 4.2 by defining \( \equiv \) and using the theorems relating \textsc{Lazy-PCF+Shar} to the intermediate semantics (Theorem 4.6) and the intermediate semantics to the call-by-name semantics (Theorem 4.15).

First we must define the relation \( \equiv \) between \textsc{Lazy-PCF+Shar} configurations and PCF expressions. We can relate \textsc{Lazy-PCF+Shar} configurations to the intermediate semantics configurations using \( \text{Fst} \), and we can relate the intermediate semantics configurations to PCF expressions using the \( \{\} \) translation. We use both of these to define \( \equiv \).

**Definition 4.16.** \( (e, A) \equiv e' \) if and only if there exists an intermediate semantics environment \( D \) such that
1. \( \text{Fst}(D) = A \)
2. \( \mathcal{P}_d(D) \)
3. \( e' = e\{D\} \)

Note that this relation is not a function in either direction. For a given configuration, \( (e, A) \) (or a given expression \( e' \)) there could be many different intermediate semantics environments \( (D) \) that satisfy the conditions and yield different results.

Finally we can prove Theorem 4.2.

**Proof of Theorem 4.2.**

\( \Rightarrow \): We are given \( (e, A) \equiv e' \) and \( (e, A) \Downarrow (v, A') \). By definition of the former we have for some \( D \): \( \text{Fst}(D) = A \), \( \mathcal{P}_d(D) \), and \( e' = e\{D\} \). According to Theorem 4.6 we then have \( \langle e, D \rangle \Downarrow \langle v, D' \rangle \) where \( \text{Fst}(D') = A' \). We can then apply Theorem 4.15 to this to get \( e\{D\} \Downarrow v' \) where \( v' = v\{D'\} \) and \( \mathcal{P}_d(D') \). We can rewrite the evaluation as \( e' \Downarrow v' \). We also can use \( D' \) to support \( (v, A') \equiv v' \) by definition.

\( \Leftarrow \): Here we are given \( (e, A) \equiv e' \) and \( e' \Downarrow v' \). Again by definition of the former we have for some \( D \): \( \text{Fst}(D) = A \), \( \mathcal{P}_d(D) \), and \( e' = e\{D\} \). So we can rewrite the evaluation as \( e\{D\} \Downarrow v' \) and use Theorem 4.15 to get \( \langle e, D \rangle \Downarrow \langle v, D' \rangle \) for some \( D' \) where \( v\{D'\} = v' \) and \( \mathcal{P}_d(D') \). Since \( \text{Fst}(D) = A \) we can also claim by Theorem 4.6 \( (e, A) \Downarrow (v, A') \) for some \( A' \) such that \( \text{Fst}(D') = A' \). And we can again support \( (v, A') \equiv v' \) by definition.

5. Related work

As stated previously, our work focuses on modeling the implementation of call-by-need evaluation, and is not concerned with optimal reductions. Thus in this section we only consider related work that has a similar goal.

Launchbury [13] presents a natural semantics of lazy evaluation that is similar to the semantics of \textsc{Lazy-PCF+Shar}. The syntax of his source language is the lambda calculus extended with (possibly) cyclic let bindings, which are used to represent re-
cursive functions. Lazy evaluation is modeled in two stages, the first of which involves normalizing the lambda terms. The terms are renamed via alpha conversion, so that all bound variables are distinct. This simplifies the semantic rules by eliminating the need for maintaining a list of fresh variables. The terms are further normalized by replacing applications \((e_1 \, e_2)\) (when \(e_2\) is not a variable) with \(\textbf{Let} \ y = e_2 \ \textbf{in} \ (e_1 \, y)\) where \(y\) is a fresh variable. This serves to establish the sharing of arguments statically, as opposed to dynamically, within the rule for application (as in \textsc{lazy-pcf+shar}). The second stage is the dynamic semantic rules, which evaluate term/heap pairs where a heap is an unordered environment with cyclic bindings. The rules are very similar to those of \textsc{lazy-pcf+shar} with the exception of the rule for variables. Since the environment is cyclic and unordered, an expression in the environment may reference any variable in the environment and should be evaluated with respect to the entire environment. In Launchbury's semantics, when a variable \(x\) is evaluated, the expression \(x\) is bound to is evaluated with respect to the entire environment \textit{excluding the binding for} \(x\). This allows the semantics to detect some infinite computation -- if an expression bound to \(x\) has a direct reference to \(x\) (not in the body of a \(\lambda\)-abstraction) then it would normally loop forever. In Launchbury's semantics this case will lead to failure (rather than infinite computation), because the binding for \(x\) will not be in the environment and no rule will apply. Though the rule allows the detection of some infinite computation it leads to the evaluation of configurations that are not valid -- the free variables of the term are not all defined in the heap. This violates the validity property, which is very useful for reasoning with the semantics.

The main difference between Launchbury’s semantics and those of \textsc{lazy-pcf+shar} is the modeling of recursion. With respect to the amount of space used to implement recursion, Launchbury's semantics does a better job of reflecting the implementation than \textsc{lazy-pcf+shar} since it uses cycles as opposed to unfolding. However, in the case where the body of the recursive function is a value, the number of steps in the evaluation will be the same for both \textsc{lazy-pcf+shar} and Launchbury's semantics. Additionally, as described in Section 2.6, the semantics of \textsc{lazy-pcf+shar} can be modified to implement such recursive functions using cycles (without sacrificing the validity of configurations), which use the same amount of space as Launchbury’s semantics during evaluation.

Another work which deserves mention is that of Yoshida [19]. This work describes a semantics for the evaluation of lambda terms which models sharing of arguments and shared evaluation. The main calculus does not specify an evaluation order and so does not directly implement lazy evaluation. However, it can be used to compare different evaluation orders by imposing an order on the existing rules. This is done in the definition of Weak Head reduction, which is analogous to the semantics of \textsc{lazy-pcf+shar}, even to the extent of updating bindings of \(x\) as they are evaluated. Weak head reduction is used later in the paper to prove optimal reductions. The weak head reduction is an implementation of lazy evaluation, but the purpose of the calculus is to study the optimality of reduction sequences in lambda calculus as opposed to providing a model for the static analysis of sharing.
6. Conclusions

In this paper, an operational semantics which captures the sharing of lazy evaluation has been presented. The semantics has been demonstrated to have certain properties such as a value characterization and subject reduction. It has also been shown that this semantics is both sound and adequate with respect to a call-by-name semantics and consequently, a denotational semantics. The semantics also captures the sharing of arguments in the environment, demonstrated by the absence of duplication of arguments, and updating values when evaluated. The semantics is also suitable for reasoning, primarily due to the definition via inferences and axioms which allows for proofs by induction on the height of the proof tree. All of these characteristics indicate that the semantics will be useful for studying the sharing of lazy evaluation and developing provably correct analyses.

References
