Journal of Computer Science 3 (8): 639-645, 2007
ISSN 1549-3636

Operational Semantics for Lazy Evaluation

Mahmoud A. Abou Ghaly, Sameh S. Daoud, Azza A. Taha and Salwa M. Aly
Mathematics Department, Faculty of Science,
Ain Shams University, Cairo, 11566, Egypt

Abstract: An operational semantics for lazy evaluation of a calculus without higher order functions was defined. Although it optimizes many aspects of implementation, e.g. there is a sharing in the recursive computation, there is no α conversion, the heap is automatically reclaimed, and an attempt to evaluate an argument is done at most once. It is still suitable for reasoning about program behavior and proofs of program correctness; this is primarily due to the definition via inferences and axioms which allows for proofs by induction on the height of the proof tree. We also proved the correctness of this operational semantics by showing that it is equivalent with respect to the values calculated to the operational semantics of LAZY-PCF+SHAR due to S. Purushothaman Iyer and Jill Seaman.

Keywords: Higher order functions, α conversion, LAZY-PCF+SHAR

INTRODUCTION

Lazy evaluation delays expression evaluation and avoids multiple evaluation of the same expression. Any implementation of lazy evaluation or call by need has two ingredients[1].

1. Arguments to functions should be evaluated only when their values are needed.
2. Arguments should only be evaluated once, further uses of them within the function body should use the values computed before. This means that there is a sharing of arguments.

The first ingredient is taken form normal order evaluation, and the second ingredient is taken from applicative order evaluation, i.e. Lazy evaluation is a normal order evaluation with sharing of arguments. In Lazy evaluation we pursue normal order evaluation and stop the evaluation when there is no top level redex. The existence of higher order functions in a calculus increase its expressive power, but makes obstacles in defining simple semantics and efficient implementations for lazy evaluation, for this sake, we will waive part of the expressive power of the calculus, by waiving higher order functions from the calculus. This is not a big problem, since higher order functions do not exist in imperative languages. So we will use a calculus with the terms; variable, number, the recursive operator μx.e, and the application of a function to arguments (with the context conditions that a function cannot be the result or the arguments of another function). We will define an operational semantics for lazy evaluation of this calculus. We call this calculus with its operational semantics rules Lmßr, since we will use multi ß reductions of the terms. Also we will compare the simplicity and the efficiency of this operational semantics with the operational semantics for lazy evaluation of the general lambda calculus.

Although Lmßr operational semantics is mainly to model sharing of arguments, it also perform many implementation optimizations, like, The heap is automatically reclaimed, since there is an automatic deletion of out of scope variables from the heap. An attempt to evaluate an argument is done at most once, since once an argument is evaluated the result of evaluation is stored and latter reference to the argument will copy this stored value directly. There is no α conversion (a renaming of variables with a completely fresh variables to avoid name clashes). There is a sharing in the recursive computation. The key reason for all such optimizations is Lmßr uses multi argument reduction for terms.

There have already been some attempts to provide semantics of lazy evaluation for lambda calculus, but all of them require extra overhead to deal with the existence of higher order functions in the calculus. For

Corresponding Author: Mahmoud Ahmed Abou Ghaly, Mathematics Department, Faculty of Science, Ain Shams University, Cairo, 11566, Egypt, Tel +20242219893 Mob +20125243452 Fax: +20248262201

example, The G-machine[8] and the Tim machine[4] perform lambda lifting as a preprocessing step to get rid of the free variables. The operational semantics LAZY-PCF+SHAR due to S. Purushothaman and J. Seaman[6], and the operational semantics due to J. Launchbury[5] are closely related to Lmβr, for simplicity, we rename them as L1 and L2 respectively. In L1 and L2, once a variable is added to the environment it is not deleted from it, the reason is, we cannot determine the end of the scope of a variable (since the result of function application can be a function). So the names of the variables must be unique. Consequently they perform α conversion, L1 do this in its {Appl} rule, while L2 do this during its normalization step. But in Lmβr, the heap is automatically reclaimed, once the function application was end, we will remove the bindings corresponding to the arguments of the function from the environment, this because, in Lmβr function application will always result with a value (not another function). So in Lmβr it is not necessary for variables names to be unique. Consequently α conversion will not happen.

Also, there are two cases in the evaluation of the recursive expression μx.e**.**

Case 1: e requires the value of x before reducing to whnf, this means that e depends directly on x, e.g. x, + x x, 2*x.

Case 2: e reduces to whnf without requiring the value of x, e.g. + 2 5.

The results of the evaluation of L1, L2 and Lmβr for these two cases are;

Case 1:

L1: there is no sharing, and the evaluation will enter an infinite loop.

L2: there is a sharing, and the evaluation will fail.

Lmβr: there is a sharing, and the evaluation will enter an infinite loop.

Case 2:

L1: there is no sharing, and the evaluation will terminate with a whnf value.

L2: there is a sharing, and the evaluation will terminate with a whnf value.

Lmβr: there is a sharing and the evaluation will terminate with a whnf value.

Where, entering an infinite loop results from using infinite data structure which is possible only with lazy evaluation. The evaluation will fail when it requires the value of a certain variable, and this variable does not exist in the environment.

In the rest of this paper, we will, define the multi-argument reduction of terms, the calculus with its operational semantics rules, and we will show the correctness of this semantics. Finally the conclusion and the bibliography.

The Syntax of Lmβr: The syntax of Lmβr is E::= v | n | $(\lambda x_1 .. x_n.e)\ e_1 .. e_n$ | μx.e i.e. a term is either a variable, constant number, application, or meu-abstraction respectively. Where, the term meu-abstraction is used for recursive computation, e.g. μx. CONS 1 x evaluates to an infinite list of 1's. And the application is an application of n expression $e_1 .. e_n$ to the function $(\lambda x_1 .. x_n.e)$ that have the n parameters $x_1 .. x_n$ and body e. We will reduce the application using multi-argument β reduction, which we will explain in the next section.

Multi Argument Reductions: It is a modified form of β-reduction, in which we may effectively perform several β-reductions at once, we explain this using the following example; consider the reduction of the following expression (λx.λy. – y x) 3 4. The basic lambda reducer, proceeds step by step like this (λx.λy. – y x) 3 4 → (λy. – y 3) 4 →- 4 3 → 1

There is no reason however, why we should not perform the λx and the λy reductions simultaneously, thus (λx.λ y. – y x) 3 4 → - 4 3 This multi argument reduction constructs an instance of the body (- y x) whilst substituting 3 for free occurrence of x, and 4 for free occurrence of y. The following observations are crucial:

i. Much is gained by performing the reductions simultaneously. Firstly it builds less intermediate structure in the heap, since the intermediate result of the λx is never constructed. Second no problems are presented by the free occurrence of x in the λy abstraction.

ii. Nothing is lost by performing λx and the λy reductions simultaneously. The result of performing the λx reduction alone is a λy abstraction, and (assuming that we perform normal order reduction until whnf is reached) no further work can be done on the λy abstraction until it is given another argument.

Hence we may as well wait until both arguments are present and then perform both reductions at once.

The Operational Semantics: The semantics we present here is an intermediate-level operational semantics, lying midway between, a straightforward denotational semantics, as that of Josephs[7] and a full operational semantics of the abstract machines[4,8]. It

actually captures sharing within lazy evaluation without requiring extra machinery either of continuations or heaps, code pointers, dumps and the like. The stack (environment) is the only computational structure required. The operational semantics rules are shown in Fig.1

$$\text{mβrVar1}\quad \frac{R\ e \longrightarrow R'\ e'}{R[D,\ x\downarrow(e,0)]\ x \longrightarrow R'[D,\ x\downarrow(e',1)]\ e'}$$

$$\text{mβrVar2}\quad R[D,\ x\downarrow(e,1)]\ x \longrightarrow R[D,\ x\downarrow(e,1)]\ e$$

$$\text{mβrVar3}^1\quad \frac{R[D]\ y \longrightarrow R'[D']\ e'}{R[D,\ x]\ y \longrightarrow R'[D',\ x]\ e'}$$

$$\text{mβrVar3}^2\quad \frac{R\ y \longrightarrow R\ e'}{R[x]\ y \longrightarrow R'\ e'}$$

$$\text{mβrVar4}\quad \frac{D[x\downarrow(\mu x.e,0)]\ e \longrightarrow D'[x]\ e'}{D[x\downarrow(\mu x.e,0)]\ x \longrightarrow D'[x\downarrow(e',1)]\ e'}$$

$$\text{mβrAppl}\quad \frac{D[x_1\downarrow(e_1,0)\ ..\ x_n\downarrow(e_n,0)]\ e \longrightarrow D'\ [x_i]\ e'}{D\ (\lambda x_1\ ..\ x_n.e)\ e_1\ ..\ e_n \longrightarrow D'\ e'}$$

$$\text{mβrRec}\quad \frac{D[x\downarrow(\ \mu x.e,0)]\ x \longrightarrow D'[x\downarrow(e',1)]\ e'}{D\ \mu x.e \longrightarrow D'\ e'}$$

$$\text{mβrError}\quad [x]\ y \longrightarrow \text{var\_have\_no\_value}$$

$$\text{mβrInt}\quad D\ n \longrightarrow D\ n$$

Fig. 1 Operational Semantics of Lmβr

Terms are evaluated with respect to a single environment called the operational semantic environment. The structure of this environment is simply a stack of a list of bindings of variable to tuples, it could be described by the following syntax rules

n ::= 0 | 1

bL ::= v↓(E, n) | bL, v↓(E, n)

D ::= ε | D[bL]

Where E is a term, v is a variable and ε is the empty brackets. That is the environment D is a stack of the binding list bL. Lmβr maps each variable to a pair of expressions, the first component of the pair represents the value of the variable and the second component acts as a marker. Originally when a binding is added to the environment the second component of the pair is set to 0 and the first component is set to the original value of the variable. Once this variable is evaluated then the second component of the pair is changed to 1 and the first component is set to the result of the evaluation.

As an example the bold x in the expression (λxy.+ * ((λx.**x**)(- 4 2)) x x) (+3 7) (* 2 6) is evaluated w.r.t. the environment [x↓(+ 3 7, 0), y↓(* 2 6, 0)] [x↓(- 4 2, 0)] that contains two bindings list, while the first light x is evaluated w.r.t. the environment [x↓(+ 3 7, 0), y↓(* 2 6), 0)] that contains only one binding list, and the seconds light x is evaluated w.r.t. the environment [x↓(10, 1), y↓(* 2 6, 0)], which is the same environment as that of the first light x, but the information that x is evaluated before is taken into consideration.

As shown from this example that the binding list bL is a list of bindings corresponding to the arguments of the function, it is pushed onto the stack before the evaluation of the function body starts, the function body is evaluated with respect to this new stack, and the stack is poped to remove this bindings list at the end of the evaluation of the function body. This corresponds to the very famous garbage collecting rule in block structured languages; last allocated first deallocated.

The coupling of an expression with an environment is referred to as a configuration and is denoted as D e for an expression e and an environment D.

The operational semantics of Lmβr are defined as a natural semantics, which define 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 evaluated 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 induction on the height of the proof justifying the evaluation relation. Natural semantics were explored by Poltkin[2] and latter by Kahn[3].

Rule {mβrInt} is used to evaluate an integer value it returns the same value with the same environment.

Evaluation of the expression μx.e with respect to the environment D, begin by applying rule {mβrRec}, which leads to an evaluation of the variable x with respect to the environment D augmented with the binding list [x↓(μx.e,0)], this applies rule {mβrVar4} which evaluates e with respect to the same environment D[x↓(μx.e,0)]. There are two cases to be considered;

Case 1: e requires the value of x before reducing to whnf, thus a reference to x during the evaluation of e causes rule {mβrVar4} to be applied again, which again evaluates e with respect to the same environment. So

rule {mβrVar4} is continuously applied with the same environment. Thus we enter an infinite loop and there is a sharing, since we use the same environment.

Case 2: e reduces to whnf without requiring the value of x. Assume e reduces to the whnf e', then the result of evaluation is e' and the binding for x is updated to x↓(e',1) to capture sharing. Following reference to x, if any, will apply rule {mβrVar2} and return the value e' directly. Thus in this case there is a sharing and the evaluation terminates with a whnf value.

Note that**;** at the end of the evaluation of the term μx.e we remove the one binding list for x that we added before, from the resulting environment, since this is the end of x scope.

The {mβrAppl} rule evaluates the application (λx_1 .. x_n.e) e_1 .. e_n in an environment D, by evaluating e (the body of the function), in the environment D augmented with the bindings list [x_1↓(e_1,0) .. x_n↓(e_n,0)]. Assume e is evaluated to the expression e' and the environment is updated to D'[x_i] (where D is modified to D' and [x_1↓(e_1,0) .. x_n↓(e_n,0)] is modified to [x_i]). Then the result of evaluation of the original redex is e' paired with environment D', and the bindings list [x_i] is deleted from the resulting environment since x_i's scopes end at this point.

Rule {mβrVar1} makes the largest contribution to the implementation of the call by need strategy. In order to determine the result of evaluating the variable x in the environment R[D, x↓(e,0)] (i.e. the binding corresponding to x is the rightmost binding in the environment). Then the expression e is evaluated with respect to R, say this result with expression e' also R may be updated to R'. Then the result of evaluation of x is e' paired with the environment R'[D, x↓(e',1)]. So the binding for x is updated to the new pair (e', 1) to capture sharing.

So arguments are stored in the environment by the {mβrAppl} and {mβrRec} rules until they are needed, at which point they are evaluated by the {mβrVar1} or {mβrVar4} rules respectively. Thus {mβrVar1} and {mβrVar4} rules correspond to the first evaluation of a variable, (it occurs when the second component of the pair of the binding corresponding to this variable is 0). The result of the evaluation is now stored as the first component of the pair in the resulting environment, to capture sharing, and the second component of the pair (the marker) is set to 1. Then following evaluation of the same variable will use the {mβrVar2} rule, since the marker is now 1. {mβrVar2} rule will return the first component in the pair directly without reevaluation, and no changes are made to the environment.

The {mβrVar3} two rules are used when the variable being looked up in the environment is not the rightmost binding in the environment it searches for the binding of that variable in the tail of the environment, propagating the results it receives. If it does not exist at all then the rule {mβrError} will raise the exception var_have_no_value.

Removing out of scope variables from the environment does not happen in L1 and L2, although it has many advantages. It saves the space occupied by those variables, and saves the time of renaming of those variables. The task of renaming of a variable consist of two steps

i) Generating a new name, where by a new name we mean a name that does not exist in the expression under evaluation and must not be added previously to the environment.

ii) Substituting this new name for the old one in the expression under evaluation.

Correctness of the Lmβr Operational Semantics: In this section, we will 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 are equivalent (with respect to the values calculated) to some accepted or standard semantics, whether operational or denotational. In this paper it will be shown that operational semantics of Lmβr are equivalent to the operational semantics of L1[7]. The correctness of L1 with respect to a standard denotational semantics is already shown in[7]. Thus, once the equivalence of Lmβr and L1 is shown, the correctness of Lmβr with respect to the standard denotational semantics follows automatically.

L1 Semantics: Since the set of terms of Lmßr is a subset of the PCF set of terms, so Fig. 2 contains only subset of L1operational semantics[7]. L1 evaluates configuration denoted by <e, D>. Where D denotes the L1 environment, it is formally described as; D::= [] | [x→e]D i.e. the environment maps each variable to its value.

Example: assume that the primitives operations +, * were added to Lmβr, and they are evaluated in a prefix form. So Fig. 3 contains an evaluation of the expression (λxy.+ * ((λx.x)(- 2)) x x) (+3 7) (* 2 6) using Lmβr and L1 semantics.

$$\{L\}\quad \langle \lambda x.e\ ,\ D\rangle \downarrow \langle \lambda x.e\ ,\ D\rangle$$

$$\{Var1\}\quad \frac{\langle e\ ,\ D\rangle \downarrow \langle e'\ ,\ D'\rangle}{\langle x,\ [x\rightarrow e]D\rangle \downarrow \langle e'\ ,\ [x\rightarrow e']D'\rangle}$$

$$\{Var2\}\quad \frac{\langle e\ ,\ D\rangle \downarrow \langle e'\ ,\ D'\rangle}{\langle y,\ [x\rightarrow e]D\rangle \downarrow \langle e'\ ,\ [x\rightarrow e]D'\rangle}$$

$$\{App\}\quad \frac{\langle e1,D\rangle\downarrow\langle \lambda x.e,D'\rangle \quad \langle e[nx/x],[nx\rightarrow e2]D'\rangle\downarrow\langle e',D''\rangle}{\langle (e1\ e2)\ ,\ D\rangle \downarrow \langle e'\ ,\ D''\rangle}$$

$$\{Rec\}\quad \frac{\langle e[nx/x]\ ,\ [nx\rightarrow \mu x.e]D\rangle \downarrow \langle e'\ ,\ D'\rangle}{\langle \mu x.e,\ D\rangle \downarrow \langle e',\ D'\rangle}$$

Fig. 2 Operational Semantics of LAZY-PCF+SHAR

Equivalence of Lmßr and L1 Semantics: The proof of equivalence is given in theorems 1, 2 it requires a condition on the Lmβr environment; if the second expression of a pair is 1 then the first expression of this pair must be in normal form. This property is referred as E_{nf}, it is defined below.

Definition: E_{nf}

1) $E_{nf}([])$
2) if $E_{nf}(D)$ then $E_{nf}(D[x\downarrow(e,0)])$
3) if $E_{nf}(D)$ and e is in normal form, then $E_{nf}(D[x\downarrow(e,1)])$

The following Lemma shows that this property is propagated by Lmβr.

Lemma:

If $E_{nf}(D)$ and D e $\longrightarrow$ D' e' then $E_{nf}(D')$ and e' $\in$ NF

Proof: the proof is by induction on the height of the inference of D e $\longrightarrow$ D' e'

In order to state the equivalence theorem it is necessary to define a translation from Lmβr environment to L1 environment. This translation will be denoted by * and will map each variable to the first expression in the pair.

Definition: (* translation)

$[]^* = []$
$(D[R, x\downarrow(e,b)])^* = [x\rightarrow e]\ (D[R])^*$ where b = 0,1

Theorem 1 is the first equivalence theorem; it states that if the Lmβr semantics produces a value in an environment having property E_{nf}, then the semantics of LAZY-PCF+SHAR produces the same value.

Theorem 1:

if D e $\longrightarrow$ D' e' and $E_{nf}(D)$ then $\exists$ D" s.t. $\langle e\ ,\ D^*\rangle \downarrow \langle e'\ ,\ D''\rangle$ and $(D')^* \subseteq$ D" where e is a Lmβr term

Proof: the proof is simply by induction on the height of the inference of D e $\longrightarrow$ D' e'
Here, we will show the {mβrVar4} and {mβrRec} cases, the other cases are straightforward inductive cases based on the definitions of E_{nf} and *

{**mβrVar4**}:Given $D[x\downarrow(\mu x.e,0)]x \longrightarrow D'[x\downarrow(e',1)]\ e'$ **(1)** and $E_{nf}(D[x\downarrow(\mu x.e,0)])$, we will search for D" s.t. $\langle x, (D[x\downarrow(\mu x.e,0)])^*\rangle \downarrow \langle e', D''\rangle$ and $(D'[x\downarrow(e',1)])^* \subseteq$ D"
The premise of **(1)** is $D[x\downarrow(\mu x.e, 0)]\ e \longrightarrow D'[x]\ e'$ **(2)** since $E_{nf}(D[x\downarrow(\mu x.e,0)])$ then applying IH to **(2)** gives $\langle e, (D[x\downarrow(\mu x.e,0)])^*\rangle \downarrow \langle e', A\rangle$ **(3)** and $(D'[x])^* \subseteq A$
=> $\langle e, [x\rightarrow \mu x.e]D^*\rangle \downarrow \langle e', A\rangle$ by Definition of *
=> $\langle e[nx/x], [nx\rightarrow \mu x.e]D^*\rangle \downarrow \langle e', A\rangle$ by α conversion
=> $\langle \mu x.e, D^*\rangle \downarrow \langle e', A\rangle$ by {Rec} rule
=> $\langle x, [x\rightarrow \mu x.e]D^*\rangle \downarrow \langle e', [x\rightarrow e']A\rangle$ by {var1} rule
The proof is completed if we take D" = $[x\rightarrow e']A$.

{**mβrRec**}**:** Given D $\mu x.e \longrightarrow$ D' e' **(1)** and $E_{nf}(D)$, we will search for D" s.t. $\langle \mu x.e, D^*\rangle \downarrow \langle e', D''\rangle$ and $(D')^* \subseteq$ D". The premise of **(1)** is $D[x\downarrow(\mu x.e,0)]\ x \longrightarrow D'[x\downarrow(e',1)]\ e'$ **(2)**. Since $E_{nf}(D)$ then $E_{nf}(D[x\downarrow(\mu x.e,0)])$. Applying IH to **(2)** yields $\langle x, (D[x\downarrow(\mu x.e,0)])^*\rangle \downarrow \langle e', A\rangle$ **(3)** and $(D'[x\downarrow(e',1)])^* \subseteq$ A **(4).**
(4) => $[x\rightarrow e'](D')^* \subseteq$ A by Definition of *
=> A = $[x\rightarrow e']$D" for some D" and $(D')^* \subseteq$ D". Then **(3)** becomes $\langle x, (D[x\downarrow(\mu x.e,0)])^*\rangle \downarrow \langle e', [x\rightarrow e']D''\rangle$
=> $\langle x, [x\rightarrow \mu x.e)]D^*\rangle\downarrow\langle e', [x\rightarrow e']D''\rangle$ by Def of *
=> $\langle \mu x.e, D^*\rangle \downarrow \langle e', D''\rangle$ by {var1} rule.

Theorem 2 is the second equivalence theorem it states that if L1 semantics produces a value in an environment D^* with the property $E_{nf}(D)$ then Lmβr semantics produces the same value

Theorem 2:

if $\langle e, D^*\rangle \downarrow \langle e', D'\rangle$ and $E_{nf}(D)$, then $\exists$ D" s.t. D e $\longrightarrow$ D" e' and $(D'')^* \subseteq$ D' where e is a Lmβr term.

Proof: The proof is by induction on the height of the inference of $\langle e, D^*\rangle \downarrow \langle e', D'\rangle$. We will show here the {Rec} case, the other cases are straightforward inductive cases based on the definitions of E_{nf} and *

{Rec} Given $\langle \mu x.e, D^*\rangle \downarrow \langle e', D'\rangle$ **(1)** and $E_{nf}(D)$, we search for D" s.t. D $\mu x.e \longrightarrow$ D" e' and $(D'')^* \subseteq$ D'. The

Lmβr	L1
□ (λxy.+ * ((λx x)(- 4 2)) x x) (+3 7) (* 2 6) [x↓(+3 7,0), y↓(+2 6,0)] + * ((λx.x)(- 4 2)) x x [x↓(+3 7,0), y↓(+2 6,0)] * ((λx x)(- 4 2)) x [x↓(+3 7,0), y↓(+2 6,0)] (λx.x)(- 4 2) [x↓(+3 7,0), y↓(+2 6,0), x↓(- 4 2,0)] x [x↓(+3 7,0), y↓(+2 6,0)] -4 2 [x↓(+3 7,0), y↓(+2 6,0)] 2 [x↓(+3 7,0), y↓(+2 6,0), x↓(2,1)] 2 [x↓(+3 7,0), y↓(+2 6,0)] 2 [x↓(+3 7,0), y↓(+2 6,0)] x [x↓(+3 7,0)] x □ +3 7 □10 [x↓(10,1)] 10 [x↓(10,1), y↓(+2 6,0)]10 [x↓(10,1), y↓(+2 6,0)] * 2 10 [x↓(10,1), y↓(+2 6,0)] 20 [x↓(10,1), y↓(+2 6,0)] + 20 x [x↓(10,1), y↓(+2 6,0)] x [x↓(10,1)] x [x↓(10,1)] 10 [x↓(10,1), y↓(+2 6,0)] 10 [x↓(10,1), y↓(+2 6,0)] + 20 10 [x↓(10,1), y↓(+2 6,0)] 30 □ 30 We begin the evaluation with an empty environment and, it ends with an empty environment.	<(λxy.+ * ((λx x)(- 4 2)) x x) (+3 7) (* 2 6), □> <(λxy.+ * ((λx x)(- 4 2)) x x) (+3 7), □ > <λxy.+ * ((λx x)(- 4 2)) x x, □> <λxy.+ * ((λx x)(- 4 2)) x x, □> < λy. + * ((λx x)(- 4 2)) nx nx, [nx→ +3 7]> (1) <+ *((λx.x)(- 4 2)) nx nx,[ny→* 2 6, nx→ +3 7]> (2) <* ((λx x)(- 4 2)) nx, [ny→* 2 6, nx→ +3 7]> < ((λx x)(- 4 2)), [ny→* 2 6, nx→ +3 7]> <(λx x), [ny→* 2 6, nx→ +3 7]> <(λx x), [ny→* 2 6, nx→ +3 7]> <z, [z→-42, ny→* 2 6, nx→ +3 7]> (3) <-4 2, [ny→* 2 6, nx→ +3 7]> <2, [ny→* 2 6, nx→ +3 7]> <2, [z→2, ny→* 2 6, nx→ +3 7]> <nx, [z→2, ny→* 2 6, nx→ +3 7]> <nx, [ny→* 2 6, nx→ +3 7]> <nx, [nx→ +3 7]> <+3 7, □> <10,□> <10, [nx→ 10]> <10, [ny→* 2 6, nx→10]> <10, [z→2, ny→* 2 6, nx→10]> <* 2 10, [z→2, ny→* 2 6, nx→10] > <20, [z→2, ny→* 2 6, nx→10] > <+ 20 nx, [z→2, ny→* 2 6, nx→10] > <20, [z→2, ny→* 2 6, nx→10] > <20, [z→2, ny→* 2 6, nx→10] > <nx, [z→2, ny→* 2 6, nx→10] > <nx, [ny→* 2 6, nx→10] > <nx, [nx→10] > <10, □> <10, □> <10, [nx→10] > <10, [ny→* 2 6, nx→10] > <10, [z→2, ny→* 2 6, nx→10] > <+ 20 10, [z→2, ny→* 2 6, nx→10] > <30, [z→2, ny→* 2 6, nx→10] > The renaming of variables occurs in steps (1), (2) and (3), and the bindings for z, ny, nx remains in the environment, although we do not need them any more.

Fig. 3 Evaluation of the Expression (λxy.+ * ((λx.x) (- 4 2)) x x) (+3 7) (* 2 6) using Lmβr and L1 Semantics

premise of (1) is <e[nx/x], [nx→μx.e]D^*>↓<e', D'> Def. of * => <e[nx/x], (D[nx↓(μx.e, 0)])*> ↓ <e', D'> (2) since E_{nf}(D) => E_{nf}(D[nx↓(μx.e,0)]), then applying IH to (2) yields D[nx↓(μx.e,0)] e[nx/x] ⟶ A e' (3) and $(A)^* \subseteq$ D' (4).

(3) => D[x↓(μx.e,0)] e ⟶ A e' (5) by α conversion. During this evaluation the binding [x↓(μx.e,0)] may updated, say to [x], then there must exist D" s.t. A = D"[x]. Then (4) becomes $(D"[x])^* \subseteq$ D' => $(D")^* \subseteq$ D' and (5) becomes D[x↓(μx.e,0)] e ⟶ D"[x] e' => [x↓(μx.e,0)]) x ⟶ D"[x↓(e',1)] e' by {mβrVar4} => D μx.e ⟶ D" e' by {mβrRec}

CONCLUSION AND FUTURE WORK

In this paper, an operational semantics for lazy evaluation has been presented. It has been shown that the semantics is correct with respect to LAZY-PCF+SHAR[7] operational semantics. Our semantics captures sharing of the arguments in the environment,

demonstrated by the absence of duplication of arguments evaluation, and updating values when evaluated. Although it optimizes many aspects of implementation, (e.g. there is no α conversion, there is a sharing in the recursive computation, and the heap is automatically reclaimed, since there is an automatic deletion of out of scope variables from the heap), it is still suitable for reasoning about program behaviour and proofs of program correctness, this is primarily due to the definition via inferences and axioms which allows for proofs by induction on the height of the proof tree. The main defect of this semantics is that, it does not allow higher order functions in the calculus. We will arrange to solve this in future work.

ACKNOWLEDGEMENTS

Many thanks to Prof. Dr. Mark Brian Josephs the director of ICR, London South Bank University, for hosting me for a 6 months visit to ICR. This paper is written during this visit**.**

REFERENCES

1. Wadsworth, C. P., 1971. Semantics and Pragmatics of the Lambda Calculus, PhD thesis, Oxford University.
2. Poltkin, G. D., 1981. A Structural Approach to Operational Semantics, Technical Report DIAMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.
3. Kahn, G., 1987. Natural semantics, Rapport de Recherche 601, INRIA, Sophia-Antipolis, France.
4. Fairbairn, J. and Stuart, W. S., 1987. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators, In Proceeding of IFIP conference on Functional Programming Languages and Computer Architecture, Portland, Springer Verlag LNCS 274, pages 34-45,
5. Launchbury, J., 1993. A Natural Semantics for Lazy Evaluation, In Proceedings of Twentieth Symposium on Principles of Programming Languages, Charleston, South Carolina, pp 144-154.
6. Seaman, J. and Purushothaman, S., 1996. Operational Semantics of Sharing In Lazy Evaluation, Science of Computer Programming Elsevier, Amsterdam, vol. 27, n^{o}3, pages 289-322 (19 ref.)
7. Josephs, M., 1989. The Semantics of Lazy Functional Languages, in TCS 68, pages 105-111,
8. Johnsson, T., 1984. Efficient Compilation of Lazy Evaluation, In Proceeding of the ACM SIGPLAN Symposium on Compiler Construction, pages 58-69