Journal of Computer Science 4 (1): 21-29, 2008
ISSN 1549-3636

Semantics of Lazy Evaluation using the Two-Level Grammar

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

Abstract: We have formalized the semantics of lazy evaluation for the lambda calculus using the two-level grammar formalism. The resulting semantics enjoys several properties, e.g., there is a sharing in the recursive computation, there is no α conversion, the heap is automatically reclaimed, an attempt to evaluate an argument is done at most once and there is a sharing in the evaluation of partial application to functions.

Key words: Heap, α conversion, partial application to functions

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[7].

- Arguments to functions should be evaluated only when their values are needed.
- 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 from normal order evaluation while the other is taken from applicative order evaluation, i.e. Lazy evaluation is a normal order evaluation with sharing of arguments. We capture laziness in two stages; the first stage is a static transformation of lambda terms to a normalized forms in which there are no free variables and the second stage is a dynamic semantics for those normalized forms using the two-level grammar formalism, separating these phases means that the dynamic semantics is much simpler than otherwise be the case.

Following Johnsson[2] these normalized expressions are called supercombinators and the transformation from lambda expressions to supercombinators are called 'lambda-lifting' since all the lambda abstraction are lifted to the top level.

There are many implementation techniques of supercombinators, the most efficient of them are the one in the G-machine[2] and the one in the Tim machine[1], both of them compile the supercombinator body into a sequence of instructions which will create an instance of this body. In this research we will not give the full details of an actual implementation of the supercombinators, but we will give only a set of rules which describe the semantics of lazy evaluation for the supercombinators in a general framework without being specific to a certain implementation. These rules could be used for reasoning and program proofs about lazy evaluation, also with little modification they could be adapted to a concrete implementation of lazy evaluation.

We call the calculus we use, with the semantics rules LTLS, since our formalism of the semantics is the two-level grammar formalism[6]. Although LTLS semantics is mainly to model sharing of arguments, it also performs many implementation optimizations, like; There is a sharing in the recursive computation. 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 this 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). And there is a sharing in the evaluation of partial application to functions. The key reason for all such optimizations is that there are no free variables.

There have already been some attempts to provide such semantics. The operational semantics LAZY-PCF+SHAR due to Purushothaman and Seaman[5] and the operational semantics due to Launchbury[4] are closely related to LTLS, (for simplicity, we rename them as S1 and S2, respectively). In S1 and S2, once a variable is added to the environment it is not deleted from it, so the names of the variables must be unique. Consequently they perform α conversion, S1 do this in

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

its {Appl} rule, while S2 do this during its normalization step. But in LTLS, the heap is automatically reclaimed, once the evaluation of function application end with a number, a special rule is applied to remove the bindings corresponding to the arguments of this function from the environment. So in LTLS it is not necessary for variables names to be unique. Consequently α conversion will not happen.

There are two cases in the evaluation of the recursive expression μx.e or equivalently letrec x = e in 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 S1, S2 and LTLS for these two cases are;

Case 1:

- S1: there is no sharing and the evaluation will enter an infinite loop.
- S2: there is a sharing and the evaluation will fail.
- LTLS: there is a sharing and the evaluation will enter an infinite loop.

Case 2:

- S1: there is no sharing and the evaluation will terminate with a whnf value.
- S2: there is a sharing and the evaluation will terminate with a whnf value.
- LTLS: there is a sharing and the evaluation will terminate with a whnf value.

Where, entering an infinite loop results from using an 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.

The rest of this research is organized as follows, after defining the normalization process we will define the two level grammar notations, then the semantics rules are given, finally the conclusion and the bibliography.

THE NORMALIZATION PROCESS

It is the process of transforming λ-terms into supercombinators; usually it is called λ-lifting.

Supercombinators: A supercombinator, $S, of arity n is a lambda expression of the form $\lambda x_1, \lambda x_2 \ldots \lambda x_n.E$ where E is not a lambda abstraction (this ensures that all the leading lambdas are accounted for by $x_1 \ldots x_n$). Such that; $S has no free variables, any lambda abstraction in E is a supercombinator and $n \geq 0$ that is there need be no lambdas at all.

A supercombinator redex consists of the application of a supercombinator to n arguments, where n is its arity. It is reduced by replacing the redex by an instance of the supercombinator body with the arguments substituted for free occurrences of the corresponding formal parameter, which is called multi argument reduction. For example, all the following expressions are supercombinators

3, (+ 2 5), λ.x.x, λ.x.+ x 1, λ.x.+ x x, λ.x. λ.y.- y x, λ.f. (λ.x.+ x x), while the following expressions are not, due to the reasons indicated beside each expression: λ.x.y (y occurs free), λ.y.- y x (x occurs free).

Such a supercombinator is somewhat analogous to a Pascal function which takes several (value) parameters, which does not refer to any global variables and which has no side-effects.

A crucial point in the definition of a supercombinator given above is that a supercombinator reduction only takes place when all the arguments are present. Real programs, of course have many lambda abstractions which are not supercombinators. It is straightforward to transform such programs so that they contain only supercombinators.

Transforming Lambda Abstraction into supercombinators 'lambda-lifting': This process could easily be explained using an example; consider the expression (λ.x. (λ.y.+ y x) x) 4 it contains two lambda abstraction that are not supercombinator. The innermost lambda abstraction λ.y. - y x has a free variable x, so it is not a supercombinator. A simple transformation will make it into one; make each free variable into an extra parameter (this is called, abstracting the free variable). Thus (λy.-y x) is transformed to (λx.λy. + y x) x. For clarity α-conversion is performed on the λx abstraction, it gives (λw.λy.+ y w) x. Now the lambda abstraction (λw.λy.+ y w) is a supercombinator. We give it the name $Y and we write it in the form $Y w y = + y w, substituting this in the original expression gives (λx. $Y x x) 4. Now the λx abstraction is also a supercombinator, we give it the name $X. Thus the original expression is transformed into the following supercombinators

$Y w y = + y w $X x = $Y x x $X 4

We can now execute our program by performing supercombinator reduction as:
$X 4 → $Y 4 4 → + 4 4 → 8

THE TWO-LEVEL GRAMMAR

Two-level grammar is a formalism for defining the syntax and the semantics of programming languages. It is used to formalize the context sensitive as well as context free aspects of programming languages. Usually, in two level grammar we will use the following terminologies[6];

- Protonotion: it is any word of lower case letters; it stands for terminals and nonterminals.
- Metanotion**:** it is any word of upper case letters, for each metanotion there must be a metarule
- Metarule**:** it states which protonotion the metanotion stands for.
- Hyper rule**:** it is a kind of abbreviation or abstraction for a number of production rules that share a common pattern. Production rules can be obtained from hyper rules by substituting the same protonotion for all occurrence of a certain metanotion in the hyper rule.
- Predicate: it is the protonotion that starts with the word where, it is used to formalize the syntax and the semantics conditions, it evaluates to true or false, it is true when it leads to an EMPTY alternative and it is false when it leads to a blind alley.

In two level grammars; Each terminal end with word sy e.g., commasy, lpasy. Semicolon is used to separate alternatives in the same rule. Comma not space is considered as the separator for protonotions in the same rule. We will use two colons in a metarule, while a single one for a hyper rule. The metanotion EMPTY represent the number zero and the Boolean value true. A positive number is a sequence of i's as represented by the metanotion TALLY, while the negative number is a sequence of i's preceded by the word negative.

THE LTLS PROGRAM

The LTLS program consists of a set of supercombinators definitions plus the expression to be evaluated. This expression plus the supercombinators body can do arithmetic calculations in infix form with the rules of precedence applied, can call supercombinators, and can contain let and letrec expressions provided that they do not introduce supercombinators definitions. This means that supercombinators definitions are allowed at the top level only.

THE LTLS SEMANTICS

The semantics we present here is an intermediate-level semantics, lying midway between, a straightforward denotational semantics, as that of Josephs[3] and a full operational semantics of the abstract machines[1,2]. It actually captures sharing within lazy evaluation without requiring extra machinery either of continuations, heaps, code pointers, dumps and the like. The stack (environment) is the only computational structure required. The semantics rules are shown in Fig. 2, while the syntax rules are omitted to save space, but they could be derived from the first and the second branches of the right hand side of the program rule. The rules in Fig. 2 depend on the conventions that: the metanotion that end with ETY corresponds to this metanotion or EMPTY e.g., TALLETY :: TALLY; EMPTY. The metanotion that end with LIST corresponds to a list of this metanotion e.g. TAGLIST :: TAG; TAGLIST commasy TAG. And the metanotion that end with S corresponds to a sequence of this metanotion e.g. DFS: DF; DFS DF.

Terms are evaluated with respect to a single environment. Rules 3 to 12 describe the structure of this environment; it is simply a stack of a list of bindings of variables to expressions. There are four kinds of binding's lists; either starts with let, letrec, args or refs. In general, the bindings in the args bindings lists are defined by the metarule DEF :: IND TAG value EXP. Where the metanotion TAG record the name of the variable and the metanotion EXP record the value of the variable. The bindings in the let and the letrec bindings lists are defined by the metarule MDEF :: DEF ENV. This means that they are the same as that of the args bindings lists, except that, they could be any other binding's lists, usually those results from the evaluation of local definitions to the let (rec) expressions.

The bindings in the refs bindings lists are defined by the rule ZDEF: IND Z_{TALLY} value EXP. We consider Z_{TALLY} as a pointer to the expression EXP. Such bindings lists is used during the evaluation of functions applications to a few arguments or to a more arguments e.g., if F is a function with three arguments then F e_1 e_2 is an application of F to a few arguments, while F e_1 e_2 e_3 e_4 is an application of F to a more arguments. In our formalization; we stipulate that the names of the pointers in the environment are unique, so we will use special names for them as; z_1, z_2, z_3… etc. To guarantee

the uniqueness of these names in the environment we will use a counter to keep track of the index of the last pointer added to the environment, say this counter have the value p then the next available pointer to be used is z_{p+1}. During the reclamation of the environment we will delete unused pointers bindings lists. For example; assume before we start the evaluation of the expression E, the value of the pointer counters is p and during the evaluation of E we have used two pointers, then these two pointes must be z_{p+1}, z_{p+2} and the pointer counter is changed to p+2. Also assume that the evaluation of E ends with a number then z_{p+1}, z_{p+2} will be deleted during the reclamation of the environment and the pointer counter will return back to the value p again.

The metanotion IND in the binding of any binding's list acts as a marker, it has one of two values var or evar. Originally when a binding is added to the environment IND is set to var and EXP is set to the original value of the variable/pointer. Once this variable/pointer is evaluated then IND is changed to evar and the result of the evaluation is stored in the metanotion EXP.

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 env args var letter x value iii plus iiiiiii var letter y value ii mult iiiiii end args var letter x value iiii minus ii end. that contains two bindings list, while the first light x is evaluated w.r.t. the environment env args var letter x value iii plus iiiiiii var letter y value ii mult iiiiii end. that contains only one bindings list and the seconds light x is evaluated w.r.t. the environment env args evar letter x value iiiiiiiii var letter y value ii mult iiiiii end. 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 args bindings list 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 when the evaluation of the function body end with a number. The let and letrec binding's lists will be treated in the same way.

How the Environment is reclaimed in LTLS; During the evaluation of a certain expression we may push binding's lists onto the environment and if the evaluation ends with a number then we no longer need these binding's lists, therefore they must be poped from the environment to free space. The question is; how many ones should be deleted? To be able to answer this question we will use a counter, to count the number of let and letrec binding's lists that are pushed onto the environment during the evaluation. We will call this counter the letr counter. Therefore, expressions are evaluated with respect to an environment plus the pointer and the letr counters. As an example; consider the following set of supercombinators definitions

$F x y z = + ($G y) ($H 2y z)
$G x = 2x
$H x y = x/y

The evaluation of the expression **$F 3, 2*1, 4** starts with an empty environment and the two counters are zeros, it is shown step by step in Fig 1. For clarity we will use the following notations <env, p, c> where p is the pointer counter and c is the letr counter, we call this triple the configuration.

<env, 0, 0> $F 3, 2*1, 4
→<env args var letter x value iii var letter y value ii mult i var letter z value iiii end, 0, 1> +($G y) ($H 2y z)
→<env args var letter x value iii var letter y value ii mult i var letter z value iiii end, 0, 0> $G y
→<env args var letter x value iii var letter y value ii mult i var letter z value iiii end args var letter x value y end, 0,1> ii mult x
→<env args var letter x value iii evar letter y value ii var letter z value iiii end args evar letter x value ii end , 0, 1> iiii
→<env args var letter x value iii evar letter y value ii var letter z value iiii end , 0, 0> iiii
→<env args var letter x value iii evar letter y value ii var letter z value iiii end , 0, 0> $H 2y z
→<env args var letter x value iii evar letter y value ii var letter z value iiii, args var letter x value ii mult y var letter y value z end, 0, 1> x div y
→<env args var letter x value iii evar letter y value ii evar letter z value iiii end, args evar letter x value iiii evar letter value iiii , 0, 1> i
→<env let var letter x value iii evar letter y value ii evar letter z value iiii , 0, 0> i
→<env let var letter x value iii evar letter y value ii evar letter z value iiii, 0, 1> iiii plus i
→<env let var letter x value iii evar letter y value ii evar letter z value iiii, 0, 1> iiiii
→<env, 0, 0> iiiii

Fig 1: Example evaluation

Note that:

- During the evaluation of the expression $G y, one bindings list was added to the environment and was

deleted at the end of this evaluation, since this evaluation ends with a number. The same are done for the expression $H 2y z.

- $G y is evaluated with the letr counter equal zero, although there is an args bindings list in the environment, this because we don't want this bindings list to be deleted when the evaluation of $G y end with a number, since we need this bindings list in the evaluation of the expression $H 2y z. If we use another copy of this bindings list in the evaluation of the $H 2y z, then we may lose sharing (e.g. the information that the binding of the variable y is updated during the evaluation of $G y must be taken into consideration during the evaluation of $H 2y z).
- Finally the environment is empty, since the evaluation of the whole expression end with a number.

The Semantics Rules: The Semantics rules are listed in Fig 2, these rules depend on the fact that the name of each supercombinator must start with the character $ and no two supercombinators have the same name (a property we guarantee from lambda lifting). The evaluation process may results with the semantics values $TAG ZLIST, $TAG ZLIST EXLIST, Z_{TALLY}, adding these values to the syntax expression EX will result with the semantics expression EXP. Similarly AC, ACC are the syntax and semantics accumulator and DF, DEF are the syntax and semantics binding respectively. The rules in Fig 2 shows that EX, AC, DF is a subset of EXP, ACC, DEF respectively, then in this context we could generally concentrate our interest on the semantics ones. ACC is used to store the temporary and the final result of the evaluation. It has two values; either acc which represents the empty store or acc e which represents a store that contains the expression e.

We also need a metanotion STATE which corresponds to 'states' existing at various stages during the computation. It is defined by the metarule STATE :: state num1 TALLYETY1 num2 TALLYETY2 ENV ACC. **Where** TALLYETY1, TALLYETY2 are the pointer and letr counters respectively. The technique we use depends on; the meaning of a program should be described essentially in terms of the correspondence it defines between its initial and final states. The topmost hyper rule is program : FUNCSETY of supercombinators gives $TAGLISTETY, vars snams $TAGLISTETY expression of EX , env, acc, ENV, AC, where FUNCSETY EX transform state num1 num2 env acc into state num1 TALLYETY1 num2 TALLYETY2 ENV AC, where reclaim num1 TALLYETY1 num2 TALLYETY2 ENV gives num1 num2 env.

Where: EX is the actual expression to be evaluated, env is the initial environment, ENV is final environment, acc is the initial store, AC is final store, TALLYETY1 is the pointer counter, TALLYETY2 is the letr counter and FUNCSETY records the set of supercombinators definitions; it is calculated once and before the evaluation of the expression EX starts. The first predicate ensures that AC store the result of evaluating the expression EX in the initial state and the second predicate ensures that the reclamation process will result with an empty environment.

Rules 1 to 26 for the basic definitions, while, the rest of the rules define the predicates part. Rules 31 to 33 correspond to the first evaluation of a variable; they are used when the indicator (IND) in the recent binding of this variable in the environment is var. Then the metanotion EXP is updated with the result of its evaluation to capture sharing and the indicator IND is set to evar. Following evaluations of the same variable will use the rule 34 or rule 35 (according to the type of bindings list), since the indicator is now evar. They will return the value EXP directly without reevaluation and no changes are made to the environment. This shows that in LTLS an attempt to evaluate an argument is done at most once. Rule 36 is used when a variable is applied to arguments e.g. x e_1 e_2 e_3, in this case the result of evaluating the expression which this variable is bound must be a partially applied function and the result of evaluation is an application of this partially applied function to these arguments. Rule 37 to 40 evaluate the predicate when the expression is a pointer. Rule 37 and 38 will find a var indicator of this pointer, so the expression bound to this pointer is evaluated and the result is stored to capture sharing. While Rule 39 and 40 will find an evar indicator, so no evaluation happens in this case and the value bound to this pointer is returned directly. Rule 50 evaluate the predicate when the expression to be evaluated is a number, in this case the result of evaluation is this number itself and the environment is reclaimed. Rule 51 to 57 evaluate the predicate which reclaims the environment, it do this, by repeatedly popping the environment to remove TALLYETY2 let, letrec and/or args binding's lists and all the pointers binding's lists that meet us during this process, where TALLYETY2 is the letr counter. Rules 63 to 77 are concerned with evaluating the predicate when the expression is a calling to a supercombinator. Rule 65 shows three cases to be considered; the first case is a calling of a supercombinator with the exact number of arguments, then an args bindings list of the arguments of the supercombinator is pushed onto the environment, the letr counter is increased by 1 and the

body of the supercombinator is executed with respect to this new configuration. The second case is a calling of a supercombinator with a few arguments, in this case a pointers bindings list of the arguments (other than pointers arguments) is pushed onto the environment, the pointer counter is increased by the number of pointers used in this pointers bindings list to record this and the result of the evaluation is a calling of the supercombinator with these pointers instead of the arguments. This result will not be evaluated according to rule 63, it remains in this form until it is given the rest of the arguments in other stages of the computations and we call it suspension. Other expressions that share this calling of the supercombinator with this few arguments, now share this suspension, so there is sharing in the evaluation of partial applications to functions. The third case is a calling for the supercombinator with a more arguments, in this case a pointers bindings list is pushed onto the environment for these more arguments and an args bindings list is pushed onto the environment for the other arguments, the pointer and the letr counters are increased to record these information, the supercombinator body is executed with respect to this new configuration, say it gives another expression G (must be a partially evaluated supercombinator) and the final result is a calling of G with the pointers in the last added pointers bindings list. Rule 64 evaluates the predicate when the expression is a calling to a supercombinator without parameters; in this case the body of this supercombinator is executed without any bindings added to the environment.

Note that: We guarantee the uniqueness of the supercombinators names, the uniqueness of the parameters names of the same supercombinator and the exclusion of the free variables, since these conditions must be syntactically checked. The rest of the rules are the rules for the arithmetic calculations, we have listed the rules for addition and multiplication, the rules for subtraction and division could be treated similarly.

1) program: FUNCSETY of supercombinators gives $TAGLISTETY, vars snams $TAGLISTETY expression of EX , env, acc, ENV, AC, where FUNCSETY EX transform state num1 num2 env acc into state num1 TALLYETY1 num2 TALLYETY2 ENV AC, where reclaim num1 TALLYETY1 num2 TALLYETY2 ENV gives num1 num2 env.

2) STATE :: state num1 TALLYETY1 num2 TALLYETY2 ENV ACC.

3) IND :: var; evar.

4) AC :: acc; acc EX.

5) ACC :: AC; acc Z_{TALLY}; acc $TAG ZLIST EXLISTETY.

6) FUNC :: fun $TAG params TAGLISTETY body EX end.

7) DF :: IND TAG value EX.

8) DEF :: DF ; IND TAG value Z_{TALLY} ; IND TAG value $TAG ZLIST EXLISTETY.

9) MDEF :: DEF; ENV.

10) ZDEF :: IND Z_{TALLY} value EXP.

11) LR :: let; letrec;

12) ENV :: env; ENV LR MDEFS end; ENV refs ZDEFS end; ENV args DEFS end.

13) TALLY :: i; TALLY i.

14) NUMBER :: TALLETY; negative TALLY.

15) ALPHA :: a;b;c;…;z.

16) TAG :: ALPHA; TAG ALPHA.

17) WEAKOP :: plus; minus.

18) SRTGOP :: times; over.

19) EXP :: EX ; Z_{TALLY}; $TAG ZLIST EXLISTETY.

20) EX :: TERM; EX WEAKOP TERM.

21) TERM :: ELEMENT; TERM STRGOP ELEMENT.

22) ELEMENT :: TAG EXLISTETY; NUMBER; lpasy EX rpasy; letsy DFS insy EX endsy; letrecsy DFS insym EX endsy; $TAG EXLISTETY.

23) ST :: num1 TALLYETY1 num2 TALLYETY2 ENV.

24) LARGS :: LR MDEFS; args DEFS;

25) XTZ :: TAG ; EX ; Z_{TALLY}.

26) EMPTY commasy ZLIST : ZLIST

27) where FUNCSETY EXP transform STATE1 into STATE2 : where FUNCSETY EX transform STATE1 into STATE2 ; where FUNCSETY Z_{TALLY} transform STATE1 into STATE2; where FUNCSETY $TAG ZLISTETY EXLISTETY transform STATE1 into STATE2.

28) where FUNCSETY EX transform STATE1 into STATE2 : where FUNCSETY TERM transform STATE1 into STATE2; where FUNCSETY EX WEAKOP TERM transform STATE1 into STATE2.

29) where FUNCSETY TERM transform STATE1 into STATE2 : where FUNCSETY ELEMENT transform STATE1 into STATE2; where FUNCSETY TERM STRGOP ELEMENT transform STATE1 into STATE2.

30) where FUNCSETY ELEMENT transform STATE1 into STATE2: where FUNCSETY TAG EXLISTETY transform STATE1 into STATE2; where NUMBER transform STATE1 into STATE2; where FUNCSETY lpasy EX rtpasy transform STATE1 into STATE2; where FUNCSETY letsy DFS insy EX endsy transform STATE1 into STATE2; where FUNCSETY letrecsy DFS insy EX endsy transform STATE1 into STATE2; where FUNCSETY $TAG EXLISTETY transform STATE1 into STATE2.

31) where FUNCSETY TAG transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 let MDEFSETY1 var TAG value EX MDEFSETY2 end ENVETY2 acc into state num1 TALLYETY11 num2 TALLYETY21 ENVETY3 let MDEFSETY3 ENVETY4 var TAG value EXP MDEFSETY2 end ENVETY2 acc EXP : where FUNCSETY EX transform state num1 num2 ENVETY1 let MDEFSETY1 end acc into state num1 TALLYETY12 num2 TALLYETY22 ENVETY5 let MDEFSETY4 ENVETY6 end acc EXP, where reclaim num1 TALLYETY12 num2 TALLYETY22 ENVETY5 let

MDEFSETY4 ENVETY6 end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY3 let MDEFSETY3 ENVETY4 end , where TAG not in ENVETY2 , where TAG not in MDEFSETY2.

32) where FUNCSETY TAG transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 letrec MDEFSETY1 var TAG value EX MDEFSETY2 end ENVETY2 acc into state num1 TALLYETY11 num2 TALLYETY21 ENVETY3 letrec MDEFSETY3 ENVETY4 var TAG value EXP MDEFSETY4 end ENVETY2 acc EXP : where FUNCSETY EX transform state num1 num2 ENVETY1 letrec MDEFSETY1 var TAG value EX MDEFSETY2 end acc into state num1 TALLYETY12 num2 TALLYETY22 ENVETY5 letrec MDEFSETY5 ENVETY6 var TAG value EXP MDEFSETY6 end acc EXP, where reclaim num1 TALLYETY12 num2 TALLYETY22 ENVETY5 letrec MDEFSETY5 ENVETY6 var TAG value EXP MDEFSETY6 end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY3 letrec MDEFSETY3 ENVETY4 var TAG value EXP MDEFSETY4 end, where TAG not in ENVETY2, where TAG not in MDEFSETY2.

33) where FUNCSETY TAG transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 args DEFSETY1 var TAG value EX DEFSETY2 ENVETY2 acc into state num1 TALLYETY11 num2 TALLYETY21 ENVETY3 args DEFSETY1 var TAG value EXP DEFSETY2 ENVETY2 acc EXP: where FUNCSETY EX transform state num1 num2 ENVETY1 acc into state num1 TALLYETY12 num2 TALLYETY22 ENVETY4 acc EXP, where reclaim num1 TALLYETY12 num2 TALLYETY22 ENVETY4 gives num1 TALLYETY11 num2 TALLYETY21 ENVETY3, where TAG not in ENVETY2.

34) where FUNCSETY TAG transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 LR MDEFSETY1 evar TAG value EXP MDEFSETY2 end ENVETY2 acc into state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 LR MDEFSETY1 evar TAG value EXP MDEFSETY2 end ENVETY2 acc EXP : where TAG not in ENVETY2, where TAG not in MDEFSETY2.

35) where FUNCSETY TAG transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 args DEFSETY1 evar TAG value EXP DEFSETY2 end ENVETY2 acc into state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 args DEFSETY1 evar TAG value EXP DEFSETY2 end ENVETY2 acc EXP : where TAG not in ENVETY2.

36) where FUNCSETY TAG EXLIST transform state num1 TALLETY1 num2 TALLETY2 ENV1 acc into state num1 TALLETY11 num2 TALLETY21 ENV2 acc EXP: where FUNCSETY TAG transform state num1 TALLETY1 num2 TALLETY2 ENV1 acc into state num1 TALLETY12 num2 TALLETY22 ENV3 acc $TAG ZLISTETY, where FUNCSETY $TAG ZLISTETY EXLIST transform state num1 TALLETY12 num2 TALLETY22 ENV3 acc into state num1 TALLETY11 num2 TALLETY21 ENV2 acc EXP.

37) where FUNCSETY Z_{TALLY} transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 refs ZDEFSETY1 var Z_{TALLY} value EX ZDEFSETY2 ENVETY2 acc into state num1 TALLYETY11 num2 TALLYETY21 ENVETY3 refs ZDEFSETY1 var Z_{TALLY} value EXP ZDEFSETY2 ENVETY2 acc EXP: where FUNCSETY EX transform state num1 num2 ENVETY1 acc into state num1 TALLYETY12 num2 TALLYETY22 ENVETY4 acc EXP, where reclaim num1 TALLYETY12 num2 TALLYETY22 ENVETY4 gives num1 TALLYETY11 num2 TALLYETY21 ENVETY3.

38) where FUNCSETY Z_{TALLY} transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 LR MDEFSETY1 refs ZDEFSETY1 var Z_{TALLY} value EX ZDEFSETY2 end MDEFSETY2 end ENVETY2 acc into state num1 TALLYETY11 num2 TALLYETY21 ENVETY3 LR MDEFSETY3 refs ZDEFSETY1 var Z_{TALLY} value EXP ZDEFSETY2 end MDEFSETY2 end ENVETY2 acc EXP: where FUNCSETY EX transform state num1 num2 ENVETY1 LR MDEFSETY1 end acc into state num1 TALLYETY12 num2 TALLYETY22 ENVETY4 LR MDEFSETY4 end acc EXP, where reclaim num1 TALLYETY12 num2 TALLYETY22 ENVETY4 LR MDEFSETY4 end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY3 LR MDEFSETY3.

39) where FUNCSETY Z_{TALLY} transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 refs ZDEFSETY1 evar Z_{TALLY} value EXP ZDEFSETY2 end ENVETY2 acc into state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 refs ZDEFSETY1 evar Z_{TALLY} value EXP ZDEFSETY2 end ENVETY2 acc EXP: EMPTY.

40) where FUNCSETY Z_{TALLY} transform state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 LR MDEFSETY1 refs ZDEFSETY1 evar Z_{TALLY} value EXP ZDEFSETY2 end MDEFSETY2 end ENVETY2 acc into state num1 TALLYETY1 num2 TALLYETY2 ENVETY1 LR MDEFSETY1 refs ZDEFSETY1 evar Z_{TALLY} value EXP ZDEFSETY2 end MDEFSETY2 end ENVETY2 acc EXP : EMPTY.

41) where TAG not in ENVETY LR MDEFS end :: where TAG not in ENVETY, where TAG not in MDEFS.

42) where TAG not in ENVETY args DEFS end :: where TAG not in ENVETY, where TAG not in DEFS.

43) where TAG not in MDEFS DEF : where TAG not in MDEFS, where TAG not in DEF.

44) where TAG not in MDEFS ENV : where TAG not in MDEFS.

45) where TAG1 not in var TAG2 value EXP : where TAG1 is not TAG2.

46) where TAGETY1 ALPHA1 is not TAGETY2 ALPHA2 : where TAGETY1 is not TAGETY2; where ALPHA1 precedes ALPHA2 in abcdefghijklmnopqrstuvwxyz; where ALPHA2 precedes ALPHA1 in abcdefghijklmnopqrstuvwxyz.

47) where TAG is not EMPTY : EMPTY.

48) where EMPTY is not TAG: EMPTY.

49) where ALPHA1 precedes APLHA2 in TAGETY1 ALPHA1 TAGETY2 ALPHA2 TAGETY3: EMPTY.

50) where NUMBER transform state ST1 acc into state ST2 acc NUMBER: where reclaim ST1 gives ST2.

51) where reclaim num1 TALLYETY1 i num2 TALLYETY2 ENVETY1 refs ZDEFSETY ZDEF end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2 : where reclaim num1 TALLYETY1 num2 TALLYETY2 ENVETY1 refs ZDEFSETY end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2.

52) where reclaim num1 TALLYETY1 num2 TALLYETY2 ENVETY1 refs end gives num1 TALLYETY11 num2

TALLYETY21 ENVETY2 : where reclaim num1 TALLYETY1 num2 TALLYETY2 ENVETY1 gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2.

53) where reclaim num1 TALLYETY1 num2 TALLY2 ENVETY1 LR MDEFSETY DEF end gives num1 TALLYETY11 num2 TALLY21 ENVETY2 : where reclaim num1 TALLYETY1 num2 TALLY2 ENVETY1 LR MDEFSETY end gives num1 TALLYETY11 num2 TALLY21 ENVETY2.

54) where reclaim num1 TALLYETY1 num2 TALLY2 ENVETY1 LR MDEFSETY ENV end gives num1 TALLYETY11 num2 TALLY21 ENVETY2: where reclaim num1 TALLYETY1 num2 TALLY2 ENVETY1 LR MDEFSETY end ENV gives num1 TALLYETY11 num2 TALLY21 ENVETY2.

55) where reclaim num1 TALLYETY1 num2 TALLYETY2 i ENVETY1 LR end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2: where reclaim num1 TALLYETY1 num2 TALLYETY2 ENVETY1 gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2.

56) where reclaim num1 TALLYETY1 num2 TALLYETY2 i ENVETY1 args DEFS end gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2: where reclaim num1 TALLYETY1 num2 TALLYETY2 ENVETY1 gives num1 TALLYETY11 num2 TALLYETY21 ENVETY2.

57) where reclaim num1 TALLYETY1 num2 ENVETY LARGS gives num1 TALLYETY1 num2 ENVETY LARGS : EMPTY.

58) where FUNCSETY lpasy EX rtpasy transform STATE1 into STATE2 : where FUNCSETY EX transform STATE1 into STATE2.

59) where FUNCSETY letsy DFS insy EX endsy transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC : where FUNCSETY EX transform state num1 TALLYETY1 num2 TALLYETY2 i ENV1 let DFS end acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC.

60) where FUNCSETY letrecsy DFS insy EX endsy transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLETY11 num2 TALLETY21 ENV2 ACC : where EX transform state num1 TALLYETY1 num2 TALLYETY2 i ENV1 letrec DFS end acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC.

61) where FUNCSETY EX WEAKOP TERM transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLYETY13 num2 TALLYETY21 ENV2 acc NUMBER: where FUNCSETY EX transform state num1 TALLYETY1 num2 ENV1 acc into state num1 TALLYETY11 num2 ENV3 acc NUMBER1, where FUNCSETY TERM transform state num1 TALLYETY11 num2 ENV3 acc into state num1 TALLYETY12 num2 ENV4 acc NUMBER2, where NUMBER1 WEAKOP NUMBER2 equal NUMBER, where reclaim num1 TALLYETY12 num2 TALLYETY2 ENV4 gives num1 TALLYETY13 num2 TALLYETY21 ENV2.

62) where FUNCSETY TERM STRGOP ELEMENT transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLYETY13 num2 TALLYETY21 ENV2 acc NUMBER: where FUNCSETY TERM transform state num1 TALLYETY1 num2 ENV1 acc into state num1 TALLYETY11 num2 ENV3 acc NUMBER1, where FUNCSETY ELMENT transform state num1 TALLYETY11 num2 ENV3 acc into state num1 TALLYETY12 num2 ENV4 acc NUMBER2, where NUMBER1 STRGOP NUMBER2 equal NUMBER, where reclaim num1 TALLYETY12 num2 TALLYETY2 ENV4 gives num1 TALLYETY13 num2 TALLYETY21 ENV2.

63) where FUNCSETY $TAG ZLIST transform state num1 TALLYETY1 num2 TALLYETY2 ENV acc into state num1 TALLYETY1 num2 TALLYETY2 ENV acc $TAG ZLIST: EMPTY.

64) where FUNCSETY1 fun $TAG params body EX end FUNCSETY2 $TAG transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC : where FUNCSETY1 fun $TAG params body EX end EX transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC.

65) where FUNCSETY1 fun $TAG params TAGLIST body EX end FUNCSETY2 $TAG ZLISTETY EXLIST transform state num1 TALLYETY1 num2 TALLYETY2 ENV1 acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC : where numberof ZLISTETY is TALLETY, where numberof EXLIST is TALLY1, where numberof TAGLIST is TALLY2, where TALLETY plus TALLY1 equal TALLY2, where FUNCSETY1 fun $TAG params TAGLIST body EX end FUNCSETY2 EX transform state num1 TALLYETY1 num2 TALLYETY2 i ENV1 args DEFS DFS end acc into state num1 TALLYETY11 num2 TALLYETY21 ENV2 ACC, where TAGLIST bind ZLISTETY EXLIST gives DEFS DFS**;** where numberof ZLISTETY is TALLETY, where numberof EXLIST is TALLY1, where numberof TAGLIST is TALLY2, where TALLETY plus TALLY1 equal TALLY3, where TALLY3 lessthan TALLY2, where FUNCSETY1 fun $TAG params TAGLIST body EX end FUNCSETY2 $TAG ZLISTETY commasy ZLIST1 transform state num1 TALLYETY1TALLY4 num2 TALLYETY2 ENV1 refs ZDEFS end acc into state TALLYETY11 TALLYETY21 ENV2 ACC, where makezbindingfrom TALLYETY1 i of EXLIST gives ZDEFS anduse ZLIST1 withlength TALLY4**;** where numberof ZLISTETY is TALLETY, where numberof EXLIST is TALLY1, where numberof TAGLIST is TALLY2, where TALLETY plus TALLY1 equal TALLY3, where TALLY2 lessthan TALLY3, where FUNCSETY1 fun $TAG params TAGLIST body EX end FUNCSETY2 EX transform state num1 TALLYETY1TALLY4 num2 TALLYETY2 i ENV1 refs ZDEFS end args DEFS DFS end acc into state TALLYETY12 TALLYETY22 ENV3 acc EXP, where FUNCSETY1 fun $TAG params TAGLIST body EX end FUNCSETY2 EXP ZLIST transform state TALLYETY12 TALLYETY22 ENV3 acc into state TALLYETY11 TALLYETY21 ENV2 ACC, where TAGLIST bindr ZLISTETY EXLIST gives DEFS DFS andtherestare EXLIST1, where makezbindingfrom TALLYETY1 i of EXLIST1 gives ZDEFS anduse ZLIST

withlength TALLY4.

66) where EMPTY bind EMPTY gives EMPTY : EMPTY.

67) where TAG commasy TAGLIST bind Z_{TALLY} commasy ZLIST EXLIST gives var TAG value Z_{TALLY} DEFS DFS: where TAGLIST bind ZLIST EXLIST gives DEFS DFS.

68) where TAG commasy TAGLIST bind EX commasy EXLIST gives var TAG value EX DFS : where TAGLIST bind EXLIST gives DFS.

69) where makezbindingfrom TALLY1 of EX commasy EXLIST gives var Z_{TALLY1} value EX ZDEFS anduse Z_{TALLY1} commasy ZLIST withlength TALLY2 i : where makezbindingfrom TALLY1 i of EXLIST gives ZDEFS anduse ZLIST withlength TALLY2**.**

70) where makezbindingfrom TALLY of EMPTY gives EMPTY anduse EMPTY withlength EMPTY: EMPTY.

71) where EMPTY bindr EXLIST gives EMPTY andtherestare EXLIST : EMPTY.

72) where TAG commasy TAGLIST bindr Z_{TALLY} commasy ZLIST EXLIST1 gives var TAG value Z_{TALLY} DEFS DFS andtherestare EXLIST2 : where TAGLIST bindr ZLIST EXLIST1 gives DEFS DFS andtherestare EXLIST2.

73) where TAG commasy TAGLIST bindr EX commasy EXLIST1 gives var TAG value EX DFS andtherestare EXLIST2 : where TAGLIST bindr EXLIST1 gives DFS andtherestare EXLIST2.

74) where numberof XTZLIST commasy XTZ is TALLY i : numberof XTZLIST is TALLY.

75) where numberof EMPTY is EMPTY : EMPTY.

76) where numberof XTZ is i : EMPTY.

77) where TALLETY lessthan TALLYETY TALLY : EMPTY.

78) where TALLYETY1 plus TALLYETY2 equals TALLYETY1 TALLYETY2.

79) where TALLYETY plus negative TALLY equals NUMBER : where TALLYETY minus TALLY equals NUMBER.

80) where negative TALLY plus TALLYETY equals NUMBER : where TALLYETY minus TALLY equals NUMBER.

81) where negative TALLY1 plus negative TALLY2 equals negative TALLY3 : where TALLY1 plus TALLY2 equals TALLY3.

82) where NUMBER1 times NUMBER2 equals NUMBER3 : where NUMBER2 times NUMBER1 equals NUMBER3.

83) where EMPTY times NUMBER equals EMPTY : EMPTY.

84) where i times NUMBER equals NUMBER : EMPTY.

85) where TALLY1 i times TALLY2 equals TALLY2 TALLY3: where TALLY1 times TALLY2 equals TALLY3.

86) where negative TALLY1 times TALLY2 equals negative TALLY3 : where TALLY1 times TALLY2 equals TALLY3.

87) where negative TALLY1 times negative TALLY2 equals TALLY3 : where TALLY1 times TALLY2 equals TALLY3.

Fig 2: The Semantics Rules

CONCLUSION

Using the two level grammar formalism we have formalized the semantics of lazy evaluation for the lambda calculus. Two level grammars are very expressive but it is commonly used for defining the syntax and semantics for imperative programming languages. This research shows that they could also be used with functional languages. 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 this semantics 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 behavior and proofs of program correctness, this is primarily due to the definition via a set of predicates which allows for proofs by evaluating the predicates. The main defect of this semantics is that, it is little bit lengthy.

ACKNOWLEDGMENTS

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. Most of this research is written during this visit**.**

REFERENCES

1. Fairbairn, J. and W.S. Stuart, 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, pp: 34-45.
2. Johnsson, T., 1984. Efficient Compilation of Lazy Evaluation, In: Proceeding of the ACM SIGPLAN Symposium on Compiler Construction, pp: 58-69.
3. Josephs, M., 1989. The Semantics of Lazy Functional Languages, in TCS 68, pp: 105-111,
4. Launchbury, J., 1993. A Natural Semantics for Lazy Evaluation, In: Proceedings of 20th Symposium on Principles of Programming Languages, Charleston, South Carolina, pp: 144-154.
5. Seaman, J. and S. Purushothaman, 1996. Operational Semantics of Sharing In Lazy Evaluation, Science of Computer Programming Elsevier, Amsterdam, 27 (3): 289-322.
6. Slonneger, K. and L.B. Kurtz, 1995. Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach, Addison-Wesley Publishing Company.
7. Wadsworth, C.P., 1971. Semantics and pragmatics of the lambda calculus. Ph.D thesis, Oxford University.