AN ALGORITHM FOR TRANSLATING LISP PROGRAMS INTO REDUCTION LANGUAGE PROGRAMS

Alexis Koster

Abstract : An algorithm for translating LISP programs into reduction language programs is presented. It allows us to compare some features of the two languages: presence/absence of environment, free variables, evaluation rules, and parallelism. It may lead to efficient execution of LISP programs on a reduction machine.

COMP-KADG, P.O.Box 28053, San Diego, Ca 92128

1 INTRODUCTION

Reduction languages (Backus 1972, 1973, 1978) are applicative languages whose syntax and semantics are simple and rigorously defined. Some of their main features are the use of functional operations, the complete absence of imperative features (Landin 1966), and the possibility of unbounded parallelism. In this paper, an algorithm for translating LISP programs into reduction language programs is presented (1). This algorithm is interesting from two points of view:

i) it sheds light on the following features of the two languages: presence/absence of environment, free variables, evaluation rules, and parallelism.

ii) it may lead to efficient execution of LISP programs. Mago (Mago 1979) has described a new type of computer, called a reduction machine, to execute reduction languages. One of the main properties of this machine is the possibility of unbounded parallelism (limited only by the size of the machine). It has been shown that this machine executes reduction language programs efficiently, owing mainly to the use of parallelism (Koster 1977). An implementation of the algorithm given in this article would allow the execution of many LISP programs on such a machine.

2 AN INFORMAL PRESENTATION OF A REDUCTION LANGUAGE

According to Backus's terminology (Backus 1973), the language presented here (henceforth called RL) is a λ-M-Red language (extended to sequences). Besides a few syntactic markers, RL has only three kinds of atomic symbols, variables (e.g. v1), objects (e.g. 1, +, TAIL, A), and the undefined symbol #. There are four kinds of composite expressions:

i) the sequence (e1 e2 ... en), n≥1,

where e1, e2,..., en are well formed expressions,

ii) the λ expression (λv1.v2...vn e), n≥1,

where v1, v2,..., vn are distinct variables, and e, the λ body, is a well formed expression,

iii) the application <e1 e2>, where e1, the operator, and e2, the operand are well formed expressions without free variables,

iv) the formal application ⌈e1 e2⌉, where e1, the operator, and e2, the operand are well formed expressions, and where at least one of them has an occurrence of a free variable.

---

(1) A full description of this algorithm is to be found in (Koster 1978)

Only applications specify computations. To evaluate an expression is to replace all its applications by their values, starting with the innermost applications. The four rules for evaluating (for reducing) an application are:

i) the primitive operation, which applies when the operator is a primitive operator. For example

<1 (A B C)> → A /the primitive operator 1 selects the first element of a sequence/

ii) the δ substitution. When the operator is an atom defined in terms of another expression, it is replaced by its definition (see LISP DEFINE).

iii) the meta composition. It applies when the operator is a sequence:

<(a1 a2 ... an) e> → <a1 ((a1 a2 ... an) e)>

iv) the λ substitution. When the operator of an application is a λ expression, the operand is substituted for all occurrences of the free variable in the λ body (case of one variable). If there are n variables (n>1), say v1, v2, ..., vn, the operand must have the form (a1 a2 ... an); a1, a2, ..., an are substituted for the free occurrences of v1, v2, ..., vn respectively. Then, in both cases, all formal applications that have lost their free variables in this substitution are transformed into applications. For example,

<(λv1.v2 {+ (v1 v2)}) (2 3)> → <+ (2 3)>

The δ substitution and meta composition are illustrated by the following example. (Note that the effect of RL primitive operators used in this article is given in Appendix A). Let the operator SECOND be defined by δSECOND= (R 1 TAIL). Then

<SECOND (A B C)> → <(R 1 TAIL) (A B C)> /δ substitution/

→ <R ((R 1 TAIL) (A B C))> /meta composition/ → <1 <TAIL (A B C)>>

/effect of operator of regular composition R/ → <1 (B C)> /effect of TAIL/ → B

The main differences between RL and reduction languages as defined in (Backus 1973) reside in the use of several λ variables in a given λ expression, the use of #, NIL, and non-NIL values for the range of predicate operations, and the disappearance of the regular composition, replaced by the primitive operator R (Backus 1978, Minne). These changes are introduced to simplify the algorithm (with respect to non-essential details), and to simplify the language. It is believed that they do not alter the character of RL as a reduction language, and that RL still possesses the Church-Rosser property (McJones 1975).

3 THE ALGORITHM

The algorithm (given in Appendix B) is described in a metalanguage similar to the one used for the LISP interpreter (McCarthy 1965). It accepts a subset of pure LISP. Dotted notation is ignored. The main restrictions are: no free variables inside LABEL expressions, no functional arguments, and no LABEL-defined functions as arguments. We also require that LAMBDA expressions have at least one variable.

The obvious feature of this algorithm is to translate LISP forms (fn arg) into RL applications or formal applications ⌈fn' arg'⌋, LISP forms (fn arg1 arg2 ... argn) into RL (formal) applications ⌈fn' (arg1' arg2' ... argn')⌋, LISP LAMBDA variables (e.g. X) into RL λ variables (e.g. x), LISP LAMBDA expressions into RL λ expressions (e.g. (LAMBDA (X) e) into (λx e') and (LAMBDA (X1 X2) e) into (λx1.x2 e')). Special treatment is needed for LISP conditional expressions, QUOTE expressions, and LABEL notation. The features not accepted by this algorithm are among the most complex of LISP (Gordon 1973). (An improvement of the algorithm to accept some of them is suggested at the end of this paper.)

The complete algorithm works in two passes. The result of the first pass may not be a well formed RL expression, if some formal applications in it have no free variables. The second pass will transform them into applications. For brevity, we often refer to the first pass of the algorithm as the algorithm.

The algorithm, Tr0, accepts the same top level pair fn a as evalquote. It transforms it into the application <fn' a'>, where fn' is the translation of fn. If a is a list of one argument, then a' is equal to this argument. Otherwise, a' = a. Two recursive functions, Tr1 and Tr2 are used in the definition of Tr0. Tr1 translates LISP functions (or operators), whereas Tr2 translates LISP forms. They have a role similar to the functions apply and eval of the LISP interpreter (McCarthy 1965). Besides the LISP element to be translated, Tr1 and Tr2 have another argument, the list of all lists of LAMBDA variables and LABEL-defined functions bound in the current environment. This list is used in the translation of conditional expressions. To continue, the previous comparison, this list can be compared to the association list, which keeps the LISP environment.

Simple examples

In the following examples, LISP programs are given as top level pairs.

LISP : CAR ((A B C)) /evaluates to A/

RL translation : <1 (A B C)> → A

LISP : CONS (A (B)) /evaluates to (A B)/ RL translation: <!IN (A (B))> → (A B)

LISP : (LAMBDA (X) (CONS (QUOTE X) (CDR X))) ((A B C)) /evaluates to (X B C)/

RL translation : <(λx ⌈UN (X ⌈TAIL x⌋)⌋) (A B C)> → <UN (X <TAIL (A B C)>)> → (X B C)

In the last example, note the different translation of the symbol X, according to its occurrence in a QUOTE expression (it is translated to the RL symbol X), or outside a QUOTE expression (it is translated to the RL variable x).

4 PROBLEMS AND SOME SOLUTIONS

We now consider some specific problems of the translation. Those problems are caused by the following features of the two languages: presence of environment in LISP/absence of environment in RL, use of data as programs, and differences of evaluation rules in LISP and RL. The effects of these features overlap and cannot always be analyzed independently.

LISP binding and RL direct substitution

In RL, variables are immediately substituted during a λ substitution. At the start of the evaluation of a LISP LAMBDA expression, the bindings of the LAMBDA variables are stored in the environment. The value of a LAMBDA variable is retrieved later, at the time of the evaluation of the variable in the LAMBDA body. The value retrieved from the activation environment (MOSES 1970) is usually the same as the value stored in the binding environment. In pure LISP, the exceptions to this rule include the use of a free variable in a functional argument (FUNARG problem, thoroughly treated in (Moses 1970)), and the use of free variables in LABEL expressions (Gordon 1973). This algorithm does not accept functional arguments for a different reason given below. It does not accept free variables in LABEL expressions for the reason just given.

Because of the direct substitution mechanism in RL, an evaluation of a variable (hence of an application) could take place in a RL expression, whereas the corresponding LISP variable and LISP form are never evaluated: this case is taken care of by the special treatment of conditional expressions described later.

QUOTE expressions

The form (QUOTE e) is translated to the RL expression (atom or sequence) e. The general rule in LISP is to evaluate a form only once (Weissman 1967). That is, e will be treated as a datum. In this case, the RL translation is correct. However, if (QUOTE e) is a functional argument, e will be evaluated again in the LISP program. In this case, the RL translation is incorrect. This problem is caused by the interchangeability of data and functions in LISP. It is not correctly handled by this algorithm, which acts as a compiler.

Conditional expressions

LISP conditional expressions differ from other pure LISP forms in the way their arguments are evaluated. The complex handling of conditional expressions by the function Tr2 of the algorithm aims to insure that the LISP arguments and their RL translations are evaluated in the same order. Consider the LISP conditional expression

g = (LAMBDA (X) (COND (p1 e1) ... (pm em)))

and its RL translation

g' = (λx ⌈(CNL ((λx p1') (λx e1')) ... ((λx pm') (λx em'))) x⌋)

The occurrences of x in the expressions p1', ..., em' will be protected from immediate substitution until the conditional expression is actually evaluated.

The following example illustrates this point. The LISP pair

e = (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (CAR X)))) (A)

evaluates to A. (CAR X) is not evaluated during the computation. The RL translation of e

e' = <(λx ⌈(CNL ((λx ⌈ATOM x⌋) (λx x)) ((λx T) (λx ⌈1 x⌋))) x⌋) A>

is computed as follows:

e' → <(CNL ((λx ⌈ATOM x⌋) (λx x)) ((λx T) (λx ⌈1 x⌋))) A>
/direct substitution does not affect occurrences of x inside the conditional expr/
→ A /since <ATOM A> → T. x in ⌈1 x⌋ is not evaluated/

A more direct translation using the operator CONDL (see Appendix A) is evaluated as follows:

<(λx ⌈CONDL ((⌈ATOM x⌋ x) (T ⌈1 x⌋))⌋) A>
→ <CONDL ((<ATOM A> A) (T <1 A>))> /immediate substitution of all variables/
→ <CONDL((T A) (T #))> → #

The reduction rule forces also the reduction of <1 A> . This reduction results in an error. Although simpler and more efficient than the first translation, this second translation is incorrect, because its execution is not consistent with LISP order of evaluation.

LABEL translation

Besides keeping the binding of LAMBDA variables, the LISP environment also keeps the binding of LABEL-defined functions. LISP LABEL notation is simulated in RL by the use of three features: meta composition, which saves the "value" of the "function" in the operand and provides for potential recursion, the definition of the operator LAB, which actually allows the recursive use of the "function", and λ substitution, which allows us to replace the occurrences of the translation of the LABEL-defined function by the translation of the definition of the function.

The LABEL expression (LABEL G (LAMBDA (X) e)) is translated to (LAB (λg.x e')). The RL operator LAB has the following effect:

<(LAB (λg.x e)) a> → <LAB ((LAB (λg.x e')) a)> /meta composition/
→ <(λg.x e') ((LAB (λg.x e')) a)> /by definition of LAB/

Note that the LISP function G is translated to the RL variable g. Following the substitution above, all free occurrences of g in e' are replaced by (LAB (λg.x e')), and al free occurrences of x in e' are replaced by a. This is the exact simulation of the computation of LABEL expressions. For example, the LISP program

d = (LABEL G (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (G (CAR X))))))

computes the first atom of its operand. When applied to (A (B)), it returns the value A. Let e' = ⌊(CNL ((λg.x ⌊ATOM x⌋) (λg.x x)) ((λg.x T) (λg.x ⌊g ⌊1 x⌋⌋))) (g x)⌋. Then the RL translation and computation of d ((A (B))) is as follows:

<(LAB (λg.x e')) (A (B))> → <(λg.x e') ((LAB (λg.x e')) (A (B)))> /effect of LAB/
→ <(CNL ((λg.x ⌊ATOM x⌋) (λg.x x)) ((λg.x T) (λg.x ⌊g ⌊1 x⌋⌋))) ((LAB (λg.x e'))
(A (B)))>
/by first λ substitution. The test <ATOM (A (B))> evaluates to NIL, and the
/computation continues with the second branch of the conditional/
→ <(λg.x ⌊g ⌊1 x⌋⌋) ((LAB (λg.x e')) (A (B)))> → <(LAB (λg.x e')) <1 (A (B))>>
/in this λ substitution, g has been replaced by its "value" (LAB (g.x e')). It is
/now recursively applied to the first element of the initial list/
→ <(LAB(λg.x e'))A> → ...

Free variables in LABEL expression

This translation is incorrect when the LABEL expression has a free variable (say Y). If Y is bound in a LAMBDA expression containing the LABEL expression, the translation y of Y will be definitively bound to a value corresponding to the LISP binding environment, rather than the activation environment, as it is required here. For example, consider the following top level pair (Gordon 1973):

e = (LAMBDA (Y) ((LABEL FN (LAMBDA (X) (COND (Y (QUOTE 1)) (X (QUOTE 2))
((QUOTE T) ((LAMBDA (Y)
(FN Y))
(QUOTE T))))))
(QUOTE NIL)))
(NIL)

The variable Y occurs free in the LABEL definition of FN. During the evaluation of e, Y is first bound to NIL. After the recursive invocation of FN that follows the evaluation of ((LAMBDA (Y) (FN Y)) (QUOTE T)), Y is bound to T so that e evaluates to 1.

Let e1' = (λfn.x y), e1'' = (λfn.x NIL), e1' = (λfn.x 1),
e2' = (λfn.x x), e2' = (λfn.x 2), e3' = (λfn.x T), e3' = (λfn.x e4'),
e4' = ⌊(λy ⌊fn y⌋) T⌋,

e5' = (λfn.x ⦃(CNL (p1' e1') (p2' e2') (p3' e3')) (fn x)⦄),

e5" = (λfn.x ⦃(CNL (p1" e1') (p2' e2') (p3' e3')) (fn x)⦄),

e6' = (LAB e5'), e6" = (LAB e5").

The algorithm would incorrectly translate the LISP pair e to the RL application <(λy ⦃e6' NIL⦄) NIL>, whose evaluation follows.

<(λy ⦃(LAB (λfn.x ⦃(CNL ((λfn.x y) e1') (p2' e2') (p3' e3')) (fn x)⦄)) NIL⦄) NIL>
→ <(LAB (λfn.x ⦃(CNL ((λfn.x NIL) e1') (p2' e2') (p3' e3')) (fn x)⦄)) NIL>
/the only free occurrence of y, in p1' = (λ fn.x y), has been replaced by NIL/
→ <e5" (e6" NIL)> /by meta composition and definition of LAB/
→ <(CNL ((λfn.x NIL) e1') (p2' e2') (p3' e3')) (e6" NIL)> /λ substitution/
/the result of the test <(λfn.x NIL) (e6" NIL)> is NIL, and so is the result of the/
/test <p2' (e6" NIL)> , so that the third branch of the conditional expression is/
/taken/ → <(λfn.x ⦃(λy <fn y>)⦄T>) (e6" NIL)> → <(λy ⦃e6" y⦄) T> → <e6" T>
/note that e6" does not contain any free occurrence of y, so that the last/
/λ substitution with y had no effect on it/
/ after a new round of meta composition, LAB application, and λ substitution/
/ the computation continues as follows/
→ <(CNL ((λfn.x NIL) e1') (p2' e2') (p3' e3')) (e6" T)>
/ Now, the test <p2' (e6" T)> = <(λfn.x x) (e6" T)> evaluates to T, and the/
/ second branch in the conditional expression is taken/
→ <e2' (e6" T) > = <(λfn.x 2) (e6" T)> → 2

In this computation, after the first substitution, NIL is definitively substituted for the free occurrence of y in p1'. Thereafter, p1", e5", and e6", which do not contain y free, are used instead of p1', e5', and e6' respectively. Gordon gives an "intuitive" evaluation of e using the binding environment to bind the free occurrence of y. The result of his computation is also 2. This is not surprising, since the RL computation just given closely simulates Gordon's intuitive evaluation.

Parallelism in pure LISP and in RL

Consider the LISP form (fn e1 e2 ... em) to be evaluated in the environment a. The LISP interpreter evaluates first fn if needed, then it evaluates the arguments e1, e2, ..., em from left to right. This order of evaluation, however, is only a consequence of the use of the recursive function evlist (McCarthy 1965), which works from left to right, as do most of the recursive functions used by evalquote. Actually, we could write

eval[(fn e1 ... em); a] = apply[fn; (eval[e1;a] ... eval[em;a]);a]

We see that e1, e2, ..., em can be evaluated in parallel because there is no side-effect from the computation of one argument on the computation of another argument; each computation uses its own copy of the environment. This parallel computation actually takes place in the computation of the RL translation; e1', e2', ..., em' are evaluated in parallel. Note that this parallelism is not limited to these m arguments. Parallel

computation is also possible inside each expression e1', e2', ..., em', so that we have, at least in theory, an unbounded number of parallel computations. As a consequence, execution of the translation of LISP programs could be efficient on a reduction machine. LISP, however, lacks primitive operators similar to reduction language primitive operators (Backus 1973, Koster 1977, Pozefsky 1977) which could take full advantage of this possibility of unbounded parallelism. (LISP 1.5 MAPCAR is such an operator).

5 AN EXTENSION OF THE ALGORITHM

We briefly indicate here how the algorithm can be modified to handle a larger subset of pure LISP as well as some features of LISP 1.5.

Primitive functions of LISP 1.5

Functions which require the evaluation of all their arguments (e.g. PLUS, TIMES) can be handled the same way as CAR or CONS. Functions such as AND and OR, which do not require the evaluation of all their arguments, would be more complex to handle.

Free variables in LABEL expressions

If a variable occurs free in a LABEL expression, but is bound in the outer LISP program, then a preliminary pass of the algorithm could transform the free variable in a bound variable.

Functional arguments with FUNCTION

In (FUNCTION e), e is treated as any other form. For example, if it is a LAMBDA expression, it is treated as any other LAMBDA expression. Moreover, if e has a free variable (e.g. X) bound in the outer program, the translation is still correct, because the direct substitution mechanism of RL will use the same value as the binding environment in LISP. This is precisely the function of FUNCTION (Moses 1970).

Functions defined with DEFINE

The mechanisms of evaluation of DEFINE-defined functions in LISP and of δ-defined function in RL are identical; in both cases, the function name is replaced by the definition of the function, when it appears in position of operator. Therefore the LISP function definition is translated as any other form, and this translation becomes the definition of the RL function. The same function name can be used in LISP and in RL. Note that the function definition in LISP cannot have free variables. Otherwise, this translation would lead to incorrectly formed RL applications, with free variables.

The last two features are illustrated by the following short example. Let G and H be two functions respectively defined by (LAMBDA (Y FUNC) (FUNC (CAR Y))) and by (LAMBDA (X Y) (G X (FUNCTION (LAMBDA (A) (CONS A Y))))).

In the definition of H, Y occurs free in the functional argument (FUNCTION (LAMBDA (A) (CONS A Y))). In the computation of the top level pair H ((A B) (C D)), Y is bound in the binding environment, hence it is bound to (C D). Following the "call" (G X (FUNCTION (LAMBDA (A) (CONS A Y)))), the functional argument will be applied to (CAR Y). Y in (CAR Y) is bound to (A B), but since Y in the functional argument is bound to (C D), the result of the computation is (A C D). If QUOTE is used instead of FUNCTION in the functional argument, Y will be bound in the activation environment, hence it will be bound to (A B). The result of the computation will be (A A B).

The RL translations of the definitions of G and H are:

G = (λy.func ‡func ‡1 y⁑⁑)

H = (λx.y ‡G (x (λa ‡UN (a y)⁑))⁑)

The pair H ((A B) (C D)) is translated and evaluated as follows.

< H ((A B) (C D))> → <(λx.y ‡G (x (λa ‡UN (a y)⁑))⁑) ((A B) (C D))>
/by § substitution/ → <G ((A B) (λa ‡UN (a (C D))⁑))> /by § substitution /
→ <(λy.func ‡func ‡1 y⁑⁑) ((A B) (λa ‡UN (a (C D))⁑))>
→ <(λa ‡UN (a (C D)) ⁑) <1 (A B)>> → <(λa ‡UN (a (C D))⁑) A>
→ <UN (A (C D))> → (A C D)

Again the RL immediate substitution mechanism translates correctly the binding of variables in the binding environment.

Other LISP constructs not mentioned in this paper are not correctly handled by the algorithm. LISP and its interpreter were designed in a very informal way (McCarthy 1978). Particular or even strange constructs are accepted by the LISP interpreter. They are evaluated in a way that may not have been intended by LISP designers. A correct handling of these constructs by a RL translator may be very difficult, or may even be impossible in the framework of a "compiler".

6 CONCLUSION

The algorithm presented in this paper shows that pure LISP (augmented with some features of LISP 1.5) and the reduction language RL are sufficiently similar in many respects to allow the translation of LISP programs into RL programs. It seems that this translation is possible because both languages are applicative languages. Free variables are a cause of problems when they are bound in the activation environment, because the RL mechanism of immediate substitution assumes variables are bound in the binding environment. Another cause of problems is the LISP property of interchangeability of data and programs (or functions). Since the algorithm translates data and functions

differently, it cannot handle an unrestricted use of data as functions (e.g. functional arguments) and of functions as data.

The meta composition has appeared as a powerful feature of reduction languages; in this paper, we have seen its use to translate the recursion introduced by LABEL notation, and to translate LISP conditional expressions. Actually, many more control structures can be implemented by meta composition. Finally, we have seen that parallel computation of the translation of LISP programs is possible. This could allow an efficient execution of LISP programs on a reduction machine.

Acknowledgments

I wish to thank Gyula Mago and Donald Stanat for their comments on an earlier version of this paper.

Bibliography

Backus, J. 1972. Reduction Languages and Variable-free programming. IBM Research Report RJ 1010. (IBM Reseach Laboratory, San Jose, California)

Backus, J. 1973. Programming Language Semantics and Closed Applicative Languages. Conference Record of ACM Symposium on Principles of Programming Languages (Boston)

Backus, J. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Communications of ACM, 21, 8, pp. 613-641.

Gordon, M. 1973. Evaluation and denotation of pure LISP programs: a worked example in semantics. (Ph.D. Thesis, University of Edinburgh)

Koster, A. 1977. Execution Time and Storage Requirements of Reduction Language Programs on a Reduction Machine. (Ph.D. Thesis, University of North Carolina at Chapel Hill)

Koster, A. 1978. An Algorithm for Translating LISP Programs into Reduction Language Programs. Technical Report TR-78-011 . (University of North Carolina at Chapel Hill)

Landin, P. 1964. The Mechanical Evaluation of Expressions. Computer Journal, 6, pp. 308-324

Landin, P. 1966. A Lambda-Calculus Approach. Advances in Programming and Non-Numerical Computations, L. Fox, ed. (Pergamon Press, New York)

Mago, G. 1979. A Network of Microprocessors to Execute Reduction Languages. International Journal of Computer and Information Sciences, 8, 586

McCarthy, J. et al. 1965. LISP 1.5 Programmer Manual. (MIT Press, MIT, Cambridge, Massachusetts)

McCarthy, J. 1978. History of LISP. Sigplan Notices, 13, 8, pp. 217-223

McJones, P. 1975. A Church-Rosser Property of Closed Applicative Languages. IBM Research Report RJ 1589. (IBM Research Laboratory, San Jose, California)

Minne, J. 1977. Programming in Backus' Red Languages. Data Flow Note 13 (University of California at Irvine, March 1977)

Moses, J. 1970. The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem. MIT Report MAC-M-428. (MIT, Cambridge, Massachusetts)

Pozefsky, M. 1977. Programming in Reduction Languages. (Ph.D. Thesis, University of North Carolina at Chapel Hill)

Scott, D. 1971. The Lattice of Flow Diagrams. Symposium on Semantics of Algorithmic Languages. (Springer-Verlag, New York)

Weissman, C. 1967. LISP 1.5 Primer. (Dickenson, Belmont, California)

Wulfe, W. and Shaw, M. 1973. Global Variables Considered Harmful. Sigplan Notices, 13, 8, pp. 217-223

Appendix A. A Partial Specification of Primitive Operators for RL

The left column contains the operator (possibly a sequence). The right column contains case specification of the operand, and the result; cases are separated by a semi-colon; for each operator, the last result applies to the operands excluded by the previous cases. Note that the undefined symbol is the result of any application containing the undefined symbol in part of the operator or the operand.

Operator	Specification
1	(x1 x2 ... xn), n≥1 → x1; #
TAIL	(x1) → NIL; (x1 x2 ... xn), n>1 → (x2 ... xn); #
UN	(x1 NIL) → x1; (x1 (x2 ... xn)), n>1 → (x1 x2 ... xn); #
ATOM	# → #; a, a is an atom → T; NIL
EQ	(x #) → #; (# x) → #; (a a), a is an atom → T; (a b), a and b are atoms → NIL; #
(LAB (λf.x g))	e → <(λf.x g) ((LAB (λf.x g)) e)>
(LAB (λf.x1.x2...xn g))	(a1 a2 ... an), n>1 → <(λf.x1.x2...xn g) ((LAB (λf.x1.x2...xn g)) a1 a2 ... an)>
(CNL (p1 e1))	x → <e1 x> if <p1 x> → b, where b is not NIL nor #; #
(CNL (p1 e1) ... (pn en))	x → <e1 x> if <p1 x> → b, where b is not NIL nor #; x → <(CNL (p2 e2) ... (pn en)) x> if <p1 x> → NIL; #
CONDL	((p1 e1) (p2 e2) ... (pn en)), n≥1 → e1 if p1 is not NIL nor #; → <(CONDL ((p2 e2) ... (pn en)) > if p1 = NIL; #
(R a1 a2 ... an)	x → <a1 <a2 ... <an x>>>, n≥1
(AA f)	NIL → NIL; (a1 a2 ... an) → (<f a1> <f a2> ... <f an>); #

Appendix B. The Algorithm

rl[X] = x

rl[(X)] = x

rl[(X1 X2 ... Xn)]= (x1 x2 ... xn)

lrl [X] =λx

lrl[(X)] = λx

lrl[(X1 X2 ... Xn)]= λx1.x2.....xn

```
Tr0[fn;a]=[null[cdr[a]] → <Tr1[fn;NIL] car[a]>;
           T → <Tr1[fn;NIL] a>]

Tr1[fn;b]=[atom[fn] → [eq[fn;CAR] → 1;
                       eq[fn;CDR] → TAIL;
                       eq[fn;CONS] → UN;
                       eq[fn;ATOM] → ATOM;
                       eq[fn;EQ] → EQ;
                       T → [rl fn]];
           eq[car[fn];LAMBDA] → (lrl[cadr[fn] Tr2[caddr[fn];cons[cadr[fn];b]]);
           eq[car[fn];LABEL] → (LAB (lrl[cons[cadr[fn];cadaddr[fn]]]
                                    Tr2[caddaddr[fn];
                                        cons[cons[cadr[fn];cadaddr[fn]];b]]))]

Tr2[e;b]=[atom[e] → rl[e];
          atom[car[e]] → [eq[car[e];QUOTE] → cadr[e];
                          eq[car[e];COND] → {Tcond[e;b] rl[car[b]]};
                          T → {Tr1[car[e]; b] Trlist[cdr[e];b]}];
          T → {Tr1[car[e];b] Trlist[cdr[e]; b]}]

Trlist[c;b]=[null[cdr[c]] → Tr2[car[c];b];
             T → cons[Tr2[car[c];b]; Trlist[cdr[c];b]]]

Tcond[(COND (p1 e1) ... (pn en)); b] = (CNL (Trc[p1;b] Trc[e1;b]) ...
                                            (Trc[pn;b] Trc[en;b]))

Trc[d;b] = (lrl[car[b]] Tr2[d;b])
```