ソフトウェア工学 57－4
（1987. 11. 25）

ガーベジセルの直接回収を行うＬＩＳＰシステムの試作

井上 克郎　　川瀬 淳　　鳥居 宏次

大阪大学基礎工学部情報工学科

筆者らは、関数型言語プログラムのソースプログラムを静的に解析して回収可能なガーベジセルの発生を検出し、実行時にそのセルを直ちに回収する方法を提案した[9]（ここではこの方法を即時回収方法と呼ぶ）。この即時回収方法の効果を確かめるために、まず、セルに関するデータ収集機能を持つＬＩＳＰインタプリタを試作し、いくつかの例プログラムを実行してみた。その結果、多くのプログラムについては、発生するガーベジセルの多くを、また、いくつかのプログラムでは、その全てを即時回収できることがわかった。また、即時回収が可能になるようソースプログラム中の関数定義を変更する方法についても述べる。

Static Detection of Garbage Cells and an Experimental Lisp System

Katsuro Inoue　　*Jun Kawase*　and　*Koji Torii*

Dept. Information & Computer Sciences, Osaka University, Toyonaka, Osaka 560, Japan

[Abstract]

We have proposed a method to detect the generation of garbage cells, by analyzing a source text of functional programming languages[9]. The garbage cells whose generation is expected are reclaimed immediately without any overhead at the execution time. We call this method *direct reclamation*. To investigate the effects of the direct reclamation, an experimental LISP interpreter has been implemented, and several sample programs are executed. We knew that for most programs, many of the garbage cells are detected and reclaimed by the direct reclamation. Programming techniques to improve the reclamability are also studied.

1. Introduction

Functional programming languages have good properties such as simply defined semantics and mathematical elegance, and the implementation of functional language has been studies. To execute those programs efficiently as much as possible, we have studied various optimization problems [10,15]. As a problem of those optimizations, we have proposed a method to reduce run-time overhead for detecting and reclaiming garbage cells [9]. We call this method *Direct Reclamation*. A *garbage cell* is a certain amount of memory space in a so-called heap area, which is inaccessible, after a certain point of the program execution from the run-time environment.

In this paper, we show an experimentally developed LISP interpreter which implements the direct reclamation. This system has a special built-in function, named *opt*, which analyses the source LISP program and expects some generation of garbage cells. *opt* also embeds, to the source program, functions named *rclm*1 and *hold* which reclaim actually generated garbage cells at the execution time. Using this system, several sample programs are executed, and various data relating to cells in the heap area and the garbage collection processes are collected. These execution results and the data will be shown in Chapter 4. We will also discuss some techniques to improve the source program so that more expressions in the source program satisfies the conditions for the direct reclamation and more garbage cells are reclaimed by the system.

The system has been developed using ordinary implementation techniques of LISP [1], and we did not special attention of the execution speed of the system, since this is an experimentally developed system for investigating the effectiveness of the direct reclamation. The data structure assumed here is the binary list, and the primitive functions *car*, *cdr*, *cons*, and so on are implemented as shown in [1].

The direct reclamation method can be applied to purely functional programs using list as its primitive data structure. In this system, user-defined functions which are defined in purely functional styles are only investigated. Function definitions which might cause side effects are not considered as candidates of the direct reclamation.

The direct reclamation method also assumes that the data types of each argument and function value are known explicitly. Here, we employ a simple type-inference method to get those types of defined functions in the LISP programs.

This system only reclaims some of *backbone* cells which will become garbage, although the proposed method is more general and can be applied to other cells. Backbone cells of a binary list are cells reachable from the root cell only using zero or more pointers stored in *cdr* (right) parts of each cell. (Note that the root cell is a backbone cell.) We suppose that reclaiming only the backbone cells would be practically sufficient in many cases, since all of garbage cells generated at execution time are reclaimed for many sample programs as shown in Chapter 4.

2. Overview of Direct Reclamation

Consider expression ...(G (F x ...) ... x ...) ... where x is a list-type variable and F is also a list-type function. The cells which appear in x (cells reachable from the root cell of a binary list representing x as components of the first arguments of F) and which do not appear in the function value of F are called *non-inherited cells* for the first argument of F. In this case, these cells are not always garbage cells after the value of (F x ...) is computed, since those cells in x may be used as an argument of G. The cells which are in the value of (F x ...), but not in any argument of F including x are called *created cells* for F.

Here, we naturally assume that created cells for F are not shared with other subexpressions, that is, there is no mechanism to share computed values of subexpressions except for subexpressions consisting only variables which are passed as parameters. With this assumption, cells which are created ones for F and which are non-inherited ones for the first argument of G become garbage cells after the value of G is computed. Since non-inherited cells and created cells for a function can dynamically depend on the values of arguments (input data), it is impossible generally to determine all of them without execution. Thus, we only discuss statically detectable cells from the program text.

A backbone cell can sometimes reachable from the root using pointers in *car* parts as seen in Figure 1. These cells are called *overlapping* cells. It is known that in actual LISP programs, there are a few percentage of overlapping cells [3,4], and sufficient conditions for created cells to be non-overlapping has been studied [9]. Here, we assume that backbone cells are non-overlapping so that the following discussion will be simplified very much. The interpreter we have constructed always checks those sufficient conditions shown in [9], and performs the optimization if those conditions hold. We only discuss backbone cells in this paper; however, our method can be easily extended to other cells as described in [9].

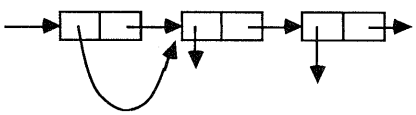

Figure 1. Example of Overlapping Cells.

2.1 Non-inherited and Created Backbone Cells

In this section, we show a method to determine whether all backbone cells of a list-type argument of a function are non-inherited cells for any input date (in such case, we call the argument a *non-inherited argument*), and whether all backbone cells of a list-type value of a function are created cells for any input date

(also, we call the argument a *created argument*). Consider a quicksort program shown in Figure 2. In this case, the first argument of defined function *qs*, represented with variable *x*, is non-inherited argument by the reason discussed later. The function *low* has only a list-type argument, represented by *x*, and it is a created argument as discussed later. Thus, for a subexpression ...*qs*(*low*(... in the definition of *qs*, backbone cells in the function value of *low* is created by *low* and not inherited by *qs*. Hence, those backbone cells can be reclaimed just after the value of *qs* is computed.

```
(de qs (x)
   (cond   [(null x) nil]
      [t (append(qs(low (cdr x)(car x)))
            (cons (car x)(qs(high (cdr x)(car x]]
(de high (x i)
   (cond   [(null x) nil]
      [(not (lessp (car x) i))(cons (car x)(high (cdr x) i]
      [t (high (cdr x) i]]
(de low (x i)
   (cond   [(null x) nil]
      [(lessp (car x) i)(cons (car x)(low (cdr x) i]
      [t (low (cdr x) i]]
(de append (x y)
   (cond   [(atom x) y]
      [t (cons (car x)(append (cdr x) y]]
```

Figure 2. Quicksort Program

2.2 Getting Equations for Inheritedness

For each list-type argument of defined functions, we employ a boolean variable which eventually indicates if all backbone cells of the argument are non-inherited or not. For the first argument of *qs*, boolean variable qs_1 is made where subscript 1 denotes the first argument. In the same manner, $high_1$, low_1, $append_1$ and $append_2$ are prepared. Also we introduce boolean constants car_1=*True* and cdr_1=*False*.

Now, we obtain sequences of functions which are applied to a list-type variable (argument) and which generate list-type result values. For the first argument of *qs*, which is represented with variable *x*, three sequences (*APPENS* (*qs* (*low* (*cdr x*) ...)) ...), (*append* ... (*cons* (*car x*) ...)), and (*append* ... (*cons* ... (*qs* (*high* (*cdr x*) ...)))) are applied to *x*. *null* is a function applied to *x*, but it generates no list-type values. *car*'s appearing at the second argument of *low* and *high* give integer values since those second arguments are known to be non-list type. These function sequences generating no list values are not necessary to be considered here.

Each function sequence is scanned from the right to the left starting at the variable, and each function symbol is replaced by a boolean variable with a subscript indicating the index of argument or a boolean constant until either of the following conditions are satisfied.

(1) *cons* is first encountered.

(2) The next function to the left of a defined function which could eventually apply *cons* directly or indirectly to the variable[9].

The obtained boolean variables and constants are connected together with *or* operator | .

For example, for (*APPENS* (*qs* (*low* (*cdr x*) ...)) ...) we get a boolean expression $cdr_1 \mid low_1$, since *low* would apply *cons* in it to (*cdr x*) and satisfies condition (2). In this function sequence, primitive function *cdr* is first applied to *x* and the value of (*cdr x*) still contains the backbone cells of *x*. Those backbone cells are also the backbone cells of the value of (*cdr x*). If the first argument of *low* is non-inherited, then this function sequence preserves no backbone cells of *x* as the result. In the case that *low* is not non-inherited, the backbone cells of *x* may be or may not be preserved, depending on the characteristic of *append* and *qs*. To analyze this, we have to know that the non-inheritedness of all cells in the binary list for *append* and *qs*, since the backbone cells of *x* might become cells other than backbones in the result of *low*. Here, we simply assume that all the functions eventually applied to the result of *low* could preserve any cell of the result, and the fact whether or not the final result of this sequence involves the backbone cells of *x* might be dominated by the fact whether or not the first argument of *low* is non-inherited.

We analyze in the same way other two function sequences. If any of these three sequences preserves the backbone cells of *x* then the first argument of *qs* is not non-inherited. We obtain the following boolean equation.

$$qs_1 = (cdr_1 \mid low_1) \,\&\, (car_1) \,\&\, (cdr_1 \mid high_1)$$
$$\text{where } cdr_1 = False,\quad car_1 = True$$

Here, we delete all applications of functions outside the application of *cons* including *cons* itself. Primitive function *car* is interpreted as *True* since it never preserves any backbone cells of its argument. For each list-type argument of the defined functions, a set of equations is constructed. We have the following set of equations. (Here, trivial simplification of the equations has been already made.)

$$qs_1 = low_1 \,\&\, high_1$$
$$high_1 = high_1$$
$$low_1 = low_1$$
$$append_1 = append_1$$
$$append_2 = \varepsilon \,\&\, append_2$$

Here, ε shows that the value of the argument can become directly the function value without any modification, and ε is interpreted as *False*, since the backbone cells can be directly become the function value.

The maximum solution of these equations with respect to *False*<*True* shows non-inheritedness of each argument of defined functions. The solution is qs_1= *True*, $high_1$= *True*, low_1= *True*, $append_1$= *True*, $append_2$= *False*.

2.3 Getting Equations for Createdness

To determine if the first argument of *low* is created, we have an equation for three function

sequences, (*cons* (*car x*) ...), (*cons* ... (*high* (*cdr x*) ...)), and (*high* (*cdr x*) ...).

$low_1 = cons_1$ & ($cons_2$ | low_1) & (low_1)
where $cons_1 = True$
$cons_2 = False$

Each sequence is scanned from the left to the right, and each function symbol is replaced by a boolean variable with a subscript indicating the index of argument until one of the following conditions are satisfied.
(1) *car*, *cdr*, or *x* itself are first encountered.
(2) The next function to the right of a defined function which could eventually apply *car* or *cdr* to the rest of the function sequence.
(3) The function which satisfies the conditions of possible creation of overlapping cells[9].

For each list-type argument of defined function, we set up the equation. For the quicksort program, we have the following equations. (Trivial simplification of the equations are made.)

$qs_1 = append_1$ & ($append_2$ | qs_1)
$high_1 = high_1$
$low_1 = low_1$
$append_1 = append_1$
$append_2 = \varepsilon$ & $append_2$

Here, ε is interpreted as *False*. The maximum solution of these equations, $qs_1 = True$, $high_1 = True$, $low_1 = True$, $append_1 = True$, $append_2 = False$, indicates the createdness of each argument of the defined functions. In order that the backbone cells of the function value of a defined function are to be created ones, all of the list-type arguments of the defined functions should be created. In this case, *qs*, *high*, and *low* have only one list-type argument and they are created. Thus, the backbone cells of the function values of these functions are always created. *append* has two list-type arguments and one of them, $append_2$ is not created. Therefore, the backbone cells of the function value may not be created ones.

3. Experimental System

3.1 Overall Structure

We have constructed an LISP interpreter system written in C running on a UNIX machine. The implementation method used for this system is an ordinary one [1]. Primitive functions implemented in this system are limited to necessary ones for executing sample programs and collecting data related to the garbage collections.

The system has special primitive functions, *rclm*0, *rclm*1, ..., *rclm*31, *hold*, and *opt*, which relate to the reclamation of garbage cells, and will be described in the next section. It has also an ordinary *mark and sweep* garbage collection process (named OGC) which marks used cells and returns unmarked cells to the available list when there are no cells in the available list. For each execution of the top-level function, the system prints out the followings.
(1) The number of extracted cells from the available list (consumed cells).
(2) The number of activated OGC processes.
(3) The number of cells returned to the available list by OGC processes.
(4) The total time required for OGC processes.
(5) The total execution time including OGC processes.

3.2 Primitive Functions for Reclamation

There is one special stack which holds pointers to the root cells of binary lists. Primitive function *hold* takes one argument and pushes the argument to this stack. The argument is expected to be the list type, and the pointer to the binary list representing the list is pushed. The value of *hold* function is its argument. Primitive function *rclm*1 pops the top element of the stack, and if it is a pointer to the root of a binary list, the backbone cells of the binary list are returned to the available list. *rclm*1 takes one argument and its value is the same as its argument.

Consider an expression ...(*F* (*G* .....))).... where *F* and *G* are list-type functions, and *F* has one list-type argument and it is non-inherited. Also, each list-type argument of *G* is created. Hence, the backbone cells of the function value of *G* can be reclaimed after the value of *F* is computed. To reclaim these backbone cells, we insert a pair of *rclm*1 and *hold* as ...(*rclm*1 (*F* (*hold* (*G* .....))))).... .

*rclm*2 reclaims all the cells reachable from the root through pointers stored in *car* parts of the cells, and *rclm*3 reclaims backbone cells and cells reclaimed by *rclm*2. In the same manner, *rclm*4 to *rclm*31 reclaim various cells specified. *rclm*0 reclaims no cells but simply pops the top of the stack.

We can insert *rclm* - *hold* pairs into the source program by hand, or for *rclm*1, *rclm*1 - *hold* pairs are inserted to the place where the conditions discussed above are satisfied, by executing a function *opt* as described in the next section.

3.3 Function to Insert *rclm*1 and *hold*

Primitive function *opt* takes one argument which is a defined function name. Here, we call this defined function the *main function*. *opt* performs the following.
(1) Defined functions which are called directly or indirectly from the main function are listed.
(2) For each definition of defined functions listed above, existence of *setq* function or existence of the defined functions which eventually execute *setq* are investigated. In this system, *setq* is only the primitive function which causes side effects. The defined function involving those functions are considered to cause side effects. The following steps are not performed for those functions.
(3) For each argument of defined functions listed, the type of argument is inferred. Here, we employed a simple method to infer types. Consider a definition of *F* such that (*de F* (x_1 ...) (.... (*g* x_1 ...) ...)). If the first argument of *g* is not the list type, it is determined that the first argument of *F* is not the list type. All

arguments of *plus*, *times*, *greaterp*, and so on are considered to be non-list arguments. The non-list arguments of defined functions already inferred are also non-list type. If the definition does not have such g, the first argument of F is considered to be the list type. If a function is defined very unusually in such a way that the data type of the argument is sometimes the list type and sometimes the non-list type, the result of the inference might be wrong and the result of the wrong inference could cause some trouble in the following steps. If we want to overcome this problem, we should employ other inference methods [12]. However, since this system is a prototype system for investigating applicability of the direct reclamation method, we used a simple way.

(4) For each list-type argument of defined functions, sequences of functions (excluding *cond*) applied to the argument are made, which are necessary for constructing boolean equations described in Section 2.2 and 2.3.

(5) For each defined function F, possible application of *car*/*cdr* and *cons* to the arguments is investigated. A flag $flag_1$ for *car*/*cdr* and a flag $flag_2$ for *cons* are prepared for F, and if the definition of F involves *car* or *cdr*, or involves a defined function whose $flag_1$ is true, $flag_1$ for F is set to be *True*. Otherwise, it is set to be *False*. $flag_2$ is set to be *True* if the definition of F involves a *cons* or a defined function whose $flag_2$ is *True*, otherwise it is set to be *False*.

(6) Two sets of equations for the non-inherited arguments and the created arguments are constructed from the information obtained at steps (4) and (5) as described in Section 2.2 and 2.3.

(7) The solutions of the sets of equations are computed using $O(n)$ algorithm described in [9].

(8) In each definition, a subexpression such as ... (*F* ... (*G* .....) ....) ... is searched for, where every argument of G is either created or the non-list type, and (*G* .....) appears as the ith argument of F and the ith argument of F is non-inherited. If it is found, the subexpression is modified to ... (*rclm*1 (*F* ... (*hold* (*G* .....)) ....)) ... .

4. Execution of Sample Programs

We have made following sample programs and executed them on the constructed interpreter.

(1) Quicksort (*qs*). (4 defined functions and 15 lines long.) This program is shown in Figure 2. The input is a random sequence of 1000 integers.

(2) Division by 2 using lists of n *nil*'s (*div*2). (4 defined functions and 17 lines long.) First a list of n *nil*'s is created and next a list of $n/2$ *nil*'s is repeatedly constructed 1200 times. This program is shown in [7]. We gave 200 as n.

(3) Full reverse function (*fullr*). (3 defined functions and 11 lines long.) This program creates an symmetric binary list of a given list [13].

(4) Prime number generator (*prime*). (8 defined functions and 28 lines long.) The algorithm used here is Eratosthenes' sieve. At first, an ascending integer sequence is generated as candidates of primes. For each integer in the sequence, a sequence of multiples is generated, and if a multiple appear in the candidate sequence, it is removed from the candidate sequence. We compute a prime number sequence less than 200.

(5) Core LISP interpreter presented in [14] (*inter*). (7 defined functions and 39 lines long.) On this interpreter, *TARAI* function [7, 16] using lists is executed with input lists of length 7, 5, and 3.

(6) GO board program (*GO*). (43 defined functions and 166 lines long.) This program takes a sequence of coordinates on the GO board as an input. These coordinates represent the locations where two players of GO game intend to place stones in turn. For each placement, a list simulating the GO board is updated according to the rule very strictly, since this program is obtained by transforming an algebraic specification of a GO rule [11]. When the game is over, the winner of the game is determined and returned as the result of this program.

For each program, function *opt* is applied to its main function. Since the type inference method used here is insufficient to detect all of non-list arguments, *rclm*1-*hold* pairs are sometimes inserted to such a subexpression as ...(*F* (*G* ...))... where G always gives non-list values. However, the inserted subexpression is executed correctly, since for non-list value on the stack top, *rclm*1 does nothing except for simply popping up the stack. The execution speed may slow down slightly.

*rclm*1-*hold* pairs are inserted automatically by performing *opt* to all subexpressions which are detected by our hand analysis of the programs. We show in the following the execution results of those sample programs. The data sizes and maximum cell capacities in the heap area are determined so that the total execution times are appropriately long and that enough garbage cells are generated. We counted the total number of the garbage cells generated, the total number of the garbage cells reclaimed by the inserted *rclm*1-*hold* pairs, the number of the OGC processes, the total time consumed for the OGC processes, and the total execution time.

(1) *qs*: Table 1 and 2 show the execution results under two different heap sizes. In Table 1, the heap area has 5000 cells and in the second case, 10000 cells. In Table 2, by applying *opt* to main function *qs*, three *rclm*1-*hold* pairs are inserted to the definition of *qs* as shown in Figure 3. Other functions are the same as Figure 1. All the garbage cells are reclaimed without using OGC processes. In the case of 5000 cells in the heap, the total times reduced by the direct reclamation is 13.6 seconds, and it is the same as the time consumed for OGC processes of the program execution without the direct reclamation (13.6 seconds). In the case of 10000 cells, however, the effect of the direct reclamation cannot be seen since the OGC processes are not time-consuming even without direct reclamation. The execution with the direct reclamation takes a little longer total time because extra *rclm*1-*hold* pairs are executed repeatedly.

Table 1. Execution of Quicksort (qs) under 5000 Cells in the Heap. (Data : 1000 random integers)

1	2	3	4	5	6	7	8	9
N	5000	17144	0	-	15	13.6	56.1	-
Y	5000	17144	17144	100%	0	0.0	42.5	0.1

1: The Direct Reclamation (N: No, Y: Yes).
2: The Total Cells in the Heap Area.
3: The mumber of the Total Garbage Cells Generated.
4: The Number of the Grabage Cells Reclaimed by RCLM1.
5: The Percentage of the Garbage Cells Reclaimed by RCLM1.
6: The Number of Activated OGC Processes.
7: The Total Time Required for OGC processes.
8: The Total Execution Time.
9: The Time Required for OPT.

Table 2. Execution of Quicksort (qs) under 10000 Cells in the Heap. (Data : 1000 random integers)

1	2	3	4	5	6	7	8	9
N	10000	17144	0	-	2	1.0	43.1	-
Y	10000	17144	17144	100%	0	0.0	44.0	0.1

Table 3. Execution of Division_by_2 (div2) under 500 Cells in the Heap. (Data: n=200)

1	2	3	4	5	6	7	8	9
N	500	120206	0	-	599	113.8	256.5	-
Y	500	120206	120200	99%	0	0.0	133.9	0.1

Table 4. Execution of Division_by_2 (div2) under 5000 Cells in the Heap. (Data: n=200)

1	2	3	4	5	6	7	8	9
N	5000	120206	0	-	25	8.0	141.0	-
Y	5000	120206	120200	99%	0	0.0	133.4	0.1

```
(de qs (x)
  (cond[(null x) nil]
     [t (rclm1(append(hold
       (rclm1(qs(hold(low (cdr x)(car x))))))
       (cons (car x)(rclm1(qs(hold(high (cdr x)(car x]]
```

Figure 3. Quicksort with Rclm1 and Hold.

(2) *div2*: Table 3 shows the execution results under 500 cells, and Table 4, 5000 cells in the heap. Four *rclm1-hold* pairs are inserted by *opt*, and most garbage cells are reclaimed by them. In the case of 500 cells without the direct reclamation, the OGC processes are activated seriously. By extending the the cell capacity in the heap, the number of the activated OGC processes are reduced, and the effect of the direct reclamation on the total execution time is reduced.
(3) *fullr*: In Table 5 and 6, we show the execution results under 5000 cells in the heap space. One *rclm1-hold* pair is inserted by *opt* as seen in Figure 4. We used two different inputs. *data1* is a linear list of 500 *nil*'s as (*nil nil nil* ..... *nil*), and *data2* is a nested list of 500 *nil*'s as (((..... (*nil*) *nil*) *nil*) .....) *nil*). All of the garbage cells can be reclaimed in both cases. It might be considered that not all garbage cells were reclaimed for *data2* since *rclm1* collects only the backbone cells; however, the reclamation is performed at each element of the input list recursively and all of the garbage cells are reclaimed.

Table 5. Execution of Full Reverse (fullr) under 5000 Cells in the Heap. (Data: (nil nil .... nil) 500 nil's)

1	2	3	4	5	6	7	8	9
N	5000	125250	0	-	31	57.2	196.9	-
Y	5000	125250	125250	100%	0	0.0	139.6	0.1

Table 6. Execution of Full Reverse (fullr) under 5000 Cells in the Heap. (Data: ((...(nil) nil) ....)nil) 500 nil's)

1	2	3	4	5	6	7	8	9
N	5000	499	0	-	0	0.0	2.9	-
Y	5000	499	499	100%	0	0.0	3.1	0.1

Table 7. Execution of Prime Number Generator (prime) under 1000 Cells in the Heap (Data: The primes less than 200)

1	2	3	4	5	6	7	8	9
N	1000	777	0	-	1	0.1	29.1	-
Y	1000	777	777	100%	0	0.0	28.9	0.3

Table 8. Execution of Core LISP Interpreter (interp) under 1000 Cells in the Heap. (Data: tarai 7, 5, 3)

1	2	3	4	5	6	7	8	9
N	1000	3128	0	-	6	1.2	18.8	-
Y	1000	3128	106	3%	6	1.4	19.1	0.3

Table 9. Execution of Core LISP Interpreter (interp) with hand embedded RCLM24 under 1000 Cells in the Heap. (Data: tarai 7, 5, 3)

1	2	3	4	5	6	7	8	9
Y	1000	3128	1811	57%	2	0.5	17.5	0.3

```
(de fullr (l)
      (do 500 (fr l]
(de fr (l)
      (cond  [(null l) nil]
             [t (rclm1 (append (hold (fr (cdr l))
                               (cons (fr (car l)) nil]]
(de append (l m)
      (cond  [(null l) m]
             [t (cons (car l)(append (cdr l) m]]
```

Figrure 4. Full Reverse Function with Rclm1 and Hold.

(4) *prime*: Table 7 shows an execution result under 1000 cells in the heap. With the direct reclamation, all garbage cells are reclaimed. However, even without the direct reclamation, only one OGC process is activated, and there was no big difference of the total execution times between two cases.
(5) *interp*: In Table 8, we show execution results under 1000 cells in the heap. By the direct reclamation, 3% of generated garbage cells are reclaimed. We further analyzed the program by hand as described in [9] and knew that a pair of *rclm24-hold* can be inserted to a subexpression for *label* in the definition of *apply1*. *rclm24* reclaims the root cell and the cell pointed by the left pointer in the root cell. The execution result of this program is shown in Table 9. The percentage of the reclaimable garbage cells increased to 57%. This is because without *rclm24*, two cells are created for two *cons*'es each time *label* is executed, which is seriously activated in the *TARAI* program, and these two cells become garbage cells. However, *rclm24* reclaims those cells, and reduces the activation of OGC processes.
(6) *GO*: Under 3000 cells in the heap, we executed a GO

Table 10. Execution of GO Board Program (Go) under 3000 Cells in the Heap under 3000 Cells in the Heap. (Data: 24 placements of stones)

1	2	3	4	5	6	7	8	9
N	3000	23144	0	-	19	2.8	265.2	-
Y	3000	23144	5757	24%	17	2.6	265.2	1.6

Table 11. Execution of GO Board Program (Go) under 3000 Cells in the Heap with a Copy Function under 3000 Cells in the Heap. (Data: 24 placements of stones)

1	2	3	4	5	6	7	8	9
N	3000	24764	0	-	22	3.3	268.1	-
Y	3000	24764	24764	100%	0	0.0	263.2	1.7

game having 24 placements of stones on a 5_by_5 board. The game starts with a placement of a black stone, and finally the black winds by one stone advantage. As shown in Table 10, 24% of garbage cells are reclaimed by the direct reclamation. We analyzed the program, and found that if the first argument of defined function *setdiff*, which computes (the set of elements in the first argument) - (the set of elements in the second argument), would be non-inherited and created, four more *rclm1-hold* pairs are inserted. To do so, we changed the definitions of *setdiff* as shown in Figure 5. (In the original definition of *setdiff*, instead of (*copy c1*), simply *c1* was used.) We introduced a copy function *copy*, and duplicated the first argument of *setdiff*. By this modification, four more *rclm1-hold* pairs can be inserted automatically, and all the garbage cells can be reclaimed as seen in Table 11. Function *copy* gives little overhead in this case, and by the effect of the direct reclamation the total execution time of the program with *copy* is shorter than that without *copy*. The time required for *opt* is 1.7 seconds and we consider that it is acceptablely small.

```
(de setdiff (c1 c2) (cond
      (null c2)      (copy c1)
      (t      (setdiff (remv c1 (car c2)) (cdr c2)))))
(de copy (l) (cond
      ((null l)      nil)
      (t      (cons (car l) (copy (cdr l)))) ))
(de remv (l e)      (cond
      ((null l) nil)
      ((equal (car l) e) (remv (cdr l) e))
      (t (cons (car l) (remv (cdr l) e))) ))
```

Figure 5. A Definition of Setdiff with Copy.

5. Programming Techniques to Get Non-inherited Arguments and Created Arguments

5.1 Extracting Variables Out of Arguments

Consider the following functions.

```
(de    g     (y1)   (loop   y1      nil) )
(de    loop  (x1 x2)        (cond
       ((end x1 x2)  x2)
       (t      (loop   (h x1) (f x1 x2)) ))
)      )
```

Function *loop* is defined in tail recursive style. Function *g* calls *loop* with y_1 and *nil* as initial value of the loop. In *loop*, the second argument is modified repeatedly and finally becomes the value of *loop* when the termination condition (*end* x_1 x_2) is satisfied.

Suppose boolean equations of created arguments as follows.

$g_1 = loop_1$
$loop_1 = \varepsilon \ \& \ loop_2$
$loop_2 = loop_1 \ \& \ loop_2$

The maximum solution is that $g_1 = loop_1 = loop_2 =$ *false*, that is, all arguments are not created. However, if we can modify the loop so that x_1 is not passed into *f*, g_1 could be *true*. For example, if the definition of *loop* can be modified as follows, g_1 will be created since the introduced functions $f_1, f_2, ..., f_n$ have no x_1 as their arguments.

```
(de    loop   (x1 x2)      (cond
       ((end  x1 x2)       x2)
       ((case1 x1 x2)      (loop   (h x1) (f1  x2)) )
       ((case2 x1 x2)      (loop   (h x1) (f2  x2)) )
       .....
       (t                  (loop   (h x1) (fn  x2)) )
)      )
```

This technique is important, since purely functional programs tend to use the tail recursive forms as loops, and created lists by the loops often become garbage cells after used for the next functions. Also, this technique can be applied to more general definition such as,

```
(de    g      (y1)   (loop   y1      nil) )
(de    loop   (x1 x2)       (cond
       ((end x1 x2)   x2)
       (t      (v      (h x1) (f x1 x2)) ))
)      )
```

where there is no explicit tail recursive loop. In the same manner as described above, we may be able to extract x_1 out of the second argument of *v* which is not created.

5.2 Introducing Auxiliary Functions

Consider an expression,

... (*g* (*f* *exp* *const*)) ...

where *exp* is some expression, *const* is a constant list having no variables (no formal parameters) in it. Assume that the argument of *g* is non-inherited and the first argument of *f* is created but the second argument is not. The optimizing function *opt* does not insert *rclm1-hold* pair to this expression since not all the arguments of *f* is created.

If we introduce an auxiliary function *f1* and modify the expression as follow, *rclm1-hold* can be inserted.

... (*g* (*f1* *exp*)) ...
.....
(*de* *f1* (x_1) (*f* x_1 *const*))

This is because *f1* has only one argument, and it becomes created one.

This technique might be unnecessary if *opt* checks not only the createdness of arguments but the existence of variables in the expressions.

5.3 Introducing Copy Function

The copy function defined as in Figure 5 simply duplicates given list by construction new backbone cells. It has one argument that is non-inherited and created. Introducing copy functions sometimes helps to improve reclaimability of garbage cells and the total execution speed as described in Section 4 (6). However, using copy functions definitely increases the number of total consumed cells, and it could slow down total execution speed by the effects of additionally inserted *rclm1-hold* pairs in some cases. Thus, we have to investigate the program carefully and know the behavior of program execution before using the copy functions.

6. Conclusions

We have discussed about our experimental LISP system which has facilities of reclaiming garbage backbone cells without extra overhead at the execution time. First, we constructed a core part of the LISP interpreter where no *opt* function had implemented yet but *rclm* and *hold* functions and facility to collect statistic data of cells had implemented. It took about 3 months for a undergraduate student to construct this core part. This LISP interpreter has over 50 built-in functions and 3200 lines long in C. Next, the program for *opt*, which is 1200 lines long in C, was added to the core interpreter. It took one and a half months for one of authors to design and implement *opt*.

In this system, only the backbone cells are reclaimed; however, for some program, the reclaimability will be higher if other cells are also reclaimed as mentioned in Section 4 (5). In the same method as described in Chapter 2, we can set equations for specific cells and determine if they are reclaimable. The time spent for solving a set of equations is small as shown in Chapter 4. Therefore, if the system makes several sets of equations and solves them in order to directly reclaim other cells as well as the backbone cells, the total performance will be improved.

In this paper, we applied the direct reclamation method to a LISP interpreter system. However, this method can be applied to the compilers of functional languages in the same way. No explicit optimization functions such as *rclm*, *hold*, and *opt* are not necessary for the input languages of the compilers, but those facilities could be embedded to the compilers and the generated codes.

References

1. Allen, J., Anatomy of LISP. *McGraw-Hill, Reading, New York* (1978).
2. Barth, J., Shifting garbage collection overhead to compile time. *Comm. ACM.* 20, 7(1977), 513-518.
3. Clark, D.W., AND Green, C.C., An empirical study of list structure in lisp. *Comm. ACM.* 20, 2 (February 1977), 78-87.
4. Clark, D.W., AND Green, C.C., A note on shared list structure in LISP. *Information Processing Letters.* 7, 6 (October 1987), 312-315.
5. Cohen, J., Garbage collection of linked data structure. *Computing Surveys.* 13, 3 (1981), 341-367.
6. Deutsh, L.P., AND Bobrow, D.G., An efficient incremental automatic garbage collector. *Comm. ACM.* 19, 9 (September 1976), 522-526.
7. Gabriel, R.P., Performance and evaluation of lisp systems. *MIT Press, Reading, Massachusetts.* (1985).
8. Hopcroft, J., AND Ullman, J., Introduction to automata theory, languages, and computation. *Addison Wesley, Reading, Massachusetts* (1979).
9. Inoue, K., Seki, H., AND Yagi, H., A method to detect unused list cells at compile time of functional program. *Papers of Technical Group on Automata and Languages, IEICE Japan.* AL85-26 (July 1985), 45-54.
10. Inoue, K., Seki, H., Taniguchi, K., AND Kasami, T., Compiling and optimizing methods for functional language ASL/F. *Science of Computer Programming.* 7, 3 (November 1986), 297-312.
11. Inoue,K., AND Torii, K., Algebraic Specification of the GO Rule., *Technical Report of Information Processing Society of Japan.* SW52-8 (February 1987), 57-64.
12. Katayama, T., Type inference and type checking for functional programming languages, -a reduced computation approach-. *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* (August 1984), 263-272.
13. Kurokawa, T., Introduction to LISP, *Baifukan, Reading, Tokyo,* (1982).
14. MaCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., AND Levin, M.I., LISP 1.5 programmer's manual. *MIT Press, Reading, Massachusetts.* (1965).
15. Seki, H., Inoue, K., Taniguchi, K., AND Kasami, T., Optimization of functional language programs. *Trans. of IEICE Japan.* J67-D, 10 (October 1984), 1115-1122.
16. Takeuchi, I., The result of the lisp contest. *Technical Report of Information Processing Society of Japan.* SYM5-3, (August 1978), 1-28.