Selectively Delayed Evaluation Through Program Transformation

KIYOSHI ONO*

By selectively delaying evaluation of S-expressions in LISP, one can avoid overheads of conventional (indiscriminately) delayed evaluation schemes without losing their advantages.

Selectively delayed evaluation is guided by the strategy "never delay the evaluation of the first argument (*car* part) of *cons*." This strategy restricts the possible forms for representing S-expressions to the following three: (1) *explicit* form not being delayed at all, (2) an *intermediate* form with only *cdr* part being delayed and (3) an *implicit* form being delayed on the whole. These forms are distinguished from one another by a translator, which generates object LISP programs which are executable in any conventional LISP system without run time checks on the forms, and whose execution corresponds, in effect, to delaying evaluation of selected S-expressions in source LISP programs.

With this method, potentially infinite lists in *cdr* direction can be dealt with, and the heap storage space requirement was reduced from $O(2^n)$ to $O(n)$ with 28% loss of run time speed for a program which generated all the elements of the powerset of a set containing *n* elements.

1. Introduction

This paper describes a *selectively* delayed evaluation scheme which avoids run time overheads of conventional delayed evaluation schemes [3, 5], but makes the best use of their advantages so as to reduce heap storage space requirement of computation at run time. Here, the heap storage space requirement means the minimum number of heap storage cells necessary for the computation to be completed. In the following, we explain this idea using terminologies of LISP, but the idea itself is applicable to other programming languages in which data structuring facilities are available and functions are treated as "values."

The delayed evaluation schemes tend to reduce the heap storage space requirement by allowing coroutine-like interactions between functions [4, 6], in which only a necessary part of a potentially large list structure (S-expression) is constructed by a function one at a time. Hence, for instance, infinite lists, such as (1 4 9 16···), can be dealt with as long as only a finite part is actually used.

However, space overheads for storing delayed S-expressions, whose evaluation is delayed, are large because of flags for run time checks on whether the S-expressions have already been evaluated and of environments in which the delayed S-expressions are evaluated later when necessary. The above intrinsic advantage of the delayed evaluation schemes for practical list processing operations dealing with finite list structures are therefore less effective.

Moreover, run time overheads are usually large because programs have to be *interpreted*, requiring run time checks, and because environments are usually implemented with *deep* binding mechanism, in which accesses to variables tend to be time-consuming. When a delayed evaluator is written on conventional LISP systems, instead of being constructed as a special LISP system, the overheads may well exceed 1000%.

This paper shows that the above overheads of space and time can be reduced by *selectively delayed evaluation* through *LISP to LISP translation* in advance of execution: source LISP programs are translated into object LISP programs so that evaluating some S-expressions in the object programs may correspond to delaying evaluation of the corresponding S-expressions in the source programs. The S-expressions whose evaluation is to be delayed are selected according to delaying strategies of the translator.

This selectively delayed evaluation generally has the following advantages:

—The translator can freely (or actively) control whether or not evaluation of an S-expression in a program should be delayed, based on delaying strategies and information gathered at translation time through some analyses, which would be too time-consuming to be repeated at run time.

—The translator could use various forms for representing a delayed S-expression because it can generate object LISP programs which consistently create and refer to the delayed S-expression in a specific form. We will call these forms *selectively delayed representation* (*SD-representation* for short) of the value of the corresponding evaluated S-expressions, for the delayed S-expressions can be considered to denote the corresponding evaluated

*Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo 113, Japan.
Present address: Science Institute, IBM Japan, Mori Buiding 21, 4-34, Roppongi 1-chome, Minato-ku, Tokyo 106, Japan.

S-expressions in a systematic way, with which writers of the translators are concerned but most LISP users are not (or should not be).

In this paper, we describe particular delaying strategies and *SD*-representations, based on differences between the roles of *car* and *cdr* parts in practical usages. The *car* part tends to point to a sub-structure as an element of a list whereas the *cdr* part joins the *car* parts to form the list. This difference is clearly shown by the existence of a list notation in LISP and *cdr*-coding LISP systems, and by empirical studies on list structures [1, 2].

Sequential processing of lists suggests an idea that a list need not exist as a whole at any time: only its first few elements will be needed in the near future. This idea motivates a delaying strategy "never delay the evaluation of the first argument (*car* part) of *cons*." This strategy implies the use of the following three *SD*-representations: (1) *explicit* form, in which conventional evaluated S-expressions represent themselves; (2) *intermediate* form, in which evaluation of only the *cdr* part is delayed; and (3) *implicit* form, in which evaluation of the entire S-expression is delayed. (See Sec. 3 for definitions.)

Through LISP to LISP translation usin gthe three *SD*-representations, infinite lists in *cdr* direction can be dealt with, and the heap storage space requirement is usually reduced from the size of the entire list structure to that of a single element with about 40% run time overheads. For example, the space requirement was reduced from $O(2^n)$ to $O(n)$ with 28% overheads for a program which generated all the elements of the powerset of a set containing n elements.

This paper is organized as follows: Sec. 2 explains overall ideas with a simple example. Sec. 3 describes three *SD*-representations and LISP primitives on them. Sec. 4 describes a LISP to LISP translation. Sec. 5 describes global delaying strategies, which specify whether or not arguments and result of a function should be evaluated before being passed among functions. Sec. 6 shows experimental results on the heap storage space requirement reduced and run time overheads incurred by object LISP programs compared with source LISP programs.

2. Preliminary Explanation with an Example

This section explains, with an example, overall ideas of this paper.

The following source LISP program is intended to sum the square of the natural numbers up to and including 100:

```
  sumlist [squarelist [1]; 100]
where
  sumlist [x; n]=
      [zerop[n]→0;
         T→plus[car[x]; sumlist[cdr[x]; subl[n] ] ] ]
  squarelist[n]=
      cons[times[n; n]; squarelist [addl[n] ] ]
```

This program, however, fails in most LISP systems because the value of *squarelist*[1] is an infinite list.

On the other hand, the object LISP program, having names of the corresponding source program suffixed by an asterisk, can produce the result:

```
  sumlist* [list [function [squarelist*]; 1]; 100]
where
  sumlist* [x: impl; n: expl]: expl=
    [zerop[n]→0;
      T→plus [car [x :=c_impl_interm[x] ];
          sumlist* [cdr[x]; subl[n] ] ] ]

  squarelist* [n: expl]: interm=
    cons [times[n; n];
        list [function [squarelist*]; addl[n] ] ]

  c_impl_interm[x]=apply [car[x]; cdr[x] ]
```

In this example and the rest of this paper, a LISP function *function* should be considered as that of LISP 1.6[7]: *function* is treated like *quote* by interpreters, and does not create a *funarg* (or *closure*).

Although the object programs could be understood as usual LISP programs, they had better be considered as "typed" functions which accept arguments and yield the result in some form. A PASCAL-like notation is used to specify the forms of arguments and the result of an object function, in which *expl*, *interm* and *impl* denote *explicit*, *intermediate* and *implicit*, respectively.

Remarks on the above object programs follow:

1) An S-expression in *implicit* form is a list whose *car* part is an object function and whose *cdr* part is an argument list to the function. For example, an S-expression (SQUARELIST* n^*), where n^* is an integer, represents an infinite list $(n^{*2}\ (n^*+1)^2\ (n^*+2)^2 \cdots)$ containing the squares of integers n^* and upwards.

2) An S-expression in *implicit* form is converted by *c_impl_interm* into an equivalent S-expression in *intermediate* form, whose *car* part is *explicit* and whose *cdr* part is *implicit*: this conversion can be considered as an incremental evaluation. Using the above example, we obtain

$$\begin{aligned}&\text{c\_impl\_interm [(SQUARELIST* } n^*)]\\ &=\text{squarelist*}[n^*]\\ &=(n^{*2}\cdot(\text{SQUARELIST* } n^*+1)),\end{aligned}$$

where n^{*2} and n^*+1 are the values of *times*[n; n] and *addl*[n], respectively.

3) No environments are retained in *implicit* and *intermediate* forms. The environments are general enough to keep values of all the variables whether or not a specific variable is referred to in future computation. The translator avoids the environments by creating *implicit* S-expressions, in which values of only the necessary variables are kept in *cdr* parts of the expressions.

4) No run time checks are necessary by LISP primitive functions. The translator inserts appropriate conversion functions, such as *c_impl_interm*, wherever necessary in object programs.

3. Selectively Delayed (SD) Representations

3.1 Definitions

An S-expression, including an atom, is represented in any of three forms, which are defined recursively as follows:

(1) *explicit* form, in which an S-expression represents itself;

(2) *intermediate* form, in which an atom represents itself, but a non-atomic S-expression represents a non-atomic S-expression in such a way that the *car* and the *cdr* parts represent the *car* and the *cdr* parts in *explicit* and *implicit* forms, respectively;

(3) *implicit* form, which must be a non-atomic S-expression whose *car* part is a function. If an S-expression *e* is in *implicit* form, *apply* [*car*[*e*]; *cdr*[*e*]] is an equivalent S-expression in *intermediate* form.

3.2 Conversion Functions

An S-expression in one form can be converted into an equivalent (but not unique) S-expression in another form. Six conversion functions are shown in the following, in which *c_form1_form2* reads 'coerce *form1* into *form2*':

```
c_impl_interm[x]=interm[x]
                =apply [car[x]; cdr[x]]
c_interm_impl[x]=list [function [identity]; x]
    where identity[x]=x
c_expl_interm[x]=
    [atom[x]→x;
      T→cons[car[x]; c_expl_impl[cdr[x]]]]
c_interm_expl[x]=
    [atom[x]→x;
      T→cons[car[x]; c_impl_expl[cdr[x]]]]
c_impl_expl[x]
    =c_interm_expl [c_impl_interm[x]]
c_expl_impl[x]
    =list [function [c_expl_interm]; x]
```

3.3 LISP Primitive Functions

Generally, LISP primitives can not be applied to S-expressions in forms other than *explicit* form. However, new functions corresponding to a LISP primitive, called its *associated* functions, can be defined for each combination of the forms of the arguments and the result.

Associated functions of typical functions are shown in the following. Not all associated functions are shown, for the others can be easily obtained by a composition of those shown and conversion functions. (Functions in the right hand side of the definitions are LISP functions.)

1. *car* & *cdr*

    car* [*x*: *interm*]: *expl*=car[*x*]

    cdr* [*x*: *interm*]: *impl*=cdr[*x*]

Note that an error occurs on the same condition in both sides of the definitions, e.g., when *car* is applied to an atomic S-expression.

2. *cons*

    cons* [*x*: *expl*; *y*: *impl*]: *interm*
      =cons [*x*; *y*]

3. *atom*

    atom* [*x*: *interm*]: *expl*=atom[*x*]

Note that a LISP primitive ***atom*** is applicable to S-expressions in not only ***explicit*** but also ***intermediate*** form. Note also that values of predicates, such as ***atom***, ***null*** and ***eq***, can be considered as also in ***intermediate*** form since the values are an atomic S-expression, *T* or *NIL*.

4. *eq*

    eq* [*x*: *interm*; *y*: *interm*]: *expl*
      =eq [*x*; *y*]

Non-atomic S-expressions can not be compared by ***eq*** unless they are in *explicit* form.

5. *rplaca* & *rplacd*

    rplaca* [*x*: *interm*; *y*: *expl*]: *interm*
      =rplaca [*x*; *y*]

    rplacd* [*x*: *interm*; *y*: *impl*]: *interm*
      =rplacd [*x*; *y*]

6. arithmetic functions and predicates

    Arithmetic functions and predicates, both accepting numeric atoms, yield numeric and Boolean atoms, respectively. Hence, the form of their arguments and result may be either *explicit* or *intermediate*.

4. LISP to LISP Translation

This section describes how to translate source LISP functions into object LISP functions, which are equivalent to given source functions except for forms of their arguments and result. Forms of arguments and result of source functions are always *explicit* whereas those of object functions are any of the three forms selected at translation time. We assume, in the following, that forms of arguments and result of object functions are selected by global delaying strategies described in Sec. 5.

Once these forms are given for an object function which is now to be generated and other functions which are called from this object function, the translation is a process of generating object expressions whose values are in a form consistent with object functions used. In particular, function applications are translated in such a way that their argument parts are in forms required by object functions in their function parts.

Consider object functions of *append*, as an example.

```
append[x; y]=
  [null[x]→y;
    T→cons[car[x]; append[cdr[x]; y]]]
```

When both arguments of an object function are to be in *implicit* and the result are to be in *intermediate* form, the following object function *append** will be generated:

```
append*[x: impl; y: impl]: interm=
  [null[x :=c_impl_interm[x]]→
```

c_impl_interm[y];
T→cons[car[x];
list[function [append*]; cdr[x]; y]]]

Note that LISP primitives *null* and *cons* are supplied with arguments in the required forms. An argument of *null* (i.e., x) is converted into *intermediate* form; the first and the second arguments of *cons* are in *explicit* and *implicit* forms, respectively, so that the value of *cons*, which is also the value of the entire conditional expression, is in *intermediate* form. Note also that a value of an object expression

list[function[append*]; cdr[x]; y]

is a valid S-expression (APPEND* *cdr*[x] y) in *implicit* form because *cdr*[x] and y are both in *implicit* form as required by *append**.

The translation will be described in terms of a translator function *trans* [*expr*; *form*], whose first argument *expr* is a source expression to be translated, and whose second argument *form* is the required form of a value of the resultant object expression. In the following, we assume that the forms of values referred to by occurrences of variables are maintained through a symbol table. The forms of variables are known at the entry to an object function, and are changed by internal *lambda* and *prog* binding or assignments.

trans[*expr*; *form*] =

Case 1 atom[*expr*]

object: (convert *expr*)
or (SETQ *expr* (convert *expr*))

where *convert* is a conversion function from the form of the current value of *expr* to *form*.

When the value of *expr* is "evaluated" in effect by *convert*, such as *c_impl_interm*, the converted value is assigned to *expr* so as to avoid reconversion.

Case 2 Constant (Quoted) expression

object: *expr* in *form*, i.e., the result of a conversion function from EXPLICIT to *form* applied to *expr*

trans[(QUOTE *expr*); *form*]
=(QUOTE c_expl_form[*expr*])

Case 3 Conditional expression

object: (COND (trans[p_1; *bool*] trans[e_1; *form*])
(trans[p_2; *bool*] trans[e_2; *form*])
. . .
(trans[p_n; *bool*] trans[e_n; *form*]))

where *expr* =(COND ($p_1 e_1$)($p_2 e_2$) . . . ($p_n e_n$))

and *bool* is either EXPL or INTERM.

Because, in LISP, any value other than *NIL* is considered *true* as a Boolean value and because we can determine whether or not a value is *NIL* when the value is in either *intermediate* or *explicit* form, the predicate $p_i s$ are translated into either *intermediate* or *explicit* form, whichever is convenient for unnecessary conversions to be eliminated.

Case 4 primitive function application

object: (*fn_assoc* trans[e_1; r_1] trans[e_2; r_2];
. . . trans[e_n; r_n])

where *expr* =(*fn* e_1 $e_2 \cdots e_n$);

fn_assoc is an associated function of *fn*;
and r_i ($i = 1, 2, \cdots, n$) is the required form of the *i*-th argument of *fn_assoc*.

The associated function *fn_assoc* and r_i are determined based on, among other things, the result form *form* and a *natural* form of the arguments e_i in such a way that redundant conversions are eliminated. Here, the *natural* form of an expression means *explicit* form for constant expressions, the form of the current value of a variable for the variable, and so on.

Case 5 non-primitive function application

expr =(*fn* e_1 $e_2 \cdots e_n$)

In the following, an object function *fn** of *fn* is assumed to accept its arguments e_i in r_i form and yield the result in *result* form. Note that if and only if *result* is INTERMEDIATE, *fn** can appear in the *car* part of S-expressions in *implicit* form.

object:

case *result* = INTERMEDIATE

i) *form* = EXPLICIT

(C_INTERM_EXPL
(*fn** trans[e_1; r_1]
trans[e_2; r_2]
. . .
trans[e_n; r_n]))

ii) *form* = INTERMEDIATE

(*fn** trans[e_1; r_1]
trans[e_2; r_2]
. . .
trans[e_n; r_n])

iii) *form* = IMPLICIT

(LIST (FUNCTION *fn**)
trans[e_1; r_1]
trans[e_2; r_2]
. . .
trans[e_n; r_n])

case *result* = EXPLICIT or IMPLICIT

(convert (*fn** trans[e_1; r_1]
trans[e_2; r_2]
. . .
trans[e_n; r_n])),

where *convert* is a conversion function from *result* to *form*

Case 6 Internal *lambda*

object: ((LAMBDA (x_1 $x_2 \cdots x_n$)
trans[*body*; *form*])
trans[e_1; r_1]
trans[e_2; r_2]
. . .
trans[e_n; r_n]),

where *expr* =((LAMBDA (x_1 $x_2 \cdots x_n$) *body*)
e_1 $e_2 \cdots e_n$),

and r_i is the natural form of e_i.

At the beginning of the translation of *body* of the *lambda* expression, forms of *lambda* variables x_i are initialized to r_i, respectively.

Case 7 Program features *prog*, *go* and *return*

We found the following strategy effective for translating expressions inside most of *prog* expressions: "S-expressions in *explicit* form should never be converted into *intermediate* or *implicit* form."

5. Global Delaying Strategies

This section describes global delaying strategies according to which programmers select forms of arguments and result of object functions.

1. Arguments should be passed in *implicit* form and result should be delivered in *intermediate* form unless otherwise stated.

This strategy assumes that *implicit* forms are more concise than corresponding *explicit* forms, and allows potentially larger list structures (including infinite lists in *cdr* direction) to be constructed element by element.

2. Some arguments should be in *explicit* form if side effects inhibit re-evaluation or re-ordering of evaluation caused by delayed evaluation.

3. So as to avoid excessive conversions from *implicit* to *explicit* form, some arguments may be preferred to be in *explicit* form at the expense of the heap storage space (but no more space than in conventional LISP systems).

```
union [x; y] =
  [null [x]→y;
    member [car[x]; y]→union[cdr[x]; y];
    T→cons [car[x]; union [cdr[x]; y] ] ]
```

In the above example, if enough space is available for accommodating the value of the second argument *y* in *explicit* form, the *y* is preferred in *explicit* form to avoid repeated conversions from *implicit* to *explicit* form inside the function *member*.

4. Results of object functions are preferred to be in *explicit* form if the results, which are assumed to be lists, are constructed from their tail to head.

For example, consider the function *reverse* defined as follows:

```
reverse [x] = prog [[v]
    Loop [null[x]→return[v] ];
        v := cons [car[x]; v];
        x := cdr[x]; go[Loop] ]
```

Note that a value of the variable *v*, whose value is returned as the value of *reverse*, is constructed by

v := cons [car[x]; v],

which adds *car*[*x*], as a new head element, to the existing list referred to by *v*. Hence, we can not know, until exit from *reverse*, what is the *car* of the value of *reverse*. Compare this with the definition of *append*[*x*; *y*] in Sec. 4, from which it is clear that the *car* of the value of *append*[*x*; *y*] is either *car*[*x*] or *car*[*y*].

5. To use several object functions derived from a single source function is sometimes advantageous in order to avoid redundant conversions.

These functions differ from each other just in the forms of their arguments and result. The source function, defined by users, accepts arguments and yields the result in *explicit* form; hence, the source function itself can also be considered as a special object function. In particular, some combination of source and object functions can eliminate frequent conversions between values in *explicit* and *implicit* forms when recursion occurs also in *car* direction like the following *equal*:

```
equal[x; y] =
  [atom[x]→eq[x; y];
    atom[y]→NIL;
    equal[car[x]; car[y] ]→equal[cdr[x]; cdr[y] ];
    T→NIL]
```

The third predicate *equal* [*car*[*x*]; *car*[*y*]] would be left unchanged in *equal** because *car*[*x*] and *car*[*y*] are in *explicit* form when control reaches this predicate.

6. Results

This section describes how much the heap storage space requirement is reduced and how much run time overhead is incurred by our LISP to LISP translation. The heap storage space requirement was analysed by hand whereas run time overhead was measured by comparing execution times of source and object programs on a conventional LISP system.

6.1 Append

Two object functions of *append*

$\text{append}_1^*[x\colon \mathit{expl};\ y\colon \mathit{expl}]\colon \mathit{interm}$

and $\text{append}_2^*[x\colon \mathit{impl};\ y\colon \mathit{impl}]\colon \mathit{interm}$

are considered.

6.1.1. Heap Storage Space Requirement

Provided that x and y refer to lists containing n elements of size m and n' elements of size m', respectively, the orders of the heap storage space requirements are as follows:

append: $nm + n'm' + n$

append_1^*: $nm + n'm' + 3$

append_2^*: $\langle x_{\text{impl}} \rangle + \langle y_{\text{impl}} \rangle + 3 + \max[m; m']$,

where $\langle x_{\text{impl}} \rangle$ and $\langle y_{\text{impl}} \rangle$ denote the size of x and y in *implicit* form, respectively. Note that $\langle x_{\text{impl}} \rangle$ is usually of the order of the size m of an element of x and independent of the length n of x in *explicit* form.

For programs processing lists sequentially, the heap storage space requirement is reduced from $O(nm)$ to $O(m)$ by using append_2^*, where n is the length of a list and m is the size of a typical element of the list.

6.1.2 Run Time Overheads

Run time overheads of *append** should be compared not with an execution time of *append* but with the total execution time T_{total} for processing lists:

$$T_{total} = T_{gen} + T_{cons} + T_{append} + T_{select} + T_{proc},$$

where T_{gen} for generating elements before constructing lists,

T_{cons} for constructing lists,

T_{append} for concatenating two lists,

T_{select} for selecting each element in turn,

and T_{proc} for processing the elements.

We separate T_{total} into two groups:

$T_{comp} = T_{gen} + T_{proc}$

and $T_{concat} = T_{cons} + T_{append} + T_{select}$.

The first group T_{comp} is independent of which function is used for *append*, whereas T_{concat} is independent of how elements are first generated and processed later, for the *car* part of S-expressions in *intermediate* form is in *explicit* form and can be directly generated and processed by user defined source functions. Hence, the ratio of run time overheads is:

$$(T^*_{concat} - T_{concat})/(T_{comp} + T_{concat}) = (R_{concat} - 1)/(1 + T_{comp}/T_{concat}),$$

where $R_{concat} = T^*_{concat}/T_{concat}$, which depends only on which function is used for *append*.

The time T_{concat} is shown for two extreme cases with the length of two arguments of *append* varied. In the following, *n* and *m* denote the length of the first and the second arguments of *append*, respectively:

Case $(n\ m) = (900\ 100)$

	append	$append^*_1$	$append^*_2$
T_{cons}	138	138	0
T_{append}	157	313	345
T_{select}	67	67	333
T_{concat}	362	518	678
R_{concat}	1.00	1.43	1.87

Case $(n\ m) = (100\ 900)$

	append	$append^*_1$	$append^*_2$
T_{cons}	140	140	0
T_{append}	18	299	65
T_{select}	69	69	324
T_{concat}	227	508	389
R_{concat}	1.00	2.23	1.71

(The unit of time is intentionally left unspecified, for only the ratio is significant.)

This result shows that $append^*_2$ can be used with at most 90% overhead: this ratio will be reduced for practical programs because complicated processing of elements makes T_{comp} large compared with T_{concat}.

6.2 Polynomial Manipulation

Polynomial manipulation programs use list processing operations, representing polynomials as lists of terms ordered in some way, say from terms having the highest power to the lowest. Addition of two polynomials is essentially a *merge* operation on two lists, which compare the first terms of the two lists, choosing the "higher" term or producing a single term from the two terms with the same power. A polynomial multiplication function *mult_poly2* may be defined as follows:

```
mult_poly2[x; y] =
  [null[x]→NIL;
    T→add_poly2[mult_term_poly[car[x]; y];
                      mult_poly2[cdr[x]; y] ] ],
```

where *mult_term_poly* multiplies a term and a polynomial.

Suppose that *poly1* and *poly2* have *n* and *m* terms, respectively, and their product has $O(n+m)$ terms. When *poly1* and *poly2* are multiplied by *mult_poly2*, the heap storage space requirement is $O(tnm)$, where *t* is the size of a term, for a value of *mult_term_poly* requires $O(tm)$ cells, and there exist *n* values of *mult_term_poly* when *add_poly2* is first entered.

On the other hand, the following object functions require only $O(t(n+m))$ cells:

mult_poly2* [*x*: *expl*; *y*: *expl*]: *expl*
mult_term_poly* [*x*: *expl*; *y*: *expl*]: *interm*
add_poly2* [*x*: *impl*; *y*: *impl*]: *interm*

The reasons are: the *n* terms (one for each *n* values of *mult_term_poly**) are generated at one time, and the rest of the terms are generated when requested by *add_poly2**. This requires $O(tn)$ cells. The result of *add_poly2** is accumulated, requiring $O(t(n+m))$ cells.

Run time overhead was measured and analysed as in the case of *append* and was about 34% when the simplest representation of a term was used: a pair of an integer coefficient and an integer power.

6.3 Powerset Generation

The heap storage space requirement is reduced from $O(2^n)$ to $O(n)$ for the program, shown in Appendix A, which generates all the elements of the powerset of a given set containing *n* elements.

The heap storage space requirement for the source program is $O(2^n)$ because the value of a *powerset*[*x*] is a list containing all the sub-sets of a set indicated by *x* and the number of the sub-sets is 2^n for the set of size *n*.

On the other hand, the heap storage space requirement for the object program is $O(n)$. A proof can be obtained by considering the successive values of *cdr*[*v*] at the point labeled by *Next* inside the loop. These values consist of $(n+1)$ substructures, which are in either I or P type shown in Fig. 1. We can represent a sub-structure as a *box* which is in either I or P state and has I- and O-ports to form a sequence of boxes. Initially, all $(n+1)$ boxes are in I state, requiring a total of $7n+2$ cells (see Fig. 2 for the case $n=3$). Each time control reaches *Next*, the boxes change their state either from P to I or from I to P, reflecting the change of the value of *cdr*[*v*] caused by *c_impl_interm*. In the change of their states, boxes as a whole behave like a binary counter, in which states I and P correspond to 0 and 1, respec-

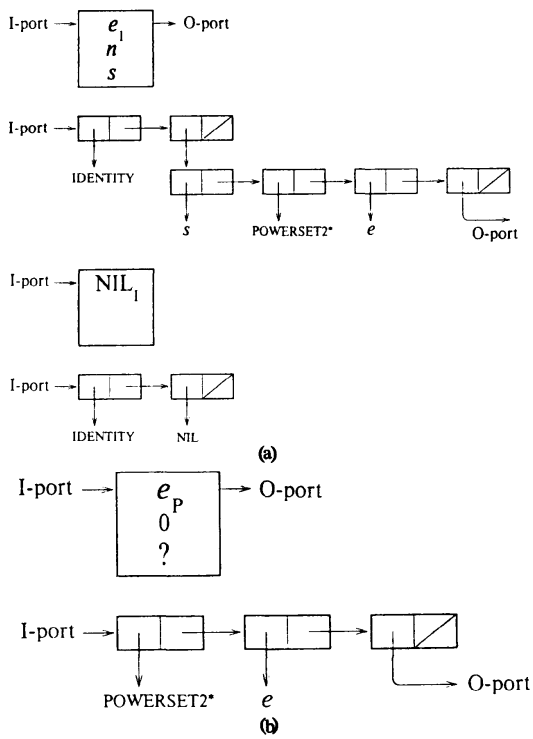

Fig. 1 Sub-structure and Box. (a) I-type sub-structure and I-state box. (b) P-type sub-structure and P-state box.

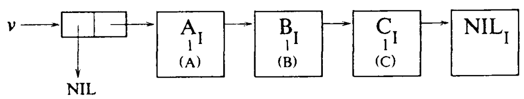

Fig. 2 Initial State of a Sequence of Boxes.

tively, and a transition from P to I generates a "carry" propagating from the left to the right in Fig. 2. During each change of a state of a box, cells are allocated or released, and cells representing *car*[*v*] are released. Considering the sequence of boxes as a binary counter, we can prove by mathematical induction that the number of cells representing successive values of *cdr*[*v*] does not exceed that of the initial value. Thus, the value of *v* can always be represented with at most $8n+3$ cells: at most *n* cells for *car*[*v*] and another cell for connecting *car*[*v*] and *cdr*[*v*].

Run time overhead was 28% when execution time of *print* [*car*[*x*]] was disregarded. This ratio will be decreased when each sub-set is processed after being generated because this processing of the sub-sets, which are already in *explicit* form, does increase the total processing time but never increase the run time overheads.

7. Concluding Remarks

A selectively delayed evaluation through program transformation was described with examples from LISP: source LISP programs are modified in advance of execution so that evaluating an S-expression in the resulting object LISP programs corresponds, in effect, to delaying the evaluation of the corresponding S-expression in the source programs. The object LISP programs are directly executable in any conventional LISP system, and may further be compiled into machine codes, in contrast to delayed evaluation schemes [3, 5], in which LISP programs are usually interpreted by a delayed evaluator.

Three forms for representing (delayed) S-expressions are introduced as *selectively delayed* (*SD*) *representations* of "evaluated" values of the delayed S-expressions. These forms are *concise*: they contain neither environments for later evaluation nor flags for the run time checks. A translator can deal with them like *data types* (i.e., the form of an expression can be determined at translation time).

This selectively delayed evaluation can reduce run time overheads, of space and time, of the delayed evaluation schemes. Therefore, some advantages of delayed evaluation schemes, though well recognized in theory but seldom used in practice because of the overheads, can now be enjoyed by practical programs. In particular, infinite lists in *cdr* direction can be dealt with, and the heap storage space requirement is usually reduced with an increase of only about 40% in run time overhead (instead of more than 1000% in the case of conventional delayed evaluators). These programs include a polynomial multiplication program. As another example, the space requirement was reduced from $O(2^n)$ to $O(n)$ with an increase of only about 28% in run time overhead for a program which generated all the elements of the powerset of a given set containing *n* elements.

A particular advantage of the *selectively* delayed evaluation is that delaying evaluation can be confined to only a few functions which are crucial to the reduction of the heap storage space requirement, while the other functions are evaluated as usual. Moreover, functions whose evaluation can not be delayed for some reasons, such as side effects, can coexist with other functions whose evaluation can be delayed safely and advantageously.

Global delaying strategies described in Sec. 5 are heuristic and do not always choose the best forms of arguments and result of object functions. Hence, our translator is currently supported by programmers through declarations, which inform the translator of the forms. More systematic strategies are needed.

Finally, it is worthwhile to note that the *selectively delayed evaluation scheme* is a general framework, in which various delaying strategies and *SD*-representations could be used depending on various requirements and language systems in which object programs are to run. In this paper, we described a particular delaying strategy "never delay the evaluation of the first argument (*car* part) of *cons*" for LISP and the corresponding three *SD*-representations. Although this strategy was found to be effective in conventional LISP systems for most programs, it can not deal with list structures with a *car* part of infinite sub-structure since it suggests *intermediate* form with the *car* part always being evaluated.

In order to deal with such lists, two approaches will be promising. The first approach is to devise new *SD*-representations (i.e., a special *intermediate* form and an *implicit* form) for each function resulting in infinite lists in both *car* and *cdr* directions. For example, for the following program resulting in an infinite binary tree with each node having a value:

```
binary_tree[n] =
    cons[n; cons [binary_tree[times[2; n] ];
        binary_tree[addl[times[2; n] ] ] ] ]
```

we would use a special *intermediate* form, *interm_eii*, in which the *car*, *cadr* and *cddr* parts are *explicit*, *implicit_eii* and *implicit_eii*, respectively, where *implicit_eii* is an *implicit* form resulting in an S-expression in *interm_eii* when *c_impl_interm* is applied to. An S-expression (BINARY_TREE*n**), where *n** is an integer, is in *implicit_eii* form provided that the following object function *binary_tree** is defined:

```
binary_tree*[n: expl]: interm_eii =
    cons[n; cons [list [function [binary_tree*];
                        times [2; n] ];
                  list [function [binary-tree*];
                        addl [times [2; n] ] ] ] ]
```

Once the special forms are devised, it will be easy to generate object functions dealing with this binary tree. It will be a challenging future study to write a program which devises special forms for a given set of functions.

The second approach is to abandon temporarily the strategy that the *car* part should always be evaluated, and to use run time checks, instead, for functions dealing with such lists. In other words, these functions are translated into object functions in which some occurrences of primitive LISP functions, such as *car* and *cdr*, are preceded by some codes for run time checks on whether or not an S-expression is in a delayed form. Note that overheads incurred by the run time checks can be minimized because the primitive functions requiring run time checks are confined in these functions. Furthermore, analyses at translation time could eliminate run time checks required by some occurrences of the primitive functions even within these functions.

Acknowledgments

Thanks are due to Professor Eiichi Goto for his encouragement and critical comments on earlier drafts of this paper, which led ultimately to a clearer presentation. Thanks are also due to Mr. Masayuki Suzuki for his critical readings of this paper.

References

1. CLARK, D. W. and GREEN, C. C. An Empirical Study of List Structure in Lisp. *Comm. ACM* **20**, 2 (Feb. 1977), 78–87.
2. CLARK, D. W. Measurements of Dynamic List Structure Use in Lisp. *IEEE Trans. Software Engineering*, Vol. SE-5, no. 1 (January 1979), 51–59.
3. FRIEDMAN, D. P. and WISE, D. S. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, S. Michaelson and R. Milner, Eds. Edinburgh, Scotland: Edinburgh Univ. Press (1976), 257–284.
4. FRIEDMAN, D. P. and WISE, D. S. Aspects of Applicative Programming for Parallel Processing. *IEEE Trans. Computers*, Vol. C-27, no. 4 (April 1978), 289–296.
5. HENDERSON, P. and MORRIS, J., Jr. A lazy evaluator. In *Proc. 3rd ACM Symp. Principles of Programming Languages* (1976), 95–103.
6. HENDERSON, P. *Functional Programming Applications and Implementation.* Englewood Cliff, New Jersey: Prentice-Hall (1980).
7. QUAM, L. H. and DIFFIE, W. *Stanford LISP* 1.6 *Manual.* Stanford, California: Stanford Artificial Intelligence Laboratory, Operating Note No. 28. 7 (1972).

Appendix A

```
Powerset Generation Program
source program:
  printsubset[x] = prog[ [v]
    v := powerset[x];
  Loop
    [null[v]→return[NIL] ];
    print[car[v] ];
    v := cdr[v]; go[Loop] ]
where
  powerset[x] =
    [null[x]→cons[NIL; NIL];
      T→powerset2[car[x]; powerset[cdr[x] ] ] ]
  powerset2[e; pset] =
    [null[pset]→NIL;
      T→cons[car[pset];
          cons[cons[e; car[pset] ];
               powerset2[e; cdr[pset] ] ] ] ]
object program:
  printsubset*[x: expl]: expl = prog[ [v]
    v := list[function[powerset*]; x];
  Loop
    [null[v := c_impl_interm[v] ]→return[NIL] ];
  Next  /* This label is referred to in the text */
    print[car[v] ];
    v := cdr[v]; go[Loop] ]
where
  powerset*[x: expl]: interm =
    [null[x]→cons[NIL; (IDENTITY NIL)];
      T→powerset2*[car[x];
                   list[function[powerset*];
                        cdr[x] ] ] ]
  powerset2*[e: expl; pset: impl]: interm =
    [null[pset := c_impl_interm[pset] ]→NIL;
      T→cons[car[pset];
          c_interm_impl[cons[cons[e; car[pset] ];
                        list[function[powerset2*];
                             e; cdr[pset] ] ] ] ] ] ]
  c_impl_interm[x] = apply[car[x]; cdr[x] ]
  c_interm_impl[x] = list[function[identity]; x]
  identity[x] = x
```

(Received April 27, 1971: revised June 3, 1972)