LISP Programming Using Ellipsis Notation

AKIRA FUSAOKA* and HIROSHI FUJITA*

This paper presents a method and system which introduces an ellipsis notation into LISP programming and computation by using a formula extrapolator. In this system, a function *factorial*[*n*] for example, can be defined as

factorial[*n*]=1*2*3*...*n

and also the data object can be represented as

1*2*3*...*100

Not only the specification but also the evaluation process of an expression is simplified by extrapolating the general stage of evaluation instead of the actual execution of annoying iteration. An algorithm of a formula extrapolator is sketched in a later section.

1. Introduction

This paper presents a method and system which introduces an ellipsis notation into LISP programming and computation. In ordinary mathematical usage, we give terms of a sequence to define an expression, counting on the inferencial ability of the reader to determine what the actual expression is. For instance, we might define a function *factorial* [*n*] as

factorial [*n*] = 1*2*3*...*n

and also represent the value *factorial* [100] as

1*2*3*...*100

This ellipsis notation** for a sequence can be used as a simple and useful means for abstracting the iterated structure of a program and computation, that is, it often allows the elimination of the need for iterating loops, recursion and induction from the specification, execution and verification of the program.

The program in the sequence form will be more readable and easier to write than the corresponding LISP expression. The following examples illustrate the specification method of program in the ellipsis notation for the sequence.

Example 1. reverse function

A common representation of the reverse function is

reverse[*x*] = [*null*[*x*] → NIL;
T → *append*[*reverse*[*cdr*[*x*]], *list*[*car*[*x*]]]]

By using the sequence, however, the same function is represented as follows.

reverse[(x_1 x_2 ... x_n)] = (x_n x_{n-1} ... x_1)

By virtue of this style of definition, the verification of properties is also simplified. For instance, the following relation of reverse can be proved via only symbolic evaluation without using any more elaborate techniques [2].

reverse[*reverse*[(x_1 x_2 ... x_n)]
= *reverse*[(x_n x_{n-1} ... x_1)]
= (x_1 x_2 ... x_n)

Example 2. filtering

The process of enumerative iteration can be also abstracted by using μ-operator and the sequence. For example,

integer-squareroot[*x*] = (μy(1 2 ... *x*)
$(y+1)*(y+1) > x$)

where ($\mu y(s_1\ s_2 \ldots s_n)$ P[*y*]) means the first s_i which satisfies P[s_i]; if no such s_i exists, the value of the form is taken as NIL. For instance,

integer-squareroot[10] = (μy(1 2 ... 10)
$(y+1)*(y+1) > 10$)

and,

$(1+1)*(1+1) = 4 < 10$ for $y=1$,
$(2+1)*(2+1) = 9 < 10$ for $y=2$,
$(3+1)*(3+1) = 16 > 10$ for $y=3$,

so we get

integer-squareroot[10] = 3

In addition to program writing, a sequence form is also useful to represent a data object. For instance, a sequence

(1 2 3 ... 100)

can be used for the representation of a list of which each element is a consecutive number from 1 to 100. This elliptical form of a data object is not only more readable but also easier to compute by avoiding the evaluation of annoying iteration whenever the result can be inferred from the sample computations. This extension is practically important because it leads to a different form of partial evaluation.

Consider the next function *foo* defined as follows,

foo[*n*, *m*] = [*greaterp*[*n*, *m*] → ( );
T → (*n* !*foo*[*n*+1, *m*])]

(We use a strip operator "!" which is evaluated immediately whenever its argument is a list.)

Then, the value *foo*[1, 100] would be evaluated by repeating the following steps 100 times.

foo[1, 100] ⇒ (1 !*foo*[2, 100])
⇒ (1 2 !*foo*[3, 100])
⇒ and so on.

However, a clever evaluator which contains the ability to extrapolate the general term of a sequence from the given sample terms might become aware of the iterated

*Central Research Laboratory Mitsubishi Electric Corporation, Amagasaki, Hyogo 661, Japan.

**Turner uses similar notation ".." representing integer interval in his KRC system [10], however, the ellipsis "..." may implicate more information than it.

pattern in the loop during a few initial computations, and infer the following general stage of the computation.

$\Rightarrow$(1 2 ... n !$foo[n+1, 100]$)

By examining the termination condition, the value in the simple sequence form

$\Rightarrow$(1 2 ... 100)

is finally attained.

Under this sequence form regime, nonterminate functions and infinite lists are quite useful as the following example illustrates.

Example 3. infinite list

The function *integer* defined as

integer[i]=(i !*integer*[$i+1$])

is nonterminate, but *integer*[0] would be evaluated to (0 1 2 ...) so that,

car[*integer*[0]]=0

cdr[*integer*[0]]=(1 2...)

This mechanism may offer another way of lazy evaluation [6].

In the evaluation process of the functions defined with the ellipsis notation as in the above examples, we require a mechanism which can extrapolate the ommited part of the elliptical form. It is also required when abstracting iterative reduction steps to get the results in an elliptical form. In both cases, the formula extrapolator plays the essential part of the system. We describe its algorithm briefly in a later section.

2. Sequence From and its Evaluation

In this section, we describe a hyper system of LISP which contains the implementation of the techniques described above. The system consists of the syntactical constitution and evaluation mechanisms for sequence forms and it allows a manipulation of the objects in the ellipsis notation.

Although a sequence is usually given by indicating the sample terms, it is treated as a function which gives the n-th term for given n. This function, which is called a sequence function, is automatically constructed from the given sample terms by using a formula extrapolator. The computation of sequence objects is, therefore, a process of program transformation for a sequence function rather than the evaluation of LISP objects. For instance, the evaluation of the following expression

eq[($s1$ $s2$ ... sn), ($t1$ $t2$ ... tm)]

is performed by checking whether the sequence function for each sequence is essentialy the same in that context.

2.1 Syntax of Sequence Form

(1) Sequence Form

A sequence form is defined as one of the following patterns.

(a) An usual LISP form, and
an infix form of arithmetic and logical expression

(b) A sequence: ($t1$ $t2$ ... tn) and ($t1$ $t2$ ...)
where $t1$, $t2$, tn are LISP forms.

(c) An expression (a sequence with an infix operator):
($s1\oplus s2\oplus\ldots\oplus sn$) and ($s1\oplus s2\oplus\ldots$)
where $s1$, $s2$, sn are LISP forms, and
$\oplus$ is an infix operator.

(d) A LIT* form:
$f[u1, f[u2, \ldots, f[un, w, vn], \ldots, v2], v1]$ and
$f[u1, f[u2, \ldots, v2], v1]$
where f is a function name, and
$u1$, $u2$, un, $v1$, $v2$, vn, w are LISP forms.

(e) A μ form: (μy($s1$ $s2$ ... sn) P[y])
where P is a predicate of y.

(f) An application: $f[x], f[x, y]$, etc.

(g) A list constructor "(", ")" and a strip operator "!" can be used to constitute a new sequence form, such as
(1 2 3 !($x1$ $x2$ $x3$ ...))

(h) A conditional form: (if p then q else r)
where p is a LISP form and q, r are sequence forms.

All through the above definition, the number of the sample terms such as $u1$, $u2$ is arbitrary, but there should be enough to be unambiguous.

(2) Sequence Constant

A sequence constant is the most reduced sequence form, which would be derived from one of the above forms. For instance,

(a) (1 2 ... 100)

(b) (1+2+...)

(c) *iplus*[1, *iplus*[2, ..., *iplus*[100, 0]...]]

(3) Slambda Function

A function defined with a sequence form is called a slambda function. It has the form as follows,

(SLAMBDA (⟨slambda variables⟩) ⟨sequence form⟩)

where ⟨slambda variables⟩ is a list of either simple atom or a sequence of subscripted variables such as ($x1$ $x2$ $x3$ ... xn).

For example,

(a) (SLAMBDA (N) (1*2*3*...*N))

(b) (SLAMBDA (($X1$ $X2$ $X3$ ... XN)
($Y1$ $Y2$ $Y3$ ... YM))
(($X1+Y1$)($X2+Y2$)$\cdots$($XN+YN$)))

A user function can be defined with the slambda via a system function DEFSEQ. DEFSEQ puts the slambda body into the function cell without any modification. For instance,

DEFSEQ[REV1 (SLAMBDA (($X1$ $X2$ ... XN))
(XN $X\langle N-1\rangle$...$X1$))]

DEFSEQ[REV2 (SLAMBDA(X)(!REV2[CDR[X]]
CAR[X]))]

(4) Internal Representation of Sequence

In the system a sequence is represented by the following 4-tuple with a marker SEQ.

(SEQ I O G F)

where I: an initial term
O: a separater (an infix operator)
G: a sequence function
F: a final term

*This list iteration LIT is slightly extended from its original definition by Barron and Strachey [1].

For example, a sequence (1 3 5 ... N) is represented internally as

(SEQ 1 "," (2*(n−1)) N)

and the sequence ((X*1)+(X*2)+···) as

(SEQ (X*1) "+" (X*n) NIL)

A LIT form has a special internal representation as follows

(LIT F L K R)

where F: a function
L: a left argument sequence
K: the innermost argument
R: a right argument sequence

For example,

$f[a1, f[a2, f[a3, \ldots, f[a_n, c, b_n], \ldots, b3], b2], b1]$

is represented internally as

(LIT f (SEQ $a1$ "," Ga an)
c (SEQ $b1$ "," Gb bn))

where Ga, Gb are sequence functions for a_i, b_i respectively.

2.2 Evaluation Mechanism

The evaluation of a sequence form is rather different from what the usual LISP eval function does in the point that it iterates nondeterministic reduction steps by using a lot of reduction rules and sometimes abstracts the reduction steps using "···" if it becomes aware of an iteration of the same pattern of a reduction.

(1) Reduction Rules for Sequence Form

The reduction rules for a sequence form are defined as follows.

(a) LISP rules: For a LISP object L,
L⇒*eval*[L]

(b) sequence reduction rules: For a sequence s=(SEQ I O G F),
s⇒(SEQ *eval*[I] O *trans*[G] *eval*[F])
where *trans*[G] means the optimized sequence function in the current context. For instance, a sequence function (X+n−1) of a sequence ((X+0) (X+1)(X+2)···) is transformed into n when a current value of X is 1.

(c) strip operator rule: For sequence form s,
!(s)⇒s

(d) For applications of primitive functions to sequence forms s=(SEQ I1 O1 G1 F1) and t=(SEQ I2 O2 G2 F2),
car-rule: *car*[s]⇒*eval*[I1]
cdr-rule: *cdr*[s]⇒s'
where s'=(SEQ *eval*[*G*[2]] O *trans*[*G*[n+1]] *eval*[F])
cons-rule: *cons*[s, t]⇒(s!t)
eq-rule: *eq*[s, t]⇒if *eval*[I1]=*eval*[I2] and O1=O2 and *trans*[G1=G2] and *eval*[F1]=*eval*[F2] then T else NIL.

(e) LIT rule: For a LIT form,
(LIT F L K R)⇒(LIT F L′ K R′)
where L≜L′, R≜R′
(≜means closure of ⇒)

(f) if-then-else rule: For a conditional form,
(if p then q else r)⇒q if p≜T
⇒r if p≜NIL

(g) μ rule: For a μ form,
(μy ($s1$ $s2$ $s3$ ... sn)P[y])
⇒Perform the reductions P[$s1$], P[$s2$], P[$s3$], ...
and returns first si' (si≜si')
such that P[si]≜T

(h) unfolding[3]:
For the application of a user defined function f to a sequence form s,
f[s]⇒definition body of f
with the actual parameter s
Binding of the sequence variable x=($x1$ $x2$ $x3$ ... xn) to a value s, is performed as follows.
If the value is a list then it is transformed into a sequence ($s1$ $s2$ $s3$ ... sn), and the binding pair
((SEQ $x1$ "," Gx xn)
(SEQ $s1$ "," Gs sn))
is generated.
If s is a sequence (SEQ I O G F), just a binding pair:
((SEQ $x1$ "," Gx xn)
(SEQ I O G F))
is generated. n is bound to the length of s.
The value of each xi is generated by using the binding pair when it is necessary.

(2) Loop Abstraction

In the process of evaluation of a recursively defined function, the stage of reduction with the similar form is usually repeated, because it contains the repetitive application of the unfolding rule to that function. If the form of the general stage of the reduction can be inferred from the first few stages, then it will be possible to generate the result in the simple form of a sequence or LIT form, avoiding the actual execution of the iteration process.

To illustrate the method of this loop abstraction, we consider the next function *foo*,

foo[x, y]=(*if* x>y *then* ( ) *else*(x !*foo*[x+1, y]))

The reduction steps of foo[1, 100] are as follows.

foo[1, 100]
=(*if* 1>100 *then* ( ) *else* (1 !*foo*[2, 100]))
⇒(1 !*foo*[2, 100]) ···if-then-else rule
⇒(1 !(*if* 2>100 *then* ( )
else (2 !*foo*[3, 100])))
···unfolding of *foo*
⇒(1 !(2 !*foo*[3, 100])) ···if-then-else rule
⇒(1 2 !*foo*[3, 100]) ···strip operator rule
⇒(1 2 !(*if* 3>100 *then* ( )
else (3 !*foo*[4, 100])))
···unfolding of *foo*
⇒and so on.

The underlined stages which are generated by the application of unfolding rule are used as the sample data for the formula extrapolator to infer the general stage:

→(1 2 ··· n
!(*if* n+1>100 *then* ( )
else (n+1 !*foo*[n+2, 100])))

A value of n should be determined to select such a program segment that the same reduction step is not repeated any more. In this case, n is bound to 100, and the reduction terminates.

$\Rightarrow$(1 2 ... 100 !( ))
$\Rightarrow$(1 2 ... 100) ···strip operator rule

2.3 Transformation

A transformation from a sequence form into a LISP object is performed by a system function TRANSSEQ which accepts a function name and a sequence form, and transforms it into a LISP function with recursion or iteration. For instance,

TRANSSEQ[FOO (1 2 3 ... 100)]

generates a LISP function of name FOO with an auxiliary recursive function FOO1 as follows.

FOO=(LAMBDA ( ) (FOO1 1))
FOO1=(LAMBDA (X) (COND ((EQ X 100)NIL)
(T(APPEND(LIST X)(FOO1(ADD1 X1))))))

3. Sequence Definability

In this system, a convenient way to specify a LISP function in the sequence form is provided. DEFINES system function, which allows the programming in the ellipsis notation, transforms a sequence form into a LISP function with recursion or iteration.

DEFINES takes a function name and a slambda function as its body, and puts the corresponding LISP function into function cells.

A function defined by DEFINES is evaluated by the usual eval function instead of applying sequence form reductions. For instance,

DEFINES[FOO (SLAMBDA(X)(1*2*3*...*X))]

defines an usual factorial function.

3.1 Slambda Variable

As are in the DEFSEQ, two types of slambda variable list are permitted in DEFINES.

The first is a list of atoms. This type of a variable is called a simple variable. The second type is a sequence of subscripted variables, that is called a sequence variable, or a list of a sequence variables. For instance,

($X1$ $X2$ ··· XN) or (($X1$ $X2$ ··· XN)
($Y1$ $Y2$ ··· YM))

In a simple variable list, one of the three special variables MAX, MIN and TILL may occur as a last element of the list. These parameters tell us when to terminate the computation. The occurence of the MAX(MIN) means that the computation is continued until the value of the next term of the sequence exceeds (becomes less than) the value of MAX(MIN). An occurence of TILL means that the number of terms for the sequence to be computed is up to the value of TILL. For instance, the function EX defined as

DEFINES[EX (SLAMBDA(X MIN)
(1+X+
(X*X/(2*1))+
(X*X*X/(3*2*1))+...)]

gives such a value as

$$\sum_{k=0}^{N-1} \frac{X^k}{k!}$$

where $X^N/N!<$MIN

On the other hand, the NTHCDR function defined as

DEFINES[NTHCDR (SLAMBDA(X TILL)
(CDR···(CDR X)···)]

applies CDR to X repeatedly to get TILL-th cdr of X.

3.2 Filtering

A filtering which is a searching process of the candidate from enumerative components of a sequence can be represented in DEFINES in the following way.

DEFINES[INTEGER-SQUAREROOT
(SLAMBDA(X)(MU Y(1 2 ··· X)
((Y+1)*(Y+1)>X))))]

where MU represents μ-operator. This definition specifies an algorithm which repeats the checking,

$2*2>X$, $3*3>X$, and so on,

until it finds out Y such that $(Y+1)*(Y+1)>X$. However, this algorithm contains a lot of unnecessary computations so that some optimization is required.

The optimization is performed as follows.

(a) A function NEXT which gives the next candidate is constructed from the sequence function for the sequence (1 2 ... X);
NEXT: $u\Rightarrow u+1$.
(b) Assume that $(u+1)*(u+1)<X$.
In this case, the next stage:
$((u+1)+1)*((u+1)+1)-X$
$=(u+1)*(u+1)-X+2*u+3$
should be computed to check if the result is positive. But the underlining part of the expression is already computed in the previous stage, so that only the other part should be computed.

This strategy leads to a simple loop shown in the Fig. 1. Also the optimized program by DEFINES is shown in the Fig. 2.

In this system, a goal directed optimizer is used in which the simplification rules:

(a) unfolding of a function and an expression
(b) replacement of expression into a variable
(c) usage of rules such as,
$car[cons[x, y]]\Rightarrow x$

are contained. These rules are also used in the sequence form reductions.

4. Formula Extrapolation

The problem of inferring a general term of a sequence from some given sample terms is called formula extrapolation. The function of formula extrapolation is one of the essential parts of this system, because the class of functions representable in this system depends upon the performance of the formula extrapolator.

In this section, we present a method of inference on which the formula extrapolator is based. Let ($T1$ $T2$ $T3$ $T4$ ...) be a given sequence where $T1$, $T2$, $T3$

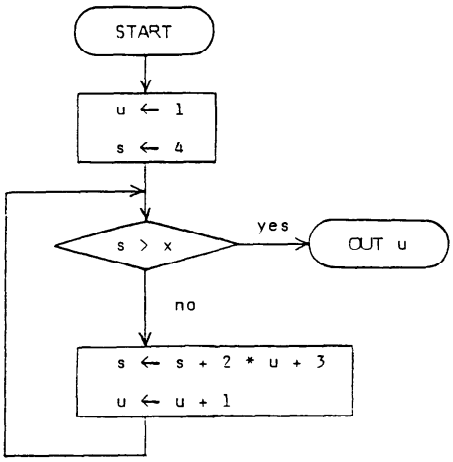

Fig. 1 A Loop for $(\mu y\ (1\ 2 \ldots x)\ ((y+1)*(y+1)>x))$.

```
(INTEGER-SQUAREROOT
  [LAMBDA  (X)
    (PROG  ((U0061  1)
            (A0067  1)
            F0069)
           (SETQ F0069  (PLUS (MINUS X)
                              4))
      L100(IF  (IGREATERP F0069 0)
               THEN  (RETURN A0067))
           (SETQ F0069  (PLUS 3 F0069
                              (TIMES U0066 2)))
           (SETQ U0066  (ADD1 U0066))
           (SETQ A0067  (PLUS A0067 1))
           (GQ L100])
```

Fig. 2 The INITEGER-SQUAREROOT program generated by DEFINES.

and $T4$ are sample terms. We assume that the given samples have uniform properties so that we can easily construct a general term.

The problem is divided into four cases, based on the length of the sample terms.

Case 1. The lengths of all the sample terms are the same: that is, for each sample term Ti,

length(Ti) = length($T\langle i+1\rangle$)

Ex. $(X-1\ \ X-2\ \ X-3 \ldots)$

Case 2. The length of the sample term is increasing: that is, for each sample term Ti,

length(Ti) < length($T\langle i+1\rangle$)

Ex. $(X\ \ X*X\ \ X*X*X \ldots)$

Case 3. The length of the sample term varies periodically.

Ex. $(X*(X-1)\ \ (X-2)\ \ X*(X-3)\ \ (X-4) \ldots)$

Case 4. The length of the sample term is decreasing: that is, for each sample term Ti,

length(Ti) > length($T\langle i+1\rangle$)

Ex. $(X+X+X+X\ \ X+X+X\ \ X+X \ldots)$

4.1 Case 1

For this case, we form a "difference sequence" from the given terms. That is, we compare the corresponding symbols of each term, and ignoring the symbols that are the same, we make a new sequence from the symbols that are different. For instance, suppose that $X-1$, $X-2$ are neighboring terms. The first symbol of each term is equal to X, and the second symbol is equal to "$-$". These symbols are ignored. The third symbol of the first term is equal to 1, the third symbol of the second term is 2, and the third symbol of the third term is 3. Thus, we get a difference sequence of (1 2 3 ...). If we can find a general term for the difference sequence, we can substitute that term back into each of the original terms. For instance, in the above example, the general term of the difference sequence is n. Substituting that back into the original terms gives $X-n$, which is the general term for the original sequence.

Of course, the original sequence could yield several separate difference sequences. For instance, if the given sample is

$(X*1+2\ \ X*2+3\ \ X*3+4 \ldots)$

we get two separate difference sequences

(1 2 3 ...) and (2 3 4 ...)

The general terms for these sequences are n and $n+1$ respectively. Therefore, the general term of the original sequence is $X*n+(n+1)$.

There may be several difference sequences, but they each contain atomic symbols rather than complex terms. We distinguish between two cases:

(1) Case 1.1

The elements of the difference sequence are numbers. In this case, the problem is reduced to the number extrapolation.

(2) Case 1.2

The difference sequence is cyclic. For example, a sequence

(SIN[X] COS[X] SIN[X]...)

gives a difference sequence (SIN COS SIN). In this case, the general term is represented by a conditional expression as follows,

(if $n=1$ (modulo 2) then SIN else COS)

Therefore, the general term of the original sequence is

(if $n=1$ (modulo 2) then SIN[X] else COS[X])

The implemented version of the system does not handle this case.

4.2 Case 2

(The length of the sample term is increasing.) In this case, we decompose the original problem into several problems of formula extrapolation. We introduce a binary relation $S1 \ll S2$ for formulas $S1$ and $S2$, meaning that $S1$ is a subformula of $S2$.

Ex. $X+Y \ll 3+X+Y-4$

If the length of the term is increasing, we cannot always give a general term as a simple algebraic expression of n. However, we present here an algorithm to construct the n-th term of the sequence for a given n.

We explain our method by examples. Suppose that the given samples are,

$T1=X+1$, $T2=X*X+2$, $T3=X*X*X+3$,
$T4=X*X*X*X+4$.

First Step

At first, we try to find out the formula sequence $S=(S1\ S2\ S3)$ which satisfies the following two properties:
For given samples ($T1\ T2\ T3\ T4$),

(1) $(S1 \ll T1)$ and $(S1 \ll T2)$,
$(S2 \ll T2)$ and $(S2 \ll T3)$,
$(S3 \ll T3)$ and $(S3 \ll T4)$,

(2) $S1 \ll S2 \ll S3$.

If there are many sequences with the above properties, we select the sequence in which the lengths of terms are maximum. In our example,

$S1=X$, $S2=X*X$, $S3=X*X*X$

that is, $S=(X\ \ X*X\ \ X*X*X)$.

Second Step

Let Ti' be the result of replacing Si in Ti by some new constant A. Then we have a new sequence T'. In our example,

$T1'=A+1$, $T2'=A+2$, $T3'=A+3$, $T4'=A+4$

Assume that we get a n-th term $Tn'=A+n$, then we can construct a n-th term of the original sequence by substituting the n-th term of S for A in Tn'. In general, the sequence T' may be length-increasing again, but it is simpler than the original sequence because the length of each term is decreased. Therefore, the original problem has been reduced to the two formula extrapolation problems, T' and S.

Third Step

In this step, we construct a general term of S. Since the sequence S has a property

$S1 \ll S2 \ll S3$

we can derive another sequence $S'=(S2'\ S3')$ by replacing $S\langle i-1\rangle$ in Si by some new constant B. In our example,

$S1=X$, $S2=X*X$, $S3=X*X*X$,

therefore,

$S2'=B*X$, $S3'=B*X$

We have another sequence extrapolation problem S'. Assume that we get the n-th term Sn' of the sequence S' by a recursive call on the formula extrapolator.

The term $S2$ is constructed by substituting $S1$ for B in $S2'$ and the term $S3$ is constructed by substituting $S2$ for B in $S3'$. Similarly, we can get the n-th term of S by iterating the substitution $n-1$ times beginning from $S2'$. In our example,

$Sn'=B*X$, and $S1=X$

therefore, by substituting X for B in $S2'$,

$S2=X*X$

and, by substituting $X*X$ for B in $S3'$,

$S3=X*X*X$

and so forth, to get finally Sn.

Fourth Step

For given n, construct n-th term of T', and substitute Sn for A in Tn'. Then we can derive the n-th term Tn of the original sequence.

Another example: Suppose that given samples are,

$T1=1$, $T2=1*2$, $T3=1*2*3$, $T4=1*2*3*4$

then,

$S1=1$, $S2=1*2$, $S3=1*2*3$

by replacing Si in Ti by a symbol A, we get T' such that

$T1'=A$, $T2'=A$, $T3'=A$, $T4'=A$

therefore, we can infer that $Tn'=A$.

On the other hand, by replacing $S\langle i-1\rangle$ in Si by a symbol B, we get,

$S2'=B*2$, $S3'=B*3$

therefore, the n-th term of S' is $Sn'=B*n$.

The n-th term of S is constructed by iterating substitution $n-1$ times. Therefore,

$S1=1$ ··· by the given sample
$S2=1*2$ ··· by substituting $S1$ for B in $S2'$
$S3=1*2*3$ ··· by substituting $S2$ for B in $S3'$
and so forth.

By substituting Sn for A in Tn', we get the n-th term of the original sequence.

4.3 Case 3

(The length of the sample term varies periodically.) In this case, the sequence is decomposed into several sequences that satisfy the condition of Case 1 or Case 2, and a general term is represented in a conditional expression in a similar way to Case 1.2. However, in the present version, this case is not implemented.

4.4 Case 4

(The length of the sample term is decreasing.) In this case, the sequence is finite. Therefore, it does not seem to be useful. This case is not implemented.

5. Implementation

The experimental system was written in QLISP[11], and several examples have been successfully run. In many cases, readability or understandability of programs written in sequence forms can be improved very much, compared with that of ordinary programs with iteration or recursion. Moreover the notation together with hierarchical program decomposition technique allows us to grasp at a glance the structure of even multiply nested iterative algorithm, abstracting the procedural detail; one could easily find the structural similarity between merge-sort and Fast-Walsh-Hadamard transform algorithms using the sequence forms (listed in Appendix).

We intend to use the system as one of the LISP packages which would present us a convenient programming environment. We also aim at incorporating our system into some formula manipulation system.

At present, however, there exists the limitation of the

use of sequence forms for program description owing to the power of the formula extrapolator. The problem has two aspects; one is the algorithmic limitation of formula extrapolation for each sequence, and the other is how to resolve the ambiguity, that is, the way to distinguish one sequence from many others which we could infer reasonably from the given finite number of sample terms.

The formula extrapolator, in the current implementation, can treat algorithmically up to fourth order of arithmetic and geometric progressions for number sequences, but practically the extrapolation for simple arithmetic and geometric progressions is sufficient in most cases. Because such number sequences often appear as subscripts for vectors and the like, especially in the iterative algorithm. For sequences of general formula, the extrapolator tries to apply the above described algorithms which essentially make use of only the symbolic difference between neighboring terms.

As for sequences which the above algorithms fail to extrapolate, the system gives rise to exceptions to break its execution. In another case, when the algorithm finds the ambiguity in the sequence caused by insufficiency of sample terms, the system arbitrarily use the strategy to choose the simplest one from the alternatives. For instance, given the sequence (1 2 ...), we should assume it to be extrapolated as (1 2 3 4 ...) instead of (1 2 4 8 ...) or others. Thus, rather peculiar sequences such as prime numbers, Fibonacci series, etc. are to be excluded from our ordinary usage of sequence forms.

6. Concluding Remarks

A program specification and its evaluation method using ellipsis notation which is based on a formula extrapolator has been described.

The ultimate goal of the work presented is the simplification of programming tasks by utilizing and mechanizing human ability of inductive inference. Similar approaches have been studied in Simon's work on sequential patterns [8], in the works on program construction from examples [9][5] and in the recent work on inductive inference by Shapiro [7].

Acknowledgements

The authors wish to thank Dr. R. Waldinger at Stanford Research Institute. The idea of a specification method of program by a sequence was given by him and the initial system which allows the program writing using sequence has been built at Stanford Research Institute. This is an extensional work to that system [4]. The authors would also like to thank Dr. C. Green at System Control Inc. for his invaluable comments and suggestions.

References

1. Barron, D. W. and Strachey, C. Programming, in Advances in Programming and Non-numerical Computation, Pergamon Press (1966).
2. Boyer, R. S. and Moore, J. S. A Computational Logic, Academic Press (1979).
3. Burstall, R. M. and Darlington, J. A transformation system for developing recursive program, *J. ACM* Vol. 24, No. 1, 44–67 (1977).
4. Fusaoka, A. and Waldinger, R. Program Writing Using Sequences, *SRI memo* (1974).
5. Green C. et al. Progress report on program understanding systems, AIM-740, Stanford Univ. (1974).
6. Henderson, P. and Morris, J. H. Lazy evaluator, *Conf. Rec. 3rd ACM sympo.* POPL, 95–103 (1975).
7. Shapiro, E. Y. Inductive inference of theories from facts, Research Report 192, Yale Univ. (1981).
8. Simon, H. A. Human acquisition of concepts for sequential pattern, in Models and Thought, Yale, 263–273 (1979).
9. Summers, P. D. A methodology for LISP program construction from examples, *Conf. Rec. of 3rd ACM sympo.* POPL, 68–76 (1975).
10. Turner, D. A. Recursion equations as a programming language, in Functional Programming and its Applications, Cambridge Univ. Press, 1–28 (1982).
11. Wilber, M. A QLISP Referential Manual, Tech. Note 118, SRI, Menlo Park, CA. (1976).

Appendix

1. Merge-Sort Algorithm

```
DEFINE[SORT (LAMBDA(X)(CAR(SORT2
        (SORT1 X)
        (ADD1(FIX(QUOTIENT(LOG(SUB1
        (LENGTH X)))(LOG 2]
DEFINES[SORT1 (SLAMBDA(X1 X2 ... XN)
                ((LIST X1)(LIST X2)...(LIST XN]
DEFINES[SORT2 (SLAMBDA(X TILL)
                (SORT3...(SORT3 X)...]
DEFINES[SORT3 (SLAMBDA(X1 X2 ... XN)
              ((MERGE X1 X2)(MERGE X3 X4)...
                (MERGE X⟨N−1⟩XN]
DEFINE[MERGE (LAMBDA(X Y)
        (COND ((NULL X)Y)
              ((NULL Y)X)
              ((GREATERP(CAR Y)(CAR X))
               (CONS(CAR X)(MERGE
                           (CDR X)Y)))
              (T(CONS(CAR Y)(MERGE X
                           (CDR Y]
```

2. Fast-Walsh-Hadamard Transform Algorithm

```
DEFINE[FWHT (LAMBDA(X)(CAR(FWHT2
              (FWHT1 X)
               (FIX(QUOTIENT(LOG
               (LENGTH X))(LOG 2]
DEFINES[FWHT1 (SLAMBDA(X1 X2 ... XN)
               ((LIST X1)(LIST X2)...(LIST XN]
DEFINES[FWHT2 (SLAMBDA((X1 X2 ... XN)
               TILL)
                (FWHT3...(FWHT3 X)...]
DEFINES[FWHT3 (SLAMBDA(X1 X2 ... XN)
      ((APPEND(FWHT4 X1 X2)(FWHT5 X1 X2))
       (APPEND(FWHT4 X3 X4)(FWHT5 X3 X4))
      ...
       (APPEND(FWHT4 X⟨N−1⟩ XN)(FWHT5
      X⟨N−1⟩ XN]
```

DEFINES[FWHT4 (SLAMBDA((*X*1 *X*2 ... *XN*)
(*Y*1 *Y*2 ... *YN*))
((*X*1 + *Y*1)(*X*2 + *Y*2)...(*XN* + *YN*]

DEFINES[FWHT5 (SLAMBDA((*X*1 *X*2 ... *XN*)
(*Y*1 *Y*2 ... *YN*))
((*X*1 − *Y*1)(*X*2 − *Y*2)...(*XN* − *YN*]

(Received January 21, 1982: revised July 23, 1982)