数理解析研究所講究録
第363巻 1979年 188-217

Associative Data Structures and Their Applications.

Masayuki Suzuki*, Eiichi Goto* **,
and Nobuyuki Inada**

*: Dept. of Information Science,
University of Tokyo,
Bunkyoku Tokyo 113 Japan.
**: Dept. of Information Science,
Institute of Physical and Chemical Reseach,
Wakoshi 351 Japan.

Abstract.

The unique representation of arbitrarily nestable ordered tuples and unordered tuples, called GID(Generalized IDentifier), is described. The LISP flag scheme and property list scheme are generalized to AMT(Associative Membership Table) and CAT(Content Addressed Table) with GIDs. Associative operations on these data types are described. Slow but operationally equivalent LISP simulator of GID, AMT and CAT is described. The effectiveness of GID, AMT and CAT is demonstrated in some programming examples.

1.Introduction

FLATS is a machine[1] for scientific computations in both numerical and algebraic fields equipped with hardware for associative capabilities, list processing, overflow free and variable precision computations. "FLATS" means "Fortran" and "Lisp" machine with "Associative" features for "Tuples" and "Sets". In this paper, we describe "ATS" part of FLATS: associative data types, basic operations and their applications. Associative capabilities for tuples(ordered n-tuples) and sets(unordered n-tuples) were implemented in HLISP [2], which is the software version for "L"-part of FLATS. These capabilities, in FLATS, are implemented by using parallel hashing hardware which consists of multiple RAM banks, hash address generators and hashing control units [3],[4].

The FLATS machine is designed to efficiently execute programs written in two languages: HLISP and BFORT(Big FORTran, i.e., FORTRAN with big number features). Portable compilers will also be written so as to enable these two languages to be run on other machines. We shall describe the associative capabilities of HLISP from the apllication programmer's view.

In HLISP, H(Hashed)-type, R(Rom)-type and two more data types, AMT(Associative Membership Table) and CAT(Content Addressed Table), are added to the data types of the "Standard LISP"[5] as shown in Table 1. H-types, AMTs and CATs which involve hashing for efficient implementation, are treated in this paper. The other types will be discussed in another paper to follow.

In hashed scheme, structural data(HVECT, HPAIR, HAMT and HCAT), are uniquely represented by making use of hashing. Equality checks for H-type data can be made very fast by pointer equality checking because of their uniqueness. In LISP, each literal atom is uniquely represented by making use of "OBLIST" mechanism, thereby enabling the equality check of two atoms to be made with the fast pointer comparison function "eq". On the other hand "equal", which is the structural equality checking function for ordered n-tuples, become quite cumbersome and time consuming. Further, that for mathematical sets(unordered n-tuple) would become even more time consuming. We describe in 2 that by virtue of H-type the equality checks for structural data(ordered n-tuple and unordered n-tuple) can be reduced to the fast pointer equality checking. H-type data are also termed GID(Generalized IDentifier), since they can be used in the same way as identifiers in LISP: flag or property list functions become applicable to GIDs in our "hashed" system. The data types, AMTs and CATs, which respectively represent unordered tuples and content addressed table, are introduced as basic building blocks for associative data structures and associative capabilities. The operations on AMTs or CATs, called AMT OPs or CAT OPs, are also constructed on the unique representation of GIDs. In 2, we describe H-type data, AMTs, CATs and the basic operations on them.

In 3, we give programming examples involving basic AMT and CAT OPs, h-OPs and GIDs placing emphasis on the best speed and time complexity. The first few are for "word count", "collated word count" and "bookkeeping of athletic clubs" having business

and data base management flavors. The last few are for polynomial manipulation. In spite of their apparent differences in flavors, the programs would turn out to be strikingly alike. They would also turn out to be fast and clean, thereby showing the importance and appropriateness of concepts and OPs relating to AMTs and CATs.

Since the clarification of concept, speed and time complexity results are the main theme of this paper, the details of the language design and implementational aspects of FLATS will not be treated.

Comparisons of data types, basic OPs and programming concept formulated in this paper with those in other languages, such as LISP, PASCAL, LEAP and SETL are discussed in 4.

2. Built-in Data Types and Basic OPs

In this section, we describe data types and basic OPs which form the basis for associative computations. Table 1 shows built-in data types of FLATS, which are classified into three types; LISP-, ROM- and H-types.

2.1 Uniquely Represented Data Types (H-type).

We introduce H-type data with the following characteristics:

(H1) Arbitrarily nested trees, lists(ordered tuples) and sets (unordered tuples) can be transformed into H-type; only one internal representative is created for each H-type datum.

(H2) Equality of two H-type data can be checked very fast(O(1) time) by the pointer checking function "eq".

Although these characteristics are realized by making use of hashing to improve time complexity, the hashing mechanism is designed to be invisible to users. Hence, a slow(O(n)) but operationally equivalent LISP simulator of H-type data is shown in the main text and a fast O(1) hash implementation is given in the Appendix.

Let h(x) be a function which turns LISP datum(S-expression), x, into a unique form so that

(H3) The pointer equality eq(h(x),h(y)) = T holds iff the equality equal(x,y) = T holds.

This can be simulated in LISP by constructing a "*HOBLIST". The "*HOBLIST" is the list $(h_m, h_{m-1}, \ldots, h_1)$ of all data which have

been subjected to the function h. Namely, the "*HOBLIST" is to be seen only from the LISP system (hereinafter, "*" prefixed literal atom is for system use only). "h(x)" is executed in the following way:

(H4) if x ATOM then h(x)=x.

Otherwise, "*HOBLIST" is searched with the structural equality checking function "equal":

if x matches, say, with h_i, then $h(x)=h_i$,

if no match, a copy of x, h_{m+1}=copy(x) is added to the "*HOBLIST" and $h(x)=h_{m+1}$.

(H5) h(x) in a LISP simulator is defined as:

```
h[x]=prog[[y];
                [atom[x] -> return[x]];
                y:=*hoblist;
              A: [null[y] -> go[B];
                  equal[x,car[y]] -> return[car[y]]];
                y:=cdr[y];
                go[A];
              B: x:=copy[x];
                *hoblist:=cons[x;*hoblist];
                return[x]]
```

For example, let x,y and z be lists as:

```
x:=list[A;B;C]          =(A B C)
y:=list[A;B;C]          =(A B C)
z:=list[A;B;D]          =(A B D)
```

For x,y and z, the pointer equality checking by "eq" is:

```
eq[x;y]=eq[x;z]=eq[y;z]=NIL
```

and the equality by "equal" is:

equal[x;y]=T

equal[x;z]=equal[y;z]=NIL.

Eq takes O(1) time but it fails in the structural equality checking. Equal can check the structural equality but it takes O(n) time for lists of length n.

For H-type lists, hx,hy and hz:

hx:=h[x] =(A,B,C)

hy:=h[y] =(A,B,C)

hz:=h[z] =(A,B,D)

the structural equality can be checked with fast O(1) time:

eq[hx;hy]=T

eq[hx;hz]=eq[hy;hz]=NIL

A considerable speed up can be achieved by using H-type when equality checks are made heavily.

2.2 AMT(Associative Membership Table)- A Generalization of the LISP Flag Scheme.

The notion of sets, a very important concept in pure mathematics, has been introduced into a number of programming languages such as LEAP[6],SETL[7] and PASCAL[8]. Historically, LISP was the first to incorporate set operations. Namely, the LISP flag scheme can be regarded as set operations. For example, let atom A be flagged with(have the following flags on its property list) TOM, DICK and HARRY; atom B be flagged with JAPAN, UK and USA. This can be interpreted as: a set {TOM DICK HARRY} with a global name A; a set {JAPAN UK USA} with a global name B. In this regard the flag functions, flagp, flag and remflag, can

be regarded as operations of membership test, element insertion and deletion, respectively.

Experiences in using the flag scheme, however, have revealed the following disadvantages:

(F1) Slow speed of operation. Because list processing is used in the implementation, O(n) time is needed for flag operation when n flags(set with n elements) are involved.

(F2) The global nature of the name of a set often causes name crashes and inconveniences in garbage collection.

(F3) Only literal atoms can be used as set elements(i.e., flags in LISP terminology).

The data type "AMT", which has the following characteristcs, was devised to overcome these disadvantages.

(A1) AMT OPs are fast O(1) time by making use of hashing.

(A2) An AMT has no global name. Note that global naming of array in Lisp 1.5 also has the same disadvantage. In Standard LISP, vectors without global names have been devised to overcome this disadvantage.

(A3) Any GID can be used as an AMT element which corresponds to a flag in the flag scheme. The H-type data as defined in (H1), (H2) and (H3), i.e., the unique forms of S-expressions, are GIDs. HAMT and HCAT to be defined later are also GIDs.

We define an AMT as the table having a current member set. We use the set notation:

$$\{ g_1^* \; g_2^* \; \cdots \; g_n^* \}$$

as the preferred standard external representation for the abstract data type AMT, where g_i^* is the external notation for a

GID. Another external notation :

$$(*AMT\ g_1^*\ g_2^*\ \ldots\ g_n^*)$$

will be used for a LISP simulator reflecting the specific data structure.

The basic OPs on an AMT are defined as follows where a AMT and m∈GID:

a:=mkamt(): make a new AMT, initially an empty set.

getamt(a,m)=[m∈a]:member ship test.

putamt(a,m)=[a:=a∪{m}]:insertion.

remamt(a,m)=[a:=a-{m}]:deletion.

The following is a LISP simulator for the AMT OPs.

(A4) For a AMT, m GID,

```
mkamt[]=cons[*AMT;NIL]
getamt[a;m]=member[a;cdr[a]]
putamt[a;m]=[getamt[a;m] -> a; T -> nconc[a;list[m]]]
remamt[a;m]=[null[cdr[a]] -> NIL;
             equal[cadr[a];m:] -> rplacd[a;cddr[a]];
             T -> remamt[cdr[a],m]]
```

2.2.1 HAMT or H-type AMT.

An HAMT is the unique representation of an AMT. Equality of two AMTs corresponds to that of sets. Let h(x) be the function which gives the HAMT corresponding to AMT x.

(HA1) "h" can be simulated in LISP by enlisting HAMTs on "*HOBLIST" similarly as in (H4) and (H5). The change to be made in (H5) is to use "seteq" instead of "equal" whenever the first list argument of x is *AMT.

```
seteq[x;y]=[(x⊂y)∧(y⊂x)]
x⊂y=[null[x]->T;
      car[x]∈y -> cdr[x]⊂y;
      T -> NIL]
```

For example, let x be an AMT, {B A C}(or (*AMT B A C) in the specific LISP simulator data structure), and y be {A B C}(or (*AMT A B C)). Suppose AMTs x and y are created as:

Program	Value in the LISP simulator
x:=mkamt[]	=(*AMT)
x:=putamt[x;B]	=(*AMT B)
x:=putamt[x;A]	=(*AMT B A)
x:=putamt[x;C]	=(*AMT B A C)

similarly,

y:=mkamt[]	=(*AMT)
. . .	
y:=putamt[y;C]	=(*AMT A B C)

For these x and y,

```
eq[x;y]=equal[x;y]=NIL
seteq[x;y]=T
hx:=h[x]              =(*AMT,B,A,C)
hy:=h[y]              =(*AMT,B,A,C)
eq[hx,hy]=T
```

Equality checks of two sets by the list processing function "seteq" is a time consuming operation($O(n^2)$ time for sets with n-elements), because the same set can be represented in many different ways such as {A B C}={B C A}= ... . A hashing method for a fast(O(1)) equality checking of sets was first given in

[9].

2.3 CAT(Content Addressed Table) - A Generalization of the LISP Property List Scheme.

Content addressed table operations have been implemented in LEAP, PASCAL and in some recent data base management systems. Historically, LISP was the first to incorporate such operations. A LISP property list can be regarded as a content addressed table. Namely, let atom A have indicators APVAL, EXPR and PNAME with respective properties 137, (LAMBDA(X)...) and A. This can be interpreted as: a content addressed table with a global name A which maps contents (symbolic addresses) APVAL, EXPR and PNAME into 137, (LAMBDA(X)...) and A, respectively.

The LISP property list scheme, however, has exactly the same disadvantages (F1),(F2) and (F3) of flags. The data type "CAT" was devised to overcome these disadvantages by imparting the same characteristics (A1), (A2) and (A3) of AMTs. Moreover, in order to overcome an inconvenience of the property list scheme when the indicator(content) is missing, an "excise value" convention has been introduced: each content addressed table has each "excise value" which is used at content missing.

We define a CAT as a system of an excise value, eϵGID, and a mapping, M, from a set of distinct GIDs, a=$\{g_1^* \ldots g_n^*\}$, into any HLISP data, $v_1^*, v_2^*, \ldots, v_n^*$, of any type. We use

$$(e\ \{g_1^*:v_1^*\ g_2^*:v_2^*\ \ldots\ g_n^*:v_n^*\})$$

as the standard external notation for a CAT. Another notation for a CAT

(*CAT e $(g_1^*.v_1^*)$ $(g_2^*.v_2^*)$ ... $(g_n^*.v_n^*)$)

will be also used for a LISP simulator.

The basic CAT OPs are defined as follows:

for $c \in CAT$, $m \in GID$, e(excise value)$\in GID$ and $a=\{g_1^* \ldots g_n^*\}$,

c:=mkcat(e):	make a new CAT with excise value $e \in GID$
getcat(c,m):	if $m=g_i^* \in a$ then v_i^*, else e
putcat(c,m,v):	if v=e and $m=g_i^* \in a$ then remcat(c,m) if v=e and $m=g^* \notin a$ then no change occurs in c. if $v \neq e$ and $m=g_i^* \in a$ then change the mapping $(g_i^*.v_i^*)$ into $(g_i^*.v^*)$. else add the mapping (m*.v*) to c.
remcat(c,m):	remove the mapping entry (m*.v*) from c.

We now show a LISP simulator for CAT OPs:

```
mkcat[e]=list[*CAT;e]
getcat[c;m]=prog[[x];
                 x:=assoc[m;cddr[c]];
                 [null[x] -> return[cadr[c]];
                  T -> return[cdr[x]]]
putcat[c;m;v]=prog[[x];
                   x:=assoc[m;cddr[c]];
                   [v=e∧x -> remcat[c;m];
                    v=e∧not[x] -> NIL;
                    v=e∧x -> rplacd[x;v];
                    v=e∧not[x] -> nconc[c;list[cons[m;v]]]];
```

```
                    return[v]]
remcat[c;m]=[null[cddr[c]] -> NIL;
            equal[caaddr[c];m] -> rplacd[c;cdddr[c]];
            T -> remcat[cdr[c];m]]
```

2.3.1 HCAT or H-type CAT.

An HCAT is the unique representation of a CAT. Two CATs, $c=(e\ \{g_1^*:v_1^*\ \ldots\ g_n^*:v_n^*\}$ and $c'=(e'\ \{g_1^{*\prime}:v_1^{*\prime}\ \ldots\ g_m^{*\prime}:v_m^{*\prime}\})$, are defined to be equal if the two excise values are equal, i.e., eq(e,e')=T (note $e,e' \in$ GID so that "eq" can be used consistently) and the two mappings are identical, i.e., the two sets of pairs

$$M= \{(g_1^*.v_1^*)\ (g_2^*.v_2^*)\ \ldots\ (g_n^*.v_n^*)\} \text{ and}$$

$$M'=\{(g_1^{*\prime}.v_1^{*\prime})\ (g_2^{*\prime}.v_2^{*\prime})\ \ldots\ (g_m^{*\prime}.v_m^{*\prime})\}$$

are the same set. Let h(x) be the function which gives the HCAT corresponding to CAT x. "h" can be simulated in LISP by enlisting HCATs on "*HOBLIST" similarly as in (H4), (H5) and (HA1). The changes to be made in (H5) is to use "equal" for checking equality of excise values and "seteq" in (HA1) for that of mappings instead of "equal" whenever the first argument of list x, is *CAT. For example,let x be a CAT, (0 {A:1 B:2}) or (*CAT 0 (A.1) (B.2)), and y be (0 {B:2 A:1}) or (*CAT 0 (B.2) (A.1)). Suppose CATs, x and y, are created as:

Program	Value in the LISP simulator
x:=mkcat[0]	=(*CAT 0)
x:=putcat[x;A;1]	=(*CAT 0 (A.1))
x:=putcat[x;B;2]	=(*CAT 0 (A.1) (B.2))
y:=mkcat[0]	=(*CAT 0)

```
y:=putcat[y;B;2]     =(*CAT 0 (B.2))
y:=putcat[y;A;1]     =(*CAT 0 (B.2) (A.1))
```

For these x and y,

```
hx:=h[x]             =(*CAT,0,(A.1),(B.2))
hy:=h[y]             =(*CAT,0,(A.1),(B.2))
eq[x;y]=equal[x;y]=NIL
eq[cadr[x],cadr[y]]=T
equal[cddr[x];cddr[y]]=NIL
seteq[cddr[x];cddr[y]]=T
eq[hx;hy]=T
```

2.4 HINTeger.

Equality checks on long integers(multiprecision) would be also important when they are used as ID(identification) numbers. In HLISP, long integers can be also transformed into H-type by the function "h".

3. Applications.

While AMT OPs realize set operations, CAT OPs realize a mechanism for associating values to GIDs. In this section, we give several examples which make use of GIDs, AMTs and CATs OPs.

3.1 Sweep OP or Forall OP.

We define "sweep" or "foreach" OP such as "(foreach M in X (<body>))". This OP is the heart of programming examples in this section. The syntax of "foreach" is:

(foreach <FORMAL> in <ID> (<body>))

The semantics is "let X be an AMT, $X=\{g_1^* \ldots g_n^*\}$, or a CAT, $X=\{g_1^*:v_1^* \ldots g_n^*:v_n^*\}$. Execute the <body> n-times with changing the formal M in the <body> as $M=g_i^*$, i=1,2,...,n upon each execution". This can be defined in a LISP simulator as follows:

```
(foreach m in x (<body>))=
    [car[x]=*AMT -> mapc[cdr[x];
                         λ[[m];prog[[]; <body>]]];
     car[x]=*CAT -> mapc[cddr[x];
                         λ[[m];prog[[]; m:=car[m];
                                        <body>]]]]
```

We now give some functions with foreach. We shall use type declaration with syntax "type(x∈CAT)" so as to improve readability of the program. Moreover, when x is type declared to be a CAT we shall use simply write "x(m)" instead of "getcat(x,m)" and

"x(m):=v" instead of "putcat(x,m,v)".

```
SIZE(X)::=(begin
                comment D is a dummy formal;
                type(X∈(AMTUHAMTUCATUHCAT));
                N:=0;
                (foreach D in X (N:=N+1));
                return(N);
            end)

CATCOPY(X,E)::=
            (begin
                type(X∈(CATUHCAT);C∈CAT);
                C:=mkcat(E);
                (foreach M in X (C(M):=X(M)));
                return(C);
            end)

CATADD(X, Y)::=
            (begin
                type(X,Y∈(CATUHCAT);C∈CAT);
                C:= CATCOPY(X,0);
                (foreach  M in Y (C(M):=C(M)+Y(M)));
                return (C);
            end)

CATSUB(X,Y)::=
            (replace "+" in CATADD by "-")
```

SIZE gives the size(cardinality) of X(AMT or CAT). Hereinafter for showing time complexities(for the fast hash method but not for the LISP interpreter), the size of data are denoted by corresponding lower case letters and specially the average length of pertinent data, by k. CATCOPY gives a new CAT with excise value E and takes O(x) time.

3.2 A "Deposits into Accounts" example.

CATADD can be used for this purpose. Namely, with X={TOM:10 JIM:20} meaning the account of TOM and JIM, and with Y={BOB:5 TOM:6} meaning deposits by BOB and TOM,

"CATADD(X,Y)={TOM:16 JIM:20 BOB:5}"

would update the accounts. Similarly, CATSUB can be used as a "draws from accounts" function. Since any GID can be used as an account identifier, account numbers(HINT) can be used instead of names(HSTR) as well.

3.3 A "Word(GID) Count" example.

Let file, T, be the record of a beginner typing exercise, supposed to be ten repetitions of "THE QUICK BROWN FOX JUMPS OVER A LAZY DOG". The following program would make word counts in CAT(0), C with O(t) time:

```
"C:=mkcat(0);
 type(C∈CAT);
 (foreach M in T (C(M):=C(M)+1));"
```

"foreach" is also applicable to a file (upon each loop, M takes on an "h-ed" value of each file item). Let paa(for print with actual arguments) be a print function and lexsort be a lexicographic sorting function.

Now, "paa(C);" would print

"C={THE:8 QUICK:10 ... }"

and "paa(lexsort(X));" would print

"lexsort(X)=((A,10),(BROWN,10),(DOG,8),(FOX,10),
(HTE,2),...,(SOG,2),(THE,8))"

showing mistouch habits in "HTE" and "SOG". It would be more convenient, however, to print those words which do not appear in another file, D called the dictionary("THE QUICK BROWN FOX JUMPS OVER A LAZY DOG" is the content of D in this case).

3.3 A "Collated word (GID) Count" example.

```
A:= mkamt();
C:= mkcat(0);
type(A∈AMT;C∈CAT);
(foreach M in D (putamt(A,M));
(foreach M in T
        (if M A then C(M):=C(M)+1 fi));
paa(lexsort(C));
```

would print out

"lexsort(x)=((HTE,2),(SOG,2))"

3.4 An example of "Bookkeeping of Clubs".

Suppose that two CATS, S and D have been created for the bookkeeping of a swimming club and a diving club. For registering TOM in the swimming club with fee 5, one would execute

```
"S("TOM"):=5"
```

with linear time O(k), where k is the number of characters, k=3 in this case. For terminating JIM's membership in the diving club, one would execute

```
"D("JIM"):=0"
```

with linear time O(k), (k=3). Suppose on December 31 we need to make the frozen records of the clubs, we would execute

```
"S12:=h(S);
 D12:=h(D);"
```

on that data with linear time O(s) and O(d). Suppose S1,S2,...,S11 and D1,D2,...,D11 are the frozen records for each month. The identity of membership now be checked with O(1) time. For example, in case there were no changes during December, "paa(eq(S12,S11));" would print "eq(S12,S11) = T". If a great number of similar questions have to be answered, the use of "h"-ed form(HCAT) would be quite advantageous, since O(k*s) time or even more, instead of O(1), would be needed in other identity check methods. An individual may be members of both clubs. For summing up the fees,

```
"T12:=h(CATADD(S12,D12));
 paa(lexsort(T12));"
```

would print

"((BOB,5),(JIM,20),(TOM,16))"

with times O(sl2+dl2) for CATADD, O(tl2) for "h".

3.5 A "Polynomial Adder" example.

Let us consider polynomials P:

$$P = \sum_{i=1}^{n} C_i T_i ,$$

said to be ST(for Sum of Term) polynomials, where$\in C_i$ HINT are called coefficients and $T_i \in$GID, term identifiers. Assuming no duplication among T_i's, P can be represented as an HCAT,

$$P=ST(P)=\{(T_i,C_i) \mid 1\leq i\leq n\},$$

called the ST HCAT form, e.g.,

P=ST(2A + 3B + 4C)

=$\{$A:2 B:3 C:4$\}$.

The same polynomial can be expressed in many different ways as

2A + 3B + 4C = 3B + 2A + 4C = ...

due to the commutativity of addition. Nevertheless, the HCAT, P=ST(P) is a unique representation, thereby enabling the identity check of two polynomials P=ST(P) and Q=ST(Q) to be made in O(1) time by an equality check of pointers, eq(P,Q). The CAT(0), P'=CATCOPY(ST(P),0) will be called the ST CAT form of P. The function CATADD readily provides as adder for two polynomials given in ST HCAT/CAT forms. Namely, "R':=CATADD(P,Q)" gives the result R' in the ST CAT form with linear time O(p+q). Extra linear time O(r') without fully using of hashing would be needed for "h"-ing R' into the SP HCAT . Hence the h-op will be used only when it is needed.

As the special case, when each term identifier T_i has the form

$$T = V_1^{E_1} V_2^{E_2} \ldots V_n^{E_n}$$

with $V_j \in$ GID and E_j being positive integers for $1 \leq j \leq n$; T_i is said to be a multivariate term and $P = \sum C_i T_i$ is said to be an SP(for Sum of Products) polynomial. T_i may be represented in different ways as in

$$T_i = A^3B^2C^1 = B^2A^3C^1 = \ldots$$

due to the commutativity of multiplication. Nevertheless, the HCAT {A:3 B:2 C:1} gives a unique representation. Hence, SP polynomials can be represented uniquely by doubly nested HCATs, to be called SP HCAT forms, as in

SP($5A^3B^2C^1 + 6X^1 + 7$)

={{A:3,B:2,C:1}:5,{X:1}:6,{}:7}

using the term of 3.2, One may say "HCATs", T_i ($T_i \in$ HCATCGID) are to be used as account identifiers in SP polynomial additions".

3.6 A "Polynomial Multiplication" example.

We now define a function CATMUL(P,Q) for multiplying two polynomials P and Q given in SP HCAT/CAT forms. Note that the multiplication of two term identifiers can be made with the function CATADD, e.g., for

$T_1 = A^2B^3$, $T_2 = A^1C^4$ and $T_1T_2 = A^3B^3C^4$, we get

h(CATADD({A:2 B:3},{A:1 C:4})

={A:3,B:3,C:4}.

Hence we readily have(the result is in the SP CAT form):

```
CATMUL(P,Q)::=
            (begin
                type(C,P,Q∈(CAT∪HCAT);T∈HCAT);
                C:= mkcat(0);
                (foreach TP in P
                   (foreach TQ in Q
                     (T:= h(CATADD(TP,TQ));
                      C(T):=C(T)+P(TP)*Q(TQ))));
                return (C);
             end)
```

which runs with $O(\sum t_{p,i} * \sum t_{q,i})$ given in [10],[11].

4. Concluding Remarks.

We described the associative data types built in FLATS, basic OPs on the data types and their applications.

The associative data types and capabilities are constructed by GIDs, AMTs, CATs and h-OP. The concept of GID is the basis of them. AMT OPs can realize fast set operations and CAT OPs can realize the associative tabulation for symbolic and/or integer arguments(GIDs). The h-OP is regarded as the generalization of LISP "intern"-OP for literal atoms and it is extended for arbitrarily nestable ordered and unordered tuples.

Equality check on long integer, ordered and unordered tuples are important for many programs, so we would propose GID and h-OP for fast identity checks when they are heavily used.

AMTs, CATs and their OPs based on "single-hit" association is simple and flexible to build up any associative data structures. In LEAP based on "A,O,V","multi-hit association" is constructed by elaborate data structures consisting of hash table and linked list. The CAT OPs may be regarded as a special subclass of LEAP OPs with data type interpretations, A CAT, O GID and V ANY. The LEAP OPs which are not needed in many cases, can be constructed by CATs and PAIRs(ordered tuple).

The concept of "hashed sets" and their OPs were reported in [9] and their applications to formula manipulation were in [10]; especially equality check for sets was improved(its time complexity is O(1)). In SETL, hashing is also used for identity check. However, it would take a long time(at least linear time) when two sets are equal. The "bit vector" feature in PASCAL

provides O(1) set OPs. It has restrictions, however,that the universe of set must be known at the compile time. The "record" concept in PASCAL is similar to "CAT". However, the structure of "record" must be also given at the compile time, while that of "CAT" is given dynamically at run time. Since the structures(AMTs and CATs) in the examples given in 3 have to be constructed dynamically at run time, the PASCAL "bit vector" and "record" features are not applicable.

The conciseness and clarity of programs in 3 owe much to sweep(foreach) OP on GIDs(i.e., the unique representative of arbitrarily nested ordered and unordered tuples).

The FLATS machine is now under development at the Institute of Physical and Chemical Research. With hardware for parallel hashing, FLATS could execute fast h-OP, AMT OPs and CAT OPs. Then, these OPs would be fully applied to formula manipulation.

Reference.

[1] Goto,E., Ida,T., Hiraki,K., Suzuki,M. and Inada,N., FLATS, A Machine for Numerical and Symbolic and Associative Computing, Proc. of the 6th Annual Symp. on Computer Architecture, (April 1979), 102-110

[2] Kanada,Y., Implementation of HLISP and Algebraic Manipulation Language REDUCE 2, Tech. Rep. of Info. Science, 75-01, Univ. of Tokyo (1975)

[3] Ida,T. and Goto,E., Performance of a Parallel Hashing Hardware with Key Deletion, Proc. IFIP Congress 77, North-Holland(1977)

[4] Ida,T. and Goto,E., Analysis of Parallel Hashing Algorithms with Key Deletion, Jour. Info. Proc. vol 1. No 1 (1978)

[5] Marti,J.B., Hearn,A.C., Griss,M.L and Griss,C., Standard LISP Report, UUCS-78-101, Univ. of Utah, (1978)

[6] Feldman,J.A. and Rovner,P.D., An Algol-Based Associative Language, CACM, 12,(1969)

[7] Schwartz,J.T., On Programming, an Interim Report of the SETL Project, Courant Inst. of Math. Sciences, New York Univ. (1973)

[8] Wirth,N., The Programming Language Pascal, Acta Informatica 1 (1971)

[9] Sassa,M. and Goto,E., A Hashing Method for Fast Set Operations, Info. Letters, 5 (1976) 31-35

[10] Goto,E. and Kanada,Y., Hashing Lemmas on Time Complexities with Application to Formula Manipulation, Proc. ACM SYMSAC 76, York Town Heights NY (1976)

[11] Goto,E., Sassa,M. and Kanada,Y, Algorithms and Programming

with CAM(Content Addressable Memories), Tech. Rept. of Info. Science, 78-04, Univ. of Tokyo (1978).

Appendix. A hash method for faster data handling.

The operations on the "*HOBLIST" in the LISP simulator for the H-type data outlined in the main text can be replaced by a hash table to be called HOBTABLE in order to increase the speed. Each entry of the hash table HOBTABLE consists of a key(hash code) and a pointer to the pertinent H-type data structure. An H-type datum is represented as a pointer to this HOBTABLE entry, and a hash code function, Hc (this is different from hash sequence function to be used for calculating the hash address) is used to generate an integer hash code for given datum x, where $0 \leq Hc(x) \leq M$(maximum integer which fits into a single word of the machine). The function, h, defined in 2, searches, inserts or deletes an H-type datum by using this hash code as the key to determine hash address sequence in the hash process. The hash code function, Hc, is constructed from the following three specific functions, Hp, Ha and Hs. Note that GIDs are constructed by starting from integers and literal atoms and by recursively nesting ordered and unordered tuples.

For integers and literal atoms, any mapping with a suitable pseudo random shuffling property, from integers or bit patterns of literal atoms into the interval [0,M] would do for "Hp".

Let $g_1^*, g_2^*, \ldots, g_n^*$ be GIDs with hash code $h_i = Hc(g_i^*)$. The hash code of a dot-end list,

$$t=(g_1^* \ \ldots \ g_{n-1}^* \ . \ g_n^*)$$

is given in terms of an asymmetric function Ha:

$$Hc(t)=Ha(h_1,Ha(.....Ha(h_{n-1},h_n)))...)$$

where $0 \leq Ha(h_1,h_2) \leq M$ and generally $Ha(h_1,h_2)=Ha(h_2,h_1)$. The hash code for AMT(unordered tuple),

$$a=\{g^*_1 \ ... \ g^*_{n-1} \ g^*_n\}$$

is given in terms of a symmetric (commutative) function Hs, so as to comply with the "unordered" nature of the sets:

$$\begin{aligned} Hc(a) &= Hs(h_1,h_2,h_3,...h_n) \\ &= Hs(h_2,h_3,h_1,...h_n) \\ &= Hs(h_3,h_1,h_2,...h_n) \\ &\quad . \ . \ . \end{aligned}$$

The speed of basic operations on AMTs, getamt, putamt and remamt can also be improved to average O(1) by using a hash table for set elements of each AMT[9].

LISP TYPE	ROM TYPE	HASHED TYPE
ATOM		
STRing	RSTR	HSTR
NUMber		
INTeger		HINT
FLOATing		HFLOAT
FUNCtion pointer	RFUNC	
PAIR	RPAIR	HPAIR
VECTor	RVECT	HVECT
AMT	RAMT	HAMT
CAT	RCAT	HCAT

Table 1. Data Types in HLISP