stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 174/81 AUGUSTUS

P. VERHELST

SOME THOUGHTS ON A TUPLE ORIENTED PROGRAMMING SYSTEM

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B05, 68B20

ACM-Computing Reviews-category: 4.13, 4.22, 4.34

Some Thoughts on a Tuple Oriented Programming System

by

Paul Verhelst

ABSTRACT

A description is given of a medium level language processor based on a single data type: tuples. This system is intended to be used as the lowest level of an interactive programming system. It is shown how abstract data types and syntax extensions may be handled in this system.

KEY WORDS & PHRASES: Programming languages, Interpreters, Programming systems, Data types, LISP

0. INTRODUCTION

A problem with current interactive computer systems is that the user is confronted with many different languages, each having its own types of objects and its own control structures. Examples are programming language, command language, edit language, etc. The non-programming languages are mainly used for operations on special types of objects; command languages allow manipulation of files, editors allow modification of text-files. The multitude of languages is not the biggest problem; in general, knowledge of a few primitive commands is enough to be able to use the system. The main problem is that operations available in one such specialised language are not available in the other languages (and especially not in the programming language). The wish to make such operations programmable has led to developments in two directions:

(1) Some of these operations have been made available to programs in the form of procedure calls (mainly for files).

(2) Specialised languages have been extended with programming language features such as control structures and operations for arithmetic and string handling (examples are the UNIX* shell command language[1] and the TECO editor).

This leads to the paradoxical situation that, although a general purpose programming language has been used to implement a specialised language, the operations of that language are not accessible from the programming language itself. This forces users who need these operations to program in languages that have cryptic and ill-defined semantics. See [2] for an extensive discussion of this subject.

This report proposes to solve this problem by defining a basic framework that supports the definition and manipulation of arbitrary types of objects (including programs). This framework is based on a single primitive data type: tuples. In the same way as is done in LISP we define a translation from programs to tuples and a "tuple processor", which executes these translated programs. This tuple form of programs allows manipulation of programs, and, by mapping the processor state on tuples, program debugging. By defining all new types of objects within this framework, the corresponding operations are automatically available in the programming language.

This report is divided into 4 sections. Section 1 gives an overview of the approach taken in LISP. Section 2 describes the tuple oriented system, and sections 3 and 4 discuss the handling of data types and syntax extensions in the proposed system.

---

*UNIX is a Trademark of Bell Laboratories.

1. THE LISP APPROACH

1.1. S-expressions and Lists

The primitive data types in LISP systems are atoms and S-expressions. Atomic values are arbitrary symbols, some of which denote integer, boolean, and floating point values. S-expressions are pairs of atoms or other S-expressions. That is, we have:

```
<value>  ::= <atom> | <S-expr> .
<atom>   ::= <id> | <int> | <real> | ... .
<S-expr> ::= '(' <value> '.' <value> ')' .
```

To manipulate these values there are a number of elementary functions available:

```
cons(v1,v2) gives S-expression with components v1 and v2;
car(s)      gives first component of S-expression s;
cdr(s)      gives second component of S-expression s;
eq(a1,a2)   compares atoms a1 and a2;
atom(v)     tests if value v is atomic;
```

A special kind of S-expressions is called "list". A list is constructed starting from the atomic value NIL:

```
<list> ::= 'NIL' | '(' <value> '.' <list> ')' .
```

The external representation of lists is "(v1 ... vn)". A list is a sequence of values, from which values may be extracted using a combination of "car" and "cdr" applications.

1.2. The LISP Meta-language

The original LISP report [3] defines a meta-language, which consists of the following constructs (we use a slightly different notation):

```
<form>   ::= <value> | <apply> | <cond> .
<apply>  ::= <fun> '(' [ <form> ( ',' <form> )* ] ')' .
<cond>   ::= '[' <form> '->' <form>
                ( ';' <form> '->' <form> )* ']'
<fun>    ::= <id> | <lambda> .
<lambda> ::= 'lambda' '(' [ <id> ( ',' <id> )* ] ')' <form> .
```

These meta-language constructs are mapped on lists in the following way:

```
<apply>  : (fun form1 ... formn)
<cond>   : (COND (form11 form12) ... (formn1 formn2))
<lambda> : (LAMBDA (id1 ... idn) form)
<value>  : (QUOTE value)
```

The elementary functions are available as CONS, CAR, CDR, etc. LISP systems provide an interpreter that processes programs translated to list-form. This interpreter is available as the function "eval". A LISP system interacts with the user by executing a "read-eval-print" loop, which repeatedly reads an S-expression, evaluates it, and prints the result.

1.3. Discussion

The most distinguishing feature of LISP systems is the equivalence of programs and data. In other computer systems executable programs are represented as sequences of machine instructions stored in files. Such code files can only be created by using compilers and linkers; incremental modification of code files is impossible. In LISP, programs can easily generate, modify, and execute other programs without any restrictions.

A problem in LISP systems is that programs must be entered in list-form, instead of in the meta-language. Consequences are an unattractive syntax and the quoting problem, i.e. list-values in the meta-language must be quoted in the hand-translated list-form. Systems like REDUCE partially solve this problem by providing a more readable representation of LISP programs (at the cost of a greater distance between internal and external representation of programs and difficulties in reconstructing the external form [4]).

The applicative form of the LISP language does not seem general enough. This is evident from the various nonfunctional extensions (RPLACA, SET, PROG, etc.). There are several reasons for these extensions: they are necessary for the implementation of history sensitive systems, they are used to write more efficient "functions", and in some algorithms the sequential notation is just simpler than the functional notation.

Another problem in LISP is the binding of identifiers, which is handled differently in the various implementations of the language. The most efficient binding mechanism for an interpreter is dynamic binding, which always takes the most recent meaning given to an identifier. Compilers produce most efficient code if static binding is used, i.e. if the meaning of an identifier can be determined from declarations of enclosing blocks only. Static binding also seems to be the best scheme for human readers of programs. Implementations of static binding in interpreters must choose between extra work for each variable reference and extra work for each function application (see Baker[5]). An example of a function

that depends on the type of binding used is:

```
(setq x 1)
(def f1 (lambda () x))
(def f2 (lambda (x) (f1)))
```

If static binding is used, (f2 2) produces 1; if dynamic binding is used, it produces 2.

2. TUPLE ORIENTED SYSTEM

Our approach is similar to that taken in LISP. Instead of lists we will take tuples as primitive data type, and instead of the applicative LISP language we will use an imperative language with possibly a functional subset. The tuple data type is similar to record and array types and can be implemented efficiently. We will not define a complete language in this report; we will only show how "normal" syntactic constructs can be mapped on tuples.

2.1. Tuples

A tuple is a sequence of zero, one or more component values. These values are taken from the following value types:

```
<value> ::= <atom> | <tuple> .
<atom>  ::= <bool> | <char> | <int> | <real> | ... .
<tuple> ::= 'tuple' '(' [ <value> ( ',' <value> )* ] ')' .
```

The empty tuple is called NIL. Atomic values are, for example:

```
<bool> ::= 'true' | 'false' .
<char> ::= '''a''' | '''b''' | ... .
<int>  ::= '0' | '1' | ... .
<real> ::= '0.0' | '1.0' | ... .
```

The set of value types can be extended arbitrarily and we will do so if this is necessary.

We define the following elementary operations on tuples:

```
ext(t,v)   extend tuple t with value v;
len(t)     length of tuple t;
sel(t,i)   i'th component of tuple t (counting from 0);
upd(t,i,v) assign value v to i'th component of tuple t;
eq(v1,v2)  compare values v1 and v2.
```

We shall abbreviate some of these operations as follows:

```
t[i]      <=> sel(t,i)
t[i] := v <=> upd(t,i,v)
v1 = v2   <=> eq(v1,v2)
```

In the definition of tuples a choice is possible between value and pointer semantics. Value semantics means that in the assignment of tuple values the tuple, with all its component tuples if present, is copied; pointer semantics means that only the address of the tuple is copied.

Both value and pointer semantics can be implemented efficiently by representing tuple values as pointers to the actual tuple objects. The difference between the two is that in value semantics a copy must be made if a component is modified. An optimisation is possible here because this copy is only necessary when there is more than one pointer to a tuple, and this can be detected if reference counts are kept with each tuple. In the majority of cases copying will not be necessary because the value is never modified.

Both because it allows a simpler implementation and because the availability of references and reference parameters can make programming easier, we shall choose for pointer semantics. With pointer semantics, however, it may still be necessary to copy values instead of addresses in some cases. Because of the inefficiencies involved in copying large values it would be useful if this copying could be postponed until one of the values (original or "copy") is actually modified. This would give great savings in both time and space requirements because only the components that are actually changed need be copied. Such a mechanism would also be an efficient alternative for recovery caches as used in e.g. SUMMER[6]. It should therefore be investigated if the optimisation techniques possible in implementations of value semantics can also be used in implementations of pointer semantics. Hibbard[7] describes a scheme used in an Algol-68 run-time system; it may be possible to adapt this.

2.2. Mapping of Programs on Tuples

2.2.1. Mapping of Syntactic Constructs

In the same way as is done in LISP we will define a tuple representation of programs. This tuple representation has the form of a tree with as nodes the syntactic constructs forming the program. We illustrate this mapping for the following miniature language.

```
<block>  ::= <prog> | <proc> .
<prog>   ::= declare ( <id> '=' <expr> )* begin <stats> end .
<stats>  ::= ( <assign> | <if> | <while> | <invoke> | <prog> )* .
<assign> ::= <id> ':=' <expr> .
<if>     ::= if <expr> then <stats> else <stats> fi .
<while>  ::= while <expr> do <stats> od .
<invoke> ::= <id> '(' [ <expr> ( ',' <expr> )* ] ')' .
<expr>   ::= <id> | <atom> | <proc> .
<proc>   ::= proc '(' [ <id> ( ',' <id> )* ] ')' <prog> .
```

We will map each construct on a tuple that has an indication of its syntactic category in the 0'th component. For this purpose we introduce a new atomic type "scode". The mapping of the various constructs now becomes:

```
<prog>   : (PROG,((id1,expr1),...,(idn,exprn)),stats)
<stats>  : (SEQ,stat1,...,statn)
<assign> : (ASSIGN,id,expr)
<if>     : (IF,expr,stat1,stat2)
<while>  : (WHILE,expr,stat)
<invoke> : (INVOKE,id,expr1,...,exprn)
<proc>   : (PROC,((id1),...,(idn)),prog)
<id>     : (ID,id)
```

This transformation of program text to program tree should be performed by a syntax oriented editor. The text form should always be reconstructable from the internal tree representation of the program, possibly using a standard layout.

2.2.2. Binding of Identifiers

The mapping given in the previous section does not allow an efficient interpretation of the program tree, mainly because of the difficulty of associating meanings with identifiers. We will therefore try to include more information in the mapping about the binding of identifiers. We will adopt the following principles:

(1) Static binding is used, i.e. the location corresponding to an identifier is determined by the declarations in the sequence of enclosing blocks.

(2) Binding should be efficient, i.e. no long search times as in the "alist" approach of LISP.

(3) Modifications in the program-tree should have only local effects (to make editing of the program tree possible).

During the execution of a program the interpreter will generate frames for holding the local variables of each block (= program or procedure). At each moment only a subset of the frames will hold locations that are accessible from the current block. The meaning of an identifier is known if we can determine frame and index in frame that correspond to the identifier. A scheme often used by compilers is to associate an address couple <i,j> with each occurrence of an identifier. Such an address couple refers to component "j" of frame "i". This frame can be found by keeping the sequence of accessible frames in a display vector. Although this mapping allows a fast interpretation of identifier references and can be made invertible, we shall not use it in the program mapping. The reason for this is that a change in the declarations near the root of the program tree may affect the address couple mapping of almost the whole program. This is unacceptable because of principle (3).

A more localised translation of identifiers is obtained as follows. We start by numbering all identifiers in a particular block in arbitrary order. Furthermore, we attach to each block an association vector, which gives a translation from these identifier numbers to either local or global variable references. For local variables the index in the local frame is given; for global variables the identifier number (index in association vector) for the enclosing block is given. Identifiers are translated to tuples of the form (VAR,i), where "i" is the index in the association vector. Entries in the association vector are of the following form:

```
(LOCV,i)  i'th component of local frame
(GLOB,i)  i'th variable of immediately enclosing block
```

The form of the tuples corresponding to blocks now becomes

```
(PROG, av, ((id1,expr1),...,(idn,exprn)), stats)
(PROC, av, ((id1),...,(idn)), prog)
```

Except for the association vector "av", the translation of a block does not depend on other blocks. The association vector only depends on the block to which it belongs and on the association vector of the immediately enclosing block. During program execution, the time necessary to find the meaning of an identifier is proportional to the number of association vectors accessed, and this depends on how deeply nested the reference is.

The association vector contains explicit information about the import of global identifiers. This information allows the editor to check for information hiding through overdeclarations. If a global identifier is used in a deeply nested block, the association vectors of the blocks in between contain entries for this identifier. The editor can give a warning if a new declaration overwrites such an entry.

The editor can "unbind" a block by replacing the global references in its association vector by the corresponding identifiers; the block can then be bound again at a different location in the program by looking up these identifiers.

Note that the static binding we use does not preclude having also some form of dynamic binding. Only the binding of identifiers to locations is static; the binding of locations to values is always dynamic. Language constructs are possible that save the contents of locations and temporarily assign new values, thus giving the effect of static binding. (For an extensive discussion of binding schemes see Lang[8].)

2.2.3. Procedure Closures

A reference to a procedure not only refers to a program, but also to the environment in which that program must be evaluated. For references to global procedures the environment can be derived from the reference itself. If procedures are passed as argument or returned as result, the environment must be made explicit. This combination of procedure and environment is called "closure". We shall demand that the closure operation is performed implicitly whenever necessary.

In LISP the closure operation must be indicated explicitly using the notation "(FUNCTION fun)" (this is not implemented in all LISP systems, probably because it is difficult to do that efficiently). By converting every procedure denotation immediately to a "closure tuple", the closure operation can be completely hidden. Such a closure tuple has the form (CLOSURE,ctx,proc), where "ctx" is a tuple representing the context and "proc" is the procedure denotation itself (see also appendix A).

The possibility of returning procedures as result prohibits the use of a stack based memory allocation scheme. All frames for local variables must be allocated from a heap. It may still be possible to optimise for almost stack-like allocation and deallocation.

3. DATA TYPES

3.1. Concrete Data Types

The system as described does not have explicit data types. For practical reasons, such as error checking and program optimisation, it may be useful to have type indications in declarations. A data type restricts the kind of values a variable can take. Each data type can be expressed as a predicate on a value. We will need a new atomic type "tcode" and a new function "vtype", which maps values on type codes. A type code has as values:

```
<tcode> ::= 'TUPLE' | 'BOOL' | 'CHAR' | 'INT' | ... .
```

The function "vtype" allows the construction of arbitrary predicates on values. Some examples:

```
v:bool          <=> vtype(v)=BOOL
v:vector(n,tc) <=> len(v)=n and ¥i:vtype(v[i])=tc
v:record(t)     <=> len(v)=len(t) and ¥i:vtype(v[i])=t[i]
v:while_node    <=> len(v)=3 and v[0]=WHILE and
                    v[1]:expr_node and v[2]:stat_node
```

This interpretation of data types is wider than that available in most languages (e.g. the type "while_node"). What possibilities this opens should be investigated more fully. The equivalence of general predicates is not decidable; restrictions on the predicates may lead to just the kind of data types available in current programming languages, i.e. records, arrays, sub-ranges, etc.

3.2. Abstract Data Types

Abstract data types are characterised by the kind of operations possible and the relations between these operations. At the moment we see a trend to make the semantics of as many syntactic constructs as possible dependent on the type of objects that are handled (such "generic" operations can, for example, be found in CLU[9] and ALPHARD[10]). Operations that should be controllable by the data type are:

(1) Object creation (v := type(e1,...,en))
(2) Component selection (v[i] or v.f)
(3) Component assignment (v[i] := e or v.f := e)
(4) Iteration (for id in v do S od)
(5) Monadic operators (op v)
(6) Output (put(v))
(7) Dyadic operators (v1 op v2)
(8) Type conversion (often implicit)
(9) Input (v := get())

We will show how these operations can be reduced to operations determined by a type description. We shall use "t$p" to denote component procedure "p" of type description "t"; "T(v)" gives the type description associated with object "v".

Operations (1) to (6) are easily transformed:

```
(1) v := type$create(e1,...,en)
(2) T(v)$select(v,i) resp. T(v)$select(v,'f')
(3) T(v)$update(v,i,e) resp. T(v)$update(v,i,'f')
(4) T(v)$iterate(v,proc (id) S)
(5) T(v)$monadic(op,v)
(6) T(v)$put(v)
```

Operations (7) and (8) are more difficult because two types are involved. A possible implementation of (7) is to try

```
T(v1)$dyadic(op,v1,v2)
```

first, and if this fails to try

```
T(v2)$dyadic(op,v1,v2)
```

The same scheme can be used for (8), trying respectively "t$convert(v,t)" and "T(v)$convert(v,t)".

Operation (9) presents most difficulties because the type of the value is not known beforehand. The operation "get" is only well defined if there is a unique external representation corresponding to each type. This implies a close relation between the "get" and "put" operation. The addition of new types and associated external representations may cause ambiguity in earlier representations. This ambiguity can only be detected if a restricted grammar is used (like e.g. LL(1)).

A type description should define the following component procedures: create, select, iterate, monadic, dyadic, convert, get, and put. This can be reduced to a procedure "select" only; other operations are then obtained by invoking the "select" procedure:

```
T(v)$p <=> T(v)$select('p')
```

This reduces the type description to a single procedure.

An instance of an abstract data type may be represented as a special tuple of the form:

```
(CAPSULE,td,value)
```

where "td" is a tuple representing the type description (possibly consisting of the "select" procedure only).

4. LANGUAGE EXTENSIONS

If we want the programming language to be extensible by the user, the new syntax must be communicated to the syntax oriented editor and the semantics must be made available to the language processor. A possible solution is to define an abstract data type corresponding to each new construct. This data type should provide information about syntax and semantics. The editor uses the syntactic information for the recognition and translation of instances of the new construct (an extension always has the form of a modification of an existing syntax rule). The semantic information can be given in the form of a procedure that can be invoked by the interpreter.

Syntax extensions give the same problems as the get/put operations described earlier. An extension should be checked for compatibility with the existing syntax rules, and this is only possible for restricted grammars.

5. CONCLUDING REMARKS

Basing a software system on a single internal representation of programs has a number of advantages. It is possible to share several general programming tools between different language implementations. For example, compilers and optimisers need only operate on the internal representation. A syntax oriented editor together with an interpreter for the internal representation forms an excellent environment for program development and testing (in the style of LISP).

Moving part of the context sensitive syntax handling to the editor makes an efficient implementation of static binding possible. It also allows the editor to do more static checks at an early moment.

A problem may be formed by the more complicated internal representation of programs compared with LISP. This may inhibit programmers to write program-generating programs. It may be better to step to a higher level representation by building programs from abstract data types on which edit, check, and execute operations are defined. The main problem that must be solved then is how to detect inconsistencies in the external representations without placing too many restrictions on the grammar used to specify them.

ACKNOWLEDGEMENTS

I want to thank Paul Klint for reading earlier versions of this report.

Appendix A. Example Interpreter

The following program fragment illustrates the interpretation process for the language defined in section 2.2.1. The context parameter has the following components:

0: the association vector of the current block
1: the frame of local variables
2: the context of the enclosing block (static environment)

```
eval_stat = proc (ctx,s)
begin
    if s <> NIL then
     if s[0] = ASSIGN then      % (ASSIGN, var, expr)
         assign(ctx,s[1],eval_expr(s[2]))
     elif s[0] = INVOKE then    % (INVOKE, var, expr1, ..., exprn)
         declare proc = eval_expr(ctx,s[1])
                 args = NIL
                 i = 2
         begin while i < len(s) do
                 ext(args,eval_expr(ctx,s[i]))
                 i := i+1
            od
            invoke(ctx,proc,args)
         end
     elif s[0] = SEQ then       % (SEQ, stat1, ..., statn)
         declare i = 1
         begin while i < len(s) do
                 eval_stat(ctx,s[i])
                 i := i+1
            od
         end
     elif s[0] = IF then        % (IF, expr, stat1, stat2)
         if eval_expr(ctx,s[1]) then
          eval_stat(ctx,s[2])
         else
          eval_stat(ctx,s[3])
         fi
     elif s[0] = WHILE then     % (WHILE, expr, stat)
         while eval_expr(ctx,s[1]) do
          eval_stat(ctx,s[2])
         od
     elif s[0] = PROG then      % (PROG, av, decls, stat)
         ctx := tuple(s[1],NIL,ctx)
         eval_decls(ctx,s[2])
         eval_stat(ctx,s[3])
     else error() fi
    fi
end
```

```
assign = proc (ctx,var,val)
begin
    if var[0] = VAR then            % (VAR, i)
        assign(ctx,ctx[0,var[1]],val)
    elif var[0] = GLOB then         % (GLOB, i)
        assign(ctx[2], ctx[2,0,var[1]], val)
    elif var[0] = LOCV then         % (LOCV, i)
        ctx[1,var[1]] := val
    else error() fi
end


invoke = proc (ctx,proc,args)
begin
    if proc[0] = CLOSURE then       % (CLOSURE, ctx, proc)
        invoke(proc[1], proc[2], args)
    elif proc[0] = PROC then        % (PROC, av, formals, stat)
        eval_stat(tuple(proc[1],args,ctx), proc[3])
    else error() fi
end


eval_expr = proc (ctx,e)
begin
    if vtype(e) <> TUPLE then return e fi
    if e[0] = VAR then              % (VAR, i)
        return eval_expr(ctx, ctx[0,e[1]])
    elif e[0] = GLOB then           % (GLOB, i)
        return eval_expr(ctx[2], ctx[2,0,e[1]])
    elif e[0] = LOCV then           % (LOCV, i)
        return ctx[1,e[1]]
    elif e[0] = PROC then           % (PROC, av, formals, stat)
        return tuple(CLOSURE, ctx, e)
    else error() fi
end


eval_decls = proc (ctx,d)
declare i = 0
begin
    while i < len(d) do             % d = ((id1,e1),...,(idn,en))
        ext(ctx[1],eval_expr(ctx,d[i,1]))
        i := i+1
    od
end
```

REFERENCES

[1] S.R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," Bell Systems Technical Journal Vol. 57, pp.1971-1990 (1978).

[2] Jan Heering and Paul Klint, "Towards Monolingual Programming Environments," To be published as MC report.

[3] J. McCarthy et al., LISP 1.5 Programmer's Manual, The MIT Press, Cambridge, Massachusetts (1965).

[4] E. Sandewall, "Programming in an Interactive Environment: the LISP Experience," Computer Surveys Vol. 10, pp.35-71 (1978).

[5] H.G. Baker, "Shallow Binding in Lisp 1.5," Communications of the ACM Vol. 21(7), pp.565-569 (1978).

[6] Paul Klint, "An Overview of the SUMMER Programming Language," Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages, pp.47-55 (1980).

[7] P.G. Hibbard, P. Knueven, and B.W. Leverett, "A Stackless Run-time Implementation Scheme," Proc. 4th International Conference on the Description and Implementation of Algorithmic Languages, pp.176-192 (1976).

[8] B. Lang, "The Binding of Variables," Proc. 4th International Conference on the Description and Implementation of Algorithmic Languages, pp.149-175 (1976).

[9] B. Liskov, "Abstraction Mechanisms in CLU," Communications of the ACM Vol. 20, pp.564-576 (1977).

[10] M. Shaw and W.A. Wulf, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," Communications of the ACM &V 20, pp.553-563 (1977).