IST. ... BIBLIOTE... Posiz. ARCHIVIO

PASSING PARAMETER TYPES IN PROGRAMMING LANGUAGES WITH DATA ABSTRACTIONS

L77-9

P. Asirelli[+], F. Gimona[++], A. Martelli[+], U. Montanari[+++]

\+ Istituto di Elaborazione della Informazione, C.N.R. - Pisa
++ Olivetti S.p.A. - Ivrea
+++ Istituto di Scienze della Informazione Università di Pisa

Abstract

Recently designed programming languages provide mechanisms which support the use of procedural and data abstraction. In particular, most of them give the possibility of defining parametric data types. However this new feature affects procedure definitions, since parameters with partially specified type must be allowed. The paper tackles the problems generated by this generalization of procedures, by showing that parameter passing requires a pattern matching mechanism. A proposal is made which keeps parameter passing and type checking separate, thus simplifying procedure calls.

Furthermore the paper analyzes the ways in which operations can be associated with abstract data types and how operations can have access to the representation of their parameters. A construct for data abstractions is proposed, which achieves great generality by separating the two above aspects.

1. Structured programming with parametric abstract data types.

Structured programming is a methodology which allows the construction of reliable programs [1-2]. According to this methodology programs are developed through successive levels of refinement. At each level the program is written using abstract objects and operations which are considered as primitives at this level, and which are implemented at lower levels. The process goes on until all abstractions are actually primitives of the programming language in use.

Although this methodology can be used for developing programs in any programming language, its effectiveness in practice depends remarkably on the choice of a suitable language. In fact, the methodology can be easily applied only if the language provides constructs which allow a clear separation between the use of an abstraction and its implementation. In this case it is also possible to add protection mechanisms which, in accordance with the methodology, prohibit the user from accessing the implementation.

Structured programming considers three kinds of abstractions, i.e. data, control and functional abstraction, to be relevant. A functional abstraction defines a transformation from a

set of input objects into a set of output objects. Most programming languages provide a suitable mechanism, the procedure, for implementing such an abstraction. A control abstraction defines a method of sequencing arbitrary actions, while a data abstraction defines a new data type in terms of a set of operations on the objects of that type. Only some recently designed programming languages provide constructs which support also these last two kinds of abstractions [3-8].

In this paper we will be mainly concerned with various aspects of the definition of data abstractions. For instance in Fig. 1 we show how the data abstraction "stack-of-integers" can be implemented in CLU [4]. The tool provided by CLU for this purpose is the cluster, which embodies the representation of objects of type "stack-of-integers" and some operations on such objects. Outside the cluster, objects of type "stack-of-integers" can be manipulated only through these operations. For instance

stack-of-integers$push (stack-of-integers$create ( ),5,

creates a stack with one element, the integer 5. Note that a parameter declared of type cvt has the abstract type outside the operation and concrete type inside the operation.

```
stack-of-integers = cluster is create, push, pop, top, empty;
rep      = array [int];
create   = proc returns (cvt)
           return rep$new ( );
           end create;
push     = proc (s: cvt, elem: int);
           rep$extendh (s, elem);
           end push;
pop      = proc (s: cvt);
           rep$retracth (s);
           end pop;
top      = proc (s: cvt) returns (int);
           return rep$fetch (s, rep$high (s));
           end top;
empty    = proc (s: cvt) returns (bool):
           return rep$size (s) = ø;
           end empty;
end stack-of-integers;
```

Fig. 1

The cluster in CLU allows the definition of only one data abstraction, whereas in some cases one would like to define more than one data abstraction at the same time [9]. For example, if we need a graph structure, we might define two types "node" and "arc", and a set of operations on objects of the two types. In

Fig. 2 we give a set of possible operations by specifying the types of the parameters and of the returned value.

```
createnode:                          →  node
createarc : node x node x info       →  arc
head      :        arc               →  node
tail      :        arc               →  node
getinfo   :        arc               →  info
outarcs   :        node              →  set [arc]
inarcs    :        node              →  set [arc]
deletenode:        node              →
deletearc :        arc               →
```

Fig. 2

There are various reasons for preferring to the cluster construct a mechanism which allows the simultaneous definition of both types. First of all, by defining "node" and "arc" with two separate clusters, there is no obvious choice of the cluster with which the operations "createarc", "head" and "tail" should be associated, since these operations have parameters of both types. Furthermore an efficient implementation can require that these operations know the representation of the objects of both types, whereas the operations in CLU can know only the representation of one type.

Finally, even if the two types "node" and "arc" can be implemented separately, they are logically connected. Thus, because one of the goals of the structured programming methodology is to join concepts which are logically related, one wants to have a mechanism for defining together these two types. Note that the module construct of MODULA [8] is a mechanism of such kind.

In Fig. 1 we have shown how to define the data type "stack-of-integers". Assume now that we need also other types" stack-of-reals", "stack-of-characters". The clusters of these types would be similar to the one in Fig. 1 except for the type of the elements. Thus it would be convenient to be able to define in the same cluster all these types, by having the type of the elements as a parameter.

Some of the languages which support data abstractions provide also this feature. For instance, in Fig. 3 we present an ALPHARD [5] implementation of the parametric type "stack". The form in ALPHARD is equivalent to the cluster in CLU. The parametric form "stack" has two parameters: the first is the type of the elements

of the stack and the second is the maximum permissible depth of the stack. Note that the form contains the representation and the implementation of the operations, together with other informations which are used for the verification. The "$<\leftarrow>$" attached to the first parameter asserts that the actual parameter must be a type for which the assignement operation is defined.

```
form stack (T:form <←>, n:integer)=
beginform
specifications
   requires n>0;
   let stack=<...x_i...> where x_i is T;
   invariant 0≤length(stack)≤n;
   initially stack=nullseq;
   function
      push(s:stack,x:T)pre 0≤length(s)<n post s=s'~x
      pop(s:stack)pre 0<length(s)≤n post s=leader(s'),
      top(s:stack)returns(x:T)
         pre 0<length(s)≤n post x=last(s'),
      empty(s:stack)returns(b:boolean)
         post b=(s=nullseq);
representation
   unique v:vector(T,1,n),sp:integer init sp←0;
   rep(v,sp)=seq(v,1,sp);
   invariant 0≤sp≤n;
   states
     mt when sp=0,
     normal when 0<sp<n,
     full when sp=n,
     err otherwise;
implementation
   body push out(s.sp=s.sp'+1∧s.v=α(s.v',s.sp,x))=
        mt,normal::(s.sp←s.sp+1;s.v[s.sp]←x);
        otherwise::FAIL;
   body pop out(s.sp=s.sp'-1)=
        normal,full::s.sp←s.sp-1;
        otherwise::FAIL;
   body top out(x=s.v[s.sp])=
        normal,full::x←s.v[s.sp];
        otherwise::FAIL;
   body empty out(b=(sp=0))=
        normal,full::b←false;
        mt::b←true;
        otherwise::FAIL;
endform;
```

Fig. 3

With the above generalization, which is obtained with the introduction of parameters, the cluster and the form become "type constructors" rather than simple type definitions; For instance,

by declaring

```
si : stack (integer, 35),
sr : stack (real   , 14);
```

we create two variables si and sr of different type: a stack of integers and a stack of reals.

The introduction of parametric data types requires also some generalizations in the definition of procedures. In fact one would like to be able to have a procedure whose parameters have a partially specified type instead of a fixed type. For instance, a procedure which replaces the top element of a stack can be declared as follows

```
procedure replacetop (s: stack (t,n), elem : t).
```

Note that the type of "elem" is not fixed, but it must always be equal to the first parameter of "stack". Some problems related to this generalization will be considered in Section 3.

2. Associating abstract data types with their operations.

As we have shown in the previous section, all languages which support data abstractions, provide a construct (cluster, form, module) for defining together the representation of a data type and all the operations which have access to it. However abstract data types can be associated with their operations in many ways, which affect the access of these operations to the representation.

A first way, which corresponds to the viewpoint of SIMULA 67 [3] and EUCLID [6], consists of associating the operations of an abstract data type **with every instance: that is, every time** an object of that type is created, the operations are associated with the object just created. For example, if x and y are two different instances of type "complex", and we suppose that the type "complex" has the operation "sum", then there is a "sum" associated with x and another one associated with y. To denote the operation "sum" associated with x, we can use the notation x.sum. This operation has only one parameter, and thus the sum of x and y can be denoted indifferently by x.sum(y) or y.sum(x).

A reasonable implementation of the operation "sum" must access the representation of both operands. It is clear that this operation can access the representation of the object with which it is associated, but the problem is how to access the representation of the parameter. A possible implementation in SIMULA 67 is the following

```
class complex;
begin
    integer re, im;
    procedure sum (y); ref (complex) y;
    begin
        re:=re + y.re;
        im:=im + y.im
    end;
    :
end complex.
```

Note that the representations of the two operands are not accessed in the same way: the components "re" and "im" of the operand with which "sum" is associated are accessed directly,whereas the corresponding components of the other operand are accessed through the "." selector. In fact, in SIMULA 67 there are no restrictions on the access, through the "." selector, to the components of an object; however this freedom is one of the reasons why this language is criticized as a language to support data abstractions, because it makes protection impossible.

More recent languages, such as EUCLID, allow the access only to those components which are explicitly exported. So, in order to hide the representation to the user of the abstraction, it is sufficient not to export those components which constitute the representation, but only the operations. However, by implementing the type "complex" in EUCLID according to this criterion, the components "re" and "im" must not be exported, and therefore it is not possible to access the representation of the parameter y of "sum".

In conclusion we remark that this first way is adequate if the purpose is to have a discipline of access to only one data structure, i.e. when the operations can be naturally associated with only one operand, but this solution is too restrictive in general.

The problem of accessing the representation of more than one operand can be partly solved by associating operations with types, instead of with every instance of a type. As we have seen in the previous section, this is the viewpoint of CLU and ALPHARD. For instance, the sum of the complex numbers x and y can be denoted in CLU by

```
complex$sum (x,y),
```

and the cluster which implements the type "complex" can be as follows

```
complex=cluster is sum,...;
rep=record[re, im:int];
sum=proc (x,y:cvt);
    x.re:=x.re + y.re;
    x.im:=x.im + y.im;
    end sum;
:
:
end complex
```

Note that in this way an operation can access only the representation of all parameters whose type is the one with which the operation is associated. Suppose now that we want to define a type constructor "matrix (n,m)", where n and m are the number of rows and columns of the matrix. Furthermore we want to define an operation "mult" for multiplying two matrices. In general, the two matrices do not have the same size, and the types of the parameters of the operation "mult" are as follows

function mult(x:matrix(n,m); y:matrix(m,p)) result matrix(n,p).

Of course we want "mult" to be able to access the representation of both parameters and of the result. However the types of the parameters and of the result (i.e. matrix (n,m),matrix(m,p) and matrix(n,p)) are all different, and thus, in the solution we are now presenting, "mult" can access the representation of only one parameter. For instance, in CLU the multiplication of two matrices M1 and M2 can be denoted as follows

```
matrix[r,s]$mult(M1, M2),
```

and therefore the operation "mult" can only access the representation of the parameter which is a matrix of size r x s.

Up to now we have dealt at the same time with two different aspects of operations: their association with abstract data types and their access to the representation of the operands. Usually a programmer defines an abstract data type together with the operations which have access to the representation, and separately he defines other operations for the same abstract data type using the previously defined operations as primitives. In the above mentioned languages these two kinds of operations are treated in different ways. For instance, in CLU, the operation "push" of "stack-of-integers" given in Fig. 1 is invoked by

```
stack-of-integers$push(s,i),
```

where s is a stack of integers and i an integer; whereas a new operation "replacetop" defined outside the cluster cannot be associated with the type "stack-of-integers", and thus it is invoked by

```
replacetop (s,i).
```

We believe that the two aspects which we have pointed out before should be kept separate, because a user of an abstraction is interested in knowing the types of the parameters of an operation, but he needs not know whether an operation has access to the representation of a parameter or not. In order to achieve this separation, we propose the following solution:

1) The operations are not associated with the type of any operand, but simply the type of every operand must be specified as usual (totally or partially: see discussion in the next section).

2) There is a construct which consists of two brackets (beginform and endform) which hide the representations of abstract data types. Thus the operations which want to access the representations must be defined within these brackets. Furthermore more than one abstract data type can be defined within the same pair of brackets.

The main advantage of this proposal is to maintain protection on the representation and at the same time to eliminate, from the user's point of view, the difference in calling style between operations accessing the representation and not. A practical disadvantage might be to require unique names for operations, since they are not qualified with argument type. A different, more sophisticated solution would be to introduce pattern directed invocation of procedures [10].

An example of the proposed construct will be given in Section 4.

Note that a language which satisfies the two points above is MODULA [8], although the lack of type constructors makes it not a very suitable language for supporting data abstractions.

3. Parameter passing and pattern matching

Every parameter actually passed during the execution of a program has normally a type and a value: the value is assigned to the formal variable, while the type is simply checked to insure type consistency. Furthermore in most languages type checking is performed once and for all at compiling or linking time and nothing is done under this aspect at run time.

Some problems arise when one wants not to specify completely the type of a parameter for guaranteeing larger applicability to the procedure. In this case the operation of passing the actual parameter should also have the effect of completely specifying the type of the formal parameter. For example, one might not want to specify the size of an array passed as a parameter, especially if parameter passing takes place by reference and thus this flexibility does not involve dynamic storage allocation. Three correct approaches are possible:

1) the array size is passed as a distinct parameter and the array is declared to be of that size. Language FORTRAN, for instance, follows this approach.

2) The formal parameter is declared to be simply an array, without specifying the size, and a built-in function exists which, given an array, returns its size.

3) The formal parameter is declared to be an array of size n, and the variable n is automatically assigned during parameter passing.

A wrong solution, instead, is to pass separately the size, if needed, for instance to control loops, without associating it with the array. This is the case of ALGOLW, where only the number of subscripts is specified for an array passed as a parameter.

The above discussion is applicable not only to the array case but is valid in general. We give here some terminology. Every possible value has a type associated with it: E.g. 25 is of type "integer". Furthermore, there are values of type "type" (e.g. "integer" can also be considered a value of type "type"). All values of type "type" are either elementary types (e.g."integer", "complex" as shown in section 2.) or terms obtained by writing a type constructor symbol followed by its (nonempty) list of arguments. For instance "stack (char, 25)", (i.e. a stack of characters of maximal depth 25, according to the example in Section 1.)is a value of type "type". Of course the types of the actual arguments of a type constructor must match the types of its formal parameters.

A type constructor itself has a type, namely the word "tconstr" followed by the list of the types of its parameters. For instance, type constructor "stack" has type "tconstr (type, integer)". Similarly, procedures and functions use keywords "proc" and "func" respectively. Furthermore, in the case of functions, the list following "func" ends with the type of the result. Assigning a type in this way to type constructors, procedures and functions allows to pass them as parameters still fully specifying their type (as in ALGOL 68, and differently from ALGOL 60, where all procedures are simply of type "proc"). For instance one could declare a type constructor "double" with two parameters. The first parameter is itself a type constructor with a single parameter of type "type", while the second parameter (of "double") is of type "type". Type constructor "double" returns a type obtained by applying twice its first parameter to its second parameter. For example, if "set" is a type constructor with a parameter of type "type", then "double (set, integer)" would be a set of sets of integers. Thus the type of "double" is the following term:

tconstr (tconstr (type), type)

Let us now go back to our problem of specifying parameter types not completely. In general, one might think to complicated conditions to be satisfied by parameter types: for instance one might have a procedure with two two-dimensional arrays as parameters, and might ask that the two arrays have the same number of elements! We restrict ourselves to simpler cases, where the condi-

tions are expressible using patterns. Patterns are terms similar to those representing the values of type "type", but where some variables (of suitable type) are present. For instance, in the example given in Section 2

```
procedure replacetop (s:stack (t,n), elem:t)
```

the term "stack (t,n)" is a pattern of type "type" where two variables t and n are present, of type "type" and "integer" respectively. A pattern represents a set of terms, namely those terms which match the pattern. For instance the term of type "type"

```
stack (record f1:stack(char);f2:array[1:100]of boolean end,31)
```

matches the pattern above, while the term

```
set-of-integers
```

does not.

Variables in patterns should be explicitely declared as such. A suitable notation would be to declare the types of variables at their first occurrence. For instance we might write the previous example as

```
procedure replacetop (s:stack(t:type; n:integer), elem:t).
```

Note that the operation of pattern matching has a twofold effect: first, to enforce the condition represented by the pattern and, second, to assign the variables in the pattern. Thus a mechanism of parameter passing where the types of formal parameters are patterns allows, as desired, both to check the type of the actual parameters and to specify completely the type of the formal parameters. For instance a procedure call to "replacetop" written as follows

```
:
a:stack (integer, 40); b:integer;
:
replacetop(a,b);
:
```

would respect type constraints and would cause the following bindings:

```
s:=a
t:=integer
n:=40
```

The use of patterns as types of formal parameters allows to represent a relatively large class of type constraints. For instance, the type constraint necessary for the procedure which multiplies two rectangular matrices (see section 2) can be written as follows:

```
function mult(a:matrix(m:integer,n:integer),b:matrix(n,p:integer))
              result matrix(m,p)
```

Similarly, the type of the function "mult" would be

```
func(matrix(m:integer,n:integer),matrix(n,p:integer))
     result matrix(m,p)
```

Note therefore that the patterns appear also in procedure types.

Passing parameter types by pattern matching, however, has the disadvantage, by its very nature, of mixing the two aspects of value passing and of type checking. Furthermore it introduces new, hidden parameters inside the type of explicit parameters. A different solution is to add also the hidden parameters to the parameter list. Declaration of procedure "replacetop" would then become:

```
procedure replacetop (t:type, n:integer, s:stack(t,n), elem:t)
```

and its type would be:

```
proc (t:type, n:integer, :stack(t,n), :t)
```

In this approach the operation of pattern matching has the only effect of checking the type. Note however that variable names cannot, as usual, be eliminated when writing the type of the procedure. A further disadvantage of the explicit approach is the necessity for the programmer to add new parameters to every procedure call. For instance the previous example would become:

```
      .
      .
      .
a:stack(integer,40); b:integer;
      .
      .
      .
replacetop(integer,40,a,b);
      .
```

By substituting "integer" and 40 for the variables t and n, the patterns representing the type of the third and fourth parameter of "replacetop" become "stack(integer,40)" and "integer" respectively. These patterns match the type of a and b. Note that the type consistency of the call to procedure "replacetop" can be statically checked since the type of "replacetop" (see above) is known and the first two actual parameters are constants.

From a methodological point of view, the extra effort for the programmer can be justified as a check for making sure he knows what he is doing. Furthermore, a similarly long notation is required by CLU. In fact a call to "push" for the above example in CLU would be:

```
stack(integer,40)$push(a,b).
```

In the next section we will suggest a construct embodying

the proposals of section 2 and 3.

4. A proposal of a construct for defining abstractions.

We will discuss an example. We want to implement a hash--table as an array where every bucket is a list of name-value pairs, but we want to leave it parameteric with respect to name and value types and, as a consequence, to the hashing function mapping the names into positions in the array. Furthermore, the size of the array must also be variable. In Fig. 4 we see a form implementing this data abstraction together with the procedure "insert". We can imagine that other procedures and functions accessing the representation of hashtable, like "create", "empty", "member", "delete", etc., might be part of the same form. In Fig.5 we see part of a program where a variable of type "hash--table (string, integer, 128)" is declared and where procedure "insert" is called.

A few comments follow. The parenthesis pair "beginform" - "endform" encloses a number of type constructor declarations, together with the declarations of all procedures and functions allowed to access the representation of the abstract data types which can be generated by these type constructors. Since some of them might have only a local meaning ,the explicit list of all exported (data and functional) abstractions is added after "beginform". Every abstraction is conceived as a separate module, with no global identifiers except the names of the other abstractions in the same form. All other nonprimitive abstractions used must be explicitely imported. This is the case of the type constructor "list" in "hashtable" and besides it, of its functions "emptylist", "addlist" in "insert". Note that the type of an abstraction must be declared when the abstraction is imported. The first (data) abstraction,"hashtable", is implemented as an array of lists of records. Note that even if the upper bound of the array is not a numerical constant, we do not necessarily need arrays with bounds computable at run time, provided that actually declared variables (like "a" in Fig. 5) have only static parameters.

Procedure "insert" has as parameters the hash-table "ht" to act upon and the pair "name"-"value" to insert. Furthermore it receives also the function "hash" and the hidden parameters "tname", "tvalue" and "m" now made explicit. A special comment is due to explain the keyword convertible added to "ht". It means that this parameter is of type "hashtable..." in the caller, and of type "array..." in the called procedure. A construct of this sort is always necessary if a form must provide protection. However three different approaches are possible.

i) Where the representation of a type constructor is known, namely inside its form, conversion is automatic and not even recognizable. Thus inside the form the name of the type construc-

```
beginform export hashtable,.... insert,...;
     typeconstructor hashtable(tname:type,tvalue:type,m:integer);
     begin
        import list:tconstr(type);
        array[1:m]of list(record n:tname;v:tvalue end)
     end hashtable
     :
     :
     procedure insert(tname:type,tvalue:type,m:integer,
               ht:convertible,
               hashtable(tname,tvalue,m),name:tname,value:tvalue,
               hash:func(:tname,:integer)result integer);
     import list:tconstr(type),
            emptylist:func(t:type,:list(t))result boolean,
            headlist:func(t:type,:list(t))result t,
            taillist:func(t:type,:list(t))result list(t),
            addlist :func(t:type,:list(t),:t)result list(t);
     type pair=record n:tname;v:tvalue end;
     begin l:list(pair);found:boolean;addr:integer;
          addr:=hash(name,m);
          l:=ht[addr];
          found:=false;
          while¬ emptylist(pair,l)and¬found do
               if headlist(pair,l).n=name then found:=true
               else l:=taillist(pair,l);
          if¬found then ht[addr]:=addlist(pair,ht[addr],
                              {n=name, v=value})
     end insert;
     :
     :
endform
```

Fig.4

```
     :
     :
import hashtable:tconstr(:type,:type,:integer),
     insert:proc(tname:type,tvalue:type,m:integer,
               :hashtable(tname,tvalue,m),:tname,:tvalue,
               :func(:tname,:integer)result integer),
     stringhash:func(:string,:integer)result integer,
     :
     :
begin a:hashtable(string,integer,128);s:string;p:integer;
     :
     :
     insert(string,integer,128,a,s,p,stringhash)
     :
     :
```

Fig. 5

tor is simply a shorthand for its representation, while outside, of course, it is an opaque symbolic name which cannot be converted. This approach is followed by PASCAL [11] based languages, like MODULA and EUCLID.

ii) Conversion must be explicitly performed using primitives "up" and "down". Of course they can be used only on type constructors defined in the same form. In particular, the attribute convertible specified for a parameter means that a "down" operation must be performed upon entrance in and an "up" operation upon exit from the procedure or function. Language CLU follows essentially this approach.

iii) Conversion can be performed only on a parameter using the convertible attribute, i.e. primitives "up" and "down" are not allowed.

The difference among the three approaches is easily shown by an example involving a recursively defined type, e.g. "list". Recursion is realized through a typed pointer(*) (see Fig. 6).

```
beginform export list,...,headlist,...;
     typeconstructor list(t:type);
     begin
         record value:t,next: ↑list(t)end
     end list;
     .
     .
     .
     function headlist(t:type,l:convertible list(t))result t;
     .
     .
     .
endform
```

Fig. 6

According to approach i), inside the form the name "list" denotes indifferently a list, or a record connected to a list, or a record with a null pointer, or a record connected to a record connected to a list, and so on. In the second approach, just after entrance in "headlist", if "l" has the convertible attribute, then its type is exactly a record connected to a list. However, applying function "down" to its dereferenced field "next", one can get a record connected to a record connected to a list. But in this case the user must take care explicitly of conversion

(*) To allow termination of a structure, pointers can always have the type "null", in alternative to their own type.

and type consistency. In the third approach "l" is again of type record connected to a list, but, as any other abstract data type, "list" is opaque and can be manipulated only using its operations. In the line of imposing a strict discipline upon the programmer, we suggest here the last approach.

We terminate our discussion on the example in Fig. 4 with two more comments: The "type" construct used to define "pair" is a PASCAL-like shorthand (similar to "equate" in CLU) while the curly brackets denote an operation which both allocates and inizializes records (similar to "cons" in LISP).

5. Conclusion.

In the paper we have proposed a new construct which allows to define a large class of abstractions with non-completely specified parameter types, still keeping value passing and type checking separated. Furthermore, it allows to define more than one type constructor in a form, and thus it is possible to have an operation which has access to the representation of many of its parameters, even if of different type.

What has not been discussed here, and will be the subject of another paper, is the possibility of checking statically the type consistency of programs with the proposed construct. In particular, the modular nature of most languages based on data and control abstractions suggests the desirability of being able to perform type checking on each module separately. Due to the strong typing mechanisms we have suggested, and in particular due to the forced type declaration for all abstractions, both imported and passed, we hope that a completely static type checking will be possible.

The research described in this paper is part of a joint project among the Italian National Research Council, the University of Pisa and Olivetti, the Italian Computer manufacturer. The project, (mainly) supported by Olivetti, has as main goal the design and implementation of an experimental symbolic interpreter for a PL/1 based language with data abstractions [12-14].

We are indebted for many interesting discussions to the other partecipants to the project, namely V. Ambriola, P. Degano, C. Lami, G. Levi, G. Pacini, F. Sirovich and F. Turini.

References

1. Dijkstra, E.W., Notes on structured programming, in Structured Programming, Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. New York, Accademic Press, 1972, 1-82.

2. Wirth, N., On the composition of well-structured programs, ACM Computing Surveys, Vol. 6, n.4, December 1974, 247-259.

3. Dahl, O.J., Myhrhaug, B., and Nygaard, K., The Simula 67 Common Base Language, Publication S-22, Norwegian Computing Center, Oslo, 1970.

4. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction mechanisms in CLU, Computation Structures Group Memo 144, M.I.T., October, 1976.

5. Wulf, W.A., London, R.L., and Shaw, M., An introduction to the construction and verification of ALPHARD programs, IEEE Trans. on Software Engineering, Vol. SE-2, n.4, December,1976, 253-265.

6. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J., Report on the programming language EUCLID, SIGPLAN Notices 12, 2.

7. Ambler, A.L., et al., GYPSY: A language for specification and implementation of verifiable programs, SIGPLAN Notices 12, 3 1-10.

8. Wirth, N., MODULA: A language for modular multiprogramming, Technical Report 18, Institut fuer Informatik, ETH, Zurich, March, 1976.

9. Ghezzi, C., and Paolini, P., Language and system supports to abstraction implementation, Fifth Annual Int. Conf. on the Implementation and Design of Algorithmic Languages, Rennes, France, May 16-18, 1977.

10. Hewitt, C., Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot, AI TR-258, M.I.T., Artificial Intelligence Laboratory, April, 1972.

11. Jensen, K., and Wirth, N., PASCAL, User Manual and Report, Lecture Notes in Computer Science, n.18, Springer-Verlag,1974.

12. Levi, G., Sirovich, F., Sviluppo di programmi a livelli: astrazioni, specifiche ed esecuzione simbolica, presented at Giornate di Studio su:"Programmazione strutturata: Esperienze e Orientamenti". Milano 23-25 June 1976, to appear in "Informatica".

13. Levi, G., Sirovich, F., Proving program properties, symbolic evaluation and logical procedural semantics, Mathematical Foundations of Computer Science 1975, (Springer-Verlag, 1975).

14. Montanari, U., L'esecuzione simbolica di programmi, N.I. B75-19, I.E.I., Pisa, October, 1975.