Separate Compilation of Type-parameterized Modules

TAIICHI YUASA*

An implementation technique is presented for separate compilation of type-parameterized modules with constraints on permissible actual types. This technique provides a simple and straightforward way to achieve dynamic type-parameter binding at run-time without loss of execution efficiency.

1. Introduction

Separate compilation of program modules is a key issue with regard to implementation of modular programming languages. It reduces compile time by avoiding reprocessing of those modules which are used in several programs. It also enhances modular programming in that internal modification of a module does not require reprocessing of other modules. In addition, use of separately generated object code facilitates efficient debugging as seen in some LISP programming systems (e.g. Interlisp[6]).

The type-parameterization mechanism, incorporated in several languages such as CLU[2], *ι*[3], and Ada[9], has been regarded as one of the important structuring concepts for modular programming. It may save programming labor because a class of related modules can be defined by a single type-parameterized module.

In particular, explicit description of constraints on permissible actual types is significant for the mechanism as it clarifies what is essential in each parameterized module. It enables the programmer to write a parameterized module independent of actual types while focusing on the fundamental structure common to all permissible actual types. Such constraints are described by 'sypes' [5] in *ι* and by 'while clauses' in CLU.

The type-parameterization mechanism causes a problem in attaining separate compilation. That is, operations on actual types to be performed during execution of parameterized modules are determined only when actual types are supplied to the modules. Solutions to this problem are classified into compile-time and run-time schemes, according to the time of type-parameter binding. (The compile-time scheme for separate compilation will be discussed later in Sec. 3.) A CLU implementation[1] adopts a run-time scheme, in which a new 'object module' of a parameterized module is created once for each distinct set of actual types. Another possible run-time scheme is to perform dynamic type-parameter binding at run-time each time a parameterized module is invoked. The idea behind this scheme is fairly simple: When a parameterized module is invoked with some actual type, the module receives information on the actual type so that actual operations can be determined by the object code for the module. It may be observed that with this scheme, language processing is simpler and more straightforward than with the former run-time scheme. The crucial point, however, is how to reduce run-time overhead attendant on run-time parameter binding with this scheme. Thus an efficient technique has to be developed for this purpose.

This paper presents such a technique adopted in the language processing of *ι*, along with discussions on implementation problems raised by the type-parameterization mechanism. (The language processing system of *ι* is embedded in an integrated programming system, called the *ι* system[4], which supports development of programs written in the language *ι*. For the overall language system, refer to [8].) Some features of the technique are:

1. It reduces run-time overhead by making the language processor do as much processing as possible during compile time.
2. It provides a simple and straightforward algorithm for compile-time processing as well as for type-parameter binding at run-time.
3. As we will see later, it is possible to write pathological programs with type-parameterization mechanisms. Although of course, the language processor ought to be able to deal with any possible cases, the technique provides efficient execution in normal and realistic cases.

The next section provides program examples used throughout this paper, along with preliminary remarks. In Sec. 4 there is a discussion of the kinds of information required for actual type parameters, and in Sec. 5, we explain how such information is constructed during execution time.

2. Concepts and Terminology

In order to locate the problem discussed in this paper, we list some programs written in the language *ι*, along with preliminary remarks. For a detailed explanation of the language, refer to [3].

Fig. 2.1 illustrates the interface part of the type module RAT which defines the type of rational numbers,

*Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan.

```
interface type RAT
  fn ZERO:           →@
       ONE:           →@
       ADD: (@, @)→@
     MULT: (@, @)→@
       NEG: @      →@
        INV: @      →@
          .
          .
end interface
```

Fig. 2.1 Interface part of type module RAT.

```
interface sype RING
  fn ZERO:           →@
       ONE:           →@
       ADD: (@, @)→@
     MULT: (@, @)→@
       NEG: @      →@
end interface
```

Fig. 2.2 Sype module RING.

where the primitive functions on RAT are declared with domain and range types. In the interface part of a type module, the type defined by the module, which is RAT in this case, is denoted as '@'.

Fig. 2.2 gives an example of another kind of modules called sype modules. The sype module RING here is supposed to define the class of types which have ring (in mathematical sense) as their substructure. If a type T belongs to the class defined by a sype S, we denote $S \lesssim T$. (Refer to [3] for the formal definition of the relation '$\lesssim$'.) From the viewpoint of the language processor of ι, the sype-type relation $S \lesssim T$ holds if, for each primitive function F of S, a function is defined on T with the same name and with the same domain and range. Thus RING$\lesssim$RAT holds.

Using the sype RING, we define a type module POLY (P: RING) which defines the type of polynomials in one variable with any coefficient type T satisfying RING$\lesssim T$.

Any type T such that RING$\lesssim T$ can be used as the actual type parameter for POLY(P: RING). For instance, since RING$\lesssim$RAT, POLY(RAT) is a type of polynomial whose coefficients are of type RAT. Thus P, which we call a type parameter of sype RING, represents the indefinite (formal) type parameter and POLY(P: RING) is said to be a type-parameterized module or simply a parameterized module. We call POLY(RAT) a definite module-instance of POLY(P: RING) since the actual type parameter RAT is a definite type. On the other hand, ARRAY(P) in the realization part of POLY(P: RING) is called an indefinite module instance because it contains the formal type parameter P of POLY(P: RING).

The realization part gives an implementation of POLY (P: RING). 'rep=ARRAY(P)' specifies that each object of type POLY(P: RING) is represented by ARRAY(P) or array of type P. (E.g. POLY(RAT) is represented by ARRAY(RAT).) In the rest of the realization part, 'rep' denotes ARRAY(P). ↓ADD implements the

```
interface type POLY (P: RING)
  fn ZERO:             →@
       ONE:             →@
       ADD: (@, @)  →@
     MULT: (@, @)  →@
       NEG:   @      →@
      COEF: (@, INT)→P
       DEG:    @     →INT
          .
          .
end interface
realization type POLY (P: RING)
  rep=ARRAY (P)
          .
          .
  fn ↓ADD (X, Y: rep) return (Z: rep)
    var I: INT;
    if HIGH (X)<HIGH (Y) then ⟨X, Y⟩:=⟨Y, X⟩ end if;
    for I from 0 to HIGH (Y) do
      X[I]:=P#ADD (X[I], Y[I])................................(*)
    end for;
    Z:=X
  end fn
          .
          .
end realization
```

Fig. 2.3 Type module POLY (P: RING).

(abstract) function ADD.

The line marked '*' in Fig. 2.3 states that the I-th components of X and Y are 'added' and the result replaces the I-th component of X. Since the components of X and Y are of type P, the addition must be that of P (i.e. P#ADD). Those functions which are actually executed in run-time at the line '*' are said to be actual ADD's for P#ADD. If the actual type parameter is RAT, the actual ADD is the ADD on rational numbers, i.e. RAT#ADD.

From the interface part of POLY(P: RING), we find another sype-type relation RING$\lesssim$POLY(P: RING). Remember that any type T such that RING$\lesssim T$ can be used as the actual type parameter for POLY(P: RING). This indicates that POLY(POLY(RAT)) is a permissible module instance.

The relation '$\lesssim$' is also defined between two sypes in the language ι. For example, suppose we have another sype module FIELD (Fig. 2.4). Since all primitive functions defined in RING are also defined in FIELD, we can denote RING$\lesssim$FIELD. In the body of STP ($P2$: FIELD)#↓ADD in Fig. 2.5, POLY($P2$)#ADD (abbreviated as rep#ADD) is called. That is, the actual type parameter which STP($P2$: FIELD) receives during execution time is passed to POLY(P: RING). This is permissible since RING$\lesssim$FIELD.

```
interface sype FIELD
  fn  ONE:           →@
       ZERO:          →@
      MULT: (@, @)→@
        ADD: (@, @)→@
         INV: @      →@
        NEG: @      →@
end interface
```

Fig. 2.4 Sype module FIELD.

```
realization type STP (P2: FIELD)
   rep=POLY(P2)
      .
      .
   fn ↓ADD(X, Y: rep) RETURN (Z: rep)
      .
      .
   Z:=rep#ADD(X, Y)
      .
      .
   end fn
      .
      .
end realization
```

Fig. 2.5 Type module STP (*P*2: FIELD).

3. The Problem and a Possible Solution

Let us return to the module POLY(*P*: RING) (in Fig. 2.3) and focus on the following problem: What should the compiler do in processing the realization part of POLY(*P*: RING), especially for the function call of *P*#ADD (marked '*')? Also what kind of information should be sent to POLY(*P*: RING) at execution time?

One possible solution is to do almost nothing with POLY(*P*: RING) itself until *P* is bound to some actual type parameter. When POLY(*T*) is used in other modules (i.e. when *P* is bound to an actual definite type instance *T*), the instance of the realization part of POLY(*P*: RING), with all occurrences of *P* replaced by *T*, is processed.

For example, when POLY(RAT) is used, the line marked '*' is replaced by:

$$Z[I] := \text{RAT}\#\text{ADD}(X[I], Y[I])$$

Then the processor knows that RAT#ADD is to be called.

Thus the processor actually regards module-instances of POLY(*P*: RING) as different type modules. In this sense, this method is nothing more than the conventional way of processing modules without the type-parameterization mechanism, and enables the language processor to generate simple and efficient object code.

However, this solution has the following deficiencies.

1. If there exists a situation where the number of module instances is infinite, the method does not work. As an example, suppose *M*(*M*(*P*)) is used in a parameterized module *M*(*P*: *S*). When *P* is bound to some type *T*, *M*(*T*) is processed. During the process another instance *M*(*M*(*T*)) is used. Since *P* is bound to *M*(*T*) this time, another instance *M*(*M*(*M*(*T*))) is used during the process of *M*(*M*(*T*)), and so on. In this way, the processor encounters an infinite number of instances of *M*(*P*: *S*). Fortunately, in the case of the language *ι*, the number of instances of a single module is always finite because dependency relationships among modules cannot be circular. (Refer to [3]. The proof of finiteness is found in [7].) However, this is not necessarily the case with other languages. See [1] for the case of CLU.
2. The bookkeeping of all instances of all parameterized modules is not a trivial task. See, for example, in Fig. 2.5 when STP(RAT) is defined, POLY(RAT) is to be automatically and implicity defined.
3. The compilation time tends to be long with multiple iterations of similar processing. In addition, a large amount of storage is required since each instance of a single parameterized module must be allocated separately.

Thus we would rather have module-wise processing where each module is independently compiled and type-parameter bindings are done dynamically.

4. Information on actual type parameters

In order to perform dynamic type-parameter binding, each parameterized module *M*(*P*: *S*) is supposed to receive data containing sufficient information on the actual type parameter *T*. Let us call the data the type description for *T*. In this section, we discuss what kind of information should be included in type descriptions.

Procedure tables for sype-type relations

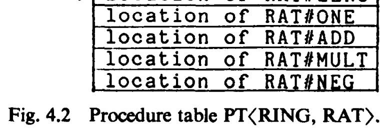
Suppose in the realization part of the module *M*1, POLY(*P*: RING)#ADD is called with the actual type parameter RAT (see Fig. 4.1).

The actual ADD for *P*#ADD in the body of POLY (*P*: RING)#↓ADD is RAT#ADD in this case. Thus the type-parameter information sent to POLY(*P*: RING) must include the location of RAT#ADD (more precisely, the entry point of the object code for RAT#ADD). Any function in RAT, corresponding to a primitive function in the sype RING, may be used in POLY(*P*: RING), and the functions actually used cannot be determined when compiling *M*1. Therefore a table must be sent to POLY(*P*: RING), which consists of the locations of all functions in RAT corresponding to the primitive functions in RING. We call such a table procedure table for RING$\lesssim$RAT and denote it as PT⟨RING, RAT⟩.

In order to reduce the time for table look-up, we make use of the order of primitive functions presented in the interface of RING. That is, the location of the function in RAT corresponding to the *i*-th primitive

```
realization M1
      .
      .
   POLY(RAT)#ADD(X, Y)
      .
      .
end realization
```

Fig. 4.1 Module *M*1.



location of RAT#ZERO
location of RAT#ONE
location of RAT#ADD
location of RAT#MULT
location of RAT#NEG

Fig. 4.2 Procedure table PT⟨RING, RAT⟩.

function in RING is contained in the *i*-th entry of the table PT⟨RING, RAT⟩. For instance, since ADD is the third function presented in sype RING (see Fig. 2.2), the third entry of PT⟨RING, RAT⟩ contains the location of RAT#ADD. Thus the table in Fig. 4.2 is constructed while processing *M*1 and is sent to POLY(*P*: RING) at run-time when POLY(RAT)#ADD is called. The object code for *P*#ADD in the realization of POLY (*P*: RING) is constructed so that the third entry of the procedure table is used in order to find the actual ADD by an indexing mechanism.

It seems that for most actual programs using type parameters, type-parameter passing is as simple as in the case of *M*1, and only procedure tables are necessary. Since procedure tables are generated in compile time and table look-up is done by indexing, run-time overhead attendant upon type-parameter passing appears to be fairly small.

Adaptor tables for sype-sype relations

Since the order of primitive functions in the interface of a sype is essential for a procedure table, we face a problem when sype-sype relations are used in programs. To illustrate, suppose we have a module *M*2 in the realization part of which STP(RAT)#ADD is called. (See Fig. 4.3. STP(*P*2: FIELD) is given in Fig. 2.5.)

As explained before, PT⟨FIELD, RAT⟩ in Fig. 4.4 is sent to STP(*P*2: FIELD) when STP(RAT)#ADD is called in executing *M*2. During execution of STP (RAT)#ADD, POLY(RAT)#ADD may be caelld and PT⟨RING, RAT⟩ is to be sent to POLY(*P*: RING). If the order of primitive functions in the interface of TING were just the same as the ones in the interface of FIELD, PT⟨FIELD, RAT⟩ could be used as PT⟨RING, RAT⟩, i.e. STP(*P*2: FIELD) could send the procedure table it received, to POLY(*P*: RING) directly. However, this is not the case. The location of RAT#ADD is found in the fourth entry in PT⟨FIELD, RAT⟩ while ADD is the third function in the interface part of sype RING (see Fig. 2.2).

Since it seems inefficient to construct PT⟨RING, RAT⟩ from PT⟨FIELD, RAT⟩ each time POLY(*P*: RING)#ADD is called from STP(RAT)#ADD, we would rather use PT⟨FIELD, RAT⟩ in POLY(*P*: RING) with some adaptations. To this end, we introduce another kind of tables, called adaptor tables. For each pair of sypes *S* and *S'* such that $S \lesssim S'$, an adaptor table AT⟨*S*, *S'*⟩ is constructed as follows. If the *i*-th primitive function of *S* is presented as the *j*-th primitive function in the interface part of *S'*, then the *i*-th entry of AT⟨*S*, *S'*⟩ has the value of *j*. In the above example, AT⟨RING, FIELD⟩ in Fig. 4.5 is constructed during compile time of STP(*P*2: FIELD). During execution of STP(RAT)# ADD, this table AT⟨RING, FIELD⟩ is linked together with the procedure table PT⟨FIELD, RAT⟩ to form the type description to be sent to POLY(*P*: RING) when POLY(RING)#ADD is called (Fig. 4.6). To find the actual ADD in the body of POLY(*P*: RING)#↓ADD, since the third entry of AT⟨RING, FIELD⟩ has the value 4, the fourth entry of PT⟨FIELD, RAT⟩ is searched, where the location of RAT#ADD is contained.

This kind of run-time linkage is necessary only when executing a parameterized module *M*(*P*: *S*) in which a sype-sype relation $S' \lesssim S$ is used for some sype *S'* and the order of primitive functions in the interface part of *S'* differs from the one in the interface of *S*.

Information on type parameters to actual type parameters

So far, we have considered only those cases where the actual type parameters to POLY(*P*: RING) are not type-parameterized. Now we explain how to deal with the cases where the actual type parameters to POLY(*P*: RING) are also parameterized.

Consider the case where POLY(*M*(*T*))#ADD is called, where *M*(*P*: *S*) is a type-parameterized type module and *T* is the actual type parameter to *M*(*P*: *S*). When evaluating *P*#ADD(*X*[*I*], *Y*[*I*]) in the body of POLY(*P*: RING)#↓ADD in Fig. 2.3, *M*(*P*: *S*)#ADD may be called with the actual type parameter *T*. In order for *M*(*P*: *S*)#ADD to be called properly with the actual type *T*, the type description sent to POLY(*P*: RING) must include information on *T*. Thus POLY(*P*: RING) is supposed to receive the type description shown in Fig. 4.7. When *M*(*P*: *S*)#ADD is called during the execution of POLY(*P*: RING), the type description for

```
realization M2
        .
        .
    STP(RAT)#ADD
        .
        .
end realization
```

Fig. 4.3 Module *M*2.

location of RAT#ONE
location of RAT#ZERO
location of RAT#MULT
location of RAT#ADD
location of RAT#INV
location of RAT#NEG

Fig. 4.4 Procedure table PT⟨FIELD, RAT⟩.

2
1
4
3
6

Fig. 4.5 Adaptor table AT⟨RING, FIELD⟩.

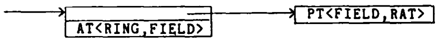


Fig. 4.6 Dynamic linkage of type description.

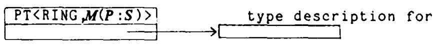


Fig. 4.7 A type description for *M*(*T*).

```
realization N(P1: S)
    .
    .
    POLY(M(P1))#ADD
    .
    .
end realization
```

Fig. 4.8 Module $N(P1: S)$.

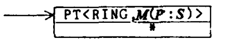


Fig. 4.9 Incomplete type description.

T is retrieved and sent to $M(P: S)$#ADD.

The type description for $M(T)$ cannot be prepared in compile time if T is not determined in compile time. Suppose that in parameterized module $N(P1: S)$, POLY $(M(P1))$ is used (see Fig. 4.8). In this case, the processor prepares an 'incomplete' type description (Fig. 4.9) in compile time of $N(P1: S)$. Then, when $N(P1: S)$ receives the actual type parameter T at run-time, the type description for T is linked from the cell marked '*' in Fig. 4.9 to form a complete type description for $M(T)$.

5. Run-time Linkage of Type Descriptions

As explained in the previous section, given a parameterized module $M(P: S)$, some of the type descriptions prepared during the compile time of $M(P: S)$ may be incomplete. To form complete type descriptions, they must be linked to the type description for the actual type T to which P is bound. The complete type descriptions constructed in this way must be retained during execution of $M(T)$. The problem here is that, during execution of $M(T)$, another instance $M(T')$ of the same module $M(P: S)$ may be used. If the incomplete type descriptions were simply linked to the type description for T', the old type descriptions might be lost.

As an example, suppose that STP(STP(RAT))#ADD is called in the module $M3$ (see Fig. 5.1). Let us consider how STP(STP(RAT))#ADD is executed. Since POLY $(P2)$#ADD is used in the realization part of STP $(P2$: FIELD) (See Fig. 2.5) and the actual type parameter is STP(RAT), POLY(STP(RAT))#ADD will be called in executing STP(STP(RAT))#ADD. Then the actual ADD for P#ADD in the realization part of POLY(P: RING) is STP(RAT)#ADD. Thus, during execution of STP(STP(RAT))#ADD, another instance of STP($P2$: FIELD), namely STP(RAT), is used.

In compile time of $M3$, a type description for STP (RAT) is prepared and is sent to STP($P2$: FIELD) at

```
realization M3
    .
    .
    STP(STP(RAT))#ADD
    .
    .
end realization
```

Fig. 5.1 Module $M3$.

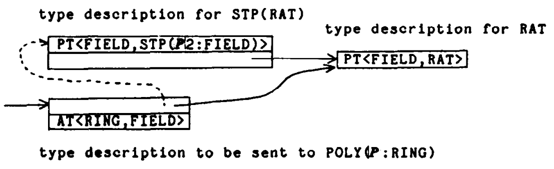


Fig. 5.2 Relinkage of type descriptions.

run-time (see Fig. 5.2). Then it is combined with the AT⟨RING, FIELD⟩ to form the complete type description to be sent to POLY(P: RING). This type description might be lost if, when STP($P2$: FIELD)# ADD is entered with the actual type parameter RAT, AT⟨RING, FIELD⟩ were combined with the type description for RAT.

One possible solution to this problem is to make copies of incomplete type descriptions whenever new type descriptions are required.

In the case of the language ι, which does not allow circularity in dependency relationships among modules, it is proved that: Given a parameterized module $M(P: S)$ and distinct types T and T', suppose that $M(T')$ is used during execution of $M(T)$. Then the execution of $M(T')$ does not require the information that the formal type parameter P of $M(P: S)$ was previously bound to T. This fact suggests a simpler and more efficient solution to the problem above. For a parameterized module $M(P: S)$, a stack is prepared which is used locally for $M(P: S)$. When P is bound to T', the type description for T is pushed on to the local stack. At the end of execution of $M(T')$, the type description for T is retrieved and relinked to the incomplete type descriptions so that the type descriptions, which existed before execution of $M(T')$, will be recovered. Fig. 5.3 illustrates how the local stack is used in the example above.

Note that, even when a number of instances of a single module $M(P: S)$ may be used, the local stack for the module need not be large, since the number of module instances of $M(P: S)$ that are used during execution of a certain module instance of $M(P: S)$ is small.

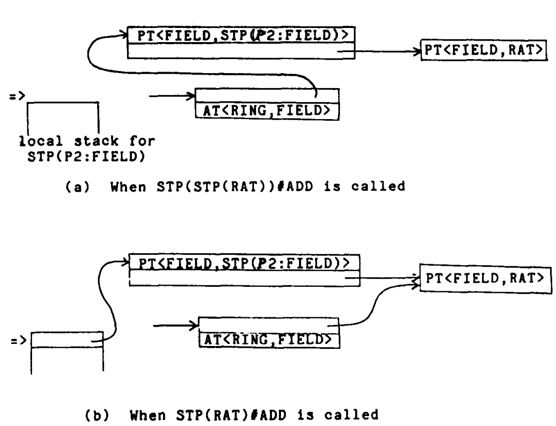


Fig. 5.3 Using local stack.

Note. Treatment of equality

Every sype or type in ℓ is supposed to have its own EQUAL function. The truth value of the equality between two objects of a type T is determined by the EQUAL of T, i.e. T#EQUAL. T#EQUAL may be defined explicitly in the type module T, or the system automatically generates code for T#EQUAL so that the EQUAL of the type by which objects of the type T are represented, is called. Thus T#EQUAL is regarded as one of the primitive functions on T.

In executing POLY(P: RING), if P#EQUAL is required, the actual EQUAL must also be retrieved from the type description that POLY(P: RING) receives. Thus we extend each procedure table PT⟨S, T⟩ so that its '0'-th entry contains T#EQUAL. For example, PT⟨RING, RAT⟩ in Fig. 4.2 is extended as shown in Fig. 6.1.

It is not necessary to use intermediate adaptor tables to find the actual EQUAL, since the location of EQUALs are always contained in the fixed entry of procedure tables. Therefore the retrieval of the actual EQUAL is faster than that of other primitive functions.

Conclusion

Problems in attaining separate compilation of type-parameterized modules with constraints on permissible actual types have been discussed and a simple and efficient implementation technique has been presented.

Although we have dealt with the type-parameterization mechanism in ℓ by the notion of sypes, the sype notion

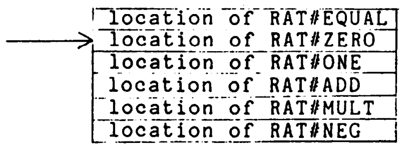


Fig. 6.1 Procedure table PT⟨RING, RAT⟩ extended.

itself is not essential to the discussion and can be substituted by other language constructs for constraints on permissible actual types, such as where clauses in CLU. In order to perform efficient run-time typeparameter binding, the implementation technique takes full advantage of the fact that the language ℓ does not allow circularity in dependency relationships among modules. Without this restriction on inter-module relationships, however, the efficiency of the technique would not be reduced excessively. Thus it should be observed that the discussions in this paper are general enough to apply to other languages with similar features.

The technique presented in this paper has been implemented in the ℓ programming system, which runs on DECsystem-20 and IBM 370 compatible machines.

Acknowledgements

The author wishes to express his appreciation to Professor Reiji Nakajima for patiently supervising this research.

References

1. Atkinson, R., Liskov, B. and Scheifler, B. Aspects of implementing CLU. Computation Structures Group Memo 167, MIT Laboratory for Computer Science (1978).
2. Liskov, B., Moss, E., Schaffert, J. C., Scheifler, B., and Snyder, A. CLU reference manual. Computation Structures Group Memo 161, MIT Laboratory for Computer Science (1978).
3. Nakajima, R., Honda, M. and Nakahara, H. Hierarchical program specification and verification—a many-sorted logical approach—. Acta Informatica (1980).
4. Nakajima, R., Yuasa, T., and Kojima, K. The ι programming system—a support system for hierarchical and modular programming—. Proc. of IFIP Congress 80, ed. S. H. Lavington, North-Holland Pub. Co. (1980).
5. Nakajima, R. Sypes—partial types—for program structuring and first order system ι logic. Research Report No. 22, Institute of Informatics, Univ. of Oslo (1977).
6. Teitelman, T. Interlisp reference manual. Xerox Palo Alto Res. Cen. (1978).
7. Yuasa, T. Supports for hierarchical software development—systems and mathematical methods—. Master's Thesis (1978).
8. Yuasa, T. Design and implementation of the ι language system —an interactive environment for modular programming—. (submitted for publication).
9. ADA reference manual. SIGPLAN Notices 6 (1979).

(Received November 26, 1981)