数理解析研究所講究録
第296巻 1977年 14-25

Proposal of Programming and Verification Scheme
— Program Verification Integrated with Structured Programming —

Reiji Nakajima,† Kyoto University & Oslo University
Michio Honda,‡ Kagawa University & Kyoto University
Hayao Nakahara,‡ Kyoto University

Summary

This paper describes a programming language and a verification system to construct and prove programs* with user-defined abstract data types. The design objective of the language is to uniformly describe programs, their formal specifications and supporting formal theories together with the characterization of the interrelations among these programming and verification concepts. On this language, rigorous program proofs become possible which match the modular and hierarchical program structures and concepts in the data abstruction environment.

§1. Introduction

The ι - iota system is an integrated system for developing and verifying well-structured programs whose design and implementaiton are in progress on DEC 20 System at Kyoto. The ι system provides an evironment in which a programmer cooperates with the system to build a good program, to prove it correct and to run it. These three functional facets are undertaken by the following subsystems:

(1) Program Developer [automates major part of coding job and helps the user to build structured and correct programs in intelligent ways.]

(2) Verifier

(3) Translator

These subsystems are highly integrated with each other on a newly developed language ι to constitute the whole ι system to provide an environment in which program verification is organically combined with programming methodology. This paper presents the method adopted for the ι verifier.

Author's present address: † Mathematics Institute, Oslo University, Oslo, Norway, ‡ Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan.
* Some examples of programs and program proofs are given in the appendices.

The ι verification system attempts to integrate program verification with the recently noted programming methodology-data abstraction. Data abstraction was originally introduced in [1], and has been remarkably developed and established the last few years. [e.g., 7, 12]

"Data abstraction should be useful in program proving." This is the point which is generally accepted. Until now, however, this seemingly obvious assertion has been repeatedly made without any sound evidence. [In [12] efforts have been made in order to combine verification with data abstraction, but there, consideration is taken only to prove the specifications of abstract data types of restricted kind (such as queues or stacks) to be correct for their implementations using some formalized models of particular kind (such as the abstract sequence.) Nothing is said about how these specifications can be applied to prove a program of an upper level which uses the abstract types.]

No example has been given in which a complete program with user-defined abstract data types is proved correct on a logically solid foundation. No discussion has been made as to how and on which foundation the correctness of such programs should be establised.

The programming and verification scheme in the ι verifier offers an answer to this issue. By the language and the proof method in this system, a complete program with data abstraction can be verified rigorously on a logically sound foundation.

1: With the scheme, one is able to organize various programming and verification concepts (such as operational and data abstractions, formal specifications and supporting formal theories) on all abstraction levels in such a way that the relations between these concepts and levels are clearly understood.

2: So called background theories or formal theories, which support the formal descriptions and verification discussions, are now disposed within the same syntactic frame work as the program itself, which makes the whole discussion lucid.

3: The user enjoys flexibility in that, to describe and prove part of a program, he can introduce a formal theory appropriate for the abstraction level on which the part is written.

We give the language description in §2 and the verification method in §3. The language features and the verificaiton method, however, are closely connected and hard to be seperately described.

§2. The ι Language - Language for describing programs, their formal specifications and supporting formal theories.

An input to the ι system is called a *program object* which consists of one or more *modules*. There are four kinds of *modules*, *procedure modules*, *type modules*, *category modules* and *theory modules*.

Procedure modules and *type modules* represent operational and data abstractions, respectively. A *category modules* defines a class of data types. A *theory module* defines a formal theory (in mathematical logic sense) on which program specifications are formulated and correctness proofs are conducted [See Apendix 1 for the description of *category modules*.]

Each *module* consists of an *interface part* and some other *parts*, which are either a *specification part*, a *pre-spec part* or a *realization part*.

An *interface part* is prepared for every *module* and contains the declaration of the *operations* (*ops*) (which are like procedures in Pascal) and *functions* (*fns*) with their domains and ranges. Syntactically, the *interface part* determines the external aspect of the *module*, i.e., only these *operations* and *functions* are visible to outer *modules*.

A *realization part* is prepared for each of the *type* and *procedure modules* and contains the implementation of the type and operational abstraction, respectively. (We borrow many of the syntactic constructs used in the *realization part* of a *type module* from CLU [4].)

A *specification part* is prepared for each *module* and contains the formal description of the *module*. It consits of *axioms*, *lemmas* and *rules* (of inference).

A *pre-spec part*, which may appear in a *type* or *procedure module*, contains the formal description of the *module* which reflects directly the implementation in the *realization part*. Usually, the *specification part*, which contains a more abstract description, is bridged to the *realization part* by the *pre-spec part* . Only the formulas in the *specification part* (not the *pre-spec part*) can be invoked in the verification discussion concerning the correctness of the other *modules* that *refer* to the *module*. Thus, the *specification part* determines the semantic aspect of the *module* seen from outer *modules*. The *realization part* and the *pre-spec part* are hidden from outer *modules*. To outer *modules*, only the *interface* and *specification parts* are visible which are independent of the actual representation and implementation taken in the *realization part*. [Refer to Figure 1 & 2 to materialize the language description]

We have adopted the "axiomatic description" (according to [5]) to write the formulas in the *specification* and *pre-spec parts*. (The free variables are universally quantified.) Note that the *interface part* provides the so called "functionality definitions" of the "algebraic specifications" proposed in [5].

In addition to the user-defined *modules*, some *system modules* are implicitly built in by the system and can be *referred* from all *modules* without explicit declaration of *refer*. The *modules* for the primitive types are such examples [Refer to Figure 3 for the *system type module* for *sequence* (array like structure of variable length)].

Finally the reader might have noticed that this language requires many syntactic redundancies. These features are intensionally introduced to clarify the program structure and concept. Since major part of coding is automatated in the ι system, this does not cause a burden on the user's side.

Moreover, a *program object* need not be complete when it is inputed to the language translator or the verifier of ι. Some *parts* of some *modules* may be left open. The system requires only those components which are logically needed to perform the task indicated by the user.

§3. Verification method

A. A *specification part* consists of some formulas which are either an *axiom*, a *lemma* or a *rule*(of inference).

In the case of a *theory module*, the *axioms* are implicit or explicit definitions of the *functions* which are declared in the *interface* of the *module*. The validity of the *axioms* and *rules* of a *theory module* is assumed to have been established in advance and need not be proved . In a *procedure* or *type module*, however, the *axioms* and *rules** must be proved to be satisfied by the *realization part* of the *module*. In a *category module*, the *axioms* and *rules** must be validated for those *type modules* which are declared to belong to this *category*.

* The *rules* are generated from the *axioms* by the "rule genrator" which is built in the system.

The *lemmas* are formulas which are proved in the formal theory determined by the *axioms* and *rules*. When it comes to proving the *specification part* to be valid for the *realization part*, the less formulas has the *specification part*, the less trouble. So we want to keep the number of the *axioms* minimum. When the *specification part* is invoked in attempting to verify another *module*, it will be nicer if the *specification part* contains some more useful formulas. This is why we introduce the *lemmas*.

In addition to the user-defined *rules*, generator induction is implicitly built in each *type module*, which can be applied in the same way as user-defined *rules*.

B. Here, we describe how verification proceeds typically in ι. Suppose the user wants to verify the *specification part* of his 'main' *procedure module*. The *axioms* placed in the *specification part* are to be proved correct for the *realization part*. For each the *axioms*, the corresponding verification condition (V.C.) is to be generated from the code in the *realization part* together with the loop invariants attached by the user.

To generate and prove the V.C., we adopt the 'top-down method' which is suggested in [3]. (Each *operation* call is replaced by the equivalent simultaneous execution of *assignments* by introducing some new functions.) Since some *functions* and *operations* defined in outer *modules* are called in the *realization part* concerned, the resulting V.C. contains these functions (some have been introduced to replace an *operation*). On the other hand, the *axiom* being proved may contain some *functions* defined in a *theory module*, and if so, these *functions* will appear in the V.C., too.

Now, to prove the V.C., invoked are the *specification part* of each *module* which defines some of these *functions* and *operations*. Often, to complete the proof, the user-system interaction finds it necessary to add some formulas as *lemmas* or *axioms* in some of the *specification parts* (each of which, of course, must be validated in due time.)

We suggest this top-down process of generating and verifying V.C.'s suits the data abstraction and modular programming environments since it reflects the modular and hierarchical program structures. [See Appendix 2 for an illustration of this process.]

On the other hand, it often happens that the proof of a program requires to be conducted on a formal theory appropriate for the abstraction level on which the program is written. Thus we introduce the syntactic concept - *theory module* so that the user can prepare a formal theory as appropriate.

In the conventional verification schemes, the dispositions of the supporting formal theories or back ground theories among the other programming and verification concepts were left rather vague. Here, after putting and arranging all thing together in the same sphere using *modules* and *parts*, we feel that the programming and verification concepts such as data and operational abstractions, their formal descriptions, and formal theories are disposed in the right position suitable for their roles.
(Moreover what is striking is that, in the scheme of the ι verifier, there is no longer distinction between formal theories and program. They are treated in the same way.)

C. Many of the formulas, which are needed to enter into the *specification part* of a *type module*, tend to be in the form of an equivalent relation between compositions of *functions* (e.g. *axiom* 2 in *specification type* POLYNOMIAL in Figure 2). If one uses Hoare's system [1], however, it is generally difficult to prove such an equivalence relation to be valid for the implementations of the functions especially when the implementations contain some loops. This is the main reason why we introduce the *pre-spec parts*.

The *pre-spec part* of a *type module* contains such formulas that are written assuming the knowledge of the actual representation of the abstract data in the *realization part*, and so, is easier to prove valid for the *realization part*. In stead of trying to verify the *axioms* in the *specification part* directly from the *realization part*, one deduces them from those formulas in the *pre-spec part* and then validates these formulas for the *realization part*. This usually makes the things easier. [Appendix 4 illustrates how this verification process works.]

References

[1] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Comm. ACM, Vol.12, (1969), 576-580.

[2] Hoare, C.A.R., "Proof of Correctness of Data Representations," Acta Informatica, Vol.1, (1972),271-281.

[3] Hoare, C.A.R. and Wirth, N., "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol.2, (1973), 335-355.

[4] Liskov, B., "An Introduction to CLU," New Direction in Algorithmic Languages 1975, 139-156, IRIA, Paris, 1976.

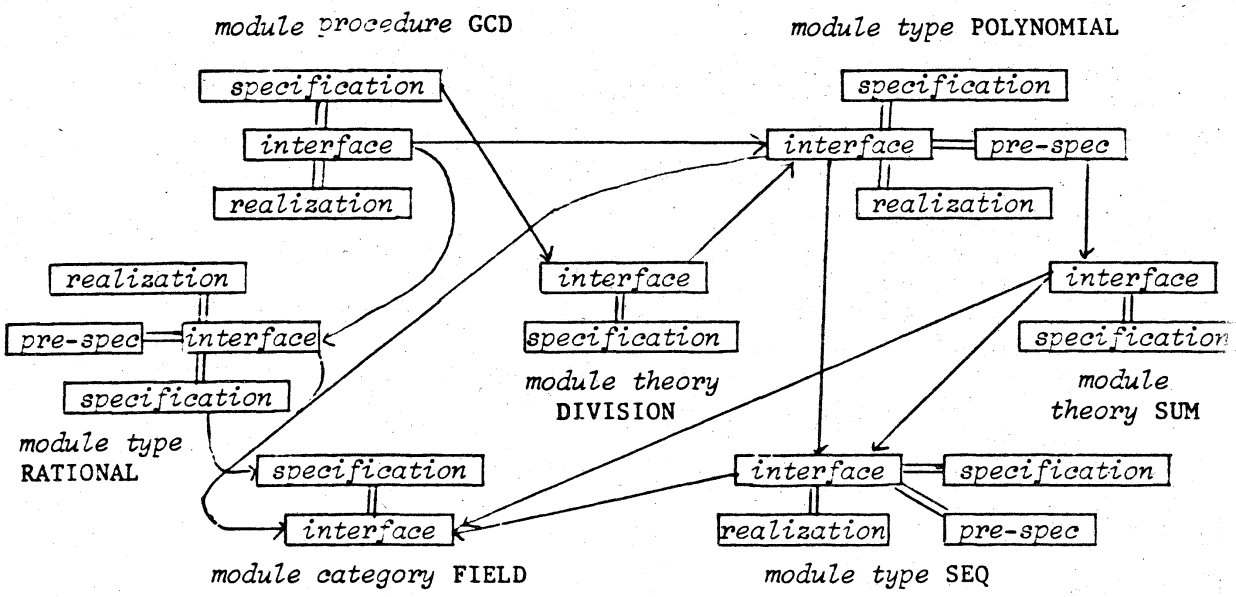
[5] Liskov, B. and Zilles, S., "Specification Techniques for Data Abstraction," IEEE Transaction on Software Engineering SE-1, Vol.1, (1975), 7-19.

[6] Wegbreit, B. and Spitzen, J., "Proving Properties of Complex Data Structures," J. ACM, Vol.23, (1976), 386-396.

[7] Wulf, W., London, R. and Shaw, M., "Abstraction and Verification in ALPHARD," New Direction in Algorithmic Languages 1975, 217-295, IRIA, Paris, 1976.

APPENDICES

Figure 1 - *Program Object* to Compute G. C. D. of Two Polynomials (I)



[A] → [B] reads *part* A *refers module* B (represented by the *interface part* of B). If the *interface part* of a *module* C *refers module* D, it means that all *parts* of C *refer* to D. Note that a *module* (the *specification* and *interface parts* of it) is visible only to those outer *modules* (or *parts*) from which an arrow is directed to the *module*.

Figure 2 — *Program Object* to Compute G.C.D. of Two Polynomials (II)

```
⌈interface category FIELD;
    thru fn  ZERO: → @ as 0 ;
                  :
                  :
             MULT: (@,@) → @ as @*@;
⌊end interface
```

[@ denotes one of the *types* which belong to FIELD.]

```
⌈specification category FIELD;
    var X,Y,Z: @;
    axiom 1: 0 + X = 0;
      :
      :
⌊end specification
```

```
⌈interface type RATIONAL; [@ denotes RATIONAL]
    is FIELD
    and thru fn ORDER: (@,@) → bool as @<@;
⌊end interface
```

```
interface type RATIONAL;
  thru fn ZERO: → @  as 0;
                :
                :
          MULT:(@,@) → @ as  @*@ ;
          ORDER:(@,@) → bool as @<@;
  end interface
```

```
⌈interface procedure GCD;
    thru fn GCD: (POLYNOMIAL(RATIONAL), POLYNOMIAL(RATIONAL))
                 → POLYNOMIAL(RATIONAL);
⌊end interface

⌈specification procedure GCD;                          ⌈DIVISION # GCD denotes the GCD
    refer DIVISION:                                    | defined in module DIVISION,
    var X,Y: POLYNOMIAL(RATIONAL);                     | while GCD(X,Y) is the function
    axiom 1: GCD(X,Y) ≅ DIVISION # GCD(X,Y);           ⌊ defined in module GCD.
⌊end specification


⌈realization procedure GCD;
    fn GCD(X,Y:POLYNOMIAL(RATIONAL)) return (Z:POLYNOMIAL(RATIONAL))
    <= avec  POLYNOMIAL do
       if DEG(X) < DEG(Y) then <X,Y> := <Y,X> end if;  [simultaneous assigment]
       while Y≠0 do
          X := (TERM(COEF(DEG(Y),Y),0)*X)-(TERM(COEF(DEG(X),X),DEG(X)-DEG(Y))*Y);
          if DEG(X)<DEG(Y) then   <X,Y> := <Y,X> end if
       end while
       end avec  ⌈avec POLYNOMIAL is used to omit POLYNOMIAL # .  Similar to
    end fn       ⌊with in PASCAL.
⌊end realization

⌈interface theory DIVISION(T:FIELD);
    thru fn GCD: (POLYNOMIAL(T), POLYNOMIAL(T)) → POLYNOMIAL(T);
            DIV: (POLYNOMIAL(T), POLYNOMIAL(T)) → bool;
            EQUIV: (POLYNOMIAL(T), POLYNOMIAL(T)) → bool as POLYNOMIAL(T)≅POLYNOMIAL(T);
⌊end interface


⌈specification theory DIVISION(T:FIELD);
    var W,X,Y:POLYNOMIAL(T:FIELD);
    axiom 1 : DIV(X,Y) ≡ ∃ W. X=W*Y;
            :
⌊end specification

⌈interface type POLYNOMIAL(T:FIELD);
    thru fn MULT: (@,@) → @ as  @*@;   [MULT(X,Y) can be abbreviated as X*Y by as]
            ZERO:  → @ as 0;
            COEF: (@,int) → T;
            DEG: @ → int;

              :

⌊end interface

⌈realization type POLYNOMIAL(T:FIELD);
    rep =SEQ(T);
    fn MULT(X,Y:rep) return (Z:rep)  [rep is like cvt in CLU]
        :
    end fn
        :
⌊end realization
```

```
⌈specification type POLYNOMIAL(T:FIELD);
    var X,Y,Z: @; ... ;                    [@ stands for POLYNOMIAL(T)]
    axiom 1: X*1 = X;
          2: X*(Y*Z) = (X*Y)*Z;
      :
    lemma 1: X≠0 ⊃ COEF(X,DEG(X)) ≠ 0;
      :
⌊end specification
```

```
⌈pre-spec type POLYNOMIAL(T:FIELD);
    refer SUM(T);
    rep = SEQ(T);                          ⌈ For COEF : POLYNOMIAL → T, ↓COEF:
    var X,Y: rep; I: int;                  | SEQ(T) → T  is the implementation of
    avec SUM, SEQ;                         ⌊ COEF.
    axiom 1: CONT( X* Y,I) = SUM(I,I, X, Y);
      .   2: ↓COEF( X,I) = CONT( X,I);
      :
⌊end pre-spec
```

```
⌈interface type SEQ(T:FIELD);
    thru fn CONT: (@,int ) → T;
          LENGTH:@ → int;                  ⌈For operations, the parameters on the left of |
       op SET: (@|int,T);                  | are called by variables and those on right of |
          :                                ⌊ are by value like PASCAL.
          :
⌊end interface
```

```
⌈specification type SEQ(T:FIELD);
    var X,Y: @; I:int;
    axiom 1:(∀I.CONT(X,I) = CONT(Y,I))∧ LENGTH(X) = LENGTH(Y) ⊃ X=Y;
      :
⌊end specification
```

```
⌈interface theory SUM(T:FIELD);
    thru fn SUM: (int,int,SEQ(T),SEQ(T)) → T;
⌊end interface
```

```
⌈specification theory SUM(T:FIELD);
    refer SEQ(T);
    var X,Y: SEQ(T); I,J:int;
    axiom 1: J<0 ⊃SUM(I,J,X,Y) = 0;
      :
    rule(P) 1: goal P(SUM(I,J,X,Y));          [P is a formula variable]
              subgoal 1: J<0 ⊃P(0);
                      2: 0≤J ⊃P(SUM(I,J-1,X,Y,)+CONT(X,J)*CONT(Y,I-J));
      :
⌊end specification
```

Figure 3 — *System Module* for *Type Sequence*

```
interface type sequence (T:any);
   thru fn create: int → @;
           length: @ → int;
           cont: (@, int) → T;
        op assign: (@|int,T);  [denoted as  x[i]:=<exp> in the actual contexts]
end interface

specification type sequence (T: any);
   var X,Y: @; I,J:int; S:T;
   axiom 1: length(create(I)) = I;
         2: 0<I≤length (X)⊃ cont(assign (X,I,S),I) = S;
         3: 0<I≤length(X)∧ 0<J≤length(X) ∧ I ≠ J
            ⊃ cont(assign(X,I,S),J) = cont(X,J);
         4: length(X)=length(Y)∧∀I.(0<I≤length(X) ⊃ cont(X,I) = cont(Y,I))
            ≡ X=Y;
      :
end specification
```

Appendix 1 - *Category Modules*

CLU [4] offers a programming mechanism called "type generator" by which a cluster can define a class of types by receiving a type as a parameter. Here, the implementation of the cluster can not assume any structure of the type passed as a parameter. (e.g. cluster STACK (T:type) defines the class of all stacks whose entries contain an element of an arbitrary type T.)

This approach does not cause any inconvenience for such data types as stacks and queues. This is not the case, however, for the type of polynomials, for example

In figure 2, *module* POLYNOMIAL defines the type of the polynimials with one variable over a coefficient field T which is passed as a parameter. Obviously the structure of T is involved in the implementation and specification of POLYNOMIAL, i.e. each *part* of *module* POLYNOMIAL is written assuming the structure of T as a field.

From a different stand point, it can be said that *module* POLYNOMIAL is written with no assumption other than that T has the abstract property of field. The abstract property is that, on T, are defined such *functions* as ZERO, ONE, ADD, ... , MULT which are bound by such *axioms* as associativity and commutativity.

Category module FIELD defines a class of the data types which have this property. The *interface part* of FIELD contains all the *functions* such as ZERO, ONE, ..., MULT and the *specification part* contains the *axioms* mentioned above.

Now the parameter T to *type module* POLYNOMIAL is declared as T: FIELD. (Our convention includes the mechanism of type generator in CLU since there is a *category any*, which is the class of all *types*.) *Type module* RATIONAL in Figure 2 defines one of the data types which belong to FIELD. [See the two way in which the *interface part* of RATIONAL is written.]

Appendix 2 - Proving *Procedure Module* GCD

We want to prove the correctness of *procedure module* GCD which is intended to compute the g.c.d. of two given polynomial with one variable over the field rational. [Figure 1 & 2]

On *module* DIVISION, the formal theory of polynomial division and g.c.d. is developed, where *function* GCD is defined from another (predicate) *function* DIV (DIV(X,Y) reads X is divisable by Y). Now, *axiom* 1 in *specification procedure* GCD:

GCD(X,Y) $\tilde{=}$ DIVISION # GCD(X,Y) for POLINOMIAL X,Y

asserts that *function* GCD computed by *module* GCD is $\tilde{=}$ to the other GCD defined in *module* DIVISION. ($X \tilde{=} Y$ means that $X = c*Y$ for some c in field T) Now to prove *axiom* 1, $Z \tilde{=}$ DIVISON # GCD(X_0, Y_0) is the goal formula since Z receives the value of GCD(X,Y) in *module* GCD. (X_0, Y_0 stand for intial values for X, Y). From this goal, the V.C.:

$(DEG(Y) \leq DEG(X) \wedge GCD(X,Y) \tilde{=} GCD(X_0,Y_0) \wedge Y = 0) \supset X \tilde{=} GCD(X_0,Y_0)$ etc.

are generated. Now, these V.C.'s are to be deduced from the *axioms* and *lemmas* in the *specification parts* of *modules* DIVISION and POLYNOMIAL.

Appendix 3. Proving Equalities

Here we discuss the important issue of the equality. As explained in [4], the equality predicate *equal* : (@, @) → *boolean as* @ = @ (for the program notations, see Figure 2) is assumed to be placed implicitly in each *type module*.

If one wants to prove an equality *equal* (X, Y) on an abstract data type, this equality is to be translated into an equality on the type structure which represents the abstract data type. [In Figure 2, to prove POLYNOMIAL# *equal*(X, Y) for POLYNOMIAL X, Y, *realization type* POLYNOMIAL is looked at. Since *type* SEQ is used as the representation for POLYNOMIAL (*rep* = SEQ(T)), SEQ#*equal*(X', Y') should be proved for SEQ X', Y' representing X, Y, respectively. Now *specification type* SEQ is searched and *axiom* 1 is found which gives a condition for SEQ equality.] The user should place some *axioms* for the equality in the *specification part* of a *type module* if he wants to establish the equality on the data type.

Incidentally, if one wants to prove an equality on *type* SEQ, which is represented by a primitive type *sequence*, then the *system type module* for *sequence* is referred, in which *axiom* 4 is the equality axiom. [See Figure 3 and Appendix 4.]