数理解析研究所講究録
363巻 1979年 247-277

RIMS Kokyuroku No. 363, Research Institute for Mathematical Science Kyoto University.

INTERACTIVE THEOREM PROVING ON HIERARCHICALLY AND MODULARLY STRUCTURED SETS OF VERY MANY AXIOMS*

Michio Honda**
Kagawa University
Dept of Information
Science
Takamatsu

Reiji Nakajima**
Kyoto University
Research Institute for
Mathematical Sciences
Kyoto

July 1979

A large number of axioms are often involved in the proof of a single theorem in many realistic applications of mechanical theorem proving such as formal verification of programs whose program domains are determined by user-defined axioms. There, fully automatic proofs are unrealistic due to the obvious constraints though a powerful machine support is highly desired. It is suggested that some meaningful structuring of theories can ease the difficulties. Several strategies are proposed to enhance efficient interactive non-resolution proofs on hierarchically and modularly structured theories with many axioms. Use of such strategies is illustrated in their application to verification of hierarchical programs with abstraction mechanisms.

Key words: interactive theorem proving, program verification, non-resolution theorem proving, structured programming, hierarchical and modular program structures

* An extended abstract of this paper was presented at IJCAI '79.
** Order is not significant

1. Introduction

For many practical applications of mechanical theorem proving, a large number of axioms are often involved in the proof of a single theorem. An example of such applications is the theorem proving required in formal program verification, where the program domains are determined by a number of user-defined axioms.

It seems that most techniques so far developed for the mechanical theorem proving are not directly applicable in such a situation. Due to the obvious time and memory constraints, fully automatic proofs are not realistic to cope with such situations. Thus, man-machine interactive non-resolution proof methods are inevitable, where the user keeps well aware of what is being done in each stage of the ongoing proof, understanding the meaning of the formulas which are generated during the proofs. With human interventions, it still can be highly difficult to conduct proofs on a large theory with many axioms. Thus some organized methods are desired to get around with this difficulty. Here we suggest that some meaningful structuring of the axiom set will ease the difficulty.

The motivation of our study presented in this paper is derived from a software development called IOTA system at Kyoto university. The IOTA system will be an interactive system for developing, debugging, verifying and executing programs written in language IOTA, where the language is designed to support hierarchical and modular program development. It is not possible to give the details of the features of the language and the system here. To make the story short, verification of programs written in language IOTA requires theorem proving on hierarchically and modularly structured theories with a large number of user-defined axioms. (In fact, we are as much concerned with how such development of theories should be made by man-machine interaction as how proofs should be done on them though this will be the subject of another paper. Often need of elaboration or modification of user-given axioms is found during proofs, which means that axiom-writing and theorem-proving must go together to some extent and this is another reason to make the system interactive.)

Section 2 presents what we mean by hierarchically and modularly structured theories. (This preparation may look too long for the modest amount of result given in the subsequent sections but should be necessary to have the proof strategies understood.) Then from section 3 through 6, we present some strategies which are intended to enhance the efficiency of interactive theorem proving on such theories.

These strategies are being implemented as IOTA prover, subsystem of the IOTA system, which will be called in the environment of program development, verification and debugging. (The most updated and readable introduction to IOTA system and language is [4]. It also contains pointers to other related literatures.)

The prover contains an automatic proof facility in addition to the proof checking facilities. The man-machine interaction (i.e. proof checker) exploits the strategies in order to reduce the proof that is beyond the limit of the capabilities of the automatic subsystem to one within the limit.

The first verison of IOTA prover currently runs on a DEC SYSTEM 20. The implementation of a more powerful version is under way.

Related works: It seems that no previous work has attempted to exploit the structuring of theories to facilitate mechanical proofs. Clear[2] seems to be somewhat in a similar direction though it uses algebraic axioms while we use first order logic. (It is beyond the scope of this paper to compare the two approaches.) The concepts of user-developed reduction rules (Section 4) are introduced earlier in LCF[3], while an idea similar to theory-focusing (Section 3) is used in [8] but with a different objective. [1] surveys numerous works on non-resolution theorem proving.

2. Hierarchical and modular theory development

Language IOTA provides a syntax by which one can build up formal theories for program specifications and write their program implementations. In order to make the discussion precise, we fix a logical system called IOTA logic [5] so that as far as specifications are concerned, IOTA is a language to form theories of IOTA logic. We seize data types in programming simply as sorts in the many-sorted logic, and so IOTA logic is a derivation of the many-sorted first order logic. Reflecting the idea of data abstraction, each sort in IOTA logic is associated with a structure which is said to be basic on the sort. The basic structure of a sort 's' is a finite set of functions PR<s> called the primitive functions on s and a finite set of axioms BA<s> called the basic axioms of s. The formulas in BA<s> are supposed to characterize properties of the functions in PR<s>. IOTA logic has the usual set of logical axioms and (logical) rules of inference as an ordinary many-sorted first order logic. In addition, the rules of IOTA logic include the generator induction rules on some of the sorts. The generator induction rule on a sort s is made of all primitive functions on s whose range is s in the well-known manner. (Examples will be given in due course.) Those sorts whose generator inductions are included among the rules of IOTA logic are called types. The rest of the sorts are called sypes. (In short we consider a fixed model for each type t, which is the variable-free terms of sort t generated from the functions in PR<t>. On the other hand we do not fix any specific model for a sype.) We wait to see how sypes can be useful until Section 5.

Language IOTA supports hierarchical and modular program building, i.e. the notion of a program in IOTA consists of a hierachy of modules, each of which is specified seperately. Thus as far as specication structures are çoncerned, program development in language IOTA is to build up theories in IOTA logic. A theory generated in this manner should be hierarchically and modularly structured. Here a module defines a pieces of the theory. We designate the theory presented by a module 'q' by TH<q>. (Later on, we will give a more precise definition of TH<q>.)

There are basically three kinds of modules --- type, sype and procedure modules.

The following is an example of a type module. (Examples given in this paper are not quite faithful to the legitimate syntax rule of language IOTA.)

```
   interface type NN
     fn ZERO : -> @ as 0
        SUC  : @ -> @
++      LESS : (@,@) -> BOOL as @≤@
++      EQUAL: (@,@) -> BOOL as @=@
   end interface

   specification type NN
     var x,y,z, u,v: @
     axiom 1: SUC(x) = SUC(y) => x=y
           2: ~SUC(x) ≤ x
           3: x≤y => SUC(x)≤SUC(y)
++         4: x≤y or y≤x
++         5: ( x≤y & y≤x ) => x=y
++         6: ( x≤y & y≤z ) => x≤z
++         7: x=x
           8: x=y => SUC(x)=SUC(y)
++         9: ( x=y & u=v ) => ( x≤u ) = ( y≤v )
++        10: ( x=y & u=v ) => ( x=u ) = ( y=v )
   end specification
```

This module presents the basic structure of a type NN (or the natural numbers). We simply say the module presents the sort of NN. The presentation is devided into two parts; the interface part which declares the primitive functions with their domains and range and the specification part in which the basic axioms are placed. @ denotes the type presented by the type module, which is NN in this case. By "as" a notational abbreviation is introduced for a function name. So LESS(x,y) may be written as x≤y. All free variables occurring in the axioms are universally quantified. Since the sort NN is a type, the generator inductions are logical rules of IOTA logic which are in the form of:

$$\frac{P\{0/x\}\ ,\quad P \Rightarrow P\{SUC(x)/x\}}{P}$$

where P is any formula, x is a variable of sort NN and P{t/x} stands for the substitution of a term t of sort NN for x in P. By saying that the rules are logical, we mean in practice that whenever the user introduces a type (module), the language processor automatically includes the generator induction rules of the type in the proof system.

Incidentally, the eqality is a function whose range is the type BOOL. In IOTA logic, there are no predicate symbols. Instead there is a special type BOOL and all terms of sort BOOL are the atomic formulas. The primitive functions on BOOL are NOT, OR, EQUAL etc., which are equivalenced with logical operators ~, or, <=> etc. by axioms.

Once we have written a type module, we may add more functions on the type by introducing them in a procedure module. These functions are said to be non-primitive because they are not included in the generator induction rules.

```
interface procedure NNMAX
  fn MAX : (NN,NN) -> NN
     MIN : (NN,NN) -> NN
end interface

specification procedure NNMAX
  var x,y:NN
  axiom 1: x≤y => MAX(x,y)=y
        2: x≤y => MIN(x,y)=x
        3: MAX(x,y)=MAX(y,x)
        4: MIN(x,y)=MIN(y,x)
      [ equality axioms for MAX, MIN ]
end specification
```

NNMAX is built upon NN and BOOL. (Notice that when we presented NN, we assumed that there was already a type module called BOOL.) This means that TH<NNMAX> is an extension of TH<NN> and TH<BOOL>. (We shall make this situation precise later.)

$$\begin{tikzcd}
 & TH\langle NNMAX\rangle \\
TH\langle NN\rangle \arrow[ur] & \\
 & TH\langle BOOL\rangle \arrow[ul] \arrow[uu]
\end{tikzcd}$$

We give another type module INTPOLY or the type of polynomials of a single variable with integer coefficients.

```
interface type INTPOLY
  fn ZERO : -> @ as 0
     TERM : (INT,NN) -> @
     TM   : (NN,@) -> @
     CM   : (INT,@) -> @ as INT.@
     ADD  : (@,@) -> @ as @+@
     COEF : (@,NN) -> INT
     DEG  : @ -> NN
end interface

specification type INTPOLY
  var x,y,z:@; m,n:NN; i:INT
  axiom 1: COEF(0,n)=0
        2: DEG(0)=0
        3: COEF(TM(n,x),n+m)=COEF(x,m)
        4: m<n => COEF(TM(n,x),m)=0
        5: x≠0 => DEG(TM(n,x))=n+DEG(x)
        6: x≠0 => COEF(x,DEG(x))≠0
        7: DEG(x)<n => COEF(x,n)=0
        8: (∀n.COEF(x,n) = COEF(y,n)) => x=y
        9: COEF(i.x,n) = i*COEF(x,n)
       10: COEF(x+y,n) = COEF(x,n) + COEF(y,n)
       11: DEG(x+y) ≤ NNMAX#MAX(DEG(x),DEG(y))
       12: COEF(TERM(i,n),n)=i
       15: m≠n => COEF(TERM(i,n),m)=0
       16: i≠0 => DEG(TERM(i,n))=n
       17: TERM(0,n)=0
end specification
```

EQUAL on INTPOLY and the equality axioms are implicit. We have not defined ≠ or < on NN. A≠B and A<B should be regarded as abbreviations of ~A=B and B≤A, respectively. Language IOTA has, in fact, this kind of macro-like facility. ZERO is the

polynomial zero, TERM(i,n) is $i.x^m$, TM(n,Q) is x^m*Q, DEG(Q) gives the degree of polynomial Q, CM(i,Q) is Q multiplied by an integer i and COEF(Q,n) is the n-th coefficient of Q. We assume that we have already a type INT or the integers on which functions like *,+,0,1 and = are primitive. A same notational abbreviation is used for functions on different sorts. For instance, EQUAL's on INTPOLY, NN, INT are all denoted by =, but the language processor will be able to distinguish them by type check. In language IOTA, the proper symbol for a function is the pair M#f, where f is the function name and M is the name of the module q in which the function is introduced. So, = on INTPOLY is properly INTPOLY#EQUAL. BUt <module name># is omitted in many cases as long as no confusion can occur.

The next example is the procedure module DVS which is built on INTPOLY, where DVS(x,y) reads x is divisible by y.

```
interface procedure DVS
  fn MULT : (INTPOLY,INTPOLY) -> INTPOLY as INTPOLY*INTPOLY
     DVS  : (INTPOLY,INTPOLY) -> BOOL
end interface

specification procedure DVS
  var x,y,z:INTPOLY; n:NN; i:INT
  axiom 1: 0*x=0
        2: x*y=y*x
        3: (x*y)*z=x*(y*z)
        4: (x+y)*z=x*z+y*z
        5: (i.x)*y=i.(x*y)
        6: TM(n,x)*y=TM(n,x*y)
        7: TERM(i,n)*x=i.TM(n,x)
        8: DVS(x,z) <=> ∃y.x=y*z
end specification
```

For different modules p and q, p is said to *depend directly on* q iff either (1) q is a type or sype module appearing in the interface part of p, or (2) at least one axiom in the specification part of p contains a function which is introduced by q. For instance, INTPOLY depends directly on NNMAX, INT, NN etc. p is said to *depend on* q iff either (1) p depends directly on q or (2) there exists a module r such that p depends directly on r and r depends on q. p is said to be *self-contained* iff p does not depend on any module. The module BOOL* is self-contained. we say that p is hierarchical iff either (1) p is self-contained or (2) the modules on which p depends are all hierarchical. It is easy to see that, if p is hierarchical, there is no module q such that p depends on q at the same time q depends on p. This means that there can be no circular chain of depending relations among hierarcnical modules. The syntax of IOTA allows only hierarchical modules and any violation will be detected by the processor.

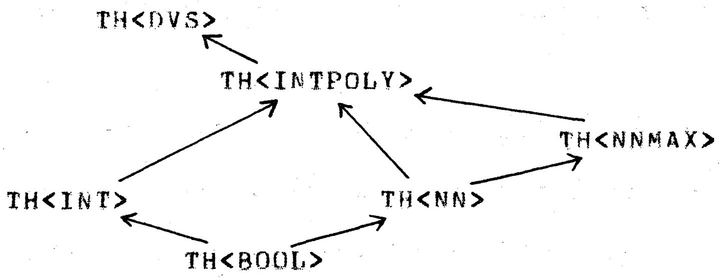
Here we are ready to make precise what is meant by a theory in IOTA logic. Although there can be different ways to define the notion of theories in IOTA logic, the following will be the most convenient for this paper. A theory T is a triple <S, F, A> where S is a set of sorts, F is a set of functions, and A is a set of formulas called the non-logical axioms of T such that

* In this paper, we are speaking informally, and so we conveniently confuse a type (sype) module name with the name of the type (sype) presented by the type (sype) module.

each function occurring in at least one of the elements of A is included in F and that for each function $f:(s1, s2, ..., sn) \to s0$ in F, si is in S for all $i=0,...,n$. we can define the notion of proofs and theorems of T in the usual manner. For two theories $T1 = <S1, F1, A1>$ and $T2 = <S2, F2, A2>$, T2 is said to be an *extension* of T1 iff $S1 \subseteq S2$, $F1 \subseteq F2$ and $A1 \subseteq A2$. By the *joint* of T1 and T2, we mean the theory $<S1 \cup S2, F1 \cup F2, A1 \cup A2>$.

Now we define the theory TH<p> of a module p, which is the semantics of the syntactic p. Let p depend on modules q1,q2, ..., qn for which TH<qi> are assumed to be defined already. Let the joint of TH<qi>, for i = 1, ..., n be $<S', F', A'>$. Then $TH<p> = <S, F, A>$ where $S = S' \cup S0$, $F = F' \cup F$, $A = A' \cup A$ such that F is the functions introduced by (the interface part of) p, A is the formulas presented by (the specification part of) p and S0 is empty if p is a procedural module whereas $S0 = \{ s \}$ if p is a type or sype module which presents the sort s. (Note that this definition of TH<p> is well defined because p is a hierarchical.)

There are several modules called system modules such as NN, BOOL, INT, which are built-in in the language. To program with IOTA is to write modules upon others starting with the system modules. Namely it is to extend theories in IOTA logic starting with the theory determined by the system modules.



The last example in this section is the type module for integer arrays.

```
interface type INTARRAY
  fn CREATE : (NN,INT) -> @
     HIGH   : @ -> NN
     FETCH  : (@,NN) -> INT
     STORE  : (@,NN,INT) -> @
end interface

specification type INTARRAY
  var x,y:@; m,n:NN; i:INT
  axiom 1: HIGH(CREATE(n,i))=n
        2: n≤HIGH(x) => FETCH(STORE(x,n,i),n)=i
        3: m≤HIGH(x) & n≤HIGH(x) & m≠n
          => FETCH(STORE(x,n,i),m)=FETCH(x,m)
        :
        :
end specification
```

The subscript of x:INTARRAY runs over NN from 0 to HIGH(x). FETCH(x,n) may be written as x[n]. CREATE(n,i) creates x such that HIGH(x) = n and x[m] = i for all 0≤m≤n.

3. Module-wise development of reduction and simplification rules

Normally, the reduction rules used in our proofs are from among the logical reduction rules i.e. the converse of the logical inference rules of the IOTA logic. The logical reduction rules are, however, designed to be valid generally on arbitrary theory and, therefore, tend to be rather inefficient. Thus, it is desirable to generate reduction rules on a specific theory from the (non-logical) axioms of the theory and use them when appropriate. (Notice that the soundness of such non-logical rules must be guaranteed, for which some kind of machine support is desired. This point seems to have received little attention so far.) But when the theory is large, it is not necessarily easy for the user to generate efficient non-logical reduction rules. An immediate application of theory modularization is in the module-wise development of reduction rules. It would be convenient to develop reduction rules on each specific subtheory defined by a module because the axioms given by a single module are supposed to relate closely to each other. The rules thus developed will not, of cause, be generally applicable but powerfull on the subtheory.

Given a module M which presents the following axioms where Q1, Q2, Q3, R are some predicate symbols.

axiom 1: Q1(x) => R(x)

```
        2: Q2(x) => R(x)
        3: Q3(x) => R(x)
```

Assume that, the predicate symbol R does not appear in any modules on which M depends. (i.e. There exists no axiom of TH<M> that contains R other than the axioms above.) Then from axioms 1, 2, 3, the following reduction rule is worked out:

```
   rule R1 ( P ): / P is a syntactic variable /
     goal   : P => R(x)
     subgoal:  P => Q1(x) or Q2(x) or Q3(x)
```

An application of this rule reduces

```
  (3.1) S(x) => R(x)   [ S(x) is a formula. ]
```

to

```
  (3.1.1) S(x) => Q1(x) or Q2(x) or Q3(x)   .
```

The following reduction rule would be worked out from the axioms of INTARRAY.

```
   rule R2 (P)
     goal     : P(FETCH(STORE(x,n,i),m))
     subgoal1: m=n & n≤HIGH(x) => P(i)
            2: m≠n & m≤HIGH(x) & n≤HIGH(x) => P(FETCH(x,m))
```

An application of rule R2 as well as some logical reduction rules reduces the formula (3.2) to the formulas (3.2.1) and (3.2.2)

```
  (3.2) ( (i<k => A[i]≤A[k]) & (m≤i<j≤k => A[i]≤A[j]) &
          (n≤i<j≤HIGH(A) => A[i]≤A[j]) & k≤HIGH(A) &
          k<n & A[k+1]≤A[k] )
```

```
       =>  ( i<k+1 => STORE(STORE(A,k,A[k+1]),k+1,A[k])[i]≤A[k] )
```

[This formula is a verification condition for a program for bubble sorting. "A" is a variable of sort INTARRAY. i,j,k,m,n are variables of sort NN.]

```
  (3.2.1) ( k+1=i & i≤HIGH(A) & (i<k => A[i]≤A[k]) &
            (m≤i<j≤k => A[i]≤A[j]) & (n≤i<j≤HIGH(A) => A[i]≤A[j])
          & k≤HIGH(A) & k<n & A[k+1]≤A[k] )
         => ( i<k+1 => A[k]≤A[K] )

  (3.2.2) ( k+1≠i & k+1≤HIGH(A) & i≤HIGH(A) &
            (i<k => A[i]≤A[k]) &
            (m≤i<j≤k => A[i]≤A[j]) & (n≤i<j≤HIGH(A) => A[i]≤A[j])
          & k≤HIGH(A) & k<n & A[k+1]≤A[k] )
         => ( i<k+1 => STORE(A,k,A[k+1])[i]≤A[k] )
```

Both formula (3.2.1) and (3.2.2) can be easily reduced to true.

4. Theory Forcusing

One of the main difficulties with a proof on a large theory derives from the wide selection of axioms to invoke at each step of the proof. It is time consuming both for the man and tne machine to search for the axiom to be used for the next reduction. Given a formula to be proved on a large theory, the validity generally depends on many axioms presented from different modules. (We will discuss this point in detail at the end of this section.) If the theory is cleanly and naturally modularized, however, one could well expect some desirable property in the proof, which we call *proof locality.* Namely, a consecutive portion of proof steps, if appropriately selected, tends to depend on axioms from only a few or preferably a single module. This property opens up the possibility of permitting the man-machine interaction to focus the attention on a particular module for a portion of the period during the proof. we collectively call such strategies theory-focusing. The successful use of theory-focusing can enhance the efficiency of the proof because it largely narrows the selection of axioms at each step and facilitates the effective applications of reduction and simplification rules on a specific module.

The many-sorted-ness of the IOTA logic provides a useful technique for theory-focusing as follows. Generally a formula, which is generated as a goal in the course of the proof, can contain terms of different sorts. For instance

$$
\begin{aligned}
(4.1)\ & x \neq 0 \ \&\ y \neq 0 \ \&\ \mathrm{DVS}(x,w) \ \&\ \mathrm{DVS}(y,w) \\
& \quad \Rightarrow \mathrm{DVS}(\mathrm{COEF}(y,\mathrm{DEG}(y)).x-\mathrm{COEF}(x,\mathrm{DEG}(x)).\mathrm{TM}(\mathrm{DEG}(x)-\mathrm{DEG}(y),y),w)
\end{aligned}
$$

```
(4.2) x≠0 & y≠0 & DVS(y,w) &
      DVS(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y),w)
      => DVS(x,w) & DVS(y,w)
```

contain terms x, y, w, COEF(y,DEG(y)).x, TM(DEG(x)-DEG(y),y), COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y) of sort INTPOLY, terms DEG(x), DEG(y), DEG(x)-DEG(y) of sort NN and terms COEF(y,DEG(y)), COEF(x,DEG(x)) of sort INT. (This happens to be one of the verification conditions of a program which computes the g.c.d. of two polynomials.) A straightforward way for theory-focusing is to replace all non-variable terms of a designated sort by variables of that sort , where the same terms are replaced by a single variable. In this way, the structure of the sort is concealed, facilitating the-focusing on the structures of the other sorts. Here, replacing the terms of sort NN and the terms of sort INT, we obtain

```
(4.1.1) x≠0 & y≠0 & DVS(x,w) & DVS(y,w)
        => DVS(r1.x-r2.TM(n,y),w)

(4.2.1) x≠0 & y≠0 & DVS(y,w) & DVS(r1.x-r2.TM(n,y),w)
        => DVS(x,w) & DVS(y,w)
```

which are free of the structures of INT and NN. Applying axioms of DVS and INTPOLY, we reduce (4.1.1) to **true** and (4.2.1) to

```
(4.2.1.1) y≠0 & r1.x=v*w =>  u.x=u*w
```

To prove (4.2.1.1), the concealed structure of INT must be recovered and r1 is changed back to COEF(y,DEG(y)). Then the proof proceeds this time using axioms of the type module of INT. (The axioms of NN are not involved.) This is an example of theory-focusing which goes upwardly in the theory hierarchy (i.e. focusing on modules which are higher in the theory hierarchy, hiding the lower modules). There are cases in which the focusing goes downwardly.

We mentioned that the validity of a formula to be proven depends generally on many axioms presented by different modules. Here we show briefly how this happens.

It is often the case that the proof of a formula requires some axioms of modules which are not referred to explicitly in the formula.

For example, the following formula contains explicitly only a function + and terms of type INTPOLY and no other module is referred to explicitly. (x, y and z are variables of type INTPOLY.)

$$x+(y+z)=z+(y+x)$$

In order to prove this formula, one would need the commutativity and associativity of + on the integers, which should be given as among the basic axioms on type INT, in addition to axiom 8 and 10 of INTPOLY.

(If we included the following two axioms as among the basic axioms on INTPOLY, the above formula could be proved using only the axioms of INTPOLY:

$$x+y=y+x$$

$$x+(y+z)=(x+y)+z$$

though these are deducible from axioms 8 and 10 together with the axioms on INT.)

5. Theory Extractions

The basic structure of the type of NN includes the theory of the total ordering as a substructure, which can be contained in the basic structures of many other types. We extract and isolate the lines preceeded by '++' in the presentation of the type module of NN to form the sype module of ORDER.

```
interface sype ORDER
  fn LESS : (@,@) -> BOOL as @≤@
     EQUAL: (@,@) -> BOOL as @=@
end interface

specification sype ORDER
  var x,y,z,u,v:@
  axiom 1: x≤y or y≤x
        2: (x≤y & y≤z) => x≤z
        3: (x≤y & y≤x) => x=y
        4: x=x
        5: (x=y & u=v) => (x≤u)=(y≤v)
        6: (x=y & u=v) => (x=u)=(y=v)
end specification
```

Thus a sype module looks quite like a type module. Sypes are another kind of sorts in IOTA logic whose basic structures are presented by sype modules in language IOTA. Namely there are primitive functions PR<s> and the basic axioms BA<s> on each sype s. The difference is that there does not exist the notion of generator inductions on sypes. Now we generally characterize the relation which holds between ORDER and NN. Let us remember that as a syntactic rule of language IOTA if a sype or type module q presents a sype or type s, the name of each function in PR<s> is in the form of T#f where T is the name of q. We will confuse all three of q, s and T.

Definition Given a sype module S and a sype or type module T, the relation $S \leq T$ holds iff

(1) For each function S#f:(s1,s2,...,sm) -> s0 in PR<S>, there exists a function T#f:(r1,r2,...,rn) -> r0 in PR<T> such that m=n, ri=T if si=S and ri=si if si≠S for i=0,1,...,m.

(2) For each formula P in BA<S>, P[S/T] is provable in TH<T>, where the formula P[S/T] is obtained from P replacing each occurrence of 'S' in P by 'T' and replacing appropriately each variable of sort S in P by a variable of sort T. (Different variables are replaced by different variables. For instance, let P be $x \leq y$ **or** $y \leq x$ in BA<ORDER> which is really ∀x.∀y.(ORDER#LESS(x,y) **or** ORDER#LESS(y,x)), then P[ORDER/NN] is ∀u.∀v.(NN#LESS(u,v) **or** NN#LESS(v,u)).)

Notice that the transformation from P to P[S/T] is determined up to alpha-conversions (or variable renamings). Since all formulas in BA<S> are closed, this does not cause any inconvenience in our arguments. Note that as the provability property in (2) of the definition is undecidable, the relation $\leq$ is undecidable, but the auther of S and T should know how to establish $S \leq T$.

Corollary In the definition above, if a formula P is provable in TH<S>, so is P[S/T] in TH<T>.

Notice that this corollary depends essentially on that there is no generator induction on the sype S.

On the other hand, the structure of the procedure module NNMAX essentially depends solely on the substructure of the total ordering on NN. So we should rather write the following type-parameterized procedure module:

```
interface procedure MAX(P:ORDER)
  fn MAX : (P,P) -> P
     MIN : (P,P) -> P
end interface

specification procedures MAX(P:ORDER)
  var x,y:P
  axiom 1: x≤y => MAX(x,y)=y
      :
      :
end specification
```

The only difference from NNMAX is that 'P' occurs in each place of 'NN'. P:ORDER is understood to be a "type parameter", which runs over all type T such that ORDER ≤ T. Substituting an "actual" type parameter 'T' for each occurrence of 'P' in the presentation of MAX(P:ORDER), we have a procedure module MAX(T). For instance MAX(NN) is isomorphic to NNMAX. (Notice that = and ≤ in MAX(P:ORDER) are P#EQUAL and P#LESS, respectively, while they are NN#EQUAL and NN#LESS in MAX(NN).)

Thus if we have already written both NN and MAX(P:ORDER), then the function MAX(NN)#MAX : (NN,NN) -> NN can be used automatically which is equivalent to NNMAX#MAX. In this way one can reduce the work of module-writing. More importantly, giving MAX(P:ORDER), the logical relation between TH<NN> and TH<NNMAX> is clarified because MAX(P:ORDER) presents only what is essential in the extention from TH<NN> to TH<NNMAX>.

As further examples, if we give RING, the sype of ring, then INTPOLY can be generalized into a more general type POLY(T:RING) which is the type of polynomials over arbitrary coeficient ring, or with ANY, the sype which has only the equality and the equality axioms, INTARRAY is generalized to ARRAY(T:ANY) which is the type of array of any object with equality. The new data type concepts of sypes generalizes so called type-parametrization structures in programming. The use of sypes is highly useful in structuring programs and theories and in simplifying verification procedures. More details on sypes as well as the formalization of the type-parametrization features within the first order logic can be found in [4]. Since the structures of theories can be simplified and clarified by introducing sypes, the use of them can be said to be helpful for theorem proving on them.

But the sype concept has a more direct application to theorem proving in the following way. There are many different theories but often quite a few of them have a common subtheory and the extraction of such common subtheory forms a sype. If a powerful reduction and simplification procedure is developed on such a sype, it can be applied to any theory that contains the subtheory. For example, all of theories of the integers, the rationals and the polynomials contain the common structure of ring. Thus any reduction and simplification strategies developed for the theory of ring can be applied to those theories.

For this purpose, the sype concept introduced in this section is rather restrictive. (These restrictions are desirable on sypes as a programming concept in order to enhance the most important goal of program structuring.) Thus we give the more general and flexible relation as follows. (Let FN<M> be the set of functions introduced by a module M and AX<M> be the set of the axioms introduced by M.)

Definition Given a sype S and a procedure, type or sype module T, we say $S \leq_H T$ for a mapping H:FN<S> -> FN<T> iff

(1) H is one to one mapping.

(2) There exists a sort t such that if f:(s1,s2,...,sn) -> s0 and H(f):(t1,t2,...,tm) -> t0, m=n and ti=si if si≠S and ti=t if si=S.

(3) For each formula P in AX<S>, Tr(P,H) is provable in TH<T> where Tr(P,H) is a formula obtained from P replacing each S#f by T#H(f) together with appropriate variable conversion from S to T.

The user can prepare a sype S which seems to be a subtheory of many theories and develop proof procedures on S. Whenever appropriate, he gives a mapping H for a module T and establishes the relation $S \leq_H T$. (The amount of work required in establishing $S \leq_H T$ should be for the most cases small or none. Often Tr(P,H) itself is found among AX<T> for many P in AX<S>.)

Corollary If $S \leq_H T$, then for any formula Q, if Q is provable in TH<S>, so is Tr(Q,H) in TH<T>. (Again this corollary essentially depends on that there is no generator induction on S.)

There can be more than one relations between the same pair of S and T. Once a relation $S \leq_H T$ is established and stored, whenever a theory concentration is made on T and the man-machine interaction finds the portion of the proof depends solely on the substructure S of T, the goal formula is mapped by H^{-1} and is tried to be reduced on S. This excludes the rest of the structure of T which is not involved for the moment and, by the reduction and simplification procedures developed on S, would speed up the proof.

There can be many candidates for sypes which need not be so elaborated as ring. For instance, it should be useful to have a simple sype consisting of two functions f and g such that they satisfy commutativity and associativity.

6. Subformula reductions

In practice, fairly large formulas are involved in proofs, especially for program verifications. They must usually be decomposed into several smaller subformulas which are easily processed. The usual technique has been to reduce such a formula to some normal form, e.g. in the form of an implication whose antecedent and consequent are a conjunction and a disjunction of atomic formulas(CD-normal form), respectively. However mechanical application of such normal form reduction can often destroy the semantic enevitability in the structure of the formula. The resulting formulas may be very hard to read and to apply the strategies presented in the previous sections.

For instance consider the following formula.

```
(6.1) ( ∀w.( DVS(x0,w)&DVS(y0,w) <=> DVS(x,w)&DVS(y,w) )
        & x≠0 & y≠0 )
      => ( DEG(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))
                                                           ≤ DEG(y)
         => ∀w.( DVS(x0,w)&DVS(y0,w)
                 <=> DVS(y,w) &
                     DVS(COEF(y,DEG(y)).x
                         -COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y), w)))
        & ( ~DEG(COEF(y,DEG(y)).x
                     -COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))≤DEG(y)
           => ∀ w.( DVS(x0,w)&DVS(y0,w)
                   <=> DVS(COEF(y,DEG(y)).x
                           -COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y), w)
                      & DVS(y,w) ))
```

(This happens to be one of the verification conditions for a program to compute the g.c.d. of two polynomials.)

The (CD-)normal forms of the formula will be:

```
(6.1.1) ( DVS(x0,w0) & DVS(y0,w0) &
          DEG(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))
```

```
                                                        ≤DEG(y)   )
        => DVS(y,w0) or DVS(x0,w0) or DVS(y0,w0) or x=0 or y=0

(6.1.2) ( DVS(x0,w0) & DVS(y0,w0) &
          DEG(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))
                                                        ≤DEG(y)   )
        => DVS(COEF(y,DEG(y)).x
                       -COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y),w0) or
           DVS(x0,w0) or DVS(y0,w0) or x=0 or y=0

(6.1.3) ( DVS(x0,w0) & DVS(y0,w0) &
          DEG(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))
                                                        ≤DEG(y)   )
        => DVS(y,w0) or DVS(x0,w0) or DVS(y0,w0) or x=0 or y=0

     :
     :
     :

(6.1.9) ( DVS(x,w0) &DVS(y,w0) &DVS(x0,w0) &DVS(y0,w0) &
          DEG(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))
                                                        ≤DEG(y)   )
        => DVS(y,w0) or x=0 or y=0

(6.1.10) ( DVS(x,w0) &DVS(y,w0) &DVS(x0,w0) &DVS(y0,w0) &
          DEG(COEF(y,DEG(y)).x-COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y))
                                                        ≤DEG(y)   )
        => DVS(COEF(y,DEG(y)).x
                       -COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y),w0) or
           x=0 or y=0

     :
     :
     :
```

Obviously these formulas (especially (6.1.10)) do not appear to teach us much. Thus we want to work out some methods which transform a large formula equivalently into several smaller ones without destroying the natural structure of the original formula.

Until now in most of theorem proving techniques, it has been usual to apply reductions exclusively to the outermost level of the formula to be proved, but careful applications of reductions to some appropriate subformulas seem to be useful for our purpose as we see in the following examples:

(6.1) is reduced to the following formulas by reducing any subformula in the form of (A => B) & (~A => C) to B when B <=> C.

```
   ( ∀w.(DVS(x0,w) & DVS(y0,w) <=> DVS(x,w) & DVS(y,w)) &
     x≠0 & y≠0   )
  => ∀w.(DVS(x0,w) & DVS(y0,w)
             <=>  DVS(y,w)  &
                  DVS(COEF(y,DEG(y)).x
                       -COEF(x,DEG(x)).TM(DEG(x)-DEG(y),y), w) )
```

Namely,to prove a formula in the form of P((A=>B) & (~A=>C)), it is sufficient to establish P(B) and B<=>C. This subformula reduction method is useful in program verification because subformulas in the form of (A=>B) & (~A=>C) often appear due to if-statement.

Often a formula to be proved contains as subformula several occurrences of a same formula, and such a subformula should not be deformed in order to preserve the natural structure of the superformula. The validity question of the original formula can be simplified by assigning *true* or *false* to such a subformula, which is more precisely described as: Given a formula P in which several occurrences of a formula F appear, in order to prove P, it is sufficient to establish F=>P{*true*/F} and ~ F => P{*false*/F} (if P is in the form of P1=>P2 then F & P1{*true*/F} => P2{*true*/f} and

~F & P1{**false**/F} => P2{**false**/F}), where P{t/F} stands for the formula obtained from replacing F by t. Then by using the simplification rules of the propositional calculus such as A=>**true** ---> **true**, the validity of the original formula can be largely simplified with the original structure of P well preserved.

```
(6.2) ( (i<k => A[i]≤A[k]) & (m≤i<j≤k => A[i]≤A[j]) &
        (n≤i<j≤HIGH(A) => A[i]≤A[j]) & k<n & A[k]≤A[k+1] &
        m≤i<j≤k+1 )
     =>  A[i]≤A[j]
```

[This formula happens to be a verification condition of bubble sort program.]

The formula above is decomposed into the following formulas by the above technique on i<j.

```
(6.2.1) ( i<j & (i<k => A[i]≤A[k]) &
          (m≤i & true & j≤k => A[i]≤A[j]) &
          (n≤i & true & j≤HIGH(A) => A[i]≤A[j]) & k<n &
          A[k]≤A[k+1] & m≤i & true & j≤k+1  )
        => A[i]≤A[j]

(6.2.2) ( ~i<j & (i<k => A[i]≤A[k]) &
          (m≤i & false & j≤k => A[i]≤A[j]) &
          (n≤i & false & j≤HIGH(A) => A[i]≤A[j]) & k<n &
          A[k]≤A[k+1] & m≤i & false & j≤k+1  )
        => A[i]≤A[j]
```

By applying the propositional reduction rules we get (6.2.1)' from (6.2.1) and **true** from (6.2.2).

```
(6.2.1)' ( i<j & (i<k => A[i]≤A[k]) &
           (m≤i & j≤k => A[i]≤A[j]) &
           (n≤i & j≤HIGH(A) => A[i]≤A[j]) &
           k<m & A[k]≤A[k+1] &  m≤i & j≤k+1 )
        => A[i]≤A[j]
```

where (6.2.1)' is proved by using the same technique on the formulas $m \leq i$, $i \leq k$ and $i < k$. In this way the natural structure of the original formula can be preserved throughout the proof.

There can be many different methods for subformula reductions and so it is desirable for the user to be able to develop such methods interactively whenever necessary. In the same way as module-wise development of reduction rules, the soundness must be established.

References

1. Bledsoe, W.W.: Non-resolution theorem proving. Artificial Intelligence 9, 1-35, 1977

2. Gordon, M., Milner, R., Morris, L., Newey, M., Wadsworth, C.: A meta language for interactive proofs in LCF. 5th ACM Conference on Principles of Programming Languages. 1978

3. Nakajima, R., Honda, M., Nakahara, H.: Describing and verifying programs with abstract data types. Formal description of Programming Concepts. (ed. Neuhold) North-Holland Publishing. Co. 1977

4. Nakajima, R., Nakahara H., Honda, M.: Hierarchical program verification --a many sorted logical approach--. RIMS-265, Research Institute for Mathematical Science, Kyoto University. 1978

5. Nakajima, R.: Sypes --partial types-- for program structuring and first order system IOTA logic. Reserch Report No. 22, Institute of Informatics, University of Oslo. 1977

6. Nelson, C., Oppen, D.C.: Simplification by cooperating decision procedures. AIM-311, Stanford Artificial Intelligence Laboratry. 1978