

Interactive Theorem Proving on Hierarchically and
Modularly Structured Set of Very Many Axioms
- Extended Abstract -

Michio Honda*
Kagawa University
Dept. of Information
Science
Takamatsu

Reiji Nakajima*
Kyoto University
Research Institute for
Mathematical Sciences
Kyoto

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.

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 is formal verification of programs whose program domains are determined by user-defined axioms. Due to the obvious time and memory constraints, fully automatic proofs are unrealistic. Thus, interactive non-resolution proofs are inevitable so that the user can keep well aware of what is being done in each stage of the ongoing proof, understanding the meaning of the formulas generated during the proof. With human assistance, it still can be highly difficult to conduct proofs on many axioms. Thus some organized methods are needed. Here we suggest that some meaningful structuring of theories or axiom bases can be useful and introduce some strategies for interactive proofs on structured theories. The motivation of this work derived from a software development called λ system[5,6] at Kyoto University. This is going to be an interactive system for developing and verifying programs in language λ which is designed to support hierarchical and modular programming. It is not possible to detail the language and system here, but in short, program development and verification in λ involves hierarchical and modular development of theories with many user-defined axioms and requires theorem proving on them. (We are as much concerned with how such theory development should be done interactively as how proofs should be done on it. 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.) The proof strategies have been implemented in λ prover, subsystem of λ system, which is invoked during program development and debugging as well as verification. (The first version of λ prover currently runs on DEC system-20[4].) The λ prover contains an automatic proof system 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. This paper is not self-sufficient or comprehensive due to space limitation. [6] is a suitable reference to the background while [4] details the prover with work-out examples.

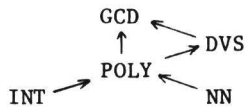
Related works: It seems that no previous work has attempted to exploit the structuring of theories to facilitate mechanical proofs. Clear [2] seems to be somewhat in a similar direction though it uses algebraic axioms while we use first order logic. (It is beyond the scope of this paper to compare the 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 [7] but with a different objective. [1] surveys numerous works on non-resolution theorem proving.

2. Hierarchical and modular theory development

The proof system is based on a deviation of the many-sorted first order logic called λ logic[6]

* Order is not significant.

which is tailored for the specification of hierarchical programs with abstraction mechanisms. Language 1 provides a syntax with which one can structuredly build up theories in 1 logic. [6] gives a formal and complete description of the logic and the language and here we only give a rough sketch. Each sort in 1 logic is characterized by a finite number of functions and axioms which are said to be the basic structure of the sort. There are two kind of sorts; types and sypes. The types are associated with the notion of generator induction rules while the sypes are not (Section 5). A syntactic unit of language 1 is called module which defines a chunk of theory in 1 logic by presenting a finite set of functions as well as axioms that characterize the functions. In the figure, NN, INT and POLY are type modules presenting the basic structures of the (abstract) types of integers, natural numbers and polynomials with integer coefficients, respectively. DVS is a procedure module which extends the theory of POLY by adding a function DIV etc. Upon these modules a procedure module GCD is written to extend the joint of the theories defined by them. GCD introduces g.c.d. function on POLY. (Logical circularity among modules is prohibited, i.e. module development must be hierarchical.) Using language 1, the module GCD is implemented; namely g.c.d. function is realized by an Algol-like routine which is written on the theory defined by DVS, POLY, NN and INT. In order to verify that the routine satisfies the axioms given by GCD, the following formula (i.e. one of the verification conditions) must be proven. $\forall w. (DIV(x0, w) \wedge DIV(y0, w) \equiv DIV(x, w) \wedge DIV(y, w)) \wedge x \neq 0 \wedge y \neq 0 \supset (DEG(COEF(y, DEG(y))).x - COEF(x, DEG(x)).TM(DEG(x) - DEG(y), y)) \leq DEG(y) \supset \forall w. (DIV(x0, w) \wedge DIV(y0, w) \equiv DIV(y, w) \wedge DIV(COEF(y, DEG(y)).x - COEF(x, DEG(x)).TM(DEG(x) - DEG(y), y), w)) [COEF: POLY \rightarrow INT, DEG: POLY \rightarrow NN, TM: (NN, POLY) \rightarrow POLY \text{ and } \therefore (INT, POLY) \rightarrow POLY \text{ are given by POLY. } DIV: (POLY, POLY) \rightarrow BOOL \text{ is by DVS. } TM(n, Q) \text{ is } Q \text{ multiplied by } x^n, r.Q \text{ is scalar product, } DIV(P, Q) \text{ means that } P \text{ is divisible by } Q, \text{ etc.}]$



3. Theory-Focusing

A major difficulty with a proof on a large theory derives from the wide selection of axioms to invoke at each step. As seen in the example of Section 2, the validity of a single formula to be proven generally depends on many axioms from different modules. 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

chosen, tends to depend on axioms from only a few or preferably a single module. This property permits 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. A successful use of theory-focusing can enhance the efficiency in proofs 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 (Section 4,5). For instance, hierarchical relations among sorts in 1 logic provides a useful technique. Generally a formula, which is generated as a goal in the course of the proof, can contain terms of different sorts. For example $x \neq 0 \wedge y \neq 0 \wedge DIV(x, w) \wedge DIV(y, w) \supset DIV(COEF(y, DEG(y)).x - COEF(x, DEG(x)).TM(DEG(x) - DEG(y), y), w)$ contains terms $COEF(x, DEG(x)).TM(DEG(x) - DEG(y), y)$, $COEF(y, DEG(y)).x$, $TM(DEG(x) - DEG(y), y)$, x , y , z of sort POLY, $DEG(x)$, $DEG(y)$, $DEG(x) - DEG(y)$ of sort NN and $COEF(y, DEG(y))$, $COEF(x, DEG(x))$ of sort INT. A straightforward way for theory-focusing is to replace all non-variable terms of a designated sort by variables of the sort, where the same terms are replaced by a same variable. Thus the structure of the sort is concealed, facilitating focusing on the structures of the other sorts. Here, replacing the terms of sorts NN and INT, we have: $x \neq 0 \wedge y \neq 0 \wedge DIV(x, w) \wedge DIV(y, w) \supset DIV(r1.x - r2.TM(n, y), w)$ which is free of the structures of INT and NN. Applying axioms of DVS and POLY, we reduce the formula above to true. 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 focusing goes downwardly. [4] illustrates use of other techniques for theory-focusing.

4. 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 1 logic. The logical reduction rules which ought to be sound and complete tend to be inefficient. Thus, it is desirable to generate reduction rules on a specific theory from the (non-logical) axioms of the theory. (Notice that the soundness of such non-logical rules must be established, for which 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 to generate efficient reduction rules. An immediate application of theory modularization is module-wise development of reduction rules. It will be convenient to develop powerful reduction rules on the subtheory

defined by a module because the axioms given by a single module are supposed to relate closely to each other.

Example Given the following axioms on the array of integers, the man-machine interaction generates the reduction rule R1.

```
axiom1: m=n ∧ m ≤ LENGTH(x)
        ⊃ FETCH(STORE(x,n,i),m)=i
2: m≠n ∧ m ≤ LENGTH(x) ∧ n ≤ LENGTH(x)
        ⊃ FETCH(STORE(x,n,i),m)=FETCH(x,m)
```

```
rule R1(P) /P is a syntactic variable/
goal      : P(FETCH(STORE(x,n,i),m))
subgoal1: m=n ∧ n ≤ LENGTH(x) ⊃ P(i)
2: m≠n ∧ m ≤ LENGTH(x) ∧ n ≤ LENGTH(x)
        ⊃ P(FETCH(x,m))
```

More examples as well as the rigid syntax for reduction rule generation are given in [4].

5. Theory extractions

Often several different theories contain a common substructure, and so do 1 modules. For example the theory by INT and POLY share the structure of ring. A sype is obtained by extracting and isolating such a substructure. Here we give a sype module RING to define the theory of ring. Like the other kinds of modules a sype module introduces finite functions and axioms. The relation between RING and POLY is given by a mapping Σ which maps the functions on RING one to one to functions on POLY. Each axiom on RING is transformed by a natural extension of Σ into some formula which is provable in theory defined by POLY. ([6] defines formally the relation and the way to establish it. In language 1 we considered only the relation between a sype and a type (or a sype) under a strong syntactical restriction but here for theorem proving purposes we utilize the relation between a sype and a module of any kind without the restriction[4]. For instance a sype with two functions which satisfy commutativity and associativity is useful toward many modules of any kind.) Then if we generate powerful simplification and reduction procedures on RING, they are automatically used on INT and POLY and many others. There are many candidates for sypes. The sype of ordering is useful for example. The user defines such a sype S and establishes interactively the relation between it and any appropriate module M. When the proof is focusing on M, the procedures on S are invoked. Note that such a strategy is useful because we consider user-defined theories. From Programming view point, the sypes are new data type concept to generalize the so-called type-parametrization mechanisms. It is useful for structuring programs and specifications as well as simplifying verification. The sypes are simply another sorts in 1 logic and differ from the types in that there is

no generator induction on them. (Note that this condition is essential in the above discussion of map Σ . Details on sypes as well as their formalization in first order logic is given in [6].)

6. Subformula reductions

In practice, fairly large formulas are involved in proofs, especially in program verification. They must usually be decomposed into smaller formulas to process. The usual technique is to reduce such a formula to a certain normal form, or to apply reduction rules exclusively to the outermost level. However blind applications of such normal form reductions often destroy the semantic inevitability in the structure of the formula, resulting formulas very hard to read and to apply reductions further. Here careful applications of reductions to some appropriate subformulas can be useful to enhance the applicability of the strategies given in the previous sections. For example

```
rule R2(P,A,B,C)
goal: P((A>B) ∧ (~A>C))
subgoal1: B ≡ C
2: P(B)
```

This rule is useful in program verification because subformulas in the form of $(A>B) \wedge (\sim A>C)$ often appear due to if-statements. There can be many useful methods for subformula reduction ([4] illustrates use of some more). Similarly to user-given reduction rules, it is desirable to be able to develop such methods interactively to guarantee the soundness.

References

- [1] Bledsoe, W.W.: Non-resolution theorem proving. Artificial Intelligence 9 '77
- [2] Burstall, R., Goguen, J.: Putting theories together to make specifications. IJCAI '77
- [3] Gordon, M., Milner, R., Morris, L., Newey, M., Wadsworth, C.: A metalanguage for interactive proofs in LCF. Proc. Principles of Prog. Lang. '78
- [4] Honda, M., Nakajima, R.: An interactive theorem proving system for large axiom bases. Res. Inst. for Math. Sci., Kyoto Univ. '79
- [5] Nakajima, R., Honda, M., Nakahara, H.: Describing and verifying programs with abstract data types. Formal description of Programming Concepts. (ed. Neuhold) North-Holland '77
- [6] Nakajima, R., Nakahara, H., Honda, M.: Hierarchical program verification - a many sorted logical approach-. Res. Inst. for Math. Sci., Kyoto Univ. '78
- [7] Nelson, C., Oppen, D.C.: Simplification by cooperating decision procedures. AIM-311, Stanford A.I. Labo '78