ICOT Technical Report: TR-588

TR-588

A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis

by
R. Hasegawa, H. Fujita & M. Fujita

August, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis

Ryuzo Hasegawa*, Hiroshi Fujita† and Masayuki Fujita*

* Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
† Mitsubishi Electric Corporation
8-1-1 Tsukaguchi-honmachi, Amagasaki, Hyogo 661, Japan

August 13, 1990

Abstract

We have been building a parallel automated reasoning system and also developing a program synthesizer, which is a promising application of the system for use on the Parallel Inference Machine(PIM).

Firstly, we will present a parallel theorem prover for first-order logic implemented in KL1, and the KL1 implementation techniques which are also useful for other related areas, such as truth maintenance systems and intelligent database systems. The MGTP prover, which has already being developed, adopts a model generation method, as used in SATCHMO, that was first implemented in Prolog by Manthey and Bry. SATCHMO tries to find ground models for the given set of clauses that satisfies a condition called range-restrictedness. The condition imposed on the clause set allows us to use only matching rather than unification during the proving process. This property is also favorable in implementing a prover in KL1 since matching is easily realized with head unification and the variables in the given clauses can be represented as KL1 variables. For ground model cases, experimental results show that the MGTP prover is more than three times faster than the SATCHMO prover on SUN3/260 even in the pseudo parallel environment of the PSI-II machine. To deal with nonground models, MGTP is extended by incorporating unification with occurs check, weighting heuristics and other deletion strategies while still keeping a model generation paradigm.

We then present a formal system for program synthesis using the MGTP prover. The sorting program is taken as an example. The program specification is expressed as a formula. Subprograms can be used in the synthesized program if the corresponding lemmas are provided to the prover. The program extraction mechanism is based on constructive logic. The proof trace obtained from the prover is translated to an LF(Edinburgh Logical Framework) proof term and a program can be extracted from this proof term.

1 Introduction

The research on theorem proving in the FGCS project aims to develop a parallel automated reasoning system applicable to various fields such as natural language processing, intelligent database designing, and other kinds of knowledge information processing on the Parallel Inference Machine (PIM).

We have implemented a theorem prover for first-order logic in KL1, a committed-choice logic language. We have also developed an experimental system for automated program synthesis as an application suitable for the theorem prover.

We will now briefly review the research on a theorem prover for first-order logic. The following are typical calculi for first-order logic that we have taken into consideration: resolution, model elimination[Lov78], the tableaux method, and the connection method[Bib86].

Recent developments in logic programming languages and machines have shed light upon the problem of how to implement these classical but powerful methods efficiently. For instance, Stickel developed a model-elimination based theorem prover called PTTP[Sti88] which is implemented in Prolog. PTTP is able to deal with any first-order formula in Horn clause form (augmented with contrapositives) without loss of completeness or soundness by employing unification with occurs check, the model elimination reduction rule, and iterative deepening depth-first search. Schumann et al. built a connection-method

based theorem-proving system, SETHEO[Sch89], in which a method identical to model elimination is used as a main proof mechanism. The system is implemented using Prolog technology and an approach similar to Stickel's is taken. Manthey and Bry presented a Tableaux-like theorem prover, SATCHMO[MB88], which is implemented in Prolog by means of a very short and simple program. SATCHMO is basically a forward-reasoning theorem prover, which also allows backward reasoning by employing Prolog over the Horn clause subsets.

The above systems all utilize the fact that Horn clause programs can be very efficiently solved. In these systems, the theorem being proven is represented with Prolog clauses and most deductions are performed as normal Prolog execution.

A method similar to the above seems to be applicable to committed-choice languages, such as KL1. If this were, we could make full use of KL1's merits, that is, the ease of writing concurrent programs and of exploiting parallelisms. However, it turns out that it is not so easy to represent a clause set for a theorem directly with KL1 clauses because a KL1 clause is not just a Horn clause; it has extra-logical constructs such as a guard and a commit operator. We should, therefore, treat the clause set as data rather than as a KL1 program.

One of the main problems here is how to represent variables appearing in a clause set for the theorem being proven. Two approaches can be considered for this problem: one is to represent them as KL1 ground terms, the other is to represent them as KL1 variables. The first approach suffers from the inefficiency of metaprogramming, that is, large interpretation overhead of objects on different layers. The second approach allows us to avoid that problem but offers problems specific to KL1, as mentioned above.

To remedy these deficiencies, while still making effective use of KL1's features, we adopted a *model generation method*, on which SATCHMO is based, as a basic proof procedure, and specialized the method to two cases: ground model case and nonground model case.

For the ground model case, a model generation prover makes it possible to use only matching rather than full unification. This suggests it is sufficient to use KL1's head unification. Therefore, we can take the second approach and achieve very efficent implementation, by using a programming trick. The MGTP theorem prover, which was built based on the model generation method, can prove a significantly large class of theorems.

For nonground model case, where a full unification with occurs check is required, we are forced to follow the first approach. Here, we used the vector facility that KL1 offers to remedy the above overhead problem. In addition, we employed traditional techniques such as clause ordering, weighting heuristics and other strategies, which are usually employed in standard resolution provers, while still staying within a model generation paradigm. For ground and nonground cases, the model generation method works very well in proving theorems and is easy to implement in KL1.

The second topic to be presented is an application for the theorem prover. Although a theorem prover for first-order logic has the potential to cover most areas of AI, it has been left behind by logic programming by ten years or more. One reason for this is the inefficiency of the proof procedure and the other is lack of useful application. However, through the research on constructive programming, we became convinced that it is very useful to apply the first-order theorem prover to programming. We believe automated program synthesis can be realized by this approach.

Curry-Howard isomorphism[Mar82], which allows us to relate programs with proofs, gives the mathematical basis for developing a formal system to synthesize a program from a proof obtained via a first-order theorem prover like MGTP.

However, there are two major problems to be solved to realize the above system: how to extract a program from a proof in first-order logic, and how to incorporate induction and equality into the system.

The first problem means that programs cannot be extracted from proofs obtained by using excluded middle as used in classical logic. This problem can be solved if the program specification is given in clausal form because a proof can be obtained from the clausal set without using excluded middle .

The second problem is that all induction schemes are expressed as second-order propositions and thus need second-order unification, which is impractical to use. However, it is possible to transform a second-order proposition to a first-order proposition if the program domain is fixed. In relation to equality, a proof of equality does not affect program extraction, so we may use any efficient algorithm for equality.

From these observations, we have developed an experimental system for automated program synthesis using the MGTP prover. The sorting algorithm is taken as an example. At present, although the system needs to be given lemmas and axioms necessary for extracting a sort program, all these can be obtained, in principle, by the theorem prover.

2 Model Generation

We adopt a method called model generation as a basic proof procedure for our prover. We assume that a theorem to be proven is negated and transformed to a set of clauses, then we try to refute the clause set as in the resolution method. We say that a clause is a *negative(positive) clause* if it has at least one negative(positive) literal but no positive(negative) literal. It is a *mixed clause* if it has both of positive and negative literals. The task of model generation is to try to construct a model for a given set of clauses and to show that no model exists for the clause set (the clause set is unsatisfiable).

The proof procedure of model generation is as follows.

- We start with a null set as a model candidate and repeatedly apply two rules shown in Fig. 1 to the clause set so as to find a model.
- If we find a model candidate under which neither rule is applicable to any clause, then the candidate is in fact a model and we conclude that the clause set is satisfiable.
- If every possible model candidate is rejected after all, then we conclude that the clause set is unsatisfiable.

There may be more than one clause to which the *model extension rule* is applicable thereby resulting in multiple extensions of a model candidate. And if every possible extension turns out not to be a model for the clause set, we conclude that the clause set has no model. Also there may be more than one literal in the consequent in the clause to which the model extension rule can be applied. If it is the case, we say the model candidate is 'expanded'. In this case we have to show that every extension of a model candidate cannot be a model in order to show the clause set has no model. The above two cases are depicted in Fig. 2. OR branches correspond to the first case and AND branches to the second case.

Note that if a clause to which the model extension rule is applicable has a literal that is unifiable with an element of the model candidate under consideration, then extending the model candidate is redundant and should be avoided. Also note that the model extension rule is always applicable to a positive clause and that extending a model candidate with the positive clause more than once is redundant and should be avoided.

A clause can be represented in an implicational form as below:

$$Antecedent \rightarrow Consequent$$

where *Antecedent* is a conjunction of positive literals and *Consequent* is a disjunction of positive literals. When we say 'clause' in the sequel, we shall mean its implicational form as well as its standard form for convenience.

The model generation method, as its name suggests, is closely related to *model elimination* method[Lov78]. There is a clear difference, however, in that model generation proceeds bottom up (as in forward reasoning) starting at positive clauses (or facts) whereas model elimination proceeds top down (as in backward reasoning) starting at a negative clause (or a query).

Deduction based on the model generation method can also be viewed as a special case of a deduction using hyper (positive) resolution. Our calculus, however, is much closer to the tableaux calculus in the sense that it explores a tree (or a tableau) in the course of finding a proof. Indeed, a closed branch in a proof tree obtained by the tableaux method corresponds exactly to an inconsistent model candidate that is to be rejected in the model generation method.

For example, consider the following set of clauses:

Problem S1[MB88]:

$$\begin{aligned}
C1: &\quad p(X), s(X) \rightarrow false.\\
C2: &\quad q(X), s(Y) \rightarrow false.\\
C3: &\quad q(X) \rightarrow s(f(X)).\\
C4: &\quad r(X) \rightarrow s(X).\\
C5: &\quad p(X) \rightarrow q(X); r(X).\\
C6: &\quad true \rightarrow p(a); q(b).
\end{aligned}$$

We start with an empty model, $M = \phi$. M is first expanded by C6 into two cases: $M_1 = \{p(a)\}$ and $M_2 = \{q(b)\}$. Then by C5, M_1 is expanded into two cases: $M_3 = \{p(a), q(a)\}$ and $M_4 = \{p(a), r(a)\}$. Further by C3, M_3 is extended to $M_5 = \{p(a), q(a), s(f(a))\}$ but M_5 violates C2, so it is marked as

Model extension rule:

If there is a positive clause or a mixed clause such that every negative literal in the clause is unifiable with an element of a model candidate, then extend the model candidate with each of the positive literals in the clause.

Model rejection rule:

If there is a negative clause such that all of its literals are unifiable with an element of a model candidate, then reject the model candidate.

Figure 1: Model generation rules

closed. On the other hand, M_4 is extended by C4 to $M_6 = \{p(a), r(a), s(a)\}$ which is also marked as closed by C1. In a similar way, M_2 is extended by C3 to $M_7 = \{q(b), s(f(b))\}$ which is marked as closed by C2. Now that there is no way to construct any model candidate for the clause set, we can conclude that the clause set is unsatisfiable.

3 KL1 Based Model Generation Theorem Prover

3.1 Variables and Unification

When implementing a parallel theorem prover in KL1, we have to first solve a problem how to represent variables appearing in the given theorems before going into the main subject how to exploit parallelisms. This problem arises whenever we consider implementing metaprograms such as a Prolog/GHC meta-interpreter in KL1.

To solve the problem, there are two approaches:

(1) Representing object level variables with KL1 ground terms

(2) Representing object level variables with KL1 variables

The first approach might be the right path in metaprogramming where object and meta levels are separated strictly, thereby giving it a clear semantics. However, it forces us to write routines for unification, substitution, renaming, and all the other intricate operations on variables and environments. These routines would become considerably larger and more complex than the main program, and introduce orders of magnitude of overhead. This deficiency would be remedied by using a partial evaluation technique. However, we have not yet developed a powerful partial evaluator sufficient to remove the above overhead.

In the second approach, most operations on variables and environments can be performed on the side of the underlying system instead of running routines on top of it. This enables a metaprogrammer to avoid writing tedious routines and to gain high efficiency. Furthermore, in Prolog, one can also use the **var** predicate to write routines such as occurrence check in order to make built-in unification sound, if necessary. This approach may not always be chosen since it makes the distinction between object level and meta level very ambiguous. However, from the viewpoint of program complexity and efficiency, the actual profit from this approach is very large.

In KL1, however, the second approach is not always possible as in the Prolog case. This is because the semantics of KL1 never allows us to use a predicate like Prolog **var**. In addition, KL1 built-in unification is not the same as Prolog's counterpart, in that unification in the guard part of a KL1 clause is limited to one way and a unification failure in the body part is considered as a program error or exception rather than a mere failure that can be backtracked. Nevertheless, we can take the second approach to implement a theorem prover where ground models are dealt with, utilizing features of KL1 as much as possible. Details of the implementation are described in the following sections.

3.2 Ground Model

The good news is that SATCHMO does not need unification where the *range-restricted*[MB88] property is enjoyed by clauses. A clause is said to be range-restricted[1] if every variable in the clause has at least one

[1] To ensure range-restrictedness a *dom* predicate may be introduced and extra clauses for it are added to the original set of clauses. This transformation does not change the satisfiablity of the original set of clauses.

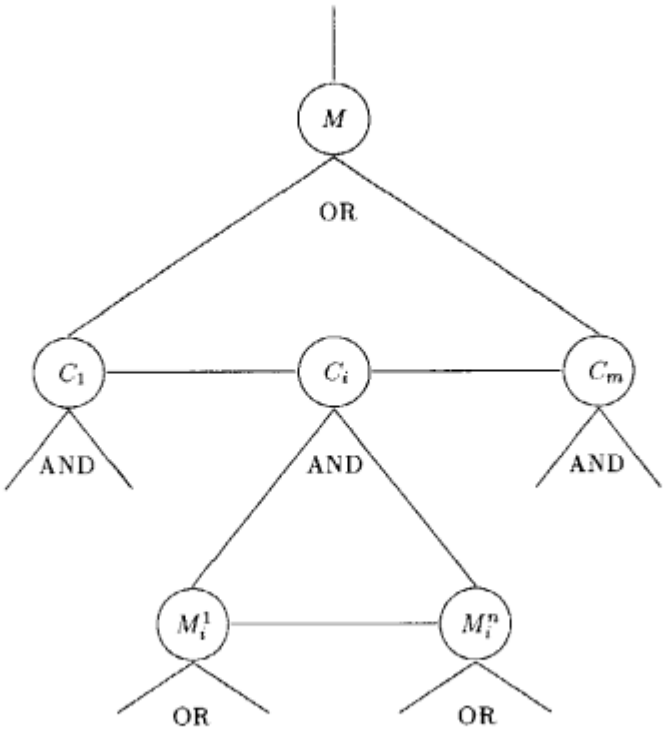

Figure 2: AND-OR tree in model generation

occurrence in its antecedent. For example, in the problem S1, all the clauses, C1 C6, are range-restricted since no variable appears in clause C6; the variable `X` in clauses C1, C3, C4 and C5 has an occurrence in their antecedents; and variables `X` and `Y` in C2 have their occurrences in its antecedent.

When range-restrictedness is satisfied, it is sufficient to consider one-way unification (matching) instead of full unification with occurs check, since a model candidate constructed by model generation rules should contain only ground atoms. Moreover, KL1 head unification is nothing but matching, so we already have a fast built-in operation for implementing model generation provers for range-restricted clause sets.

There is, however, a little problem concerning variables shared among literals in a clause, which requires a programming trick. To see this, consider the previous example, S1. The original clause set is transformed into a set of KL1 clauses as shown in Fig. 3. In `c(N,P,GS,R)`, `N` indicates clause number, `P` is a literal to be matched against an element in a model candidate, `GS` is a list of atoms appearing to the left of `P`, and `R` is the result returned when the match succeeds. Notice that clause C1 is represented by two KL1 clauses. In the first KL1 clause, the first literal of C1, `p(X)`, is matched against an element in the model candidate. If the match succeeds, the interpreter retains an instance of `p(X)` and proceeds to match the second literal, `s(X)`. At this point, the interpreter calls the second KL1 clause for C1 with an element of the model candidate as the first argument and the instance of `p(X)` as the second argument. Notice that whenever this call succeeds, the variable `X` in `s(X)` will get instantiated to the same value as `X` in `p(X)`.

3.3 The Interpreter

Fig. 4 shows a MGTP interpreter written in KL1.

Given a model candidate `M` and the number of clauses `N`, the interpreter tries to prove whether the given set of clauses is satisfiable or not. If it terminates with `A=sat(A=unsat)` then the answer is "satisfiable"("unsatisfiable"), otherwise it fails to prove.

`satisfy_clauses` explores every possible expansion of the model, `M`, checking satisfiability of every clause by spawning a **`satisfy_ante`** for each clause.

The results from the clauses are combined by **`and_sat`**. If all of the clauses return **`sat`** as the answer,

```
:- module mgtp_problem.
:- public model/1,nc/1,c/4.

model(M):-true|M=[].
nc(NC):-true|NC=6.

c(1,p(X),[],     R):-true|R=cont.
c(1,s(X),[p(X)],R):-true|R=false.          % C1:  p(X),s(X)->false.
c(2,q(X),[],     R):-true|R=cont.
c(2,s(Y),[q(X)],R):-true|R=false.          % C2:  q(X),s(Y)->false.
c(3,q(X),[],     R):-true|R=[s(f(X))].     % C3:  q(X)->s(f(X)).
c(4,r(X),[],     R):-true|R=[s(X)].        % C4:  r(X)->s(X).
c(5,p(X),[],     R):-true|R=[q(X),r(X)].   % C5:  p(X)->q(X);r(X).
c(6,true,[],     R):-true|R=[p(a),q(b)].   % C6:  true->p(a);q(b).
otherwise.
c(_,_,_,R):-true|R=fail.
```

Figure 3: S1 problem transformed to KL1 clauses

the model candidate, `M`, is indeed a model. Otherwise, some clause would return `unsat` thereby indicating that the clause set is unsatisfiable under the model candidate, `M`.

For each clause, `satisfy_ante` exhaustively makes combination of atoms out of `M` for generating different instantiation of the clause.

If a clause is found such that its antecedent is satisfied by `M` at `satisfy_ante1`, and that its consequent is not satisfied by `M` at `satisfy_cnsq`, the model candidate, `M`, is expanded to several branches according to the number of disjuncts in the consequent of the clause.

The results from the branches rooted at `M` are combined by `and_unsat`. If all of the branches return `unsat` as the answer, the clause set is unsatisfiable under the model candidate, `M`. Otherwise, some branch would return `sat` thereby indicating that the clause set is satisfiable under some of the expansions of the model candidate.

There are two kinds of parallelism, AND-parallelism and OR-parallelism. AND-parallelism is exploited in the current interpreter at `extend_model` where more than one `false` processes may be spawned. OR-parallelism could be introduced at `satisfy_clauses` where more than one clause is violated thereby becoming candidates for extending the current model.

3.4 Performance Comparison for Ground Model

Table 1 shows performance comparison among the systems, PTTP, SATCHMO and MGTP.

SATCHMO is faster than PTTP for all the problems S1, S2 and S3. MGTP is three to four times faster than SATCHMO for problems S1 and S3. For problem S2, however, SATCHMO is faster than MGTP since the former solves parts of the given problem directly as a Prolog program thereby avoiding interpretation overhead.

Fuchi developed a method to translate a problem together with its proof procedure into a KL1 program[Fuc90]. Fuchi's program runs about three times faster than our system. This difference of speed is smaller than the amount to be normally expected when interpretation overhead is taken into account. By applying partial evaluation technique it would be possible for our interpretive method to obtain performance comparable to Fuchi's compilation method.

4 Extension of MGTP

4.1 Nonground Model

With MGTP, we can prove a large class of theorems very efficiently. There are, however, more difficult theorems hard to prove with this type of prover as mentioned in [MB88]. For example, to prove that the following set of clauses is unsatisfiable is a very difficult problem for the MGTP prover.

```
:-module mgtp. :-public do/1.

do(A):-true|
    mgtp_problem:model(M),
    mgtp_problem:nc(N),
    satisfy_clauses(0,N,M,A,_).

satisfy_clauses(_,_,_,_,unsat):-true|true. alternatively.
satisfy_clauses(J,N,M,A,B):-J<N,J1:=J+1|
    satisfy_ante(J,[],[true|M],M,A1,B),
    and_sat(A1,A2,A,B),
    satisfy_clauses(J1,N,M,A2,B).
satisfy_clauses(N,N,_,A,_):-true|A=sat.

and_sat(sat,sat,A,_):-true|A=sat.
and_sat(unsat,_,A,B):-true|A=unsat,B=unsat.
and_sat(_,unsat,A,B):-true|A=unsat,B=unsat.

satisfy_ante(_,_,_,_,_,unsat):-true|true. alternatively.
satisfy_ante(J,S,[P|M2],M,A,B):-true|
    mgtp_problem:c(J,P,S,R),
    satisfy_ante1(J,R,P,S,M2,M,A,B).
satisfy_ante(_,_,[],_,A,_):-true|A=sat.

satisfy_ante1(J,fail,P,S,M2,M,A,B):-true|
    satisfy_ante(J,S,M2,M,A,B).
satisfy_ante1(J,cont,P,S,M2,M,A,B):-true|
    satisfy_ante(J,[P|S],M,M,A1,B),
    and_sat(A1,A2,A,B),
    satisfy_ante(J,S,M2,M,A2,B).
satisfy_ante1(J,false,P,S,M2,M,A,B):-true|B=A,A=unsat.
satisfy_ante1(J,F,P,S,M2,M,A,B):-list(F)|
    satisfy_cnsq(F,F,M,A1,B),
    and_sat(A1,A2,A,B),
    satisfy_ante(J,S,M2,M,A2,B).

satisfy_cnsq(_,_,_,_,unsat):-true|true. alternatively.
satisfy_cnsq([D1|Ds],F,M,A,B):-true|satisfy_cnsq1(D1,Ds,F,M,M,A,B).
satisfy_cnsq([],F,M,A,_):-true|
    mgtp_problem:n(N),
    extend_model(F,M,N,A,_).

satisfy_cnsq1(D,Ds,F,[D|M2],M,A,_):-true|A=sat.
satisfy_cnsq1(D,Ds,F,[],M,A,B):-true|satisfy_cnsq(Ds,F,M,A,B). otherwise.
satisfy_cnsq1(D,Ds,F,[_|M2],M,A,B):-true|satisfy_cnsq1(D,Ds,F,M2,M,A,B).

extend_model(_,_,_,_,sat):-true|true. alternatively.
extend_model([D|Ds],M,N,A,B):-true|
    satisfy_clauses(0,N,[D|M],A1,_),
    and_unsat(A1,A2,A,B),
    extend_model(Ds,M,N,A2,B).
extend_model([],_,_,A,_):-true|A=unsat.

and_unsat(unsat,unsat,A,_):-true|A=unsat.
and_unsat(sat,_,A,B):-true|A=sat,B=sat.
and_unsat(_,sat,A,B):-true|A=sat,B=sat.
```

Figure 4: A MGTP interpreter in KL1

Table 1: Performance comparison

Problem	S1	S2*	S3**
PTTP†	86msec	24sec	?(>30min)
SATCHMO†	16msec	68msec	6.3sec
MGTP‡	5.4msec	107msec	1.5sec
(Number of reductions)	(390)	(8,495)	(118,237)

† (Sicstus Prolog V0.6 on SUN3/260)
‡ (pseudo-Multi-PSI single-PE on PSI-II)
* S2: Schubert's Steamroller problem
** S3: Pelletier and Rudnicki's problem

Problem Imp[Ove90]:

$$p(i(i(i(X,Y),Z),i(i(Z,X),i(U,X))))$$
$$\neg p(i(X,Y)) \mid \neg p(X) \mid p(Y)$$
$$\neg p(i(i(i(a,b),a),a))$$

Since the first clause is not range restricted we need to add a *dom* predicate as follows.

$$\begin{aligned} dom(X), dom(Y), dom(Z), dom(U) &\rightarrow p(i(i(i(X,Y),Z),i(i(Z,X),i(U,X)))) \\ p(i(X,Y)), p(X) &\rightarrow p(Y) \\ p(i(i(i(a,b),a),a)) &\rightarrow false \\ true &\rightarrow dom(a) \\ true &\rightarrow dom(b) \\ dom(X), dom(Y) &\rightarrow dom(i(X,Y)) \end{aligned}$$

We have not yet obtained a proof for this problem with MGTP. The difficulty is due to the combinatorial explosion caused by blindlessly, yet systematically generated *dom* atoms. To avoid this it is necessary to incorporate a mechanism to selectively generate such dom atoms that are relevant to prove the theorem. An idea similar to *magic set*[BMSU86] might be applicable to this problem. Another way of solving this problem is to employ a learning mechanism. By training with a number of theorems of a similar sort, a prover might learn which dom atoms are more relevant to the class of theorems.

More drastic approach to solving the problem is doing without dom predicate. In this case, we have to consider models whose elements may be nonground. We also need full unification with occurs check when applying model generation rules to clauses.

4.2 Variables and Unification Revisited

If the nonground model approach is taken, MGTP's approach of representing object variables in KL1 variables can no longer be effective, instead, the other approach should be taken to represent variables with KL1 ground terms. Also, as mentioned in section 3.1, routines for unification, substitution, and householding of the environment have to be written on top of KL1.

When representing variables with KL1 ground terms, it is possible to consider two alternatives:

- Representing a variable with a KL1 atom, and an environment with a KL1 list, or
- Representing a variable with a KL1 integer, and an environment with a KL1 vector.

Routines based on the first alternative can be specified at a high level while they are very inefficient to execute. On the other hand, routines written according to the second alternative turn out to be fairly efficient, though their specification looks very low level. In fact, with KL1 vectors, you can program routines as if you are operating memory cells or pointers directly, thereby gaining performance comparable to those obtained on firmware level.

Table 2 shows performance comparison between unification routines. It should be worthwhile to note that the integer&vector version takes only three times as much CPU-time as KL1 built-in unification.

Table 2: Unification in KL1

Problem	builtin	atom&list	integer&vector
1	—	9.36	0.51
2	0.15	4.89	0.46
3	0.46	33.1	1.29

(msec)

1: $i(A, i(A, A)) = i(i(i(B, C), D), i(i(D, B), i(E, B)))$
2: $i(X, i(Y, Z)) = i(i(i(B, C), D), i(i(D, B), i(E, B)))$
3: $p(h(X_1, X_1), h(X_2, X_2), Y_2, Y_3, X_3) = p(X_2, X_3, h(Y_1, Y_1), h(Y_2, Y_2), Y_3)$

4.3 Avoiding Redundancy

To obtain a proof faster, it is very important (even essential) to avoid, as much as possible, redundant computation that are repeated for identical arguments and results. The idea is sketched as follows. For a kernel clause having two negative literals we need a pair of positive electrons to perform hyper-resolution. The pair will be obtained from the current set of unit clauses (or resolvents), M. After performing one step hyper-resolution for each pair taken out of M as electrons, we may obtain another set of unit clauses, ΔM. Then, in the next step, we will have to choose electron pairs out of $M + \Delta M$. The number of such pairs amounts to

$$(M + \Delta M)^2 = M \times M + M \times \Delta M + \Delta M \times M + \Delta M \times \Delta M.$$

Notice, however, that $M \times M$ number of pairs are those which have been selected in the previous step. Hyper-resolution steps for such pairs are just superfluous but those steps which are performed on pairs containing at least one electron from ΔM are only meaningful. This argument generalizes to kernel clauses that have more than two negative literals.

There is a way to avoid redundancy in which a unit clause is retained in a process and a process network is constructed dynamically as shown in Fig. 5.

The sketchy algorithm for the method is as follows:

- A model element generated by applying the extension rule is entered into a queue.
- A process, C_i, is created for each literal, L_i, in the antecedent of each clause.
- Each process, C_i, picking up a model element out of the queue, tries to match with the literal, L_i.
- If the match succeeds, C_i creates C_{i+1} for the next literal, L_{i+1}, and sends to it an instance of the literal, L_i, as the result of the matching. If the match fails, no process is created for L_i.

This method is easy to implement. One should, however, i) copy a process network, or ii) attach a color to an instance flowing in a stream in order to share the network, when the consequent of the clause consists of more than one literal.

Another way is to retain in a stack, instead in a process, a literal instance as a result of matching the literal against an element of a model. Fig. 6 illustrates an implementation of the method where S_0 is a stack for storing model elements generated; S_i is a stack for storing results from matching literal L_i against a model element; e is the most recent model element pushed onto S_0; and δ_i is a set of literal instances generated at the stage triggered by e.

The task performed at literal L_i is to make combinations of literal instances, $\delta_{i-1} \times S_0$, and $S_i \times e$ and then to store into S_i the resulting literal instances, δ_i.

If the consequent consists of more than one disjunct, stack S_0 is expanded to corresponding branches. Thus S_i may form a tree.

Although this method is rather conventional compared to the previous method, the problems described above can be solved effectively.

4.4 Heuristics

Also important is weighting heuristics. We used two sorts of measure for weighting: total number of symbols contained in a literal and an index that indicates, in a sense, the balancedness of a term tree.

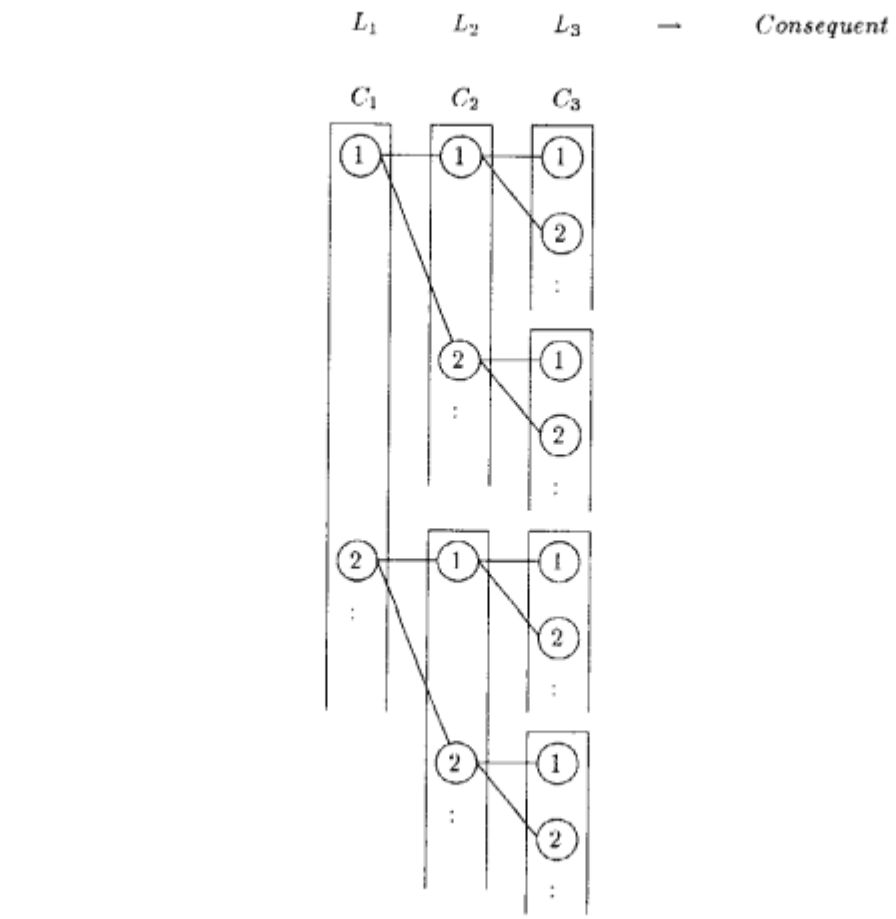

Figure 5: Process network

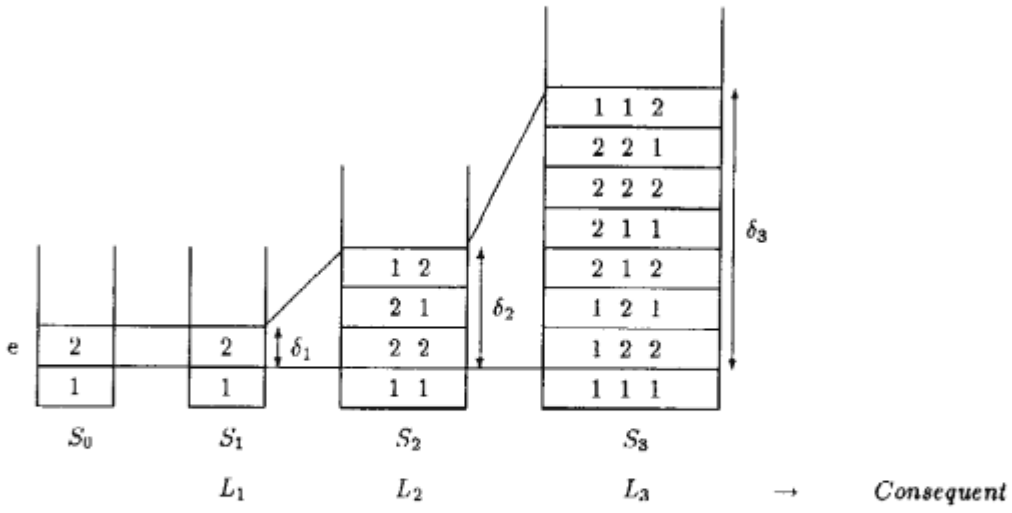

Figure 6: Literal instance stacks

Table 3: Performance comparison

PTTP†	?(>2hour)
SATCHMO†	?(>24hour)
Otter‡	32sec
ESP*	7.4sec
Extended MGTP**	17sec

† (Sicstus Prolog V0.6 on SUN3/260)
‡ C on SUN3/260 (including preprocessing)
* ESP/SIMPOS Version 6.0D5 on PSI-II
** pseudo-Multi-PSI/V2 on PSI-II (single-PE)

The former takes effect in improving performance in many cases, whereas the latter has an effect heavily dependent on the given problem.

Furthermore, in solving the Imp problem in section 4.1, we found that the mixed use of these two measures is better than single use of them. The mixed measure, M, is calculated as follows:

$$M = r \times (C/\overline{C}) + (1 - r) \times (B/\overline{B})$$

where C is the number of total symbols in a literal; $\overline{C}$ is the expected maximum value for C; B is the difference of numbers of symbols between left and right branches of a term, $i(L, R)$. $\overline{B}$ is the expected maximum value for B; and r is a factor for the mixture ranging between 0 and 1. Setting $r = 0.8$, we obtained the shortest proof for the theorem. With the values $r = 0.7$ or $r = 0.9$ the proof becomes much longer.

4.5 Performance Comparison for Nonground Model

The performance comparison for solving the Imp problem is shown in Table 3.

With PTTP, SATCHMO and MGTP for ground model, we have not obtained a proof after hours of running. We first solved the problem with an ESP program which is a specialized version only to solve the Imp problem, This program employs:

- Unification with occurs check
- Redundancy avoiding based on a method similar to the second approach described in section 4.3
- Weighting hueristics

while they are not for general purpose.

A proof of 22 steps is obtained in 7.4 seconds in the ESP program, whereas Otter gives 20 steps of proof in 32 seconds including preprocessing time. A version of MGTP extended for nonground models is written using similar techniques to those used in the ESP version. The extended MGTP takes about three times longer than the ESP version to solve the Imp problem. This difference in speed is mainly due to the difference of overhead in unification routines, since more than 80 percent of the running time is consumed by unification.

5 Program Synthesis by Parallel Prover

In the following sections, we present a new approach to realize an automated program synthesis employing the model generation theorem prover MGTP.

5.1 Framework of Program Synthesis

5.1.1 Proof and Program

Many attempts to realize program synthesis by theorem proving have been made in recent twenty years. For instance, Manna and Waldinger[Man80] proposed a method embedded in the Tableau Method and extracted a program from a proof by hand. Traugott[Tra89] extended this method and applied to a variety

$$\begin{array}{ccccc} \text{Type} & = & \text{Proposition} & \Leftrightarrow & \text{Specification} \\ \vdots & & \vdots & & \vdots \\ \lambda\text{-term} & = & \text{Proof} & \Leftrightarrow & \text{Program} \end{array}$$

Figure 7: The Curry-Howard Isomorphism

of sort algorithms. But the method is rather ad hoc and has insufficient justification of the correctness of the extracted program.

A more strict approach is to extract programs from proofs in the constructive mathematics in the way that is based on the Curry-Howard isomorphism[Mar82]. The basic idea, first proposed at the end of the 1960's, is based on the important correspondence between two pairs of a typed lambda term and its type and a proposition and its proof. Such correspondency is called proposition as type principle, or Curry-Howard isomorphism. Curry-Howard isomorphism is shown in Fig. 7

The major advantage of this approach is the correspondence between the soundness of the extracted program and the correctness of the proof. A program is mathematically assured if the proof is correct.

Our program extraction method is based on [Tak87]. But our program extraction algorithm is so powerful that such programming facilities as modular and the recursion in various data types are introduced.

5.1.2 Program synthesis by theorem prover

The set of proofs in intuitionistic logic is a subset of the set of proofs in the classical logic on which general provers are based. So general provers can in principle be utilized to find intuitionistic logic. Although it has been concensus that it is difficult to make a high performance theorem prover, it is a fascinating idea that a theorem prover can be applied for program synthesis.

We recognize three ways toward program synthesis :

(1) The main part is proved by hand, and easy parts are proved by the theorem prover.

(2) Under some useful lemmas, the provided theorem is proved by the theorem prover.

(3) The whole proof is generated by the theorem prover from general knowledge.

We approach the sort problem from the position (2), we provide essential lemmas for each algorithms to the theorem prover.

5.2 The problems and the solutions

5.2.1 Intuitionistic logic and prover

Some proofs in classical logic have no corresponding programs. Usual first-order provers generate proofs based on classical logic. This suggests to us that current prover technologies are useless for program synthesis. The difference between the classical logic and the intuionistic logic is that excluded middle($\forall$ P P $\vee$ $\neg$ P) cannot be provable in intuitionistic logic. Excluded middle is equivalent to reduction to absurdity. Reduction to absurdity has nothing to do with programs. Reduction to absurdity is a proof method that to assume the negation of a conclusion and to deduce the contradiction. You can get no information from contradiction. If the conclusion is the spec of the program, reduction to absurdity is prohibited.

The efficient provers today handle formulas in the clausal form. There is only one deduction rule for the clausal form : Resolution Principle. Transformation of formulas into clausal form is beyond intuitionistic logic. For example, you cannot deduce $\neg$ A $\vee$ B from A$\supset$B. However the Resolution Principle is within intuitionistic logic. Hence, the problems in using clausal form based provers are combined to the transformation to the clausal form. If the specification can be initially expressed in a clausal form, all the prover based on the clausal form can be used. After the transformation of premises to the clausal form, you can prove some propositions which cannot be deduced in intuitionistic logic. In order to prove such propositions in intuitionistic logic, some formulas are in the form of excluded middle, such as Q A $\vee$ $\neg$A where Q is a sequence of universal quantification. In the case of quick sort problem, no extra formulae are needed.

5.2.2 The Recursion and the Induction

Many complicated controls of iteration can be expressed in a simple recursion. The recursion corresponds to the induction in proofs. All induction schemas are based on the well-founded induction on the ordinal number. Each schemas is correspond to each data domain such as list or natural number.

Induction can be expressed as a second-order proposition. For example, the induction schema for list is as follows.

$$\forall\ \mathbf{P{:}list} \rightarrow \mathbf{prop}$$
$$(\forall\ \mathbf{L{:}list}\ (\forall\ \mathbf{L'}\ \mathbf{:\ list\ L'} \prec \mathbf{L} \supset \mathbf{P(L')}) \supset \mathbf{P(L)}) \supset \forall\ \mathbf{L{:}list\ P(L)}$$

Second-order propositions cannot be utilized by the first-order provers because of their second-order variables with which provers need the second order unification. Second order unification is not practical because of its inefficiency.

If the second-order variables are assigned by a proposition, the induction schema comes to be a first-order proposition. For example, the above schema for sort algorithm becomes as follows.

$$[\forall\ \mathbf{Z{:}list}\ (\forall\ \mathbf{Y{:}list\ Y} \prec \mathbf{Z} \supset \exists\ \mathbf{U1{:}list\ perm(U1,\ Y)} \wedge \mathbf{ord(U1)}) \supset \exists\ \mathbf{U2{:}list\ perm(U2,\ Z)} \wedge \mathbf{ord(U2)}]$$
$$\supset \forall\ \mathbf{Z{:}list}\ \exists\ \mathbf{S{:}list\ perm(S,\ Z)} \wedge \mathbf{ord(S)}$$

Where $Y \prec Z$ means a partial order relation that Y is a sublist of Z (there is a one to one mapping from Y to Z). perm(a, b) represents that a is a permutation of b, and ord(s) means that s is ordered. This is a first-order proposition, which first-order provers can handle.

Induction rule must be expressed in the clausal form in MGTP. We used the following formulae to prove a quick sort problem.

$\neg$ **ord(U2)** | $\neg$ **perm(U2,** s**)** | **ord(sort(X))**
$\neg$ **ord(U2)** | $\neg$ **perm(U2,** s**)** | **perm(sort(X), X)**
(if the proposition is satisfied for the introduced constant, it is satisfied for any list)

$\neg$ **Y** $\prec s$ | **ord(sort(Y))**
$\neg$ **Y** $\prec s$ | **perm(sort(Y), Y)**
(all sublists of the list introduced by this rule can be sorted)

Although these rules are in a form very different from that of the original formula, it is easy to see the logical equivalence.

5.2.3 Equality

In order to prove theorems which relate to programs the equality axiom (ex. $A = B \rightarrow P(A) \supset P(B)$) is quite important, because the same data can be created in various manipulations. For example, cons(car(X), cdr(X)) = X if X is not a null list. The equality axiom is a higher order proposition and is beyond the area of first-order provers. But it can also be resolved in the same way as induction. In fact it is not practical because it is not so general that you need many rules for each problem.

As there is no program information in the proof of equality nor in the proof with equality axiom, this part can be checked by any problem solver. Some efficient algorithms of equality solving, which seem promising to utilize to our prover are developed. Term Rewriting System(TRS) is one of them and we are planning to use it in our prover.

5.3 Sort problem

We chose the "sort problem" as the first example of automated program synthesis using the theorem prover. The reason for this choice is that the "sort problem" is difficult enough for the problem of program synthesis, but not so much difficult it prevents research. Moreover, various algorithms have already been invented, so it would be interesting to try to investigate how many algorithm are found in what way by the theorem prover. Also, from the viewpoint of concurrent programming that is one of main target of ICOT, the sorting problem will give us various interesting problems.

5.3.1 Propositional Expression of the Sorting Problem

It is necessary to express the problem as a mathematical formula to give it to the theorem prover. The expression which we used is as follows.

$$\forall\ \mathbf{Z{:}list}\ \exists\ \mathbf{S{:}list}\ \mathbf{perm(Z,\ S)}\ \wedge\ \mathbf{ord(S)}$$

Where Z:list is a syntax sugar of

$$\forall\ \mathbf{Z{:}list}\ \ldots\ \rightarrow\ \forall\ \mathbf{Z}\ \in\ \mathbf{list} \supset \ldots,$$
$$\exists\ \mathbf{Z{:}list}\ \ldots\ \rightarrow\ \exists\ \mathbf{Z}\ \in\ \wedge\ \ldots$$

Predicate "perm(X, Y)" means that the list X is permutation of the list Y.
Predicate "ord(X)" means that X is increasingly ordered.
Note: the problem representation cannot be uniquely determined and neither can the proof.

The followings are the major functions, definitions, and theorems provided to get the proof of the quick sort.

- functions
  car, cdr, cons, append, less, gre

- definitions
  ord:list → prop,
  perm:list → list → prop,
  subbag:list → prop less-set:list → nat → prop,
  gre-set:list → nat → prop,
  X ∈ list ⊃ X = nil ∧ X = cons(car(X), cdr(X)),
  X ∈ list ⊃ perm(X, X),
  ord(nil),
  ∀ Z Z ⊂ list ⊃ ∀ E E ∈ nat ⊃ ¬ Z = nil ⊃ less-set(less(Z), E),
  ∀ Z Z ∈ list ⊃ ∀ E E ∈ nat ⊃ ¬ Z = nil ⊃ gre-set(gre(Z), E).

- lemmas

  /divide lemma/
  ∀ L L ∈ list ⊃ ∀ E E ∈ element ⊃
  ∃ Great, Less Great, Less ∈ list ∧
  perm(append(less, great), L) ∧
  less-set(Less, E) ∧
  gre-set(Gre, E) ∧
  subbag(Less, L) ∧
  subbag(Gre, L),

  /ordered append/
  ∀ E E ∈ element ⊃ ∀ Great, Less Great, Less ∈ list ⊃
  less-set(Less, E) ∧
  gre-set(Gre, E) ∧
  ord(Less) ∧
  ord(Gre) ⊃ ord(append(Less, cons(E, Gre))).

5.3.2 Finite Domain Expression of Problems

We have introduced dom-predicate in order to get the proof corresponding to quick sort to use the finite domain prover. Dom-predicate is the predicate which generates ground terms of the Herbrand universe, and has a great effect on the performance of the prover. That is, without generating the term as the counter example, refutation would never succeed.

5.3.3 Efficient Proof Search

As we described in the previous section, dom-predicate has a great effect on the performance of solving non-range-restricted problems by MGTP. The tactics which we adopted for general use for automated programming are as follows. We add the term included in the literal which is in the model, and its

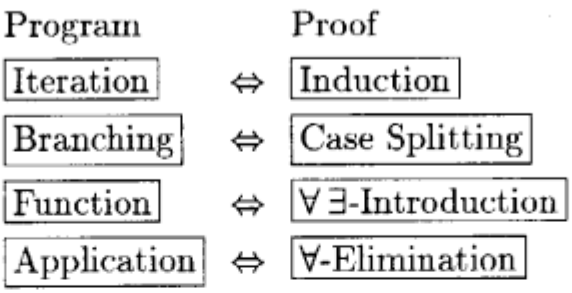

Figure 8: The Correspndence between Proofs and Programs

subterm to dom. This method of generating dom corresponds to searching an example in the proof constructively. Because the bottom-up method is one of the proof tactics, the model generated from MGTP can be regarded as the lemmas. The term included in this lemma is the value which we can construct, considering each of the propositions as a program. There will be no term in the program except such terms. And the proof from which a program is extracted must be constructed by such terms.

5.4 Program Extraction

In this section, we show the program extraction method by the quick sort example.

5.4.1 Program Extraction Rules

Fig. 8 will be of good help for intuitionistic understanding of the relation between a program and a proof. We briefly describe a corresponding program extraction rule to each inference rule in logic.

- **∀, ∃-introduction**
  In the proof using ∀, ∃-introduction rule, the proposition ∀ x:t.some y:t.p(x, y) should be proved by practically constructing y which satisfies p(x, y).In this case, the program corresponding to such a proof is the function whose argument is x, and it returns y which satisfy p(x, y). So, if we call the function f then p(x, f(x)) is true for all x.

- **∀-elimination**
  In the proof using ∀-elimination rule, the proposition q(a) should be proved by showing that q(x) is true for all x, that is, all x:t.q(x).In this case, the program extracted from the proof of q(a) is the application of a to the program corresponding to the proof of all x:t.q(x). The program extracted from the proof of∀x:t.q(x) is a function whose argument is x.

- **∨-elimination**
  Or-elimination rule can be considered as the divide-and-conquer method. In the proof using or-elimination rules, a given proposition should be proved by dividing into some cases and showing that the proposition could be derived in each case, and that the division could be valid.
  In this case, the program extracted from the proof using or-elimination forms the conditional expressions, that is, if - then - else - formation, and each condition corresponds to each of the divided cases.

- **use lemma**
  Obviously, using lemma in the proof is to refer to the fact that has been proved in the context previously by using the lemma name.By the analogy between proof and program, it is shown that the program extracted from the proof of the lemma is referred in the program extracted from the proof. So, using lemma corresponds to using a subroutine.

- **Induction**
  The program extracted from the proof using the induction forms the recursive function, that is, the function in which the function itself is used in the description. In such a program, the sub-program extracted from the sub-proof which uses the inductive assumption is in the form of the application of the program itself. For the inference rules of inductions, there are program extracting rules which have the fixed point operator μ. The recursive function can be defined by operator μ.

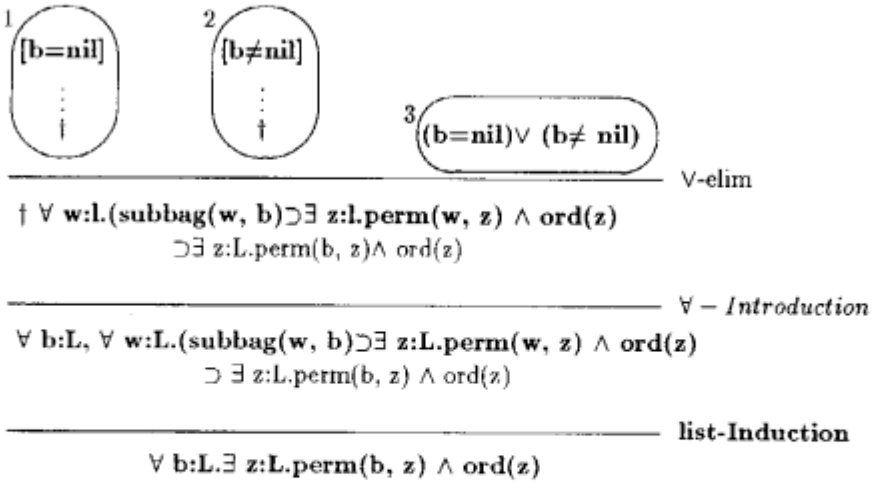

Figure 9: Proof Structure of Quick Sort(1)

Now we will show how the program is extracted from the proof of the quick sort problem. We can make program extracting possible by applying the program extracting rules corresponding to each inference rules. The following is the observation of extracting programs from each sub-proof structure.

On the sorting problem, the theorem can be written as the following.

$$\forall\ \mathbf{X{:}list.}\exists\ \mathbf{Y{:}list.}\ \mathbf{perm(X,\ Y)} \wedge \mathbf{ord(Y)}$$

However there are many proofs of sorting problems, so we will consider one example, shown in Fig. 9

In this proof, we first use induction for string(what we call string-induction), and use ∀-introduction.Then we get the following subgoal.

$$(\forall\ \mathbf{W{:}list.(subbag(W,\ B)} \supset \exists\ \mathbf{Z{:}list.perm(W,\ Z)} \wedge \mathbf{ord(Z)}) \supset \exists\ \mathbf{Z{:}list.perm(B,\ Z)} \wedge \mathbf{ord(Z)}$$

In order to prove this subgoal, we use the or-elimination rule(divide-and-conquer).In this case, we divide into two cases, b=nil, and b≠nil.

First, Fig. 10 shows that the extracted programs corresponding to three sub-proofs, b=nil, b≠nil, and the proof of validity of the division. In the indexed block 1, 2, lambda x is the abstraction of the assumption b=nil and b≠nil, and lambda y is the abstraction of the assumption of (string-)induction.

Second, Fig. 11 shows that the extracted programs corresponding to the sub-proof include an or-elimination rule.As we described above, the extracted program from the subproof using the or-elimination rule forms the conditional expression.Here, lambda y is the abstraction of the inductive assumption. So, this program does not form a recursive function.

Finally, Fig. 12 shows that the extracted program correspond to the whole proof includes a string-induction rule. In this program, there is no λ y that is the abstraction of the inductive assumption.This program forms the recursive function z in which function z is used.The function z is described as recursion using fixed point operator μ.

The extracted programs from the proof of the sorting problems is the following.

μ **z.** λ **x.if x=nil**
then nil
else append(z(first(qdiv(head(x), tail(x)))),
cons(head(x),

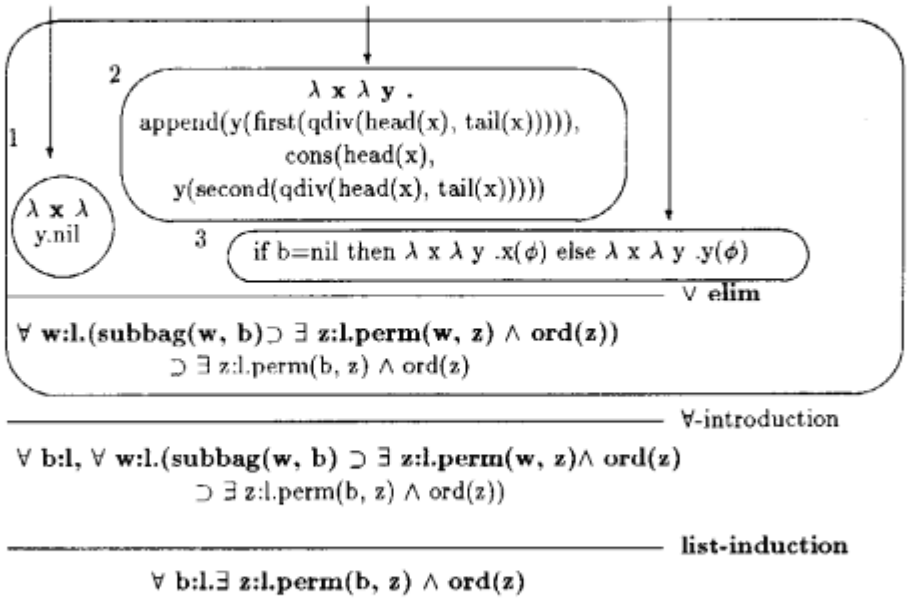

Figure 10: Proof Structure of Quick Sort(2)

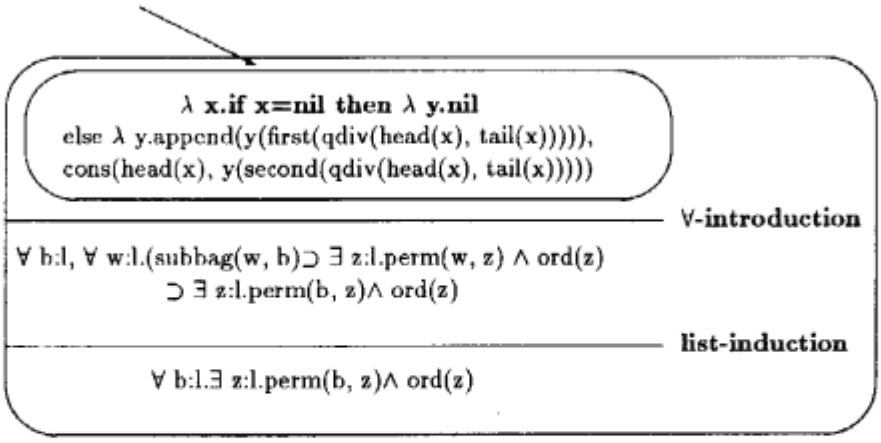

Figure 11: Proof Structure of Quick Sort(3)

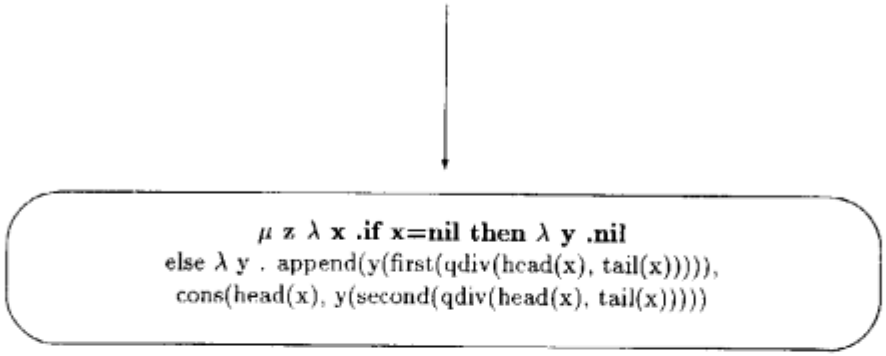

Fig 6.

Figure 12: Proof Structure of Quick Sort(4)

z(second(qdiv(head(x), tail(x)))).

In the description, qdiv is the function extracted from lemma 'qdiv', so qdiv is the subroutine in this function. The arguments of qdiv are element a and string x, and it returns tuple of string, (x1, x2), in other words qdiv(a, x) = (x1, x2). x1 consists of elements in x which are all smaller than a. And x2 consists of elements in x which are all larger than a. first and second are functions which return the first element and the second element of the given tuple.
The extracted program in this example is a well-known sorting program, what we call, quick sort.

6 Conclusion

We have presented a model-generation based theorem prover, MGTP, which is implemented in KL1. The implementation techniques developed in KL1 are also useful for constructing other systems which treat ground models, such as truth maintenance systems and intelligent database systems. We have also presented a program synthesis system which uses the MGTP prover.

Model-generation provers seem to be much more efficient than model-elimination provers such as PTTP for a large class of problems, including Shubert's Steamroller problem, that essentially deal with finite domains. Model generation also makes it easy for us to capture the proof process and to design an efficient prover based on it. In particular, we largely owe our success in the implementation of MGTP in KL1 to the fact that it needs only to match (one-way unification). In MGTP, variables appearing in the input formulas are represented by KL1 variables, and fresh variables for a different instance of the formula are created automatically by simply calling corresponding KL1 clauses. The transformation of the input formula into corresponding KL1 clauses is performed quite mechanically with little computational cost.

SATCHMO-like provers, however, are less effective in proving theorems dealing with infinite domains, which are typical mathematical problems. This is because they attempt, in a coarse manner, to generate the whole Herbrand universe for the given formula. In such cases, appropriate strategies to restrict search space need to be incorporated into the prover. Otherwise, the use of full unification with occurs check might be indispensable.

We have also had some experiences with solving more difficult problems to which MGTP is not suited, by using Prolog, ESP and KL1 programs. All these programs are designed specifically for each problem while still being based on model generation. Several versions of MGTP for nonground models are currently being constructed and tested. They adopt ground-term representation for object level variables and employ full unification procedure with occurs check. Performance evaluation of the provers on MULTI-PSI is now being carried out intensively.

It is often pointed out by some logic programming researchers that a full first-order theorem prover has few applications except for mathematics. However, we think that automated programming is one of the most promising applications for the first-order prover, now that automated program synthesis, which has been a difficult problem for a long time, has the chance to be realized through a thoerem prover technology.

Curry-Howard Isomorphism is an elegant method for relating proofs with programs. This principle together with the prover technology makes automated programming possible. Through the experiments, on sort problems, of theorem proving and program extraction, we have shown that this approach is practical. The following are problems that we faced and solved.

- Not all proofs generated by a theorem prover have corresponding programs, but all programs have corresponding proofs and the theorem prover can find all of them. Proper control of provers can avoid useless proofs.
- Induction, which is essential for program synthesis, can be represented as first-order axioms.
- The equality axiom is also essential but the proof of equality has nothing to do with programs, so we can use any equality solver such as a term rewriting system for improving the efficiency of the prover.

Acknowledgement

We would like to express our gratitude to Dr. Kazuhiro Fuchi for giving us the opportunity of doing this research and for showing us his original programs, which helped us to develop MGTP. We also wish to thank Dr. Koichi Furukawa for introducing us to other related works and for his advice.

References

[Bib86] Bibel, W., *Automated Theorem Proving,* Vieweg, 1986.

[BMSU86] Bancilhon, F. Maier, D. Sagiv, Y. and Ullman, J.D., Magic Sets and other strange ways to implement Logic Programs, in *Proc. of the ACM SIGACT-SIGMOD Symp. on Princ. of Database Systems,* 1986.

[FH90] Fujita, H. and Hasegawa, R., Implementing A Parallel Theorem Prover in KL1, in *Proc. of KL1 Programming Workshop '90,* pp.140-149, 1990 (in Japanese).

[Fuc90] Fuchi, K., Impression on KL1 programming – from my experience with writing parallel provers –, in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990 (in Japanese).

[Lov78] Lovelend, D.W., *Automated Theorem Proving: A Logical Basis,* North-Holland, 1978.

[Man80] Manna, Z. and Waldinger R., A deductive approach to program synthesis, in *ACM Trans. Programming Languages and Systems***2**(1), pp.91-121.

[Mar82] Martin-Löf P., Constructive mathematics and computer programming in *Proc. International Congress for Logic, Methodology and Philosophy of Science,* pp. 153-175, 1982.

[MB88] Manthey R. and Bry, F., SATCHMO: a theorem prover implemented in Prolog, in *Proc. of CADE 88, Argonne, illinois,* 1988.

[Ove90] Overbeek, R., private communication, 1990.

[Sch89] Schumann, J., SETHEO: User's Manual, Technical report, ATP-Report, Technische Universitat Munchen, 1989.

[Sti88] Stickel, M.E., A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, in *Journal of Automated Reasoning* **4** pp.353-380, 1988.

[Tak87] Takayama, Y., Writing Programs as QJ Proof and Compiling into Prolog Programs, in *Proc. of IEEE The Symposium on Logic Programming '87,* pp.278-287, 1987.

[Tra89] Traugott, J., Deductive System of Sorting Programs, in

[Wos88] Wos, L., *Automated Reasoning – 33 Basic Research Problems –,* Prentice-Hall, 1988.