TR-606

A Model Generation Theorem Prover in KL1. Using a Ramified-Stack Algorithm

by
H. Fujita & R. Hasegawa

December, 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 Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm

Hiroshi FUJITA* Ryuzo HASEGAWA
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
fujita@sys.crl.melco.co.jp hasegawa@icot.or.jp

Abstract

This paper presents a model-generation based parallel theorem prover for first-order logic implemented in KL1. The prover, called MGTP, is efficient in solving range-restricted non-Horn problems. The range-restrictedness condition allows us to represent object-level variables with KL1 variables, and to use only matching rather than full unification, thereby obtaining an interpreter that is very simple yet efficient. The implementation techniques developed are also useful for other related areas, such as truth maintenance systems and intelligent database systems. To improve the efficiency of MGTP, we also developed a *ramified-stack* (RAMS) algorithm for removing redundant computation during conjunctive matching phase of a proving process. Experimental results show that an MGTP prover with RAMS can attain orders of magnitude speedup over the naive one without RAMS, and can achieve good performance for non-Horn problems on a non shared memory multiprocessor, Multi-PSI.

1 Introduction

We have been conducting research on theorem proving under the Fifth Research Laboratory at ICOT[FH90, HFF90, FKKFH90, HKFFK90]. The objective of this research project is to develop a parallel automated reasoning system on the parallel inference machine, PIM, based on KL1 and PIMOS technology[CSM88]. We aim at applying this system to various fields such as natural language processing, intelligent database systems, and automated programming.

From the viewpoint of logic programming, the motive for the research is twofold. First, the research would contribute to those aiming at extending languages and/or systems from Horn clause logic to full first-order logic. Secondly, theorem proving is one of the most important applications that could be built upon the logic programming systems.

From the viewpoint of theorem proving, on the other hand, it seems that the logic programming community is in a rather mature state for dealing

*Present address: Mitsubishi Electric Corporation
8-1-1 Tsukaguchi-honmachi, Amagasaki, Hyogo 661, Japan

with more classical and difficult problems that remain unsolved or have been abandoned. We might achieve a breakthrough if we apply logic programming technology to theorem proving.

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[Lov78] based theorem prover called PTTP[Sti88]. PTTP is able to deal with any first-order formula in Horn clause form (augmented with contrapositives) without loss of completeness or soundness. It works by employing unification with occurs check, the model elimination reduction rule, and iterative deepening depth-first search. Schumann et al. built a connection-method[Bib86] based theorem-proving system, SETHEO[Sch89], in which a method identical to model elimination is used as a main proof mechanism. Manthey and Bry presented a tableaux-like theorem prover, SATCHMO[MB88], which is a very short and simple program in Prolog.

The above systems all utilize the fact that Horn clause problems 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. However, that approach cannot be taken in KL1 because a KL1 clause is not just a Horn clause; it has extra-logical constructs such as a guard and a commit operator. Therefore, we have first to solve the problem of how to handle variables appearing in the given theorems before going into the main subject of how to exploit parallelisms. This problem arises whenever we consider implementing meta-programs such as a Prolog or a GHC meta-interpreter in KL1.

To solve the above problem, we adopted a *model generation method*, on which SATCHMO is based, as a basic proof procedure, and restricted the method to the ground model case. It turns out that this method is well suited to our purpose of implementing a parallel theorem prover in KL1.

In the next Section, the problem of meta-programming in KL1 is discussed. In Section 3, a model generation method, on which our theorem prover is based, is explained. In Section 4, a simple MGTP interpreter is presented as a solution for the above problem. In Section 5, a ramified-stack algorithm[HFF90] is presented, which improves the performance of a model generation prover greatly. In Section 6, performance results are presented together with a consideration of parallel execution of the MGTP prover on a prototype parallel inference machine, Multi-PSI[NIIRC89]. After discussing about issues concerning further improvements of the prover in section 7, we conclude the paper in section 8.

2 Meta-programming in KL1

As mentioned in Section 1, it is not possible to represent a clause set for a theorem being proven directly with KL1 clauses. We should, therefore, treat the clause set as data rather than as a KL1 program. In this case, the inevitable problem is how to represent variables appearing in a given clause

set. Two approaches can be considered for this problem:

(1) Representing object-level variables with KL1 ground terms, or

(2) Representing object-level variables with KL1 variables.

The first approach might be the right path in meta-programming 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 large and complex compared to the main program, and make overhead bigger. 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 routines running on top of it. This enables a meta-programmer to prevent from writing tedious routines and to gain high efficiency. Furthermore, in Prolog, one can 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- and meta- levels very ambiguous. However, this approach makes the program much more simple and efficient.

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 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 that cannot 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 Model generation

Throughout this paper, a clause is represented in an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

where $A_i (1 \leq i \leq n)$ and $C_j (1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, A_2, \ldots, A_n$; the consequent is a disjunction of $C_1, C_2, \ldots, C_m$. A clause is said to be *positive* if its antecedent is $true (n = 0)$, and *negative* if its consequent is $false (m = 0)$.

There are the following two rules in the model generation method.

- Model extension rule: If there is a clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model M and $C\sigma$ is not satisfied in M, then extend the model M by adding $C\sigma$ into the model M.

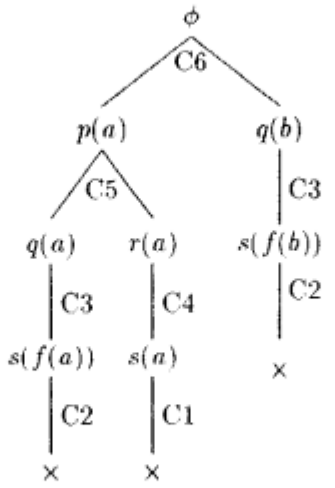

Figure 1: A proof tree for S1

- Model rejection rule: If there is a negative clause whose antecedent $A\sigma$ is satisfied in a model M, then reject the model M.

We call the process of obtaining $A\sigma$ a *conjunctive matching* of the antecedent literals against elements in a model. Note that the antecedent $(true)$ of a positive clause is satisfied by any model.

The task of model generation is to try to construct a model for a given set of clauses starting with a null set as a model candidate. If the clause set is satisfiable, a model should be found. The method can also be used to prove that the clause set is unsatisfiable, by exploring every possible model candidate to see that no model exists for the clause set.

For example, consider the following set of clauses, S1[MB88]:

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

A proof tree for the S1 problem is depicted in Fig. 1. We start with an empty model, $M_0 = \phi$. M_0 is first expanded into two cases: $M_1 = \{p(a)\}$ and $M_2 = \{q(b)\}$, by applying the model extension rule to $C6$. 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 $M_5 = \{p(a), q(a), s(f(a))\}$. Now with M_5 the model

rejection rule is applicable to $C2$, thus M_5 is rejected and marked as closed. On the other hand, M_4 is extended by $C4$ to $M_6 = \{p(a), r(a), s(a)\}$ which is rejected by $C1$. In a similar way, the remaining model candidate M_2 is extended by $C3$ to $M_7 = \{q(b), s(f(b))\}$, which is rejected by $C2$. Now that there is no way to construct any model candidate, we can conclude that the clause set S1 is unsatisfiable.

The model generation method, as its name suggests, is closely related to the model elimination method. However, the model generation method is a restricted version of the model elimination method in the sense that the polarity of literals in a clause of implicational form is fixed to either positive or negative in the model generation method, whereas it is allowed to be both positive and negative in the model elimination method. Moreover, from the procedural point of view, the model generation is restricted to proceed bottom-up (as in forward reasoning) starting at positive clauses (or facts). These restrictions, however, do not hurt the refutation completeness of the method.

The model generation can also be viewed as *unit hyper-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 branch in a proof tree obtained by the tableaux method corresponds exactly to a model candidate.

4 MGTP for ground model

The model generation method does not need full unification during conjunctive matching if the *range-restrictedness* condition[MB88] is imposed on problem clauses. A clause is said to be range-restricted if every variable in the clause has at least one occurrence in its antecedent. For example, in the S1 problem, 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[1] is satisfied, it is sufficient to consider one-way unification, or matching, instead of full unification with occurs check because a model candidate constructed by the 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.

4.1 Transforming problem clauses to KL1 clauses

The program of our MGTP prover consists of two parts: an interpreter written in KL1 and a set of KL1 clauses representing a set of clauses for the given

[1]To ensure range-restrictedness, a *dom/1* predicate is added to the antecedents of problem clauses and extra clauses for the predicate are added to the original set of clauses, if necessary. This transformation does not change the satisfiability of the original set of clauses.

```
c(1,p(X),[],     R):-true|R=cont.
c(1,s(X),[p(X)],R):-true|R=false.
c(2,q(X),[],     R):-true|R=cont.
c(2,s(Y),[q(X)],R):-true|R=false.
c(3,q(X),[],     R):-true|R=[s(f(X))].
c(4,r(X),[],     R):-true|R=[s(X)].
c(5,p(X),[],     R):-true|R=[q(X),r(X)].
c(6,true,[],     R):-true|R=[p(a),q(b)].
otherwise.
c(_,_,_,R):-true|R=fail.
```

Figure 2: S1 problem transformed to KL1 clauses

problem. During a conjunctive matching, an antecedent literal expressed in the head of a KL1 clause is matched against a model element chosen out of a model candidate which is retained in the interpreter.

Although the conjunctive matching can be implemented straightforwardly in KL1 as above, we need a programming trick for supporting variables shared among literals in a problem clause. The trick concerns how to propagate the binding for a shared variable from one literal to another.

To see this, consider the previous example, S1. The original clause set is transformed into a set of KL1 clauses as shown in Fig. 2. In `c(N,P,S,R)`, `N` indicates clause number; `P` is an antecedent literal to be matched against an element taken out of a model candidate; `S` is a pattern for receiving from the interpreter a stack of literal instances appearing to the left of `P`, that have already matched model elements; and `R` is the result returned to the interpreter when the match succeeds.

Notice that original clause $C1$ ($p(X), s(X) \rightarrow false.$) is translated to the first two KL1 clauses. The conjunctive matching for $C1$ proceeds as follows. First, the interpreter picks up some model element, E_1, out of a model candidate and tries to match the first literal $p(X)$ in C1 against E_1 by initiating a goal, $c(1, E_1, [], R_1)$. If the matching fails, then the result $R_1 = fail$ is returned by the last KL1 clause. If the matching succeeds, then the result $R_1 = cont$ is returned by the first KL1 clause and the interpreter proceeds to the next literal $s(X)$ in C1, picking up another model element, E_2, out of the model candidate and initiating a goal, $c(1, E_2, [E_1], R_2)$. Since the literal instance in the third argument, $[E_1]$, is ground, the variable X in $[p(X)]$ in the head of the second KL1 clause gets instantiated to a ground term. At the same time, the term $s(X)$ in that head is also instantiated due to the shared variable X. Under this instantiation, $s(X)$ is checked whether it matches E_2, and if the matching succeeds then the result, $R_2 = false$, is returned.

4.2 A simple MGTP interpreter

With the problem clauses transformed to KL1 clauses as above, a simple interpreter is developed as shown in Fig. 3[2].

```
clauses(_,_,_,_,quit):-true|true.
alternatively.
clauses([J|Cs],C,M,A,B):-true|
    ante(J,[true|M],[],C,M,A1,B),
    sat(A1,A2,A,B),
    clauses(Cs,C,M,A2,B).
clauses([],_,_,A,_):-true|A=sat.

ante(_,_,_,_,_,_,quit):-true|true.
alternatively.
ante(J,[P|M2],S,C,M,A,B):-true|
    mgtp:c(J,P,S,R),
    ante1(J,R,P,S,M2,C,M,A,B).
ante(_,[],_,_,_,A,_):-true|A=sat.

ante1(J,fail,_,S,M2,C,M,A,B):-true|
    ante(J,M2,S,C,M,A,B).
ante1(J,cont,P,S,M2,C,M,A,B):-true|
    ante(J,M,[P|S],C,M,A1,B),
    sat(A1,A2,A,B),
    ante(J,M2,S,C,M,A2,B).
ante1(_,false,_,_,_,_,M,A,B):-true|
    A=unsat,B=quit.
ante1(J,F,_,S,M2,C,M,A,B):-list(F)|
    cnsq(F,F,C,M,A1,B),
    sat(A1,A2,A,B),
    ante(J,M2,S,C,M,A2,B).

cnsq(_,_,_,_,_,_,quit):-true|true.
alternatively.
cnsq([D1|Ds],F,C,M,A,B):-true|
    cnsq1(D1,M,Ds,F,C,M,A,B).
cnsq([],F,C,M,A,_):-true|
    extend(F,M,C,A,_).

cnsq1(D,[D|_],_,_,_,_,A,_):-true|A=sat.
cnsq1(_,[],Ds,F,C,M,A,B):-true|
    cnsq(Ds,F,C,M,A,B).
otherwise.
cnsq1(D,[_|M2],Ds,F,C,M,A,B):-true|
    cnsq1(D,M2,Ds,F,C,M,A,B).

extend(_,_,_,_,_,quit):-true|true.
alternatively.
extend([D|Ds],M,C,A,B):-true|
    clauses(C,C,[D|M],A1,_),
    unsat(A1,A2,A,B),
    extend(Ds,M,C,A2,B).
extend([],_,_,_,A,_):-true|A=unsat.

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

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

Figure 3: A simple MGTP interpreter

The interpreter, given a list of numbers identifying problem clauses and a model candidate, checks whether the clauses are satisfiable or not. The top-level predicate, `clauses/5`, dispatches a task, `ante/7`, for checking whether each clause is satisfied or not in the current model. If all the clauses are satisfied in the current model, the result, `sat`, is returned by `sat/4` combining the results from `ante` processes.

For each clause in the given clauses, conjunctive matching is performed between the elements in the model candidate and the literals in the antecedent of the clause with `ante/7` and `ante1/9` processes. The conjunctive matching for the antecedent literals proceeds from left to right, by calling `c/4` one by one. An `ante` process retains a stack, S, of literal instances. If the match succeeds at a literal, L_i, with a model element, P, then P is pushed onto the

[2]In the program, '`alternatively`' is a KL1 compiler directive which gives a preference among clauses for evaluating guards of them in such a way that clauses above `alternatively` are evaluated before those below it. The preference, however, may not be strictly obeyed depending on implementation.

stack S, and the task proceeds to matching the next literal, L_{i+1}, together with the stack, $[P|S]$.

According to the result of `c/4`: `fail`, `cont`, `false` or `list(F)`, an `ante1/9` process determines what to do next. If the result is `cont`, for example, `ante1` will fork multiple `ante` processes to try to make every possible combination of elements out of the current model for the conjunctive matching.

If the conjunctive matching for all the antecedent literals of a clause succeeds, a `cnsq/6` process is called to check the satisfiability of the consequent of the clause. `cnsq1/8` checks whether a literal in the consequent is a member of the current model. If no literal in the consequent is a member of the current model, the current model cannot satisfy the clause. In this case, the model will be extended with each disjunct literal in the consequent of the clause by calling an `extend/5` process.

After extending the current model, a `clauses/5` process is called for each extension of the model, and the results are combined by `unsat/4`. When a `clauses` process for some of the extended models returns `sat` as the result, it means that a model is found and the clause set is known to be satisfiable. If every extension of the model leads to `unsat`, the current model is not a part of any model, if any, for the given set of clauses.

Thus, if the top-level `clauses/5` process returns `sat` as the result, then the given clause set has a model and is satisfiable, and if it returns `unsat`, then the given clause set has no model and is unsatisfiable.

5 Avoiding redundancy in conjunctive matching

To improve the performance of the model generation provers, it is essential to avoid, as much as possible, redundant computation in conjunctive matching. This section will go into implementation techniques on this subject in more detail.

5.1 Redundancy in the basic algorithm

The basic algorithm for model generation employed by the simple interpreter given in Fig. 3 contains much redundancy as well as SATCHMO does.

Let us consider a clause having two antecedent literals. For this clause, we need a pair of model elements to perform conjunctive matching. The pair will be chosen out of the current model, M. After performing one step conjunctive matching for each pair taken out of M, we may obtain another set of atoms, δ, with which the model is extended. Then, in the next step, we will have to choose pairs out of $M + \delta$. The number of such pairs amounts to

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

Notice, however, that $M \times M$ number of pairs are those which have been selected in the previous step. Conjunctive matching steps for such pairs are just superfluous but only those steps which are performed on pairs containing at least one atom from δ are needed. This argument generalizes to clauses that have more than two antecedent literals.

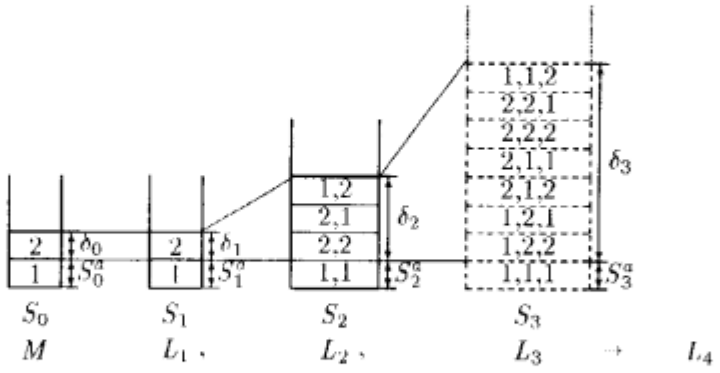

Figure 4: Literal instance stacks

A Rete-like algorithm could be used to avoid this sort of redundancy and is easy to implement with processes and streams provided by KL1, in which a literal instance is retained in a process and a process network is constructed dynamically. However, if the given clause is non-Horn for which the Rete algorithm is not designed, we would have to consider a sort of multiple world problem. In this case, one should 1) copy the whole process network having been created for each different model, or 2) attach a color to a literal instance flowing in a stream for indicating to which model the instance belongs in order to share the network.

5.2 Ramified-stack algorithm

Now we present a new method to avoid the redundancy stated above and enable us to share the common model. This method retains in a stack, instead in a process, an instance as a result of matching a literal against an element of a model. The method is illustrated in Fig. 4, where model elements are represented by numbers, and works for a Horn clause as follows:

- A stack, S_0, is assigned to a model candidate for storing model elements generated.
- A stack called *a literal instance stack* (LIS), $S_i(i > 0)$, is assigned to each literal, L_i, in the antecedent of a clause for storing literal instances. Note that LIS for the last literal expressed in dashed boxes needs not be allocated actually.
- Stack $S_i(i \geq 0)$, is divided into two parts: δ_i and S_i^a where δ_0 is the most recent model elements pushed onto S_0; $\delta_i(i > 0)$ is a set of literal instances generated at the current stage triggered by δ_0; and S_i^a is those created at the previous stages.

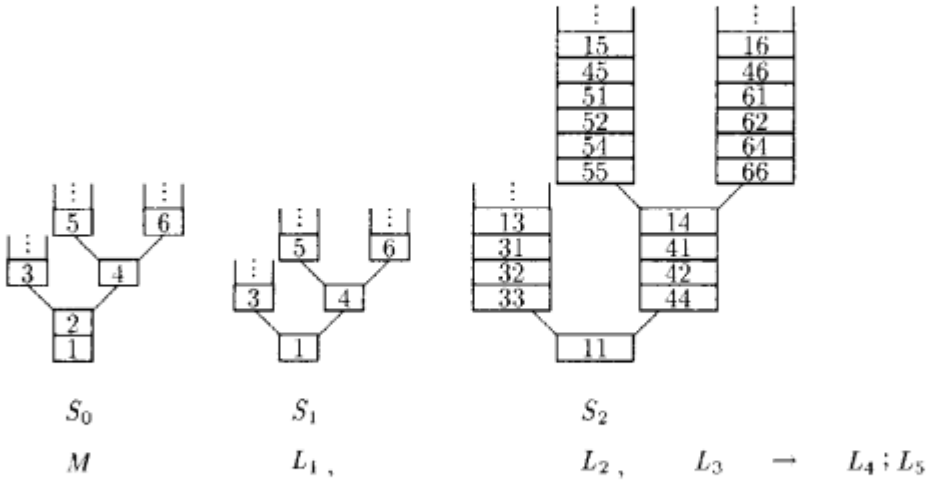

Figure 5: A system of ramified stacks

- A task, T_i, being performed at each literal, L_i, computes the following:

$$\delta_i := \delta_{i-1} \times \delta_0 \cup \delta_{i-1} \times S_0^a \cup S_{i-1}^a \times \delta_0$$

  where, $A \times B$ denotes a set of pairs of an instance taken out of A and that of B, and $S \cup T$ denotes set-union of S and T.

- Conjunctive matching for a clause with n-literals consists of a sequence of tasks $T_1, T_2, \ldots, T_n$ performed from left to right.

Now we introduce the notion of *ramified stacks* (RAMS) for a non-Horn clause. The algorithm with RAMS is based on that with LIS described above. The idea is as follows:

- A model is represented by a branch of a ramified stack and the model is extended only at the top of the current stack.

- After applying the model extension rule to a non-Horn clause, the current model may be extended to multiple descendant models.

- Every descendant model that is extended from a parent model can share its ancestors with other sibling models just by pointing the top of the stack corresponding to the parent.

- Each descendant model can extend the stack for its own sake independently of other sibling models.

Table 1: Performance comparison

Problem	Provers				MGTP-S / MGTP-R
	PTTP	SATCHMO	MGTP-S	MGTP-R	
S1	86msec	16msec	5.4msec	3msec	1.67
S2	24sec	68msec	107msec	66msec	1.57
S3	—	6.3sec	1.5sec	319msec	4.86
6-queens		899sec	253sec	650msec	390

As a result we obtain a system of ramified stacks as depicted in Fig. 5 in a straightforward manner, where it is assumed that the first literal L_1 fails to match the model element number two.

Note that ramifying a queue is not very easy in contrast to doing with a stack as in the RAMS method, as long as the list structure is used for implementing it. The MGTP interpreter with the ramified-stack algorithm is shown in Appendix A.

6 Performance evaluation

In this section, the performance of MGTP provers are compared to some of other theorem proving systems, and the effect of a ramified-stack algorithm is measured. Also shown are ways of exploiting parallelism in the MGTP prover and several experimental results of its parallel execution on a Multi-PSI.

6.1 Performance of MGTP provers on PSI-II

The provers being compared are PTTP and SATCHMO, both written in SICSTUS Prolog and run on a SUN3/260, a simple MGTP interpreter (MGTP-S), and the one with RAMS (MGTP-R). Both MGTP-S and MGTP-R run on a Pseudo-Multi-PSI system with single processor mode on a PSI-II. Table 1 shows the performance comparison among these systems where problems S2 and S3[MB88] are the Shubert's Steamroller problem and the Pelletier and Rudnicki's problem, respectively. These problems are thought rather difficult to prove by resolution provers.

Comparing PTTP with SATCHMO, we can see that there is a large difference in their performance. Problems S3 and 6-queens cannot be solved by PTTP within 30 minutes. On the other hand, SATCHMO and MGTP-S are comparable in terms of the order of magnitude. MGTP-S is three to four times faster than SATCHMO for problems S1,S3 and 6-queens. For problem S2, however, SATCHMO is faster than MGTP-S since the former uses backward reasoning as well as forward reasoning and evaluates Horn clause subset by Prolog, whereas MGTP-S and MGTP-R use only forward reasoning.

The speedup of MGTP-R compared to MGTP-S, as the effect of RAMS, ranges from 1.5 to about 400 depending on the problems given. MGTP-R shows rather small improvements in speed over MGTP-S for problems S1-S3,

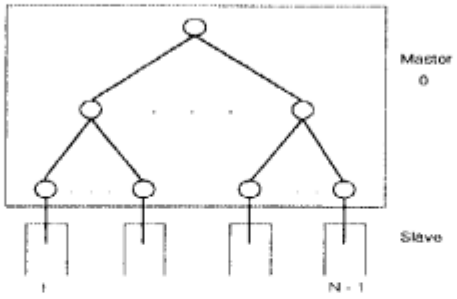

Figure 6: Simple allocation scheme

but achieves a remarkable improvement of 390 speedup for 6-queens problem. The greater the number of conjuncts in the antecedent of a clause, the more speedup can be observed.

6.2 Performance of MGTP-R on Multi-PSI

There are major three sources for parallelizing the proving processes in the MGTP prover:

- Multiple model candidates in a proof,
- Multiple clauses to which model generation rules are applied, and
- Multiple literals in conjunctive matching.

Let us assume that the prime objective to use the model generation method is to find a model as a solution. There may be alternative solutions or models for a given problem. We take it as OR-parallelism to seek these multiple solutions at the same time. According to our assumption, multiple model candidates and multiple clauses are taken as sources for exploiting the OR-parallelism. On the other hand, multiple literals are the source of AND-parallelism since all the literals in a clause relate to a single solution, where shared variables in the clause should have compatible values.

Processor allocation

With the current version of the MGTP prover, we attempt to exploit only OR-parallelism on the Multi-PSI machine.

Processor allocation methods we have adopted achieve 'bounded-OR' parallelism in the sense that OR-parallel forking in the proving process is suppressed so as to meet restricted resource circumstances.

One simple way of doing this, called *simple allocation*, is depicted in Fig. 6. We expand model candidates starting with empty model using a single master-processor until the number of the candidates exceeds the number of available processors, then distribute the remaining tasks to slave-processors. Each slave processor explores the branches assigned without further distributing tasks to any other processors. This simple allocation scheme for task distribution works fairly well since communication cost can be minimized.

Table 2: Performance of MGTP-R on Multi-PSI

Problem	Number of processors				
	1	2	4	8	16
4-queens					
time(msec)	40	40	39	44	44
speedup	1.00	1.00	1.02	0.90	0.90
Kred	1.45	1.47	1.48	1.50	1.50
6-queens					
time(msec)	650	407	266	189	154
speedup	1.00	1.59	2.44	3.44	4.22
Kred	23.7	23.7	23.7	23.8	23.8
8-queens					
time(msec)	12,538	6,425	3,336	1,815	1,005
speedup	1.00	1.95	3.76	6.91	12.5
Kred	460	460	460	460	460
10 queens					
time(msec)	315,498	159,881	79,921	40,852	21,820
speedup	1.00	1.97	3.94	7.72	14.5
Kred	11,117	11,117	11,117	11,117	11,117

6.2.1 Speedups with multiple processors

One of the examples we used is the N queens problem. The problem can be expressed by the clause set as follows:

$$
\begin{aligned}
C_1 &: \quad true \rightarrow p(1,1); p(1,2); \ldots; p(1,n). \\
C_2 &: \quad true \rightarrow p(2,1); p(2,2); \ldots; p(2,n). \\
&\quad \ldots \\
C_n &: \quad true \rightarrow p(n,1); p(n,2); \ldots; p(n,n). \\
C_{n+1} &: \quad p(X_1,Y_1), p(X_2,Y_2), unsafe(X_1,Y_1,X_2,Y_2) \rightarrow false.
\end{aligned}
$$

The first N clauses just expresses every possibility of placing queens on the N by N chess board. The last clause expresses the constraint that a pair of queens must satisfy. So, the problem would be solved just when either a model (one solution) or all of the models (all solutions)[3] are obtained for the clause set. The KL1 clauses for 4-queens problem are shown in Appendix B.

The performance has been measured on the MGTP-R prover running on the Multi-PSI with simple allocation method. Table 2 gives the result of the all solution search with the N-queens problem. Here we should note that

[3] All models can be obtained, if they are finite, by the MGTP interpreter with all-solution mode.

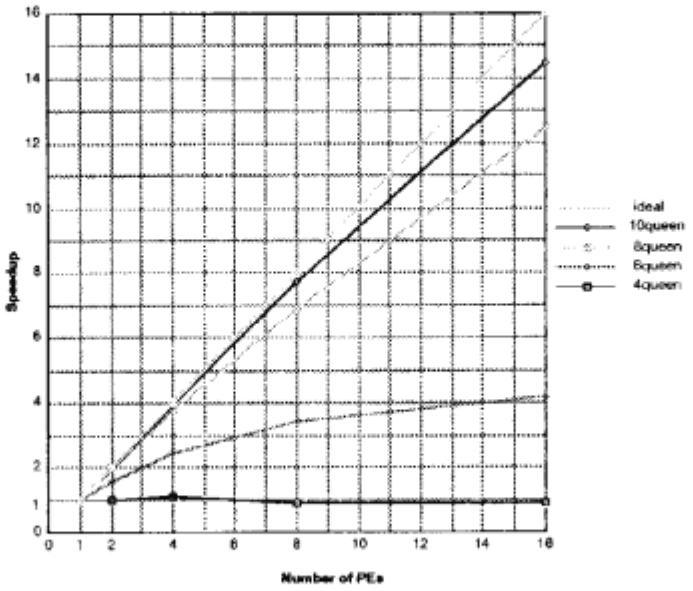

Figure 7: Speedup of MGTP-R on Multi-PSI

the total number of reductions stays constant even though the number of processors used increases. That means no extra computation is introduced by distributing tasks. Speedups obtained using up to 16 processors are shown in Fig. 7. For the 10-queens problem, almost linear speedup is obtained as the number of processors increases. Only for the 4-queens problem, the speedup rate is rather small. This is probably because in such a small size of problem, the constant amount of the interpretation overhead would dominate the proper tasks for the proving process.

7 Discussions

The MGTP prover using a ramified-stack algorithm achieves a good performance yet is not optimal. There are four main issues that we are taking into consideration for further improving the performance of the prover: indexing, pruning search spaces, removing interpretation overhead, and exploiting AND-parallelism. These will be discussed briefly in the following subsections.

7.1 Indexing

If the number of predicate symbols appearing in the given set of clauses is large, or the number of non-unifiable literals in the clause set is large, then by introducing clause-indexing, substantial amount of matching tasks that would eventually fail can be avoided. To do this, the ramified stacks for a given set of clauses should be rearranged so that a single literal stack is assigned for each distinct predicate or distinct literal rather than for each literal in each clause.

A more sophisticated indexing mechanism could be employed as in [Sti89], which retrieves any term rather than mere predicate symbols. This technique will become crucial in solving a class of problems whose proof size is large. However, since the task of computing indexing itself is not very small, significant overhead would also be introduced. With the Multi-PSI machine, a firmware-level support for the term-indexing mechanism would be desirable.

7.2 Pruning search spaces

The MGTP prover is complete in the sense that for any given unsatisfiable set of clauses it can construct a closed proof tree. However, it is not necessarily able to find the simplest proof since its control strategy for searching for a proof is fixed in the current version where a LIFO-like approach is employed for scheduling candidates for model extension. However, other control strategies, such as FIFO, could also be employed without affecting the basic function of the RAMS algorithm.

With regard to the pruning method, another approach is now being taken into consideration. That is the *relevancy testing* in [WL89], which is also closely related to an indexing method. With this method only such an atom that is relevant to a negative clause is generated as a model element. The idea behind this is that when the prover is used in refutation mode, not all of the atoms that will be added to a model candidate by applying the model extension rule can be used for rejecting the model.

By employing this technique, substantial speedup may be obtained for a class of problems, for example, those dealing with predicates from a number of disjoint domain of knowledge.

7.3 Partial evaluation

The MGTP prover runs in an interpretive manner. This means that an optimization is possible by removing an interpretation overhead on the basis of partial evaluation technique.

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 prover. 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.

The use of the partial evaluation technique will become crucial when we develop a more complicated interpreter than the MGTP prover for ground models. Indeed, in order to deal with problem clauses which do not meet the range-restrictedness condition, we will be forced to take a ground term approach for representing object-level variables and will need to incorporate routines for unification, copying, and environment handling into the interpreter, all of which will cause substantial overhead.

To remove these overheads, we need an automatic partial evaluator that

can unfold KL1 clauses and evaluate guard goals symbolically without changing the synchronization condition as well as the input/output condition. Further, it might be possible to optimize the partial evaluation process if a self-applicable partial evaluator were available.

7.4 AND parallelism

In our terminology, AND-parallelism corresponds to a parallelism inherent in conjunctive matching processes of the MGTP prover. Though the parallelism is large, it is not exploited in the current implementation of MGTP since the Multi-PSI machine is not suited to exploit this kind of fine-grained AND-parallelism. This is because AND-parallel processes usually share a large amount of information to perform their tasks, and they require heavy communications which tend to incur a serious bottleneck in a local-memory architecture such as Multi-PSI. We are now, nevertheless, attempting to exploit as much AND-parallelism as possible by separating conjunctive matching tasks into independent subtasks.

8 Conclusion

We have presented a model-generation theorem prover, MGTP, which is implemented in KL1 and evaluated its performance. For range-restricted problems, model-generation provers need to use only matching rather than full unification and can make full use of KL1 variables, thereby achieving good efficiency.

The key techniques for implementing MGTP in KL1 are: (1) Translating a given set of clauses of implicational form to a corresponding set of KL1 clauses, (2) Emulating a conjunctive matching by head unification for the KL1 clauses, and (3) Obtaining fresh variables automatically just by calling a KL1 clause. These techniques are very simple and straightforward yet effective.

To improve the efficiency of MGTP, we developed a ramified-stack algorithm which enables us to avoid redundant computations in conjunctive matching phase of the prover. We have also obtained good performance results by running the MGTP prover in the parallel environment on the MULTI-PSI.

For solving non-range-restricted problems, the MGTP prover would require further extension or modification. If the problem is Horn, it can be solved with the MGTP prover extended with unification with occurs-check for ground-represented variables without changing the basic structure of the prover. For non-Horn problems, however, substantial change in the structure of the prover would be required since the ramified-stack algorithm is not designed appropriately so as to manage shared variables in literals in the consequent of a clause.

Acknowledgements

We would like to thank 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 Koichi Furukawa for introducing us to

other related works and for his advice. Thanks are also due to M. Koshimura at JBA Co. Ltd., M. Fujita at 5th Research Laboratory at ICOT, and J. Iimura and F. Kumeno at MRI Inc. for tuning up programs and for measuring the performance of MGTP.

References

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

[CSM88] Chikayama, T., Sato, H., and Miyazaki, T., Overview of the Parallel Inference Machine Operating System (PIMOS), in *Proc of FGCS'88*, 1988.

[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).

[FKKFH90] Fujita, H., Koshimura, M., Kawamura, T., Fujita, M., and Hasegawa, R., A Model-Generation Theorem Prover in KL1, *Joint US-Japan Workshop*, 1990.

[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).

[HFF90] Hasegawa, R., Fujita, H. and Fujita, M., A Parallel Theorem Prover in KL1 and Its Application to Program Sysnthesis, *Italy-Japan-Sweden Workshop*, ICOT TR-588, 1990.

[HKFFK90] Hasegawa, R., Kawamura, T., Fujita, M., Fujita, H., and Koshimura, M., MGTP: A Hyper-matching Model-Generation Theorem Prover with Ramified Stacks, *Joint UK-Japanese Workshop*, 1990.

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

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

[NIIRC89] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K., and Chikayama, T., Distributed Implementation of KL1 on the Multi-PSI/V2, in *Proc. of 6th ICLP*, 1989.

[Sch89] Schumann, J., SETHEO: User's Manual, Technical report, ATP-Report, Technische Universität München, 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.

[Sti89] Stickel, M.E., The Path-indexing method for indexing terms, Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1989.

[WOLB84] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., *Automated Reasoning: Introduction and Applications*, Prentice-Hall, 1984.

[Wos88] Wos, L., *Automated Reasoning – 33 Basic Research Problems –*,

Prentice-Hall, 1988.

[WL89] Wilson, D.S. and Loveland, D.W., Incorporating Relevancy Testing in SATCHMO, CS-1989-24, Department of Computer Science, Duke University, Durham, North Carolina, 1989.

A An MGTP interpreter with ramified stacks

To use the interpreter, the user initiates a goal, `do(M,FLIS,LIS,U,D,S)`, by giving a nil to `M` as an initial model candidate, a list of initial literal instance stacks for negative clauses to `FLIS`, a list of initial literal instance stacks for mixed clauses to `LIS`, a list of consequents of positive Horn clauses to `U`, a list of consequents of positive non-Horn clauses to `D`. The result of the goal is returned in a stream `S`. If the result is nil, then there is no model for the problem. Otherwise, `S` contains (finite) models for the given problem.

```
:- public do/6. :- module mgtp_i.

do(M, FLIS,LIS, [P|DisjU],DisjO, Sat) :- true |
    member(P,M, YN),
    (YN = yes   -> do(M, FLIS,LIS, DisjU,DisjO, Sat);
     YN = no    -> do1(M,P, FLIS,LIS, DisjU,DisjO, Sat)).
do(M, FLIS,LIS, [],[Dis|DisjO], Sat) :- true |
    checkConsq(Dis, M, S),
    (S = sat    -> do(M, FLIS,LIS, [],DisjO, Sat);
     S = unsat  -> expand(Dis, M, FLIS,LIS, DisjO, Sat)).
do(M, _,_, [],[], Sat) :- true | Sat = [M].

do1(M,P, FLIS,LIS, DisjU,DisjO, Sat) :- true |
    NM = [P|M], DM = [P],
    clauses(NM,DM, FLIS,NFLIS, [],False, _,_),
    do1Decide(False, NM,DM, NFLIS,LIS, DisjU,DisjO, Sat).

do1Decide([], M,DM, FLIS,LIS, DisjU,DisjO, Sat) :- true |
    clauses(M,DM, LIS,NLIS, DisjU,NDisjU, DisjO,NDisjO),
    do(M, FLIS,NLIS, NDisjU,NDisjO, Sat).
do1Decide(False, _,_, _,_, _,_, Sat) :- False \= [] | Sat = [].

expand([], _, _,_, _, Sat) :- true | Sat = [].
expand([P|Ps], M, FLIS,LIS, DisjO, Sat) :- true |
    Sat = {Sat1,Sat2},
    do1(M,P, FLIS,LIS, [],DisjO, Sat1),
    expand(Ps, M, FLIS,LIS, DisjO, Sat2).

clauses(M,DM, [{I,LiS}|LIS],NLIS, Si,So, Di,Do) :- true |
    NLIS = [{I,NLiS}|NLIS1],
    clause(I,LiS,NLiS, M,DM, Si,Sm, Di,Dm),
    clauses(M,DM, LIS,NLIS1, Sm,So, Dm,Do).
clauses(_,_, [],NLIS, Si,So, Di,Do) :- true |
    NLIS = [], So = Si, Do = Di.

clause(ID,[],NLiS, M,DM, Si,So, Di,Do) :- true |
    NLiS = [],
    mgtp:c(ID, DM, Si,So, Di,Do).
clause(ID,[S|LiS],NLiS, M,DM, Si,So, Di,Do) :- true |
```

```
    NLiS = [NS|LiS1],
    mgtp:c(ID, DM, S,NS, [],DS),
    ante(ID,LiS,LiS1, M,DM, S,DS, Si,So, Di,Do).

ante(ID,[],NLiS, M,DM, Stack,DStack, Si,So, Di,Do) :- true |
    NLiS = [],
    literal(ID, M,DM, Stack,DStack, Si,So, Di,Do).
ante(ID,[S|LiS],NLiS, M,DM, Stack,DStack, Si,So, Di,Do) :- true |
    NLiS = [NS|NLiS1],
    literal(ID, M,DM, Stack,DStack, S,NS, [],DS),
    ante(ID,LiS,NLiS1, M,DM, S,DS, Si,So, Di,Do).

literal(_, _,_, [],[], Si,So, Di,Do) :- true | So = Si, Do = Di.
literal(ID, M,DM, Stack,[S|DStack], Si,So, Di,Do) :- true |
    literal(ID, M,S, Si,Sm, Di,Dm),
    literal(ID, M,DM, Stack,DStack, Sm,So, Dm,Do).
literal(ID, M,[P], [S|Stack],[], Si,So, Di,Do) :- true |
    mgtp:c(ID, [P|S], Si,Sm, Di,Dm),
    literal(ID, M,[P], Stack,[], Sm,So, Dm,Do).

literal(_, [],_, Si,So, Di,Do) :- true | So = Si, Do = Di.
literal(ID, [P|Ps],S, Si,So, Di,Do) :- true |
    mgtp:c(ID, [P|S], Si,Sm, Di,Dm),
    literal(ID, Ps,S, Sm,So, Dm,Do).

member(_,[], YN) :- true | YN = no.
member(X,[X|_], YN) :- true | YN = yes.
otherwise.
member(X,[_|Xs], YN) :- true | member(X,Xs, YN).

checkConsq([],_, S) :- true | S = unsat.
checkConsq([X|Xs],M, S) :- true |
    member(X,M, YN),
    (YN = yes   -> S = sat;
     YN = no    -> checkConsq(Xs,M, S)).
```

B KL1 clauses for 4-queens problem

The 4-queens problem given in section 6 is transformed to the following KL1 clauses where $unsafe(X1, Y1, X2, Y2)$ predicate is represented by the guard goals in the second `c/6` clause.

```
:- module mgtp. :- public do/1,c/6.

do(S) :- true | mgtp_i:do([],[{1,[[]]}],[],[],
    [[p(1,1),p(1,2),p(1,3),p(1,4)],[p(2,1),p(2,2),p(2,3),p(2,4)],
     [p(3,1),p(3,2),p(3,3),p(3,4)],[p(4,1),p(4,2),p(4,3),p(4,4)]],S).

c(1,I,Si,So,Di,Do):-I=[p(X1,Y1)]|So=[I|Si],Do=[I|Di].
c(1,I,Si,So,Di,Do):-I=[p(X2,Y2),p(X1,Y1)],
    X1=\=X2,0=:=(Y1-Y2)*(X1-X2-Y1+Y2)*(X1-X2+Y1-Y2)|
    So=[false|Si],Do=Di.
otherwise.
c(_,_,Si,So,Di,Do):-true|So=Si,Do=Di.
```