Equational Prover of THEOREMA*

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University
A-4040, Linz, Austria
kutsia@risc.uni-linz.ac.at

Abstract. The equational prover of the THEOREMA system is described. It is implemented on Mathematica and is designed for unit equalities in the first order or in the applicative higher order form. A (restricted) usage of sequence variables and Mathematica built-in functions is allowed.

1 Introduction

The THEOREMA[1] system [7] is an integrated environment for proving, solving and computing built on the top of Mathematica [32]. It is based on early papers by Buchberger (e.g. [5, 6]) and provides a front end for composing formal mathematical text consisting of a hierarchy of axioms, propositions, algorithms etc. in a common logic frame with user-extensible syntax, and a library of provers, solvers and simplifiers for proving, solving and simplifying mathematical formulae.

The equational prover of THEOREMA is one of such provers, designed for unit equality problems in the first order or in the applicative higher order form. The input may contain sequence variables. A (restricted) usage of Mathematica built-in functions is allowed. The prover has two proving modes: unfailing completion [3] and simplification (rewriting/narrowing). It consists of the preprocessor, the kernel and the proof presenter parts. The preprocessor checks the input syntax, sets option values, Skolemizes, chooses a proving mode, an ordering and passes the preprocessed input to the kernel. The kernel runs a proof procedure with the chosen settings and passes the output to the proof presenter, which structures it, deletes redundant steps, introduces lemmata and constructs the proof object.

2 Proof Procedure: Algorithm and Implementation

The unfailing completion procedure is implemented as a given-clause algorithm [23], where proof search is organized as a DISCOUNT [1] loop. The input of the procedure is a set of (implicitly) universally closed equalities $\mathcal{E}$, a ground goal $\mathcal{G}$ and a (ground total) reduction ordering $>$, which can be either the lexicographic path ordering (LPO), Knuth-Bendix ordering (KBO) or the lexicographic extension of the multiset path ordering with sequence variables (MPOSV [18]). Before

* Supported by the Austrian Science Foundation (FWF) under Project SFB F1302.

[1] `http://www.theorema.org/`.

calling the completion procedure, we Skolemize all equations in the given proving problem. If the equation in the hypothesis contains existentially quantified variables, we proceed in a standard way introducing a new function symbol eq and two new constants, $true$ and $false$. Then we add two new equations $eq(x,x) = true$ and $eq(s,t) = false$ to $\mathcal{E}$, where x is a variable and s and t are sides of the hypothesis, and $true = false$ becomes a goal.

The proving procedure saturates $\mathcal{E}$ and works on a set of active facts $\mathcal{A}$, participating in the inference, and on a set of passive facts $\mathcal{P}$, waiting to become members of $\mathcal{A}$. The completion loop is shown on Fig. 1. It is essentially the same as the loop WALDMEISTER implements (see [20]).

Algorithm 1. Completion Loop

```
Function ProveByCompletion(E, G, >)
 1: (A, P):=(∅, E)
 2: while ¬trivial(G)∧P ≠ ∅ do
 3:     e :=Select(P); P := P\{e}
 4:     if ¬orphan(e) then
 5:         e :=Normalize>_A(e)
 6:         if ¬redundant(e) then
 7:             (A, P1):=Interred>(A, e)
 8:             A := A ∪ {Orient>(e)}
 9:             P2 :=CP>(e, A)
10:             P := P∪Normalize>_A(P1 ∪ P2)
11:             G :=Normalize>_A(G)
12:         end
13:     end
14: end
15: return trivial(G)
```

Fig. 1. Main loop for proving by unfailing completion mode.

The predicate *trivial* on line 2 is true on an equality $s = t$ iff s and t are identical. The function *Select* on line 3 decides which equality should be selected from passive facts for activation. It has to guarantee that every passive fact eventually becomes active, thus ensuring the fairness of the procedure. This function selects a fact with the minimal weight. If there are several such facts, then it chooses the oldest one. Moreover, once in each five iterations *Select* takes the smallest fact where $false$ occurs, if there is such a fact in $\mathcal{P}$[2]. The predicate *orphan* on line 4 is true on e iff a parent equation of e has been reduced. The predicate *redundant* on line 6 is true on e iff either e is trivial, is subsumed by an equation in $\mathcal{A}$ or is ground joinable. The interreduction function *Interred* on line 7 takes active facts that are reducible by e out of $\mathcal{A}$ and puts them into $\mathcal{P}_1$. The *Orient* function on line 8 orients e with respect to $>$, if possible. The

[2] BARCELONA and FIESTA implement such a selection criterion [26].

function CP on line 9 generates all possible critical pairs between e and $\mathcal{A}$. The function $Normalize$ does normalization with the active facts only. Passive facts are normalized only on their generation and after selection.

We store $\mathcal{P}$ as a heap of heaps, following the WALDMEISTER approach. It allows efficient "orphan murder" and fast selection of the minimal equation. We use Mathematica arrays to implement the heap. Terms are kept as stringterms.

Normalization of the selected equation, new critical pairs or the goal, using active facts, is an example of forward simplification. Since the set of active facts grows larger and larger, fast identification of the appropriate fact for rewriting (generalization retrieval) becomes crucial. There are various indexing techniques suitable for this operation, like code trees [30], substitution trees [14], context trees [12] or various versions of discrimination trees [9, 22, 15]. However, in our case, instead of implementing an indexing technique, we decided to rely on Mathematica rewriting tools, as its programming language is a rewrite language [4][3]. This approach is similar to what Stickel proposed in [27], using the Prolog implementation technology for model elimination. We call it Mathematica Technology Term Rewriting (MTTR) and show on the example below how it works.

Let R and E be respectively the set of rules and the set of equations in $\mathcal{A}$. Assume we have a rule for the associativity law in R: $f(f(x,y),z) \to f(x,f(y,z))$. Every active and passive fact has its unique label, an integer associated with it. Let the label for the rule above be 5. Then we transform the rule into a Mathematica assignment as follows: First, we normalize the variable names and transform each variable in the left hand side of the rule into Mathematica patterns getting `f[f[x1_, x2_], x3_]`. Next, we make the Mathematica assignment:

```
f[f[x1_, x2_], x3_, _ : {5}] :=(AppendTo[$LABELS, {5}]; f[x1, f[x2, x3]])/;
        ($PHASE === "Rewriting")
```

where `$LABELS` is a global variable, initialized with the empty list every time before normalizing a term. After a term is normalized, `$LABELS` stores the list of labels of those active facts which participated in the normalization. In the condition, `$PHASE` is a global variable specifying the deduction phase. It prevents unexpected evaluations. In the example above, the assignment becomes applicable only at the "Rewriting" phase and not, for instance, at the "Subconnectedness checking" phase, where we only need reducibility. The entire main loop runs in the "Neutral" phase, switching to the specific phases when needed.

Transformation of equalities from E into delayed assignments is done in the similar manner, but we add the ordering check additionally. For instance, an equality $f(x,f(y,z)) = f(y,f(z,x))$ is transformed into two assignments:

```
f[x1_, f[x2_, x3_], _ : {"L", 6}] :=
   (AppendTo[$LABELS, {"LR", 6}]; f[x2, f[x3, x1]])/;
    ($PHASE ==" Rewriting"∧$GREATER[f[x1, f[x2, x3]], f[x2, f[x3, x1]]])
```

[3] We tried to implement discrimination trees, but since the low-level programming capabilities are very restricted in the high-level Mathematica programming language, which itself is not very fast, we did not get a reasonable performance.

```
f[x1_, f[x2_, x3_], _ : {"RL", 6}] :=
   (AppendTo[$LABELS, {"RL", 6}]; f[x3, f[x1, x2]])/;
   ($PHASE === "Rewriting" ∧ $GREATER[f[x1, f[x2, x3]], f[x3, f[x1, x2]]])
```

where `$GREATER` is a global variable whose value is the function specifying the given reduction ordering. We make sure that for equalities like the commutativity law, only one delayed assignment is made, instead of two. Note that MTTR approach treats constants as nullary function symbols.

When we need to rewrite a term t in a phase p, we simply call the function `rewrite` on t and p. `rewrite` is implemented as follows:

```
Clear[rewrite];
rewrite[term_, phase_] := Module[{ans},
   With[{y = $SIGNATURE}, $PHASE = phase; Map[Update, y]];
   ans = term; $PHASE = "Neutral"; ans];
```

where `$SIGNATURE` is a global variable whose value is a list of all constants and function symbols occurring in the problem.

MTTR is probably one of the fastest ways of doing rewriting in Mathematica, but it has disadvantages as well, namely, we have minimal control on rewriting strategies, can not keep track of results of single rewrite steps, and should put additional control to prevent unexpected evaluation.

We found useful to use Mathematica matching mechanism in forward subsumption as well, which is one of the redundancy criteria for the selected equation. Another redundancy criterion is the ground joinability test [2], implemented for associative-commutative symbols only: we add to the active facts a ground complete subset for an AC symbol f consisting of AC axioms and the additional equation $f(x, f(y, z)) = f(y, f(x, z))$. Any other equation joinable modulo AC can be deleted. Such an equation is called ground joinable. To test ground joinability we use the following trick: for each AC symbol `f` we create a new function symbol `nf`, make a list of transformation rules `$AC-SYMBOLS`$=\{$`f` $\rightarrow$ `nf`, $\ldots\}$ and set attributes of `nf` to {`Flat, Orderless, OneIdentity`}. Then testing whether `s=t` is ground joinable reduces to testing whether `s/.$AC-SYMBOLS` and `t/.$AC-SYMBOLS` are identical[4].

Mathematica delayed assignment rules are employed also in caching for term orderings. The orderings like LPO or MPOSV compare the same term pairs many times. To avoid a repeated work we store the result of comparison between two terms s and t as `cachedComparison[s,t]:=result`, where `result` is either `True` or `False` and look it up whenever s and t have to be compared again[5].

With interreduction and critical pair generation the situation becomes more complicated. Here we need to perform instance and unifiable retrieval on the set of active facts. Mathematica does not provide mechanisms which would make possible to implement these operations in the same spirit as we did for generalization retrieval. In order to perform instance retrieval in more or less reasonable way, we had to implement some kind of indexing for the terms in active facts. We chose path indexing [28], because it does not involve backtracking, insertion

[4] in Mathematica `/.` is a short notation for the function `ReplaceAll`.

[5] The idea of caching was implemented earlier in Dedam by RPO caching [26].

and deletion can be done more efficiently than for other indexing techniques, is economical in terms of memory usage, and is useful for retrieving instances (see [25]). One of the main disadvantages of path indexing is that it requires costly union and intersection operations to combine intermediate results. We use Mathematica built-in functions `Union` and `Intersection` for these operations.

The main loop for proving by simplification, unlike Algorithm 1, does not perform interreduction and orphan testing, and does not generate critical pairs unless at least one of the parent equations contains a term with the head *eq*. The simplification mode has one more specific feature: optionally, all equations can be oriented from left to right. In this case, of course, termination of rewriting is not guaranteed and the prover issues the corresponding warning.

3 Extensions

3.1 Sequence Variables

A sequence variable is a variable that can be instantiated with an arbitrary finite, possibly empty, sequence of terms. To distinguish, we call ordinary variables individual variables. Sequence variables are allowed to appear only as arguments of flexible arity symbols. The main difficulty in deduction with sequence variables is infinitary unification, even in the syntactic case ([19]). However, it was shown to be decidable in [17] and a theorem proving procedure with constraints à la Nieuwenhuis/Rubio [24] was proposed in [18].

The equational prover of THEOREMA implements unfailing completion with sequence variables, occurring only in the last argument positions in terms (as, e.g. in $f(a, f(x, \overline{x}), g(\overline{x}), \overline{y})$, where $\overline{x}$ and $\overline{y}$ are sequence variables). It makes unification unitary. A rule-based unification algorithm is shown on Fig. 2.

Algorithm 2. Unification with sequence variables in the last argument positions	
Function unify(s, t), s and t are not sequence variables	
1: unify$(t, t) := \{\}$	
2: unify$(x, t) := \{x \leftarrow t\}$	if $x \neq t$ and $x \notin vars(t)$
3: unify$(t, x) := \{x \leftarrow t\}$	if $x \neq t$ and $x \notin vars(t)$
4: unify$(f(s, \tilde{s}), f(t, \tilde{t})) :=$ compose$(\sigma,$ unify$(f'(\tilde{s})\sigma, f'(\tilde{t})\sigma))$	if $\sigma =$ unify(s, t) and $\sigma \neq$ `fail`
5: unify$(f(\overline{x}), f(\tilde{t})) := \{\overline{x} \leftarrow \tilde{t}\}$	if $\overline{x} \neq \tilde{t}$ and $\overline{x} \notin vars(\tilde{t})$
6: unify$(f(\tilde{t}), f(\overline{x})) := \{\overline{x} \leftarrow \tilde{t}\}$	if $\overline{x} \neq \tilde{t}$ and $\overline{x} \notin vars(\tilde{t})$
7: unify$(s, t) :=$ `fail`	otherwise

Fig. 2. Unification for terms with sequence variables in the last argument position

The input of the algorithm are two terms, which are not sequence variables themselves and all occurrences of sequence variables happen only in the last argument positions of subterms. The *compose* function on Line 3 returns `fail`,

if at least one of its arguments is `fail`, otherwise it composes substitutions in its first and second arguments. $\tilde{s}$ and $\tilde{t}$ denote arbitrary finite, possibly empty, sequences of terms. $vars(t)$ (resp. $vars(\tilde{t})$) is a set of all individual and sequence variables of a term t (resp. sequence of terms $\tilde{t}$). x is an individual variable, $\overline{x}$ is a sequence variable. The function symbol f' on line 4 is a new flexible arity symbol, if the symbol f on the same line has a fixed arity, otherwise f' and f are the same. The function symbol f on 5 and 6 has a flexible arity.

In the simplification mode we use sequence variables without any restrictions on their occurrence. It means that in this case unification is infinitary and matching is finitary. For the moment we do not allow existential goals in this setting, and therefore unification problems do not appear. As for matching/rewriting, MTTR follows the Mathematica strategy, choosing from the finite alternatives the matcher that assigns the shortest sequences of terms to the first sequence variables that appear in the pattern[6].

Unrestricted quantification over sequence variables takes the language beyond first-order expressiveness. In [17] we considered an extension of the language with constructs called pattern-terms, which abbreviate term sequences of unknown length matching certain "pattern", like, for instance, all the terms in the sequence having the same arguments, but different top function symbols. Such pattern-terms are naturally introduced via Skolemization. In the unfailing completion mode of the prover we allow pattern-terms to occur only in the last argument positions in terms whose top function symbol has a flexible arity. A pattern-term can be unified only with a sequence variable which does not occur in it, or with an identical pattern-term.

The MTTR technique has to be extended to terms with sequence variables and pattern-terms. First, each sequence variable should be transformed into the corresponding pattern (an identifier with three underscores). Second, since individual variables match neither sequence variables nor pattern-terms, we have to restrict Mathematica patterns that correspond to individual variables. Thus, a term $f(x, f(\overline{g(\overline{x})}), \overline{x})$, where $\overline{x}$ is a sequence variable, $\overline{g(\overline{x})}$ is a pattern-term and x is an individual variable, will be transformed into the pattern `f[x1_?¬MatchQ[#,.var[.seq[_]]] ∧ ¬MatchQ[#,.seq[_]]&,.seq[g[x2___]],x2___]`[7].

We also extended the path indexing technique to index terms with sequence variables and pattern-terms in the last argument positions. However, more effort has to be made here to improve efficiency of retrieval operations.

3.2 Problems in Applicative Higher Order Form

Warren introduced in [31] a method to translate expressions from higher order applicative form into first order syntax, preserving both declarative and narrow-

[6] Recently, [21] proposed an approach to gain more control on rewriting with sequence variables in Mathematica.

[7] `.var` and `.seq` are THEOREMA tags respectively for variables and for sequences. `.var[.seq[_]]` is a Mathematica pattern which can match any THEOREMA sequence variable and `.seq[_]` can match any THEOREMA pattern-term.

ing semantics [13]. With this translation, for example, the higher order equation $twice(F)(X) = F(F(X))$ is translated into the equations $twice(F, X) = apply(F, apply(F, X))$, $apply(twice_0, F) = twice_1(F)$, $apply(twice_1(F), X) = twice(F, X)$, where *apply* is a new binary function and $twice_0$ and $twice_1$ are new constructors representing partial applications. Since THEOREMA syntax allows higher order expressions, the equational prover can accept such an input. Then the input is translated by Warren's method into the first order form, on which the proving procedure is applied. The output is translated back into higher order form. Thus, the user sees only higher order input and output.

Optionally, if the proving mode is simplification and the goal is universal, MTTR can be applied immediately, without Warren's translation, because the Mathematica programming language supports rewriting with higher order constructs. In this case currently the equalities are oriented from left to right, but we intend to implement HORPO [16] as well.

At the current stage sequence variables and pattern-terms can be used in higher order problems only in the simplification mode with universal goals.

3.3 Using Mathematica Built-in Functions

We incorporate Mathematica built in functions in the proving task in the following way: On the one hand, to be interpreted as a Mathematica built-in function it is not enough for a function in the proving problem to have a syntax of a Mathematica function. It has to be stated explicitly that it is a built-in function (THEOREMA has a special construct `built-in` for that). Moreover, a function can get its built-in meaning only when it appears in the goal. After normalization, the goal is checked on joinability modulo built-in meaning of the Mathematica functions in it, but the built-ins are not used to derive new goals. On the other hand, the approach is not completely sceptical: after a built-in function is identified, it is trusted and the result of computation is not checked. Therefore, when Mathematica functions are involved in the task, in the prover output it is stated: "If the built-in computation is correct, the following theorem holds...". The integration tool is still at the experimental level and needs further development, e.g. integrating existing frameworks of combining computer algebra and theorem proving/rule based reasoning (with [8] as a particular example).

4 Proof Presentation

We use the Proof Communication Language (PCL) [11] to describe proofs. A slight modification is needed to represent reduction steps, because MTTR does not show intermediate rewriting results. Proofs are structured into lemmata. Proofs of universally closed theorems are displayed as equational chains, while those of existential theorems represent sequences of equations. The symbols *eq*, *true* and *false* are not shown. In failing proofs, on the one hand, the theorems which have been proved during completion are given, and on the other hand, failed propositions whose proving would lead to proving the original goal are

displayed, if there are any. They are obtained from descendants of the goal and certain combinations of their sides.

5 Examples

Example 1 (Insertion sort). This is an example of using sequence variables and Mathematica functions. The assumptions specify the insertion sort:

$\texttt{Assumption}[\text{“}\texttt{1}\text{”}, \underset{\texttt{n}}{\forall}(\texttt{insert}[\texttt{n}, \langle\rangle] = \langle \texttt{n} \rangle)]$

$\texttt{Assumption}[\text{“}\texttt{2}\text{”}, \underset{\texttt{n,m,}\overline{\texttt{x}}}{\forall}(\texttt{insert}[\texttt{n}, \langle \texttt{m}, \overline{\texttt{x}} \rangle] = \texttt{prepend}[\texttt{max}[\texttt{m}, \texttt{n}], \texttt{insert}[\texttt{min}[\texttt{n}, \texttt{m}], \langle \overline{\texttt{x}} \rangle]])]$

$\texttt{Assumption}[\text{“}\texttt{3}\text{”}, \texttt{sort}[\langle\rangle] = \langle\rangle]$

$\texttt{Assumption}[\text{“}\texttt{4}\text{”}, \underset{\texttt{x,}\overline{\texttt{y}}}{\forall}(\texttt{sort}[\langle \texttt{x}, \overline{\texttt{y}} \rangle] = \texttt{insert}[\texttt{x}, \texttt{sort}[\langle \overline{\texttt{y}} \rangle]])]$

$\texttt{Assumption}[\text{“}\texttt{5}\text{”}, \underset{\texttt{x,}\overline{\texttt{y}}}{\forall}(\texttt{prepend}[\texttt{x}, \langle \overline{\texttt{y}} \rangle] = \langle \texttt{x}, \overline{\texttt{y}} \rangle)]$

where `min` and `max` are interpreted as the Mathematica `Min` and `Max` functions:

```
Built-in[“MinMax″,
  min → Min  “Minimum″
  max → Max  “Maximum″]
```

We would like to prove the following proposition:

$\texttt{Proposition}[\text{“}\texttt{sort}\text{”}, \underset{\overline{\texttt{x}}}{\exists}(\texttt{sort}[\langle 1, 3, 2.4, -4 \rangle] = \langle \overline{\texttt{x}} \rangle)]$

The equational prover is called to prove the proposition under the given assumptions and built-ins. In addition, numbers are treated in the Mathematica built-in way and all the equations are oriented from left to right:

```
Prove[Proposition[“sort″], using → {Assumption[“1″], Assumption[“2″],
  Assumption[“3″], Assumption[“4″], Assumption[“5″]},
  by → EquationalProver,
  built-in→{Built-in[“MinMax″], Built-in[“Numbers″]},
  ProverOptions → {EqPrOrdering → “LeftToRight″}]
```

The output of the prove call is placed in a new notebook. THEOREMA displays it in an elegant way, with natural language text, hierarchically nested cells, hyperlinks, colors, etc. We show the (final part of the) generated proof below [8]:

...

To prove (Proposition (sort)), we have to find $\overline{x}^*$ such that

(1) $\quad sort[\langle 1, 3, 2.4, -4 \rangle] = \langle \overline{x}^* \rangle.$

We will use the following assumptions, referring to them as axioms:

(Axiom 1) $\quad sort[\langle\rangle] = \langle\rangle.$

[8] Koji Nakagawa provided a tool to translate the proofs from the THEOREMA proof format into LaTeX form.

(Axiom 2) $\underset{x1}{\forall}\,(insert[x1, \langle\rangle] = \langle x1\rangle)$.

(Axiom 3) $\underset{x1,\overline{x2}}{\forall}(prepend[x1, \langle\overline{x2}\rangle] = \langle x1, \overline{x2}\rangle)$.

(Axiom 4) $\underset{x1,\overline{x2}}{\forall}(sort[\langle x1, \overline{x2}\rangle] = insert[x1, sort[\langle\overline{x2}\rangle]])$.

(Axiom 5) $\underset{x1,x2,\overline{x3}}{\forall}(insert[x1, \langle x2, \overline{x3}\rangle] =$
$prepend[max[x2, x1], insert[min[x1, x2], \langle\overline{x3}\rangle]])$.

We choose

$\overline{x}^* = Sequence[3, 2.4, 1, -4]$

and show that the equality (1) holds for this value (assuming that the built-in simplification/decomposition is sound):

(Theorem) $sort[\langle 1, 3, 2.4, -4\rangle] = \langle 3, 2.4, 1, -4\rangle$.

Proof.

$sort[\langle 1, 3, 2.4, -4\rangle] = \langle 3, 2.4, 1, -4\rangle$

if and only if (by (Axiom 4) LR, (Axiom 4) LR, (Axiom 4) LR, (Axiom 4) LR, (Axiom 1) LR, (Axiom 2) LR)

$insert[1, insert[3, insert[2.4, \langle -4\rangle]]] = \langle 3, 2.4, 1, -4\rangle$

if and only if (by (Axiom 5) LR, (Axiom 2) LR, (Axiom 3) LR)

$insert[1, insert[3, \langle max[-4, 2.4], min[2.4, -4]\rangle = \langle 3, 2.4, 1, -4\rangle$

if and only if (by (Axiom 5) LR, (Axiom 5) LR, (Axiom 2) LR, (Axiom 3) LR, (Axiom 3) LR, (Axiom 5) LR, (Axiom 5) LR, (Axiom 5) LR, (Axiom 2) LR, (Axiom 3) LR, (Axiom 3) LR, (Axiom 3) LR)

$\langle max[max[max[-4, 2.4], 3], 1], max[max[min[2.4, -4],$
$min[3, max[-4, 2.4]]], min[1, max[max[-4, 2.4], 3]]],$
$max[min[min[3, max[-4, 2.4]], min[2.4, -4]],$
$min[min[1, max[max[-4, 2.4], 3]], max[min[2.4, -4],$
$min[3, max[-4, 2.4]]]]], min[min[min[1, max[max[-4, 2.4], 3]],$
$max[min[2.4, -4], min[3, max[-4, 2.4]]]], min[min[3, max[-4, 2.4]],$
$min[2.4, -4]]]\rangle = \langle 3, 2.4, 1, -4\rangle$

if and only if (by the built-in simplification/decomposition)

$\langle 3, 2.4, 1, -4\rangle = \langle 3, 2.4, 1, -4\rangle$

which, by reflexivity of equality, concludes the proof. □

Example 2 (Combinatory logic). This is an example of a problem in applicative higher order form. The strong fixpoint property holds for the set consisting of the combinators B and W only[9]. The prover uses Warren's method to translate the problem into first order form, proves it with unfailing completion procedure and shows the output again in the higher order form. The proof is given below:

Prove:

(Proposition (goal)) $\mathit{strong\text{-}fixed\text{-}point}[\mathit{fixed\text{-}pt}] = \mathit{fixed\text{-}pt}[\mathit{strong\text{-}fixed\text{-}point}[\mathit{fixed\text{-}pt}]]$

...

We will use the following assumptions, referring to them as axioms:

(Axiom 1) $\underset{x,y}{\forall}\ (x[y][y] = W[x][y])$.

(Axiom 2) $\underset{x,y,z}{\forall}\ (B[x][y][z] = x[y[z]])$.

(Axiom 3) $B[W[W]][B[W][B[B][B]]] = \mathit{strong\text{-}fixed\text{-}point}$.

We need the following propositions:

(Lemma 1) $\underset{x1}{\forall}\ (W[W][W[B[B[x1]]]] = \mathit{strong\text{-}fixed\text{-}point}[x1])$

Proof. We take all variables arbitrary but fixed.

$W[W][W[B[B[x_1]]]] =$	(by (Axiom 2) RL,(Axiom 2) RL)
$W[W][B[W][B[B][B]][x_1]] =$	(by (Axiom 2) RL)
$B[W[W]][B[W][B[B][B]]][x_1] =$	(by (Axiom 3) LR)
$\mathit{strong\text{-}fixed\text{-}point}[x_1]$.	□

(Lemma 2) $\underset{x1,x2}{\forall}\ (x1[W[x2][x2]] = W[W[B[B[x1]]]][x2])$

Proof. We take all variables arbitrary but fixed.

$x_1[W[x_2][x_2]] =$	(by (Axiom 1) RL)
$x_1[x_2[x_2][x_2]] =$	(by (Axiom 2) RL)
$B[B[x_1]][x_2][x_2][x_2] =$	(by (Axiom 1) LR)
$W[W[B[B[x_1]]]][x_2]$.	□

[9] In TPTP this problem is stated in the first order form in COL003-12.pr.

(Lemma 3) $W[W][W[B[B[fixed\text{-}pt]]]] = fixed\text{-}pt\,[W[W][W[B[B[fixed\text{-}pt]]]]]$

Proof.

$W[W][W[B[B[fixed\text{-}pt]]]] =$ (by (Axiom 1) RL)

$W[W[B[B[fixed\text{-}pt]]]][W[B[B[fixed\text{-}pt]]]] =$ (by (Lemma 2) RL)

$fixed\text{-}pt[W[W[B[B[fixed\text{-}pt]]]][W[B[B[fixed\text{-}pt]]]]] =$ (by (Axiom 1) LR)

$fixed\text{-}pt[W[W][W[B[B[fixed\text{-}pt]]]]]$. □

(Proposition 1) $fixed\text{-}pt[strong\text{-}fixed\text{-}point[fixed\text{-}pt]] = strong\text{-}fixed\text{-}point[fixed\text{-}pt]$.

Proof.

$fixed\text{-}pt[strong\text{-}fixed\text{-}point[fixed\text{-}pt]] =$ (by (Lemma 1) RL)

$fixed\text{-}pt[W[W][W[B[B[fixed\text{-}pt]]]]] =$ (by (Lemma 3) RL)

$W[W][W[B[B[fixed\text{-}pt]]]] =$ (by Lemma 1) LR)

$strong\text{-}fixed\text{-}point[fixed\text{-}pt]$. □

Now, we prove (Proposition (goal)).

(Theorem) $strong\text{-}fixed\text{-}point[fixed\text{-}pt] = fixed\text{-}pt[strong\text{-}fixed\text{-}point[fixed\text{-}pt]]$

Proof.

$strong\text{-}fixed\text{-}point[fixed\text{-}pt] =$ (by (Proposition 1) RL)

$fixed\text{-}pt[strong\text{-}fixed\text{-}point[fixed\text{-}pt]]$. □

6 Performance and Future Development

From 431 unit equality problems in TPTPv2.4.0 [29] the prover solved 180 (42%) within 300 seconds on a Linux PC, Intel Pentium 4, 1.5GHz, 128Mb RAM[10]. The performance is lower than the one of, for instance, WALDMEISTER (85%), DISCOUNT (70%), FIESTA (68%) or CIME [10] (53%), but it should be taken into account that Mathematica is fundamentally an interpreter, the prover does not have many heuristics and is not tuned for any specific class of problems. It is still at the experimental level, and there is a room to improve in many parts

[10] This does not include problems proved with some user interaction (e.g., choosing appropriate precedence, ordering, age-weight ratio, etc.), but only those ones proved in the autonomous mode.

of it. Especially, the autonomous mode can be strengthened with a structure detection facility, and instance and unification retrievals can be done more efficiently. Proving with constraints involving sequence variables might be another interesting future development.

Strong sides of the prover are its abilities to handle sequence variables and problems in applicative higher form, and to interface Mathematica functions. THEOREMA provides yet another advantage – a convenient, user-friendly interface, and human-oriented proof presentation tools.

7 Acknowledgments

My thanks go to Prof. Bruno Buchberger and all the THEOREMA group members.

References

1. J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, editor, *Proceedings of the 6th RTA*, volume 914 of *LNCS*, pages 397–402, Kaiserslautern, Germany, 1995. Springer.
2. J. Avenhaus, Th. Hillenbrand, and B. L¨ochner. On using ground joinable equations in equational theorem proving. *J. Symbolic Computation*, 2002. To appear.
3. L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. A¨ıt-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Elsevier Science, 1989.
4. B. Buchberger. Mathematica as a rewrite language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings of the 2nd Fuji Int. Workshop on Functional and Logic Programming*, pages 1–13, Shonan Village Center, Japan, 1–4 November 1996. World Scientific.
5. B. Buchberger. Symbolic computation: Computer algebra and logic. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, Applied Logic Series, pages 193–220. Kluwer Academic Publishers, 1996.
6. B. Buchberger. Using Mathematica for doing simple mathematical proofs (invited paper). In *Proceedings of the 4th Mathematica Users' Conference*, pages 80–96, Tokyo, Japan, 2 November 1996. Wolfram Media Publishing.
7. B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema project: A progress report. In M. Kerber and M. Kohlhase, editors, *Proceedings of Calculemus'2000 Conference*, pages 98–113, St. Andrews, UK, 6–7 August 2000.
8. R. Bündgen. Combining computer algebra and rule based reasoning. In J. Calmet and J. A. Campbell, editors, *Integrating Symbolic Mathematical Computation and Artificial Intelligence. Proceedings of AISMC-2*, volume 958 of *LNCS*, pages 209–223, Cambridge, UK, 3–5 August 1994. Springer.
9. J. Christian. Flatterms, discrimination trees, and fast term rewriting. *J. Automated Reasoning*, 10(1):95–113, 1993.
10. E. Contejean, C. Marche, B. Monate, and X. Urbain. CiME version 2, 2000. `http://cime.lri.fr/`.
11. J. Denzinger and S. Schulz. Analysis and representation of equational proofs generated by a distributed completion based proof system. SEKI-report SR-94-05, University of Kaiserslautern, Germany, 1994.

12. H. Ganzinger, R. Nieuwenhuis, and P. Nivela. Context trees. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. Proceedings of the IJCAR'01*, volume 2083 of *LNAI*, pages 242–256, Siena, Italy, June 2001. Springer.
13. J. C. González-Moreno. A correctness proof for Warren's HO into FO translation. In D. Saccà, editor, *Proc. of the 8th Italian Conference on Logic Programming (GULP'93)*, pages 569–585, Gizzeria Lido, Italy, June 1993. Mediterranean Press.
14. P. Graf. Substitutin tree indexing. In J. Hsiang, editor, *Proceedings of the 6th RTA*, volume 914 of *LNCS*, pages 117–131, Kaiserslautern, Germany, 1995. Springer.
15. T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. WALDMEISTER – high-performance equational deduction. *J. Automated Reasoning*, 18(2):265–270, 1997.
16. J.-P. Jouannaud and A. Rubio. The higher order recursive path ordering. In *Proceedings of the 14th annual IEEE symposium LICS*, Trento, Italy, 1999.
17. T. Kutsia. Solving and proving in equational theories with sequence variables and flexible arity symbols. Technical Report 02-09, PhD Thesis. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2002.
18. T. Kutsia. Theorem proving with sequence variables and flexible arity symbols. In M. Baaz and A. Voronkov, editors, *Logic in Programming, Artificial Intelligence and Reasoning. International Conference LPAR'02*, volume 2514 of *LNAI*, pages 278–291, Tbilisi, Georgia, 2002. Springer.
19. T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Proceedings of Joint AISC'2002 – Calculemus'2002 conference*, volume 2385 of *LNAI*, Marseille, France, 1–5 July 2002. Springer.
20. B. L¨ochner and Th. Hillenbrand. A phytography of WALDMEISTER. *AI Communications*, 15(2,3):127–133, 2002.
21. M. Marin. Introducing Sequentica, 2002. `http://www.score.is.tsukuba.ac.jp/~mmarin/Sequentica/`.
22. W. W. McCune. Experiments with discrimination-tree indexing and path-indexing for term retrieval. *J. Automated Reasoning*, 9(2):147–167, 1992.
23. W. W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, US, January 1994.
24. R. Nieuwenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *J. Symbolic Computation*, 19:321–351, 1995.
25. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1853–1964. Elsevier Science, 2001.
26. J. M. Rivero. Data structures and algorithms for automated deduction with equality. PhD Thesis. Universitat Politècnica de Catalunya, Barcelona, Spain, 2000.
27. M. Stickel. A Prolog Technology Theorem Prover: implementation by an extended Prolog compiler. *J. Automated Reasoning*, 4:353–380, 1988.
28. M. Stickel. The path indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.
29. G. Sutcliffe and C. Suttner. The TPTP Problem Library for Automated Theorem Proving. `http://www.cs.miami.edu/~tptp/`.
30. A. Voronkov. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *J. Automated Reasoning*, 15(2):237–265, 1995.
31. D. H. D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence*, volume 10, pages 441–454. Edinburgh University Press, Edinburgh, UK, 1982.
32. S. Wolfram. *The Mathematica Book*. Cambridge University Press and Wolfram Research, Inc., fourth edition, 1999.