

The IJCAR 2004 Workshop on Empirically Successful First Order Reasoning

Geoff Sutcliffe¹

University of Miami

Stephan Schulz²

Technische Universität München and IRC/irstr

Tanel Tammet³

Tallinn Technical University

Abstract

Key words: Empirically Successful First Order Reasoning

The IJCAR 2004 workshop on Empirically Successful First Order Reasoning brings together practitioners and researchers who are concerned with the implementation and deployment of working automated reasoning systems for first order logic. The focus is on first order logic because it has adequate expressive power for many applications, has well understood and manageable computational properties. The automated reasoning community has much experience with the implementation and application of automated reasoning systems for the logic. As a result there exists, and is potential for further, empirically successful first order reasoning. The workshop discusses “really running” systems, and not theoretical ideas that have not yet been translated into working systems.

The workshop has two main topic areas:

Systems

- Implementation techniques and comparisons

¹ Email: geoff@cs.miami.edu

² Email: schulz@informatik.tu-muenchen.de

³ Email: tammet@cc.ttu.ee

- Data structures and algorithms for the efficient representation of terms, formulae, search states, etc., e.g. new indexing techniques, efficient implementation of simplification orderings, etc.
- Higher level data structures and formats for the representation of proof tasks and derivations, proof and lemma storage, etc.
- Implemented and evaluated heuristics

Applications

- Descriptions of automated reasoning solutions in application domains
- Experiences with practical applications
- Encoding of domain problems into logic, and decoding of logic solutions into the domain
- Special automated reasoning techniques for applications
- User interfaces (to whole systems, not just the automated reasoning component)
- System integration

Additionally, the workshop includes **system and application demonstrations**.

Participants come from several sources:

- Researchers who have developed and implemented successful automated reasoning techniques and systems.
- Practitioners who have deployed automated reasoning systems in working applications.
- Users who have already attempted to apply automated reasoning in their domains, and now wish to learn more.
- Potential users who are interested in learning how automated reasoning may be used in their domains.

The concrete application areas include the fields of verification, deductive databases, mathematics, knowledge representation, semantic web, etc. The workshop provides a forum for discussion of the techniques necessary to take automated reasoning from the lab and into the “real world”. The workshop enables the attendees to learn from each others’ practical experiences, and will document their state-of-the-art techniques.

Darwin: A Theorem Prover for the Model Evolution Calculus

Peter Baumgartner¹

*Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany*

Alexander Fuchs²

*Universität Koblenz-Landau
Fachbereich Informatik
Universitätsstraße 1
D 56070 Koblenz, Germany*

Cesare Tinelli³

*Department of Computer Science
The University of Iowa
14 MacLean Hall
Iowa City, IA 52242, USA*

Abstract

Darwin is the first implementation of the Model Evolution Calculus by Baumgartner and Tinelli. The Model Evolution Calculus lifts the DPLL procedure to first-order logic. *Darwin* is meant to be a fast and clean implementation of the calculus, showing its effectiveness and providing a base for further improvements and extensions.

Based on a brief summary of the Model Evolution Calculus, we describe in the main part of the paper *Darwin*'s proof procedure and its data structures and algorithms, discussing the main design decisions and features that influence *Darwin*'s performance. We also report on practical experiments carried out with problems from the CADE-18 and CADE-19 system competitions, as well as on results on parts of the TPTP Problem Library.

Key words: Automated Theorem Proving,
Davis-Putnam-Logemann-Loveland procedure

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

1 Introduction

In propositional satisfiability the DPLL procedure [5,4], is the most popular and successful method for building complete SAT solvers. Its success is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics and constraint propagation techniques for reducing the search space. Thanks to these heuristics and to very careful engineering, the best SAT solvers today can successfully attack real-world problems with hundreds of thousands of variables and clauses.

Although the DPLL method is usually described procedurally, its essence can be captured declaratively by means of a sequent-style calculus [16]. The DPLL calculus has been recently lifted to the first-order level in [2]. The result is a sound and complete calculus, called the Model Evolution calculus, or \mathcal{ME} calculus for short, for the unsatisfiability of first-order clauses (without equality)⁴.

One of the main motivations for developing the Model Evolution calculus was the possibility of migrating to the first-order level some of those very effective search techniques developed by the SAT community for the DPLL procedure. This paper describes *Darwin*, a first implementation of the calculus designed to incorporate these techniques—or better, their first-order equivalents. The current version of *Darwin* implements a first-order version of unit propagation [18], a form of simplification, and backjumping, a form of intelligent backtracking (which seems to have been used for the first time for a first-order theorem prover in [12]). The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned for the next version.

Although *Darwin* is still at a prototype stage, it borrows many advanced techniques from the first-order theorem proving world—such as term indexing, subterm sharing, redundancy elimination, and so on. The overall rationale for developing this system was to get an initial sense of the performance potential of the \mathcal{ME} calculus, to constitute a robust code base for further improvements on the implementation, and for future extensions of the calculus.

This paper provides a fairly high level description of *Darwin*'s architecture and implementation, usually providing more details only on those implementation aspects that are specific to the \mathcal{ME} calculus—as opposed to a first-order calculus in general.

¹ Email: baumgart@mpi-sb.mpg.de

² Email: alexf@uni-koblenz.de

³ Email: tinelli@cs.uiowa.edu

⁴ The \mathcal{ME} calculus extends and significantly improves on the FDPLL calculus [1], which was the first successful attempt to lift the DPLL calculus to the first-order level.

2 The Model Evolution Calculus

We start by providing a summary description of the Model Evolution calculus and its main features, concentrating on those aspects that are relevant to the understanding of the implementation. More details on the calculus can be found in [2,3].

The DPLL procedure can be described as one that attempts to find a model of a given formula, input as a set of clauses, by starting with a default interpretation in which all input atoms are false and incrementally modifying it until it becomes a model of the input formula, or all alternative modifications have been considered with no success. The \mathcal{ME} calculus can be seen as lifting this “model evolution” process to the first-order level.

The goal of the calculus is to construct a Herbrand model of a given set Φ of clauses, if any such model exists. To do that, during a derivation the calculus maintains a *context* Λ , a finite set of (possibly non-ground) literals. The context Λ is a finite—and compact—representation of a Herbrand interpretation I_Λ , serving as a candidate model for Φ . The denoted interpretation I_Λ might not be a model of Φ because it does not satisfy some instances of clauses in Φ . The purpose of the main rules of the calculus is to detect this situation and either *repair* I_Λ , by modifying Λ so that it becomes a model of Φ , or recognize that I_Λ is unrepairable and fail. In addition to these rules, the calculus contains a number of simplification rules whose purpose is, like in the DPLL procedure, to simplify the clause set and, as a consequence, to speed up the computation.

The rules of the calculus manipulate sequents of the form $\Lambda \vdash \Phi$, where Λ is the current context and Φ is the current clause set. The initial sequent is made of a context standing for an initial interpretation and of the input clause set.

To describe the rules we need to introduce a few technical preliminaries first.

2.1 Technical Preliminaries

Contexts are finite sets of possibly non-ground literals built over terms as usual, however over two types of variables: *universal* variables—or simply *variables*—drawn from an infinite set X and denoted here by x, y, z , and *parametric* variables—or simply, *parameters*—drawn from an infinite set V disjoint with X and denoted here by u, v, w . Context literals are either *universal*, that is parameter-free, or *parametric*, that is, variable-free. By contrast, clause literals, that is, literals occurring in the clause set Φ of a sequent, are all parameter-free. For all purposes, the literals of a context can be considered variable and parameter disjoint with each other—in tableaux terms, neither parameters nor variables are rigid.

Each context can be seen as the finite specification of a certain Herbrand

interpretation. Roughly speaking, within a context both universal and parametric literals stand for their ground instances. The main difference is that universal literals always stand for all of their ground instances, whereas parametric literals may stand only for a subset of all of their ground instances. The precise way in which context literals denote ground instances and how that is used to associate a Herbrand model to a context is defined in [2,3]. Here we will limit ourselves to introduce a few notions that involve parameters and are needed to describe the rules of the calculus.

Let us consider the set of substitutions defined over the set $X \cup V$. We say a substitution is *parameter-preserving*, or *p-preserving* for short, if its restriction to the set V of parameters is a renaming over V in the standard sense—i.e., it is a permutation of V . A substitution is a *p-renaming* if it is a p-preserving renaming.

We say a term s is a *p-preserving variant* of a term t , or *p-variant* for short, if there is a p-renaming ρ such that $s\rho = t$. We say that s is *p-preserving more general than* t , iff there is a p-preserving substitution σ such that $s\sigma = t$. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t 's variables and by $\mathcal{P}ar(t)$ the set of t 's parameters. These definitions stated for terms also apply to literals and clauses in the obvious way.

We assume an infinite supply of *Skolem constants* disjoint with the set of constants occurring in any given input clause set. We write L^{sko} to denote the result of applying some substitution to the literal L that replaces each variable in L by a fresh Skolem constant. We write \bar{L} to denote the complement of L .

A literal L is *contradictory with* a context Λ iff there is a p-variant K of some literal in Λ and a p-preserving substitution σ such that $L\sigma = \bar{K}\sigma$.

Definition 2.1 (Context Unifier) Let Λ be a context and

$$C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_n$$

a parameter-free clause, where $0 \leq m \leq n$. A substitution σ is a *context unifier of C against Λ with remainder* $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ iff there are fresh p-preserving variants K_1, \dots, K_n of context literals such that

- (i) σ is a most general simultaneous unifier of $\{K_1, \bar{L}_1\}, \dots, \{K_n, \bar{L}_n\}$,
- (ii) for all $i = 1, \dots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,
- (iii) for all $i = m + 1, \dots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$.

A context unifier σ of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ is *admissible* iff for all distinct $i, j = m + 1, \dots, n$, $L_i\sigma$ is parameter- or variable-free and $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.

The existence of an admissible context unifier Λ between a context and a clause indicates that the interpretation I_Λ denoted by Λ falsifies the clause.⁵

⁵ Strictly speaking, this is true if the context unifier is also productive (see [2]). But the

The rules of the \mathcal{ME} calculus use context unifiers as a way to discover that the interpretation associated with the current context falsifies one of the current clauses, and decide how to “repair” the context.

Context unifiers are at the core of the \mathcal{ME} calculus because they are used by all of its non-optional derivation rules. In fact, context unification is the computational bottleneck of our current implementation as most of Darwin’s run time is spent on computing context unifiers. Darwin’s algorithm and data structure to compute context unifiers are described in Section 4.6 below.

2.2 The Derivation Rules

The derivation rules of the calculus are described below. We follow the version of rules given in [3] as those described in [2] are a somewhat simplified but less powerful version. Except for **Compact**, which is a simplification rule that applies only to contexts with variables/parameters, all the other rules are direct first-order liftings of the rules of the DPLL calculus, and reduce to those rules when the input clause set is ground.

$$\text{Split} \quad \frac{\Lambda \vdash \Phi, C \vee L}{\Lambda, L\sigma \vdash \Phi, C \vee L \quad \Lambda, (\overline{L\sigma})^{\text{sko}} \vdash \Phi, C \vee L} \quad \text{if } (*)$$

$$\text{where } (*) = \begin{cases} C \neq \square, \\ \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \\ \text{with remainder literal } L\sigma, \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\text{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

Split is the only non-deterministic rule of the calculus. As mentioned earlier, the existence of an admissible context unifier σ of $C \vee L$ against Λ indicates that I_Λ falsifies $(C \vee L)\sigma$. The left conclusion of the rule tries to fix this problem by adding to the context a literal $L\sigma$ from σ ’s remainder. The alternative right conclusion—needed for soundness in case the repair on the left turns out to be unsuccessful—adds instead the skolemized complement of $L\sigma$, i.e. the results of replacing all universal variables of $L\sigma$, if any, by fresh Skolem constants. The addition of $(\overline{L\sigma})^{\text{sko}}$ prevents later splittings on L but leaves the possibility of repairing the context by adding another of σ ’s remainder literals. When the rule is applicable, we call $L\sigma$ a *split literal*.

difference can be ignored here.

$$\text{Assert} \quad \frac{\Lambda \quad \vdash \Phi, C \vee L}{\Lambda, L\sigma \vdash \Phi, C \vee L} \quad \text{if} \quad \left\{ \begin{array}{l} \sigma \text{ is a context unifier of } C \text{ against} \\ \Lambda \text{ with an empty remainder,} \\ L\sigma \text{ is universal and} \\ \text{non-contradictory with } \Lambda, \\ \text{there is no } K \in \Lambda \text{ s. t. } K \text{ is} \\ \text{p-preserving more general than } L\sigma \end{array} \right.$$

When **Assert** applies, the only way to find a model for the clause set based on the current context or any extension of it is to satisfy every ground instance of $L\sigma$. The addition of $L\sigma$ makes sure that this is the case. Applications of **Assert** are highly desirable in practice because i) they strongly constrain further changes to the context, thereby limiting the non-determinism caused by the **Split** rule, and ii) they cause more applications of the three simplification rules below. When the rule is applicable, we call $L\sigma$ an *assert literal*.

$$\text{Subsume} \quad \frac{\Lambda, K \vdash \Phi, L \vee C}{\Lambda, K \vdash \Phi} \quad \text{if } K \text{ is p-preserving more general than } L.$$

Subsume removes clauses that are “permanently satisfied” by the context, that is, satisfied by the interpretation denoted by the current context or any context that extends the current one. **Subsume** is not needed for completeness but might improve the performance of an implementation.

$$\text{Resolve} \quad \frac{\Lambda \vdash \Phi, L \vee C}{\Lambda \vdash \Phi, C} \quad \text{if} \quad \left\{ \begin{array}{l} \text{there is a context unifier } \sigma \text{ of } L \\ \text{against } \Lambda \text{ with an empty remainder} \\ \text{such that } C\sigma = C \end{array} \right.$$

Resolve simplifies the clause set by removing literals from clauses. Like **Subsume** it is not needed for completeness. **Resolve** is the only rule of the calculus that is not implemented in its full generality in *Darwin*. In the current implementation **Resolve** is only applied for the special case in which there is a K in Λ s.t. $\neg K$ is p-preserving more general than L .

$$\text{Compact} \quad \frac{\Lambda, K, L \vdash \Phi}{\Lambda, K \quad \vdash \Phi} \quad \text{if } K \text{ is p-preserving more general than } L$$

Compact simplifies the context by removing literals which are instances of other literals.⁶ **Compact** is another optimization rule.

$$\text{Close} \quad \frac{\Lambda \vdash \Phi, C}{\Lambda \vdash \square} \quad \text{if} \quad \left\{ \begin{array}{l} \Phi \neq \emptyset \text{ or } C \neq \square, \\ \text{there is a context unifier } \sigma \text{ of } C \text{ against } \Lambda \\ \text{with an empty remainder} \end{array} \right.$$

⁶ The literals K and L are meant to be distinct.

Close detects a context which falsifies the clause set and cannot be modified in order to satisfy it. When the rule is applicable, we call σ a *closing context unifier*.

2.3 Derivation Tree

Definition 2.2 (Derivation Tree) A *derivation tree* is a labeled tree inductively defined as follows:

- (i) a one-node tree is a derivation tree iff its root is labeled with a sequent of the form $\Lambda \vdash \Phi$, where Λ is a context and Φ is a clause set;
- (ii) A tree \mathbf{T}' is a derivation tree iff it is obtained from a derivation tree \mathbf{T} by adding to a leaf node N in \mathbf{T} new children nodes N_1, \dots, N_m so that the sequents labeling N_1, \dots, N_m can be derived by applying a rule of the calculus to the sequent labeling N . In this case, we say that \mathbf{T}' is *derived from* \mathbf{T} .

Split as the only non-deterministic rule introduces two children nodes, every other rule only one child node.

Definition 2.3 (Open, Closed) A branch in a derivation tree is *closed* if its leaf is labeled by a sequent of the form $\Lambda \vdash \square$; otherwise, the branch is *open*. A derivation tree is *closed* if each of its branches is closed, and it is *open* otherwise.

Definition 2.4 (Derivation) A *derivation* is a possibly infinite sequence of derivation trees $(\mathbf{T}_i)_{i < \kappa}$, such that for all i with $0 < i < \kappa$, \mathbf{T}_i is derived from \mathbf{T}_{i-1} .

For a given input clause set Φ , derivations are started with the sequent $\neg v \vdash \Phi$ in the root node. Here, the pseudo-literal $\neg v$ causes the interpretation denoted by the context to falsify every atom by default.

A derivation ending with a closed derivation tree is a proof of the unsatisfiability of Φ . An exhausted branch, i.e. a branch to whose leaf no derivation rules apply, is a proof for the satisfiability of Φ , its context denotes a model for the clause set.

An important aspect to guarantee refutational completeness is to equip the calculus with a suitable notion of *fairness*. We will not describe it here and refer to [2,3] instead. We note, however, that it enables proof procedures emphasizing *don't-care* nondeterminisms. The sole form of *don't-know* nondeterminism is caused by the branching nature of the **Split** inference rule.

3 The Proof Procedure

The proof procedure implemented in *Darwin* follows the main loop described below. Similarly to the DPLL procedure, Darwin's procedure basically corresponds to a depth-first, or more precisely an iterative-deepening, exploration

of a derivation tree of the calculus. At any moment, the procedure stores in its data structures a single branch of the tree, where split nodes correspond to decision points. The procedure grows a branch until

- the branch can be closed, in which case it backtracks to a previous choice point and regrows the branch in the alternative direction, or
- the branch cannot be grown further, which means that a model of the input set has been found, or
- a depth limit is reached, in which case the procedure restarts from the beginning, but with an increased depth limit.

At any moment, in addition to the current context and the set of current clauses, the procedure maintains a set of *candidate literals*, literals that could be added to the context as a consequence of the application of the **Assert** or **Split** rule. Before entering the main loop, the candidate set is initialized with all the literals that could be added to the initial context by an application of **Assert**, which are just the unit clauses from the given clause set.

The main loop of *Darwin's* proof procedure consists of the following steps:

(i) Candidate Selection

If the candidate set is empty the problem is proven satisfiable and the procedure ends returning the current context, which denotes a model of the input clause set. Otherwise, a literal is chosen from the candidate set based on selection heuristics described in Section 4.7. The heuristics are based on various measures but it always prefers **Assert** candidates over **Split** candidates, in order to minimize the creation of backtrack points.

(ii) Context Evolution

If the selected literal is a **Split** literal, a backtrack point is created (corresponding to the left part of the application of the **Split** rule). Then, the literal is added to the context, the **Compact** rule is exhaustively applied to the new context, and the **Subsume**, and **Resolve** rules are exhaustively applied to the current clause set using the new context literal.

(iii) Context Unifier Computation

All possible context unifiers between current clauses and the new context are computed which involve the new context literal. If this leads to the computation of a *closing context unifier*, a context unifier with an empty remainder, the current branch is immediately closed, forcing the procedure to backtrack.

(iv) Backtracking

If a closing context unifier is found in the previous step, the current context does not satisfy the input clause set and is unrepairable. The procedure then backtracks to a previous choice point, undoing all changes to the context and the clause set done from that choice point on. Since the choice point corresponds to the left part of the application of the **Split** rule which added a literal L to the context, the right part of the

application is then tried. The skolemized complement of L is selected for addition to the context and the computation continues with Step [ii](#).

If there are no more choice points to backtrack to, the input set is proven unsatisfiable and the procedure quits.

(v) Candidate Generation

If no closing context unifier is found in Step [iii](#), the procedure extracts from the computed context unifiers all those literals that are suitable for an application of **Split** or **Assert**, adds them to the candidate sets, and goes back to Step 1.

A high-level pseudocode description of the proof procedure is provided in Figure [1](#). For simplicity we describe a non-restarting recursive version of the procedure implementing naive chronological backtracking. When it terminates the procedure either returns a set of literals, representing the most recent context and denoting a model of the input clause set, or raises the exception **CLOSED**, to denote that the clause set is unsatisfiable. In the backjumping version the exception **CLOSED** would also carry dependency information used to decide whether to ignore right splits or not. The following example is intended to demonstrate the working of the proof procedure.

Example 3.1 Let Φ be the following clause set.

$$p(x, a) \vee s(a) \tag{1}$$

$$q(x, y) \vee q(y, x) \tag{2}$$

$$r(f(x, y)) \vee \neg p(x, y) \tag{3}$$

$$\neg p(a, a) \vee \neg q(x, y) \vee \neg r(f(a, y)) \tag{4}$$

After initializing its variables Λ and L , the proof procedure in Figure [1](#) first determines an initial set of candidates CS . Because Φ contains no unit clause, CS is the empty set and the function me is called as $me(\Phi, \emptyset, \neg v, \emptyset)$.

The set of new candidates CS' determined then consists of the two split literals $p(x, a)$ and $q(u, v)$. They originate from clause 1 and from clause 2, respectively. Simplification of Φ has no effect, and so Φ' is the same as Φ . The current context Λ' becomes $\{\neg v\}$. Because of $CS' \neq \emptyset$, line 20 is reached, and the selection heuristics chooses $p(x, a)$ as the literal L to consider for the next inference step (the literal $p(x, a)$ is preferred over the other split literal, $q(u, v)$, because it is universal, while $q(u, v)$ is not; cf. Section [4.7](#) for details). Because $p(x, a)$ is a split literal, line 25 is reached, which results in the call $me(\Phi, \{\neg v\}, p(x, a), \{q(u, v)\})$. In its execution, the new assert candidate $r(f(x, a))$ is determined (from $p(x, a)$ and clause 3) and thus gets added to the given candidates, yielding $CS' = \{r(f(x, a)), q(u, v)\}$. This time, simplification does show an effect: with the given literal $p(x, a)$, which belongs to the current context as noted on line 7, clause 1 is subsumed, and the first literal of clause 4 is resolved away. The new clause set Φ' thus is

$$q(x, y) \vee q(y, x) \tag{2}$$

```

1 function darwin  $\Phi$ 
2   // input: a clause set  $\Phi$ 
3   // output: either "unsatisfiable"
4   //           or a set of literals encoding a model of  $\Phi$ 
5   let  $\Lambda = \emptyset$  // set of literals
6   let  $L = \neg v$  // (pseudo) literal
7   //           //  $\Lambda \cup \{L\}$  is the current context
8   let  $CS =$  set of assert literals consisting of the unit clauses in  $\Phi$ 
9   //           // the candidate set
10  try  $me(\Phi, \Lambda, L, CS)$ 
11  catch CLOSED -> "unsatisfiable"
12
13 function  $me(\Phi, \Lambda, K, CS)$ 
14   let  $CS' = add\_new\_candidates(\Phi, \Lambda, K, CS)$ 
15   let  $\Phi' = \Phi$  simplified by Subsume and Resolve
16   let  $\Lambda' = \Lambda \cup \{K\}$  simplified by Compact
17   if  $CS' = \emptyset$  then
18      $\Lambda'$  //  $\Lambda'$  encodes a model of  $\Phi'$ 
19   else
20     let  $L = select\_best(CS', \Lambda')$ 
21     if  $L$  is an assert literal then
22        $me(\Phi', \Lambda', L, CS' \setminus \{L\})$  // assert  $L$ 
23     else
24       try
25          $me(\Phi', \Lambda', L, CS' \setminus \{L\})$  // left split on  $L$ 
26       catch CLOSED ->
27          $me(\Phi', \Lambda', \bar{L}^{sko}, CS' \setminus \{L\})$  // right split on  $L$ 
28
29 function  $add\_new\_candidates(\Phi, \Lambda, L, CS)$ 
30   adds to  $CS$  all assert literals from context unifiers involving  $L$ 
31   and one split literal from each remainder of a context unifier involving  $L$ 
32   raises the exception CLOSED if it finds a closing context unifier
33
34 function  $select\_best(CS, \Lambda)$ 
35   returns the best assert or split literal in  $CS$ 

```

Fig. 1. Darwin's proof procedure as pseudo code.

$$r(f(x, y)) \vee \neg p(x, y) \tag{3}$$

$$\neg q(x, y) \vee \neg r(f(a, y)) \tag{4'}$$

Next, $p(x, a)$ is moved to the current context, yielding $\Lambda' = \{\neg v, p(x, a)\}$. The execution of the pseudocode reaches line 20, and among the current candi-

dates $CS' = \{r(f(x, a)), q(u, v)\}$ the literal $r(f(x, a))$ is selected by the heuristics for further processing (see again Section 4.7). Because $r(f(x, a))$ is an assert literal, line 22 is reached and $me(\Phi, \{\neg v, p(x, a)\}, r(f(x, a)), \{q(u, v)\})$ is called. On execution, the newly asserted literal $r(f(x, a))$ together with the clause 4' gives rise to the new assert candidate $\neg q(x, a)$. Notice that in the underlying **Assert** rule application the context literal $r(f(x, a))$ gets instantiated to $r(f(a, a))$ – with a parametric literal $r(f(u, a))$ instead, $\neg q(x, a)$ could not be derived as an assert candidate. Now, $\neg q(x, a)$ is chosen to be asserted, and the next call thus is $me(\Phi, \{\neg v, p(x, a), r(f(x, a))\}, \neg q(x, a), \{q(u, v)\})$. Because for the context $\{\neg v, p(x, a), r(f(x, a)), \neg q(x, a)\}$ a closing context unifier exists (it uses clause 2), the exception **CLOSED** is raised. Notice that the parametric literal $p(u, v)$ from the set of candidate literals was never chosen to derive this closed branch.

The exception raised is caught by the first incarnation of me . Its execution thus reaches line 27 and tries the right alternative of that **Split** application. Because the split literal was $p(x, a)$ the corresponding call to me uses the complement of the Skolemized version of $p(x, a)$, say, $\neg p(c, a)$. On the execution of $me(\Phi, \{\neg v\}, \neg p(c, a), \{q(u, v)\})$, the new assert candidate $s(a)$ is derived from $\neg p(c, a)$ and clause (1). It will indeed be asserted, and for the next call to me only one candidate will be available, which is the split literal $q(u, v)$. After choosing it and calling me again no more candidate can be determined. The execution of me thus terminates and returns the context $\{\neg v, \neg p(c, a), s(a), p(u, v)\}$ to indicate satisfiability of the given clause set.

4 Implementation

The description of the proof procedure in the previous section omits most implementation details and also leaves room for certain improvements. We provide some of these details as implemented in *Darwin* next, focusing more on those that are significant for its performance.

4.1 Term Database

During the derivation hundreds of thousands of terms may be created, easily consuming hundreds of megabytes of memory. Many of these terms are dropped soon after creation, e.g. in backtracking or when a new context literal permanently satisfies a number of remainders. This causes a lot of time spent by the garbage collector.

To lessen the problems caused by high memory consumption, terms are stored in a compact way. Terms are represented in a natural way as tree-like data-structures. However, at a lower representational level, *Darwin* uses a database technique similar to the one used in the Vampire prover [13].⁷ Compared to a naïve representation of terms, it allows for vastly reduced memory

⁷ Similar techniques are also used in Otter [11] and E [14].

consumption by sharing common subterms. For instance, the terms $p(f(a))$ and $g(f(a))$ share the common subterm $f(a)$, which needs to be represented only once in memory.

As in Vampire, usage of terms is managed by associating counters with them. Requesting a term increments its usage counter, explicit deregistration decrements it. When a term's usage counter drops to zero, the term is removed from the database and garbage collected.⁸ Contrary to term requests, which are processed immediately, deregistration requests for a term are stored in a buffer and processed (in order of arrival) only when the buffer is full, effectively delaying by the buffer's length the decrement of the term's usage counter. Since quite often the same terms are requested and released as part of the candidate set management process, this delay in processing deregistration requests reduces the number of times those terms are actually removed from and reinserted into the database.

The overhead of retrieving a term from the database is reduced by means of an efficient hashing on the terms. Furthermore, we gain the possibility of implementing term equality tests as constant-time pointer equality tests, and we save in term creation and garbage collection, leading in practice to performance improvements in some cases.

4.2 Backjumping and Dynamic Backtracking

The simplest backtracking strategy for a search procedure is (naïve) chronological backtracking, which backtracks to the most recent choice point in the current branch of the search tree. A more effective form of chronological backtracking, implemented instead in *Darwin*, is *backjumping*, which takes dependencies between choice points into account. The idea of backjumping is best explained in terms of the calculus: suppose the derivation subtree below a left node introduced by a **Split** rule application is closed *and* the literal added on the left conclusion by that application is not needed to establish that the subtree is closed. Then, the **Split** rule application can be viewed as not being carried out at all. The proof procedure thus may neglect the corresponding choice point on backtracking and proceed to the previous one.

Backjumping is well known to be one of the most effective improvements for propositional SAT solvers. Its implementation is not too difficult and is based on keeping track of which context literals and clauses are involved in particular in **Assert** and **Close** rule applications. Backjumping is an example of a successful propositional technique that directly lifts to the proof procedure of *Darwin*.

⁸ Note that *Darwin* does not implement its own garbage collector. Since *Darwin* is written in OCaml, removing the term from the database is enough to make the memory locations it occupies available to OCaml's garbage collector. To eliminate the overhead of explicit registrations and deregistration in the database we plan to reimplement the term database using OCaml's weak hash tables, which effectively push the registrations and deregistration activities down to the compiler level.

A smarter technique than backjumping has been proposed under the name of *dynamic backtracking* by Ginsberg [7]. It can be adapted to our proof procedure and it is currently implemented in *Darwin* as an alternative to backjumping. The idea is that a choice point (and associated state) not involved in establishing that a branch is closed is not discarded as in backjumping, but kept if it does not depend on a discarded choice point. Conceptually, the choice points are no longer seen as nodes in a tree but as nodes of a dependency graph. Discarding a choice point does not automatically invalidate all later created choice points as well, but only those dependent on it. Thus dropping and possibly recomputing a still valid and potentially useful part of the derivation is avoided.

A disadvantage of dynamic backtracking versus backjumping is that its implementation is more involved and requires a more complex type of dependency analysis. This causes non-negligible runtime overhead. Furthermore, because derivations are in general not shorter than with backjumping, it is not yet clear at the moment when it is best to use dynamic backtracking instead of backjumping.

4.3 Iterative Deepening over Term Depth

The refutational completeness of the proof procedure is ensured by using iterative deepening over a bound on term depth, i.e. over the depth of terms seen as trees. The proof procedure never adds a literal to the context if its depth exceeds the current term depth bound. Thus, when the inference rule applications to the current context are exhausted⁹ and leave it open, the procedure has to check if a candidate literal has been ignored because it exceeded the depth bound. If so, the procedure will restart with a completely new derivation and an increased term depth bound; otherwise, it reports the discovery of a model for the input set.

A benefit of the scheme described is that possibly many candidates for **Assert** and **Split** rule applications with deep terms will be dropped. This vastly decreases the memory requirements for some problems which have a refutation using only comparatively shallow terms but have lots of candidates with deeper terms.

Currently, no information from a previous round is kept after a restart. A valuable improvement of *Darwin* might be to avoid this and keep growing the current branch under the increased term depth bound. Asymptotically, though, there should be no difference. Another related improvement would be to compute permanent lemma clauses as a side effect of derivations, as can be commonly found in SAT solvers.

Alternative measures for literal complexity than taking the term depth

⁹ By the design of the inference rules it is impossible that a context contains two or more p-variants of the same literal. This property implies the termination of exhaustive inference rule applications under a term depth bound.

could be used as well. For instance, the hyper tableau prover KRHyper [17] uses iterative deeping over term weights, which are computed as the number of symbols in a term. The resolution prover Otter [11] offers sophisticated control facilities to weigh a term. There is considerable room for further experimentation.

4.4 Initial Default Interpretation

As mentioned in Section 2.3, the pseudo-literal $\neg v$ that constitutes the initial context assigns by default false to all ground atoms. Instead of $\neg v$, the pseudo-literal v may be used, assigning true to all ground atoms. It is indeed often plausible to take v , given that many theorem proving benchmarks consist of an “axiom part” and a “theorem” part. The theorem part quite often consists of one or more negative clauses. These theorem clauses are falsified in the interpretation associated with the pseudo-literal v . Now, the calculus considers for **Split** rule applications only clause instances that are falsified in the current interpretation. This means that then theorems are used early in the derivation, de-emphasizing, in particular, the use of positive clauses from the axiom part. This way the calculus becomes more goal-oriented than it would be with $\{\neg v\}$ as the initial context.

Nevertheless, and somewhat surprisingly (to us), the overall performance on many TPTP problems that have the structure mentioned is much better with $\neg v$ than with v . This phenomenon should be investigated further.

4.5 Unification with Offsets

In order to avoid creating variants of terms when needed for unification, Otter and KRHyper use so called *contexts*. A compile time limit for the number of variables per term is imposed, e.g. 64 variables per term in the case of KRHyper. Each variable in a term is identified by a number less than the limit. During unification a context – containing a multiplier – is associated with each term. The effective id of a variable during the unification is the limit multiplied by the associated context’s multiplier plus the variable’s real id. E.g. if the limit is 64, y has the id 1 in $p(x, y)$, and for a given unification the multiplier of the context associated with $p(x, y)$ is 3, then the effective id of y during the unification is $64 * 3 + 1 = 193$. To avoid exceeding the compile time limit, terms are normalized when constructed so that the variable ids are enumerated from 0 on.

Inspired by this idea *Darwin* uses *offsets*, which avoid the compile time limitation. During unification the terms of each clause are associated with an offset unique for the unification. The unification operates on “terms” of the form *offset:term*. For example, if the clause $p(x) \vee p(f(x))$ is unified with two variants of the context literal $\neg p(u)$, the offset 0 may be associated with the clause, and the offsets 1 resp. 2 with the two occurrences of the context literal. Then the terms $0:p(x)$ and $1:\neg p(u)$ are unified, and the terms $0:p(f(x))$ and

$2:\neg p(u)$ are unified, yielding the unifier $\{0:x \mapsto 1:u, 2:u \mapsto 0:f(1:u)\}$ where $1:u$ and $2:u$ are in fact two different variables.

4.6 Context Unifiers

Recall that Step 3 of *Darwin's* proof procedure computes all possible context unifiers involving the context literal just added. To be precise, the system computes context unifiers of input clauses in order to identify literals that can be added to the context by the **Split** rule, and computes context unifiers of subsets of input clauses in order to identify literals that can be added by the **Assert** rule. To speed up this computation, context unifiers are partially precomputed and cached as described below. For simplicity, we consider here only the computation of the context unifiers for **Split**. Figure 4.6 illustrates this process and its embedding in the proof procedure.

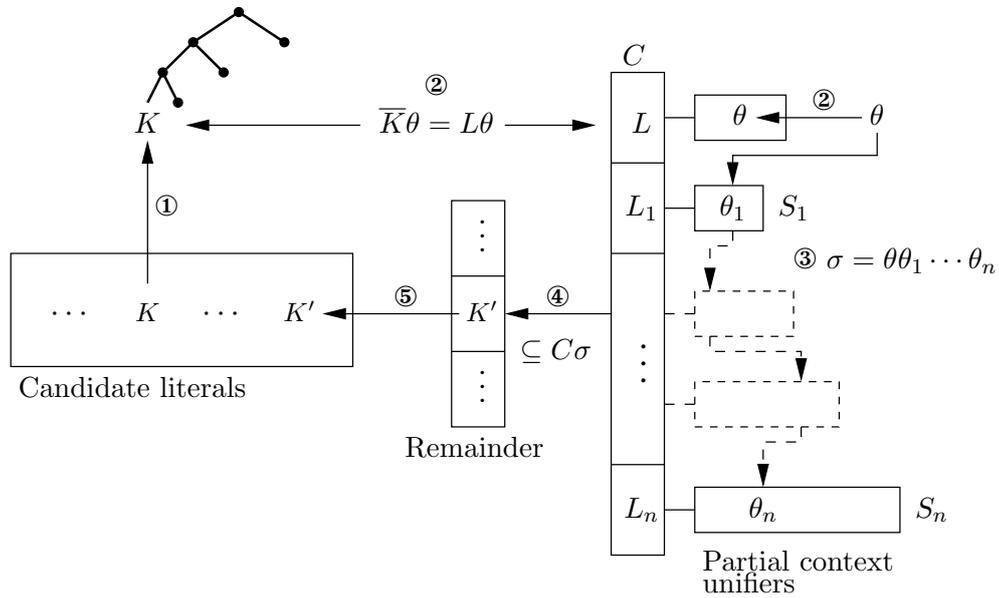


Fig. 2. Computation of context unifiers and its embedding in the proof procedure.

Each input literal has an associated list of *partial context unifiers*. A partial context unifier is merely a unifier between the input literal and a literal from the current context. If a literal occurs in several input clauses at the same position these occurrences share one list.

The bindings of the stored partial context unifiers are kept in a database similar to the term database. Especially for some Horn problems, where many very similar terms are computed, the unifiers tend to share most bindings. Using the database leads to significant memory savings.

When a new literal K is added to the context (step i in the proof procedure, step ① in Figure 4.6), the system computes all partial context unifiers between (a fresh variant of) \bar{K} and each input literal. Then it stores each computed unifier on the list of the corresponding input literal. This is depicted in Figure 4.6 as step ②, however for only one input literal. After that,

for each literal L that unifies with \overline{K} and for each input clause C containing L , the system attempts to find all possible context unifiers of C against the current context. This is done as follows.

Assume that C is of the form $L \vee L_1 \vee \dots \vee L_n$, θ is the partial context unifier between L and \overline{K} , and S_i is the set of partial context unifiers stored in L_i 's list. Then the system considers each tuple of partial unifiers in $\{\theta\} \times S_1 \times \dots \times S_n$ and attempts to merge the elements of that tuple into a single unifier (step ③ in Figure 4.6). When the merge succeeds, the resulting substitution is a context unifier of C against the current context.¹⁰

To minimize recomputation, the merged unifiers are computed incrementally by traversing the partial context unifier lists for the clause C in a depth-first fashion. The root node of the depth-first traversal is θ , its children are all the partial context unifiers of L_1 , the children of each of the root's children are all the partial context unifiers of L_2 , and so on. Partial context unifiers are merged incrementally as they are visited along a path of this imaginary tree, and the merged unifier computed along a path is reused for all the extensions of that path.

Clearly, less work is done if the tree is slim at the top, as less merging operations are then necessary. To achieve this the lists associated with the literals L_1, \dots, L_n in C are actually first ordered by increasing length before starting the traversal. This is indicated in Figure 4.6 by boxes of growing length for S_1 to S_n in this order.

Each newly computed context unifier determines a remainder (step ④ in Figure 4.6), and every such (non-empty) remainder provides one new candidate literal that gets added to the candidate set in Step v of the proof procedure¹¹ (step ⑤ in Figure 4.6, where the new candidate literal is denoted as K').

Note that for each candidate literal, the system maintains a reference to the remainder and the context unifier it came from. This entails that *all* the computed context unifiers are permanently kept in memory.¹²

It is interesting to point out though that the calculus does not prescribe that at all: the **Split** inference rule (and similarly **Assert**) admits implementations that compute remainders only “locally”, during the **Split** rule application, and discards them afterwards. Thus, for a given context, the possible context unifiers of a clause could be computed, say, one after the other until an admissible one is found. At this point **Split** could be applied using that unifier and the unifier could then be immediately discarded. Memory consumption

¹⁰ The context unifier is converted into an admissible context unifier afterwards. But we can ignore this issue here.

¹¹ It can be shown that it is indeed enough to consider only one **Split** literal per remainder without affecting the calculus' completeness.

¹² This is not entirely accurate, since only context unifiers with remainders containing *active* candidate literals are explicitly kept (see Section 4.8). It is accurate though if we assume that the set of active candidates is large enough to contain all candidates.

under such a scheme would be obviously far less. Nevertheless, the approach as used in *Darwin* has a big advantage: because at any point in the derivation all the theoretically necessary context unifiers and their remainders are explicitly stored, they are available for inspection and comparison. Because both the choice of a remainder from the set of all possible (admissible) remainders, and the choice of a literal from it to split with are *don't-care* nondeterministic choices, arbitrary heuristics can be employed for their computation.

Furthermore, for each pairing of an input literal L with a context literal, the computation of the context unifier for the clause containing L is attempted *exactly once* in the current derivation tree branch. This avoids the re-computation of the same context unifier that would happen in the more naïve scheme indicated above.

These two considerations are the main rationale for the design decisions that led to the described data structures and algorithms described above. Fortunately, memory problems seem seldom to happen. But clearly more experimental results explicitly monitoring memory consumption are needed.

4.7 Selection Heuristic

As explained in the previous section, all theoretically necessary remainders are explicitly stored in memory, at any point in the derivation, which supports the effortless implementation of heuristics to select a literal to split with. The heuristics for selecting a literal from the candidate set to be added to the context is based on the following criteria. The overall heuristics is determined by the induced lexicographic ordering over these criteria, with “Universality” being the most significant criterion, and “Generation” the least significant one.

(i) Universality

Universal literals (which includes ground literals as well) are preferred to parametric literals as they impose stronger constraints on the context. Furthermore, as soon as the context contains parameters the number of computed remainders and thus split candidates might increase significantly.

(ii) Remainder Size¹³

Recall that candidate literals for **Split** are drawn from the remainder of some context unifier. Now, if the problem is satisfiable, at least one remainder literal of every remainder must be satisfied by the context. Because of this, candidate literals originating from smaller remainders are preferred over literals from larger remainders. The rationale is that backtracking is minimized this way. For an extreme case, note that for

¹³ This applies only to **Split** candidates, not **Assert** candidates. **Assert** is always preferred over **Split** in order to emphasize redundancy elimination. Recall that **Assert** literals are always universal. Therefore criterion (i) is always satisfied for **Assert** candidates, while criteria (iii) and (iv) still take effect.

Split literals coming from a singleton remainder applying the right side of Split is pointless because it immediately produces a closed branch. As a consequence *Darwin* does not even generate a choice point when it adds such literals to the context.

(iii) Term Weight

The number of symbols in a literal has shown to be useful information that should be exploited. This emphasizes the use of “lighter” literals. Because variables are excluded from counting, additional preference is given to literals with variables instead of parameters or other terms at the variable positions.

(iv) Generation

This is a measure of how close in the derivation the candidate is to the original clause set. The generation of a context literal is -1 for $\neg v$, and the generation of the corresponding candidate otherwise. The generation of a candidate is the maximum of the generations of the context literals used in its context unifier incremented by one. That is, candidates whose context unifier is solely based on $\neg v$ are of generation 0.

Candidates with a smaller generation are preferred. The intention is to keep the derivation close to the problem set, similar to bidirectional search. For some problems this is the key to their solutions; on average it is a slight improvement.

Recall that the term depth is not needed as part of the heuristic as it is implicitly imposed by the depth bound (see Section 4.3).

4.8 Inactive Candidates

In order to decrease the memory usage there is a limit on the number of *active* candidates, i.e. the candidates stored together with the remainder they came from and additional information for the selection heuristic. For the other candidates—the *inactive* candidates—only the clause and the context literals used in the computation of the context unifier are stored. Due to the term database this amounts to a few pointers per inactive candidate. The management of active/inactive candidates in *Darwin* is analogous to the management of active/passive clauses in recent versions of Waldmeister [6]. Specifically, however, it works as follows.

When the limit of active candidates is reached a new candidate is first compared with the worst active candidate, according to the ordering relation described in the previous subsection. If the new candidate is better, the worst active candidate is made inactive, and the new candidate is added to the active candidates; otherwise the new candidate is put into the set of inactive candidates. The best inactive candidates are moved to the active set as active candidates are selected for addition to the context and removed from the set. When an inactive candidate is made active, the context unifier has to be

recomputed from the clause and context literals.

4.9 Substitution Tree Indexing

The context is basically a set of literals. The preconditions of **Split**, **Assert**, and **Subsume** require, in essence, to search the context for literals that unify with, subsume, or are subsumed by a given literal. Some of these queries are applied to every computed candidate at least once in order to immediately drop invalid, e.g. subsumed, candidates. In order to avoid a linear scan of the context to perform each of these checks, *Darwin* uses term indexing for the context based on substitution trees [9].

Substitution trees index terms by abstracting over identical subterms. E.g. the terms $f(g(a))$ and $f(g(b))$ are represented by a node containing $f(g(x))$ and two children containing the substitutions $\{x \mapsto a\}$ and $\{x \mapsto b\}$. Thus for the term $f(h(a))$ the non-unifiability is detected at the node $f(g(x))$ for both children. In general substitution trees seem to be best suited for deep terms containing variables. For shallow ground terms, e.g. for clause sets stemming from Bernays-Schönfinkel problems, *Darwin*'s implementation of substitution trees actually produces slower performance than no indexing at all.

For comparison an alternative indexing scheme based on imperfect discrimination trees has been implemented. Their performance is quite close to substitution trees for non Horn problems and slightly superior for Horn problems. This might be due to an inefficient implementation of the significantly more complex substitution trees, too small indexes – rarely larger than some ten thousand terms –, unsuitable terms or a bad query to maintenance operation ratio.

4.10 Close Look-ahead

A branch is detected as unsatisfiable as soon as **Close** applies, which happens when a context unifier with an empty remainder is computed for an input clause. It is easy to see however, that when two contradictory **Assert** candidates are computed, the branch containing them can be closed after one **Assert** application. Now, due to the fact that candidate literals wait for their turn in the candidate set, in unlucky cases two contradictory candidates might be ignored for a long time. To avoid this problem, **Assert** candidates are stored in a substitution tree (Section 4.9). Each new candidate is checked against the tree for a contradiction. As soon as this check succeeds **Close** can be triggered by adding the new candidate to the context.¹⁴

It is not clear yet if this in general improves the performance by leading to shorter derivations or decreases the performance by introducing too much overhead. First tests seem to indicate the first, but further tests are needed.

¹⁴ Actually, it is not added but the derivation is immediately backtracked.

4.11 Programming Language

Darwin is implemented in *OCaml*¹⁵. OCaml is—among other things—a fast strongly-typed functional language based on ML. OCaml—and thus *Darwin*—is available for several Unix-like operating systems including Linux and Mac OS X, and for the Windows family. OCaml has previously been successfully used for the implementation of the theorem prover KRHyper¹⁶ at the University of Koblenz and for the solver ICS¹⁷ at SRI International, among others.

Though the programming background of the second author, the main developer of *Darwin*, was mostly in OO-style C++, he quickly enjoyed using OCaml. Among other things OCaml's strong-typing, garbage collection, extremely short compile times, and informative news group made up for the paradigm shift. At the current stage of development we find that the higher level of abstraction provided by OCaml constructs—and thus the better readability and maintainability of the code, compared to e.g. C—amply compensate for possible performance losses when compared to implementations in lower level languages like C.

5 Performance Evaluation

As mentioned we have just started evaluating the performance against the TPTP problem library¹⁸. Because *Darwin*'s input language is clause logic, and *Darwin* does not (yet) have dedicated inference rules for equality, we concentrated on the clausal problems without equality.

Furthermore, in order to compare *Darwin* with other current provers, we separately list results for some of the problem sets used in the last two CASC competitions, i.e. CASC-18 in 2002 and CASC-19 in 2003¹⁹. Equality was handled by including the axioms of equality as provided in the TPTP.

All tests were run on a Pentium IV 2.4Ghz computer with 512MB of RAM. The imposed time limit was 300 seconds for the tests on the clausal problems of the TPTP without equality, and 500 seconds for the CASC tests; the memory limit was 500 MB in both cases. Experiments showed that for the CASC competitions slower machines by a factor of three were used. As most problems are solved within 100 seconds the results are comparable nevertheless.

Table 1 summarizes the results for the former problems and Table 2 summarizes the result for the CASC problems. For each problem set the name and the number of problems are given, followed by the results for the tested *Darwin* configurations. Each result is stated as the number of problems solved and the average CPU time spent on it.

¹⁵ See <http://caml.inria.fr/>.

¹⁶ See <http://www.uni-koblenz.de/~wernhard/krhyper/>.

¹⁷ See <http://www.icansolve.com/>.

¹⁸ See <http://www.cs.miami.edu/~tptp/>.

¹⁹ Available at <http://www.cs.miami.edu/~tptp/CASC/18/> and <http://www.cs.miami.edu/~tptp/CASC/19/>.

Name	# Problems	Default	Dyn.Bt.	v	Inact.	Discr.	-Subs.
HNE	753	591/6.5	591/6.5	591/6.5	600/7.6	592/6.0	591/6.4
NNE	1172	803/4.1	804/3.9	730/9.0	802/4.0	801/3.7	802/4.1

Table 1. Results for *Darwin* test runs on the clausal problems of the TPTP problem library (version 2.6) without equality, divided in Horn problems (“HNE”) and non-Horn problems (“NNE”). Table entries are of the form “Number of problems solved”/“average CPU time”. See text for further explanations.

Name	# Problems	Best	Default	Dyn.Bt.	v	Inact.
CASC-18						
HNE	35	34	18/24.0	18/24.0	18/24.0	19/23.0
HEQ	35	33	9/23.6	9/23.6	9/23.6	9/24.2
EPS	35	27	28/11.1	30/3.8	29/26.9	28/10.0
EPT	35	34	33/16.8	32/9.9	28/7.1	33/18.1
NNE	35	33	16/7.4	16/7.7	11/6.8	17/19.7
SNE	35	28	9/14.5	9/15.5	6/0.0	9/14.1
CASC-19						
HNE	20	18	10/64.9	10/64.9	10/64.9	10/33.8
HEQ	20	18	0/0.0	0/0.0	0/0.0	0/0.0
EPS	35	34	31/6.2	31/5.0	31/28.5	31/5.4
EPT	35	33	31/5.6	31/7.6	30/29.3	32/19.1
NNE	20	18	12/16.5	10/16.8	8/53.9	13/31.0
SNE	35	34	3/0.0	3/0.0	2/0.0	3/0.0

Table 2. Results for *Darwin* test runs on CASC-18 and CASC-19 problem sets. Problem names: HNE – Horn with No Equality; HEQ – Horn with some (but not pure) Equality; EPS – Effectively Propositional non-theorems (satisfiable clause sets); EPT – Effectively Propositional Theorems (unsatisfiable clause sets); NNE – Non-Horn with No Equality; SAT with No Equality. Table entries are of the form “Number of problems solved”/“average runtime”. See text for further explanations.

In Table 2, “Best” is the number of problems solved by the best prover for this problem set at the CASC competition. In both tables, “Default” refers to *Darwin* with all inference rules enabled, backjumping enabled, the initial context $\{-v\}$, the initial depth bound set to 2, and the use of substitution tree indexing. The remaining columns represent modifications of this default setting: “Dyn.Bt.” means dynamic backtracking instead of backjumping is used, “ v ” means the initial context is set to $\{v\}$, “Inact.” picks the best inactive literal instead of the oldest, i.e. candidate selection is better informed but leads to more memory consumption²⁰, “Discr.” uses discrimination trees instead of substitution trees and “-Subs.” does not apply the **Subsume** rule.

²⁰ Note that we described this improved behavior in Section 4.8.

Note that the backtracking method and default interpretation does not matter for Horn problems, as, first, no backtracking happens at all, and, second, no splitting occurs²¹ — the pseudo-literal $\neg v$ (or v) of the initial context is never used. Further note that *Darwin* is among the best provers for the EPS and EPT divisions, which consist of clause sets without function symbols except constants.

The results show that backjumping and dynamic backtracking are close in the number of solved problems, though they do not solve exactly the same problems. The “ v ” setting is clearly inferior to the “ $\neg v$ ” setting in terms of performance. In addition, “ v ” and “Inact.” are the only configurations exceeding the memory limit repeatedly in the CASC tests. “Inact.” makes up for this with the best result for HNE. Discrimination trees are in general noticeably faster than substitution trees for Horn problems, and similar for non-Horn problems. Deactivating **Subsume** leads to a slight performance decrease. Altogether the best configuration seems to be “Default + Inact. + Discr.” for Horn problems, and “Default + Inact.” for non-Horn problems, as the current discrimination tree implementation does not support productivity checks.

Updates of experimental results and more detailed information, including *Darwin*’s time and memory consumption individually for each problem, can be found on *Darwin*’s web page, <http://www.mpi-sb.mpg.de/~baumgart/DARWIN/>.

6 Conclusions and Future Work

The purpose of this paper was to describe the design of the *Darwin* theorem prover, its proof procedure, data structures and algorithms. One of the main motivations for developing *Darwin*’s calculus, Model Evolution, was the possibility of migrating to the first-order level some of those very effective techniques developed by the SAT community for the DPLL procedure. This goal has been achieved to a certain degree: the current version of *Darwin* implements a first-order version of unit propagation, a form of simplification, and backjumping, a form of intelligent backtracking. These features, which are considered absolutely critical for the good performance of propositional DPLL-based SAT solvers, where the most immediately implementable as the Model Evolution calculus itself [2,3] was already designed with them in mind.

Yet, much remains to be done. Various alternatives and modifications to *Darwin*’s data structures and algorithms have been identified in Section 4. Among these, perhaps the most significant one concerns the selection heuristics explained in Section 4.7.

It will be interesting to adapt to *Darwin* some of the heuristics that have proven useful with the propositional DPLL procedure. For instance, we are

²¹ Assert and Close are sufficient for completeness, Split is not needed.

considering implementing a literal selection heuristics that prefers candidates from recent *conflict sets*, i.e., recently responsible for the closure of a previous branch [8]. Since conflict sets are already computed in *Darwin* as they are used for backtracking (see Section 4.2) this heuristics should be quite easy to incorporate. The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned for the next version. Adding lemmas, however, will require some more theoretical work on the calculus level first.

Fairness of derivations is currently realized through iterative deepening over term depth. It will be interesting to experiment with alternatives like iterative deepening over derivation length. Different iterative deepening strategies are known to have drastical impact on the search space exploration of model *elimination* provers, and it seems plausible to expect the same for *Darwin*.

We also reported on practical experiments carried out with problems from the CADE-18 and CADE-19 system competitions, as well as on results on parts of the TPTP problem library. When assessing the performance of *Darwin* compared to other provers, we believe one should take into account that the Model Evolution calculus is a very recent development. A great deal of knowhow has been developed over the last decades for the implementation in particular of resolution and model elimination based systems. Although the techniques employed there can be partially exploited (and we tried so for *Darwin*), new algorithms and data structure tailored for the Model Evolution calculus are probably needed. Similarly, more work is necessary to identify successful proof strategies and heuristics for the calculus. The same applies to other instance-based methods such as, e.g., the disconnection tableau calculus [10], which presently seems to be the only calculus of this kind for which a competitive prover exists [15]. Despite a lack of established knowhow we find our first experimental results very encouraging. In particular, *Darwin* performs very well on clause sets stemming from Bernays-Schönfinkel problems. It is among the best provers for the EPS and EPT divisions of the TPTP library.

Darwin is available from the authors on request; we would be glad if others found it useful.

Acknowledgements

We thank the anonymous referees for their insightful comments on how to improve the paper and for their valuable suggestions on how to improve *Darwin*'s implementation as well.

The work of the second and third authors was partially supported by Grant No. 237422 from the National Science Foundation.

References

- [1] Peter Baumgartner. Fdpll – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.
- [2] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.
- [3] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. Fachberichte Informatik 1–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.
- [4] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
- [5] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
- [6] J.-M. Gaillourdet, Th. Hillenbrand, B. Löchner, and H. Spies. The new WALDMEISTER loop at work. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, volume 2741 of *LNAI*, pages 317–321. Springer-Verlag, 2003.
- [7] Matthew L. Ginsberg, James M. Crawford, and David W. Etherington. Dynamic backtracking, 1996.
- [8] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.
- [9] Peter Graf. Substitution tree indexing. Research Report MPI-I-94-251, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1994.
- [10] Reinhold Letz and Gernot Stenz. Proof and Model Generation with Disconnection Tableaux. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba*, volume 2250 of *Lecture Notes in Computer Science*. Springer, 2001.
- [11] William W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, National Laboratory, Argonne, IL, 1994.
- [12] F. Oppacher and E. Suen. HARP: A Tableau-Based Theorem Prover. *Journal of Automated Reasoning*, 4:69–100, 1988.
- [13] Alexandre Riazonov and Andrei Voronkov. Vampire 1.1 (system description). In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

- [14] S. Schulz. System Abstract: E 0.3. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 297–301. Springer, 1999.
- [15] Gernot Stenz. DCTP 1.2 - System Abstract. In Uwe Egly and Christian G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Copenhagen, Denmark, July 30 - August 1, 2002, Proceedings*, volume 2381 of *Lecture Notes in Computer Science*, pages 335–340. Springer, 2002.
- [16] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
- [17] Christoph Wernhard. System Description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.
- [18] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

Simple and Efficient Clause Subsumption with Feature Vector Indexing

Stephan Schulz¹

*ITC/irst, Trento, Italy
and*

Technische Universität München, Germany

Abstract

We describe *feature vector indexing*, a new, non-perfect indexing method for clause subsumption. It is suitable for both forward (i.e., finding a subsuming clause in a set) and backward (finding all subsumed clauses in a set) subsumption. Moreover, it is easy to implement, but still yields excellent performance in practice. As an added benefit, by restricting the selection of features used in the index, our technique immediately adapts to indexing modulo arbitrary AC theories with only minor loss of efficiency. Alternatively, the feature selection can be restricted to result in *set subsumption*.

Feature vector indexing has been implemented in our equational theorem prover E, and has enabled us to integrate new simplification techniques making heavy use of subsumption. We experimentally compare the performance of the prover for a number of strategies using feature vector indexing and conventional sequential subsumption.

Key words: automated theorem proving, saturation, subsumption, indexing

1 Introduction

First-order theorem proving is one of the core areas of automated deduction. In this field, saturating theorem provers have, in the last few years, developed a significant lead compared to systems based on other paradigms, such as top-down reasoning or instance-based methods.

There are a number of reasons for this. At least one of these reasons is the compatibility of saturating calculi with a large number of redundancy elimination techniques, as e.g. tautology deletion, rewriting, and *clause subsumption*. Subsumption allows us to discard a clause (i.e., exclude it from

¹ Email: schulz@eprover.org

further proof search) if a (in a suitable sense) more general clause exists. In many cases, subsumption can eliminate between 50% and 95% of all clauses under consideration, with a corresponding decrease in the size of the search state.

Subsumption of multi-literal clauses is an NP-complete problem [5]. If some attention is paid to the implementation, the worst case is rarely (if ever) encountered in practice, and single clause-clause subsumption tests rarely form a critical bottleneck. However, the sheer number of possible subsumption relations to test for means that a prover can spend a significant amount of time in subsumption-related code. Even in the case of our prover E [12,13], which, because of its DISCOUNT loop proof procedure, minimizes the use of subsumption, frequently between 10% and 20% of all time was spent on subsumption, with much higher values observed occasionally. The cost of subsumption systematically increases if other simplification techniques based on subsumption are implemented.

In a saturating prover, we are most often interested in subsumption relations between whole sets of clauses and a single clause. In *forward subsumption*, we want to know if *any* clause from a set subsumes a given clause. In *backward subsumption*, we want to find all clauses in a set that are subsumed by a given clause.

We can use this observation to speed up subsumption, by using *indexing techniques* that only return candidates suitable for a given subsumption relation from a set of clauses, thus reducing the number of explicit subsumption tests necessary. A *perfect index* will return exactly the necessary clauses, whereas a *non-perfect* index should return a superset of candidates for which the desired relationship still has to be verified.

Term indexing techniques have been used in theorem provers for some time now (see [7] for first implementations in Otter or [2,3,14] for increasingly up-to-date overviews). However, lifting term indexing to clause indexing is not trivial, because the associative and commutative properties of the disjunction and the symmetry of the equality predicate are hard to handle. In many cases, (perfect) term indexing is only used to retrieve subsumption candidates, i.e., to implement non-perfect clause indexing (see e.g. [17]). Moreover, often two different indices are used for forward- and backward subsumption, as e.g. in the very advanced indexing schemes currently implemented in Vampire [10].

We suggest a new indexing technique based on *subsumption-compatible* numeric clause features. It is much easier to implement than known techniques, and the same, relatively compact data structure can be used for both forward- and backward subsumption. We have implemented the new technique for E 0.8, and in a more polished and configurable way, for E 0.81, with excellent results.

In this paper, we will, after some initial definitions, describe the new technique. We will also discuss how it has been integrated into E, and how it also serves to speed up *contextual literal cutting*, a subsumption-based simplifica-

tion technique that has given another boost to E. We present the results of various experiments to support our claims.

2 Preliminaries

We are primarily interested in first order formulae in clause normal form in this paper. We assume the following notations and conventions. Let F be a finite set of function symbols. We write $f|_n \in F$ to denote f as a function symbol with arity n . Functions symbols are written as lower case letters, we usually use a, b, c for function symbols with arity 0 (constants), and f, g, h for other function symbols. Let V be an enumerable set of variable symbols. We use upper case letters, usually X, Y, Z to denote variables. The set of all *terms* over F and V , $Term(F, V)$, is defined as the smallest set fulfilling the following conditions:

- (i) $X \in Term(F, V)$ for all $X \in V$
- (ii) $f|_n \in F, s_1, \dots, s_n \in Term(F, V)$ implies $f(s_1, \dots, s_n) \in Term(F, V)$

We typically omit the parenthesis from constant terms, as for example in the expression $f(g(X), a) \in Term(F, V)$.

An (equational) *atom*² is an unordered pair of terms, written as $s \simeq t$. A *literal* is either an atom, or a negated atom, written as $s \not\simeq t$. We define a negation operator on literals as $\overline{s \simeq t} = s \not\simeq t$ and $\overline{s \not\simeq t} = s \simeq t$. If we want to write about arbitrary literals without specifying polarity, we use $s \dot{\simeq} t$, or, in less precise way, l, l_1, l_2, \dots . Note that \simeq is commutative in this notation.

A *clause* is a multiset of literals, interpreted as an implicitly universally quantified disjunction, and usually written as $l_1 \vee l_2 \dots \vee l_n$. Please note that in this notation, the \vee operator is associative and commutative (but not idempotent). The empty clause is written as \square , and the set of all clauses as $Clauses(F, V)$. A *formula* in clause normal form is a multiset of clauses, interpreted as a conjunction.

A *substitution* is a mapping $\sigma : V \rightarrow Term(F, V)$ with the property that $Dom(\sigma) = \{X \in V \mid \sigma(X) \neq X\}$ is finite. It is extended to a function on terms, atoms, literals and clauses in the obvious way.

A *match* from a term (atom, literal, clause) s to another term (atom, literal, clause) t is a substitution σ such that $\sigma(s) \equiv t$, where \equiv on terms denotes syntactic identity and is lifted to atoms, literal, clauses in the obvious way, using the unordered pair and multiset definitions.

² For our current discussion, the non-equational case is a simple special case and can be handled by encoding non-equational atoms as equalities with a reserved constant $\$true$. We will still write non-equational literals in the conventional manner, i.e., $p(a)$ instead of $p(a) \simeq \$true$.

3 Subsumption

If we consider a (multi-)set of clauses, that is, a formula in clause normal form, not all of the clauses necessarily contribute to the meaning of it. Often, some clauses are *redundant*. Some clauses do not add any new constraints on the possible models of a formula, because they are already implied by other clauses. Depending on the mechanism of reasoning employed, we can delete some of these clauses, thus reducing the size of the formula (and hence the difficulty of finding a proof). In the case of current saturating calculi, *subsumption* is a technique that allows us to syntactically identify certain clauses that are implied by another clause, and can usually be discarded without loss of completeness. We can specify the (multiset) subsumption rule as a deleting simplification rule (i.e., the clauses in the precondition are *replaced* by the clauses in the conclusion) as follows:

$$(CS) \frac{\sigma(C) \vee \sigma(R) \quad C}{C} \quad \begin{array}{l} \text{where } \sigma \text{ is a substitution, } C \\ \text{and } R \text{ are arbitrary (partial)} \\ \text{clauses} \end{array}$$

In other words, a clause C' is subsumed by another clause C if there is an instance $\sigma(C)$ that is a sub-multiset of C' .

This version of subsumption is used by most modern saturation procedures. It is particularly useful in reducing search effort, since it allows us to discard larger clauses in favor of smaller clauses. Smaller clauses typically have fewer inference positions and generate fewer and smaller successor clauses.

Individual clause-clause subsumption relations are determined by trying to find a simultaneous match from all literals in the potentially subsuming clause to corresponding literals in the potentially subsumed clause. This is usually implemented by a backtracking search over permutations of literals in the potentially subsumed clause (and in the equational case, permutations of terms in equational literals). The first order clause subsumption problem is well-known to be NP-complete. However, there are a number of implementation techniques that can usually avoid the worst case, so that in practice individual subsumption attempts can be completed in acceptable time.

Most of the techniques used to speed up subsumption try to detect failures early by testing necessary conditions. Those include compatibility of certain clause measures (discussed in more detail below) and existence of individually matched literals in the potentially subsumed clause for each literal in the potentially subsuming clause. Additionally, in many cases certain permutations of literals can be eliminated by partially ordering clauses with a suitable ordering.

However, whereas individual subsumption attempts are reasonably cheap in practice, the number of potential subsumption relations to test for in saturation procedures is very high. Using a straightforward implementation of subsumption, we have measured up to 100 000 000 calls to the subsumption

subroutine of our prover E in just 5 minutes on a 300 MHz SUN Ultra-60 for some proof tasks. Thus, the overall cost of subsumption is significant.

3.1 Subsumption Variants

In addition to standard multiset subsumption, there are a number of other subsumption variants and related techniques. We will shortly discuss some of them.

The definition of *set subsumption* is identical to that of multiset subsumption, except in that clauses are viewed as sets of literals (i.e., no multiple occurrences of the same literal are allowed). This allows for a slightly stronger subsumption relation: $p(X) \vee p(Y)$ can subsume $p(a)$ with set subsumption, but not with multiset subsumption. Set subsumption can be used in preprocessing or by provers not based on saturation. For most saturation-based calculi (especially those for which factorization is an explicit inference rule), the fact that a clause can subsume some of its factors causes loss of completeness.

Subsumption modulo AC is a stronger version of multiset or set subsumption, where we do not require that the instantiated subsuming clause is a subset of the subsumed clause, but only that it is equal to a subset modulo a specified theory for associative and commutative function symbols. For example, if f is commutative, then $p(f(a, X))$ subsumes $p(f(b, a)) \vee q(a)$. We are not aware of any system currently using subsumption modulo AC, however, it is generally believed to be useful for reasoning modulo AC.

Equality subsumption allows an equational unit clause to potentially subsume another clause with an equational literal implied by it. It can be described by the following simplification rule:

$$(ES) \frac{s \simeq t \quad u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t}$$

It is typically only applied if $s \simeq t$ cannot be used for rewriting. This rule is implemented by E and a number of other provers, including at least the completion-based systems Waldmeister [6] and DISCOUNT [1].

Finally, a simplification rule that has been popularized by implementation in SPASS [19] and Vampire [9], and is sometimes called *subsumption resolution*, combines resolution and subsumption to cut a literal out of a clause. In the context of a modern superposition calculus, we believe the rule can be better described as *contextual literal cutting*:

$$(CLC) \frac{\sigma(C) \vee \sigma(R) \vee \sigma(l) \quad C \vee \bar{l}}{\sigma(C) \vee \sigma(R) \quad C \vee \bar{l}} \quad \begin{array}{l} \text{where } \bar{l} \text{ is the negation of } l \text{ and} \\ \sigma \text{ is a substitution} \end{array}$$

It can be implemented via a standard subsumption engine (by negating each individual literal in turn, and then testing for subsumption) and is implemented thus at least in E and Vampire. Depending on how and when this

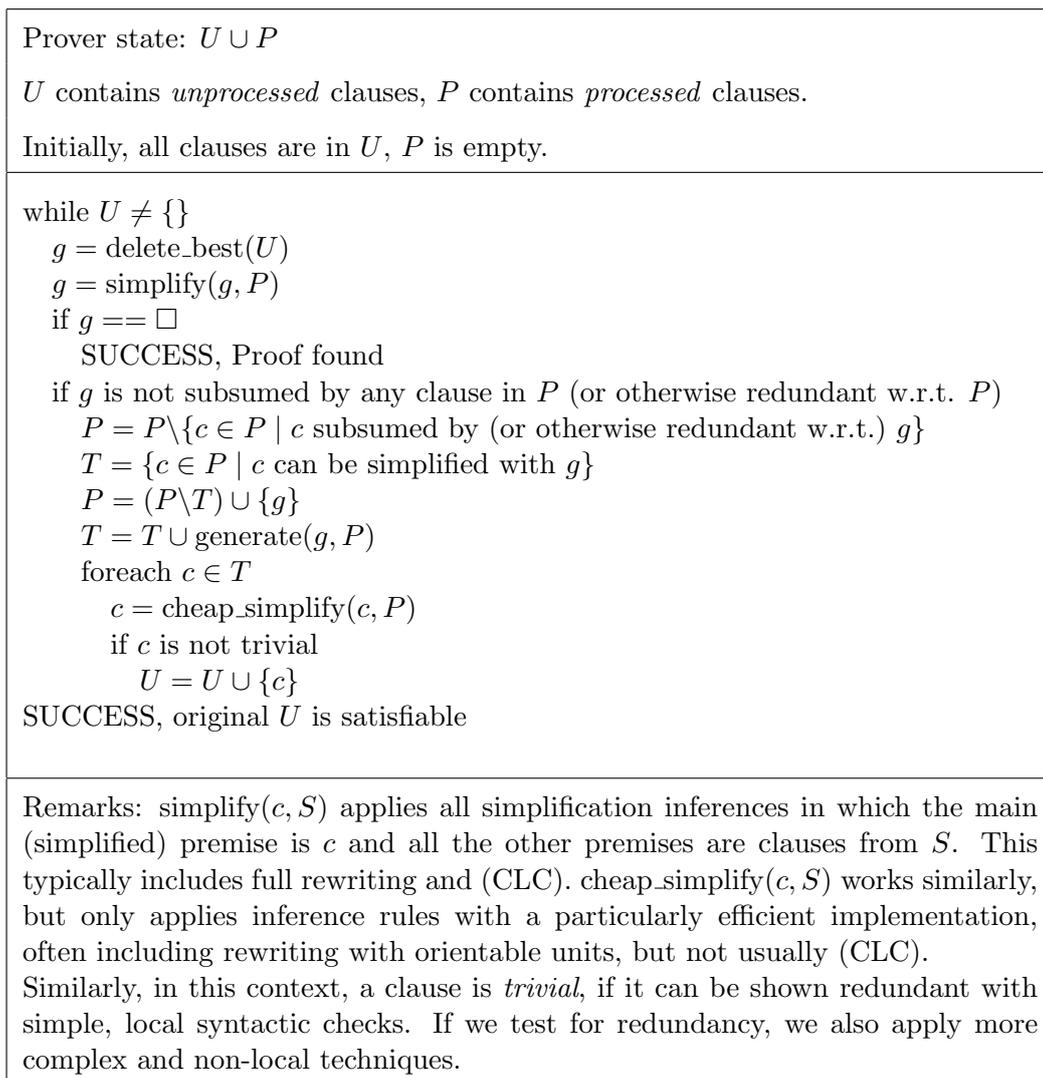


Fig. 1. Saturation procedure of E

rule is applied, it can increase the number of required subsumption tests by many orders of magnitude.

3.2 Saturation Procedures and Clause Set Subsumption

Figure 1 shows a sketch of the main proof procedure of our prover E. It is representative of a modern high-performance prover using a variant of the *DISCOUNT loop* proof procedure (in which unprocessed clauses are *passive*, i.e., not even used as side premises for simplification). The alternative *Otter loop* primarily differs in that simplification and subsumption are also performed between clauses in U and using clauses from U as side premises for simplification of all clauses.

Please observe that standard subsumption appears in exactly two different places and exactly two different roles in this procedure: First, we test if the *given clause* g is subsumed by *any* clause in P . In other words, we want to

know if a single clause is subsumed by any clause from a set. This is usually called *forward subsumption*.

If the given clause is not redundant, we next want to find *all* clauses in P that are subsumed by g . Again, we have an operation between a single clause and a whole set, in this case called *backward subsumption*.

It is obvious that we can implement forward and backward subsumption naively by sequentially testing each clause from P against g . This implementation is e.g. used in early versions of SPASS [19], and was used in E up to version 0.71. However, this does not make use of the fact that we are interested in subsumption relations between individual clauses and usually only slowly changing *clause sets*. The idea behind clause indexing is to *preprocess* the clause set so that subsumption queries can be answered more efficiently than by sequential search.

4 Feature Vector Indexing

Indexing for subsumption is used by a number of provers. Most existing implementations [17,18,7] use a variant of *discrimination tree indexing* on terms to build an index for forward subsumption, often for non-perfect indexing. Indexing for backward subsumption is less frequent, and usually based on a variant of path indexing. We will now present a new and much simpler technique suitable for both forward and backward subsumption.

Our technique is based on the compilation of necessary conditions on numeric clause features. Essentially, a clause is represented by a vector of feature values, and subsumption candidates are identified by comparisons of feature vectors. Feature vectors for clause sets are compiled into a *trie* data structure to quickly identify candidate sets.

4.1 Subsumption-Compatible Clause Features

A (*numeric*) *clause feature function* (or just *feature*) is a function mapping clauses to natural numbers, $f : \text{Clauses}(F, V) \rightarrow \mathbf{N}$. We call f *compatible with subsumption* if $f(C) \leq f(C')$ whenever C subsumes C' . In other words, if f is a subsumption-compatible clause feature, then $f(C) \leq f(C')$ is a necessary condition for the subsumption of C' by C . Unless we specify a particular subsumption variant, we assume standard multiset subsumption.

We will define a number of clause features now, all of which are compatible with multiset subsumption, and many of which are compatible with other subsumption variants.

Let C be a clause. We denote the sub-multiset of positive literals in C by C^+ , and similarly the sub-multiset of negative literals by C^- . Please note that both C^+ and C^- are clauses as well. $|C|$ is the number of literals in C . $|C|_f$ is the number of occurrences of the symbol f in C , e.g. $|p(a, b) \vee f(a, a) \neq a|_a = 4$.

Let t be a term, and let $f|_n$ be a function symbol. We define $d_f(t)$ as

follows:

$$d_f(t) = \begin{cases} 0 & \text{if } f \text{ does not occur in } t \\ \max\{1, d_f(t_1) + 1, \dots, d_f(t_n) + 1\} & \text{if } t \equiv f(t_1, \dots, t_n) \\ \max\{d_f(t_1) + 1, \dots, d_f(t_m) + 1\} & \text{if } t \equiv g(t_1, \dots, t_m), g|_m \neq f \end{cases}$$

Intuitively, $d_f(t)$ is the depth of the deepest occurrence of f in t (or 0). The function is continued to atoms, literals and clauses as follows:

$$\begin{aligned} d_f(s \simeq t) &= \max\{d_f(s), d_f(t)\} \\ d_f(s \dot{\simeq} t) &= d_f(s \simeq t) \\ d_f(l_1 \vee \dots \vee l_k) &= \max\{d_f(l_1), \dots, d_f(l_k)\} \end{aligned}$$

The feature functions defined by the following expressions are compatible with standard subsumption, subsumption modulo AC, and equality subsumption: $|C^+|$, $|C^-|$, $|C^+|_f$ (for all f), $|C^-|_f$ (for all f). The argument is essentially always the same: instantiation can only add new symbols, and a superset (super-multiset) or superstructure always contains at least as many symbols as the subset or substructure.

The feature functions defined by the following expressions are compatible with standard subsumption, set subsumption, and equality subsumption: $d_f(C^+)$ (for all f), $d_f(C^-)$ (for all f). The argument is similar: Instantiation can only introduce function symbols at new positions, never take them away at an existing depth.

If any two feature functions f_1 , f_2 are compatible with a certain subsumption type, then any linear combination of the two with non-negative coefficients is also compatible with that subsumption type. That is, $f(C) = af_1(C) + bf_2(C)$ with $a, b \geq 0$ is also a compatible feature function.

Many provers already use the criterion that a subsuming clause cannot have more function symbols than the subsumed one. In our notation, this can be described by the requirement that $\sum_{f \in F} |C|_f \leq \sum_{f \in F} |C'|_f$. This will, on average, already decide about half of all subsumption attempts. However, by looking at and combining more fine-grained criteria, we can do a lot better.

4.2 Clause Feature Vectors and Candidate Sets

Let π_n^i be the projection function for the i th element of a vector with n elements. A *clause feature vector function* is a function $F : \text{Clauses}(F, V) \rightarrow \mathbf{N}^n$. We call F subsumption-compatible (for a given subsumption type) if $\pi_n^i \circ F$ is a subsumption compatible feature for each $i \in \{1, \dots, n\}$. In other words, a subsumption compatible feature vector function combines a number of subsumption compatible feature functions. We will now assume that F is a subsumption-compatible feature function. If $F(C) = v$, we call v the feature vector of C .

We define a partial ordering \leq_s on vectors by $v \leq_s v'$ iff $\pi_n^i(v) \leq \pi_n^i(v')$ for all $i \in \{1, \dots, n\}$. By definition of the feature vector, if C subsumes C' ,

then $F(C) \leq_s F(C')$. This allows us to succinctly identify the candidate sets of clauses for forward- and backward subsumption. Let C be a clause and P be a clause set. Then

$$\text{candFS}_F(P, C) = \{c \in P \mid F(c) \leq_s F(C)\}$$

is a superset of all clauses in P that subsume C and

$$\text{candBS}_F(P, C) = \{c \in P \mid F(C) \leq_s F(c)\}$$

is a superset of all clauses in P that are subsumed by C . As our experiments show, if a reasonable number of clause features are used in the clause feature vector, these supersets are usually fairly small. Restricting subsumption attempts to members of a suitable candidate set reduces the number of attempts often by several orders of magnitude.

4.3 Index Data Structure

Whereas it is possible to store complete feature vectors with every clause in a set, this approach is rather inefficient in terms of memory consumption, and still requires the full comparison of all feature vectors. If, on the other hand, we compile feature vectors into a *trie*-like data structure, with all clauses sharing a vector stored at the corresponding leaf, large parts of the vectors are shared, and candidate sets can be computed much more efficiently.

Assume a (finite) set P of clauses with associated feature vectors $F(P)$ of length n . A *clause feature vector index* for P and F is a tree of uniform depth n (i.e., each path from the root to a leaf has length n). It can be recursively constructed as follows: If n is equal to 0, the tree consists of just a leaf node, which we associate with all clauses in P . Otherwise, let $D = \{\pi_n^1(F(C)) \mid C \in P\}$, let $P_i = \{C \mid \pi_n^1(F(C)) = i \mid i \in D\}$ (the set of all clause for which the first feature has a given value i , and let $F' = \langle \pi_n^2, \dots, \pi_n^n \rangle \circ F$ (shortening the original feature vectors by the first element). Then the index consist of a root node with successors T_i , such that each T_i is an index for P_i and F' . Inserting and deleting is linear in the number of features and independent of the number of elements in the index.

As an example, consider F defined by $F(C) = \langle |C^+|_a, |C^+|_f, |C^-|_b \rangle$, the clauses $C1 = p(a) \vee p(f(a))$, $C2 = p(a) \vee \neg p(b)$, $C3 = \neg p(a) \vee p(b)$, $C4 = p(X) \vee p(f(f(b)))$, and the set of clauses $P = \{C1, C2, C3, C4\}$. The feature vectors are as follows: $F(C1) = \langle 2, 1, 0 \rangle$, $F(C2) = \langle 1, 0, 1 \rangle$, $F(C3) = \langle 0, 0, 0 \rangle$, $F(C4) = \langle 0, 2, 0 \rangle$. Figure 2 shows the resulting index.

4.4 Forward Subsumption

For forward subsumption, we do not need to compute the full candidate set $\text{candFS}_F(P, C)$. Instead, we can just enumerate the elements and stop as soon as a subsuming clause is found. Assume a clause set P , a feature function F

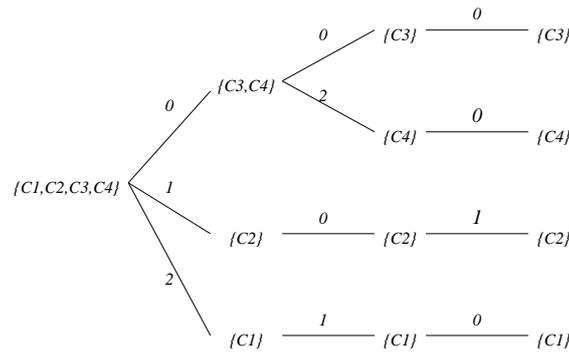


Fig. 2. Example of Clause Feature Vector Index

with feature vector length n , and an index I . We denote by $I[v]$ the subtree of I associated with value v . The clause to be subsumed is C . Figure 3a) shows the algorithm for indexed subsumption.

Note that it is trivial to return the subsuming clause (if any), instead of just a boolean value. We traverse the subtrees in order of increasing feature values, so that (statistically) smaller clauses with a higher chance of subsuming get tested first.

The subsumption test in the leaves of the tree is implemented by sequential search. In particular, finding the candidate sets and applying the actual subsumption test are clearly separated, i.e., it is trivially possible to use any subsumption concept as long as F is compatible with it.

4.5 Backward Subsumption

The algorithm for backward subsumption is quite similar, except that we traverse nodes with feature values greater than or equal to that of the subsuming clause, and that we cannot terminate the search early, since we have to find (and return) *all* subsumed clauses. We use the same conventions as above. Additionally, $mv(I)$ is the largest feature value associated with any subtree in I . Figure 3b) shows the algorithm.

4.6 Optimizing the Index Data Structure

Each leaf in the feature vector index corresponds to a given feature vector. If we ignore the internal structure of the trie, and the order of features in the vector, we can associate each leaf with an unordered set of tuples $(f, f(C))$ of individual feature functions and corresponding feature value. It is easy to see that any order of features in the feature vector will generate the same number of leaves, and that each leaf is either compatible with a given set of feature function/feature value tuples, or not. Thus, at least for a complete search as in the backward subsumption algorithm, we always have to visit the same number of leaves.

However, we can certainly minimize the internal number of nodes in the

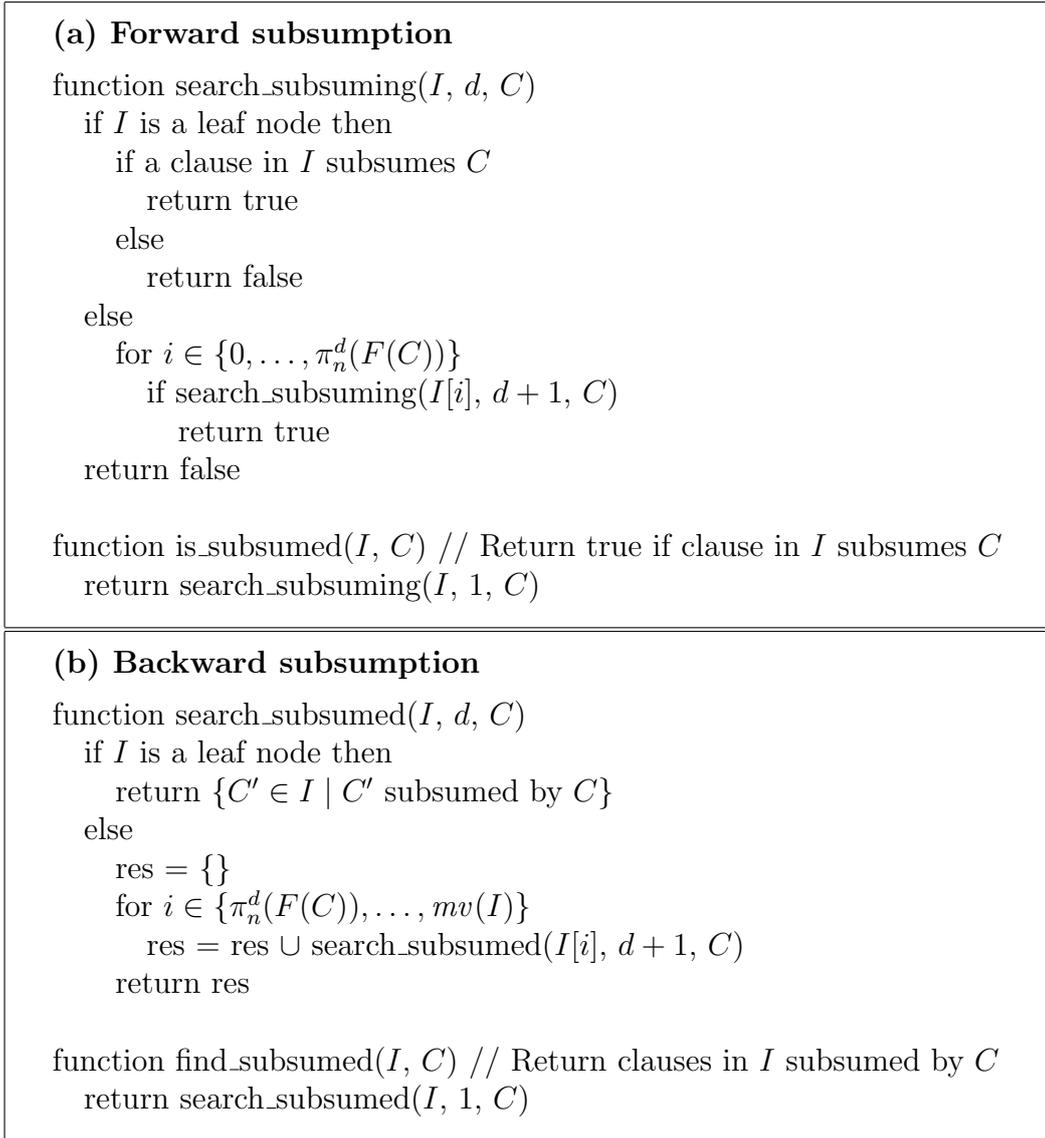


Fig. 3. Forward and backward subsumption with feature vector indexing

trie, and thus the total number of nodes. Consider for a simple example feature vectors with two features f_1, f_2 , where f_1 yields the same value for all clauses from a set P , whereas f_2 perfectly separates the set into n individual clauses. If we test f_1 first, our tree has just one internal node (plus the root). Traversing all leaves touches $n + 2$ nodes (counting the root). If on the other hand we evaluate the more informative f_2 first, we will immediately split the tree into n internal nodes, each of which has just one leaf as the successor. Thus, to traverse all leaves we would touch $2n + 1$ nodes, or, for a reasonably sized n , nearly twice as many nodes.

This example easily generalizes to longer vectors. In general, we want the least informative features first in a feature vector, so that as many initial paths as possible can be shared. This is somewhat surprising, since for most exclu-

sion tests it is desirable to have the most informative features first, so that impossible candidates are excluded early. Of course, if we have totally uninformative features, we can just as well drop them completely, thus shrinking the tree depth.

Unfortunately, we have to determine the feature vector function before we start building the index, i.e., in practice before the proof search starts. We can only estimate the informativeness of a given feature by looking at the distribution of its values in the initial clause set, and assume that this is typical for the later clauses.

For best results, we could view application of a feature function to a clause as a probability experiment and the results on the initial clause set as a sample. We could then sort features by increasing estimated entropy³ [15] or even conditional entropy. However, we decided to use a much simpler estimator first, namely the range of the feature value over the initial clause set. We have implemented three different mappings: *Direct mapping*, where the place of a feature in the vector is determined by the internal representation of function symbols used by the system, *permuted*, where features are sorted by feature value range, and *optimized permuted*, where additionally features with no estimated usefulness (i.e., features which evaluate to the same value for all initial clauses) are dropped off.

Our experimental results show that both permuted and optimized permuted feature vectors perform much better than direct mapped ones, with optimized permuted ones being best if we allow only a few features, whereas plain permuted ones gain if we allow more features. Generally, we can decrease the number of nodes in an index by about 50% using permuted feature vectors. We explain this behaviour by noting that the degree of informativeness is generally estimated correctly, but the prediction whether a feature will be useful at all is less precise. We have especially observed the situation that only a single negative literal occurs in the initial clause set (e.g. all unit-equational proof problems with a single goal), and hence all features restricted to negative literals have an initial range of zero, although a large and varied set of negative literals is generated during the proof search.

5 Implementation Notes

We have implemented clause feature vector indexing in our prover E, using essentially simple versions of standard trie algorithms for inserting and deleting feature vectors (and hence clauses), and the algorithms described in section 4.4 and 4.5 for forward and backward subsumption. We are using subsumption only between the set of processed clauses P and the given clause g and vice

³ The *entropy* of a probability experiment is the expected information gain from it, or, in other words, the expected cost of predicting the outcome. In our case, a feature with higher entropy splits the clause set into more (or more evenly distributed) parts. See e.g. [11] or, for a more comprehensive view, [4].

versa, but we have also implemented contextual literal cutting using the index. It can be optionally applied either to the newly generated clauses during simplification (using clauses from P for cutting) or between g and P , in both directions.

Feature vector indexing is used for forward and backward non-unit multiset subsumption, all versions of contextual literal cutting, (unit) equality backward subsumption, and backward simplify-reflect (equational unit cutting, see [12]) inferences. Forward equality subsumption and forward simplify-reflect have been implemented using discrimination tree indexing (on maximal terms in the unit clause used) since early versions of E.

Our standard multiset subsumption code, used both for conventional subsumption and to check indexed candidates for actual subsumption, already is fairly optimized. It uses a number of simple criteria to quickly determine unsuitable candidates, including tests based on literal- and symbol count, and individual literal matching. Only if all these tests succeed do we start the recursive permutation of terms and literals to find a common match.

The feature vector index is implemented in a fairly straightforward way, using a recursive data structure. Note that all our features in practice yield small integers. Hence we have implemented the mapping from a feature value to the subtree as a (dynamic) array. The only special case we support is the case that a node has exactly one successor. In this case we do not use an array, but just store the feature value and a pointer to the successor, to avoid the memory overhead of the array.

Clauses in a leaf node are stored in a simple set data structure (which is implemented throughout E as a *splay tree* [16] using pointers as keys). Empty subtrees are deleted eagerly.

It may be interesting to note that the first (and working) version of the indexing scheme took only about three (part-time) days to implement and integrate from scratch. It took approximately 7 more days to arrive at the current (production-quality) version that allows for a large number of different clause feature vector functions to be used and applies the index to many different operations. Compared to other indexing techniques, feature vector indexing seems to be easy to implement and easy to integrate into existing systems.

6 Experimental Results

We used all 5180 clause normal form problems from TPTP 2.5.1, without equality axioms, but otherwise unchanged. All test runs were performed on a cluster of 300MHz SUN Ultra-60 workstations with a time limit of 300 seconds (or equivalent configurations). The memory limit was 192 MB.

The indexed version of the prover uses a maximum feature vector length of 75. Features used in the vector are $|C^+|$, $|C^-|$, $|C^+|_f$ and $|C^-|_f$ (for some function symbols f). The vector might be shorter than 75 elements if only a

few symbols occur in the input formula. The results here were obtained with (plain) permuted feature vectors; features are ordered by increasing value range in the input clause set.

You can download detailed results of these and additional test runs at http://www.eprover.org/feature_vector_indexing.html.

6.1 Results with Aggressive Contextual Literal Cutting

Most of the strongest search strategies we found so far apply contextual unit cutting only to the given clause. To measure the effect of indexing for the worst-case scenario, we ran the system with a strong standard strategy, but with contextual literal cutting applied even to passive clauses, so that subsumption attempts are maximized. In this case, the version without feature vector indexing was able to solve 2671 problems within the time limit, whereas the prover with indexing solved 2717 problems (a strict superset). On the common subset, the indexed version needed 19857 s, whereas the plain system used 32140 s, for a speed-up of nearly 40% for the whole prover. For several harder examples, the indexed version ran as much as 5 times faster than the conventional prover.

Figure 4 shows the scatter plot of times for the conventional over the indexed version of the prover. Very few examples perform worse with indexing, and the drop is usually not very significant. A number of examples cluster around equal performance, and the majority shows moderate to large speed-ups.

We manually reran some of the few cases where the indexed version of the system was significantly slower than the non-indexed version on different hardware (Generic Intel Pentium-III/Pentium-4 PCs with GNU/Linux and a PowerPC G4 notebook with MacOS-X). In no case could we reproduce the slow-down (although it is reproducible on SUN hardware). Thus, we currently believe that it is caused by some unfortunate interplay between features of the SPARC architecture and our straight-forward recursive implementation. In particular, we suspect the large register window spills caused by deep recursions on SPARC.

Figure 5 shows how many direct clause-clause subsumption calls have been used by the conventional and the indexed version for the problems solved by both. Note the double logarithmic scale necessary to adequately display the large variation in numbers. The conventional version needs, over all problems, about 30 times more calls than the indexed version. For individual problems, the improvement factor varies from 1 (for some trivial problems) to approximately 7500 (for the problem SYN738-1, where the number of calls dropped from 22 552 to 3). The largest number of subsumption calls was observed for SYN711-1 with 444 793 509. For this problem, indexing reduced the number of calls by a factor of nearly 200 to 2 229 754, and the run time from 235s to 79s.

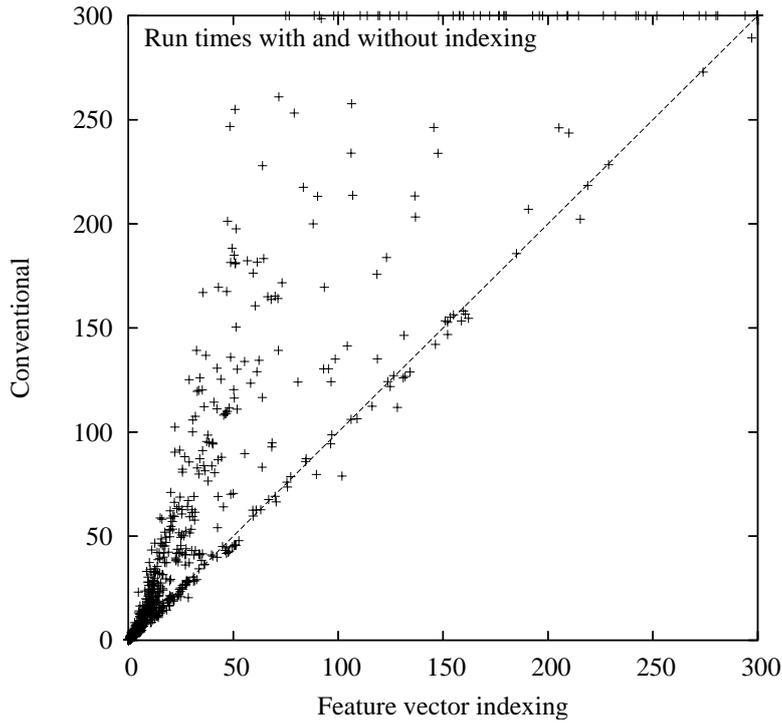


Fig. 4. Run times of indexed versus conventional implementation for aggressive (CLC)

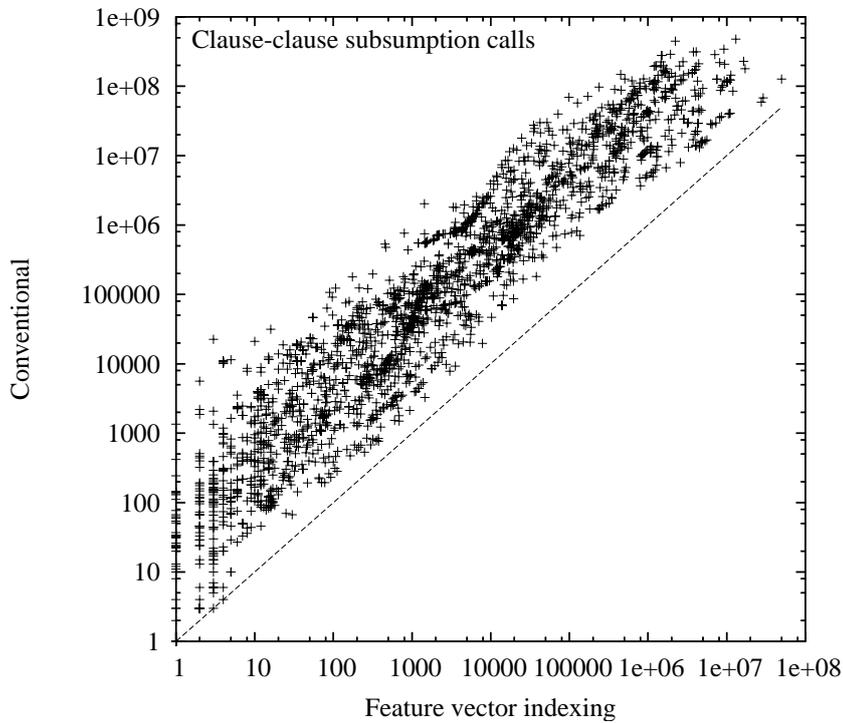


Fig. 5. Subsumption calls of indexed versus conventional implementation

6.2 Results for the Automatic Mode

Most users use E in *automatic mode*, where the prover analyses the problem, and then configures itself to use a strategy that has performed well on similar problems. In this mode, contextual literal cutting is usually only applied to the given clause, and thus the overall cost of subsumption-related techniques is lower to begin with. We have performed various experiments to measure the effect of feature vector indexing for this scenario as well. Figure 6 compares the run times of E with and without feature vector indexing using the automatic mode included with E 0.8. The conventional version solves 3405 problems, whereas the version using feature vector indexing proves a superset of 3438 problems. On the subset solved by both systems, the indexed version uses 34438 s, whereas the conventional one uses 40238 s, for a speed-up of about 15%.

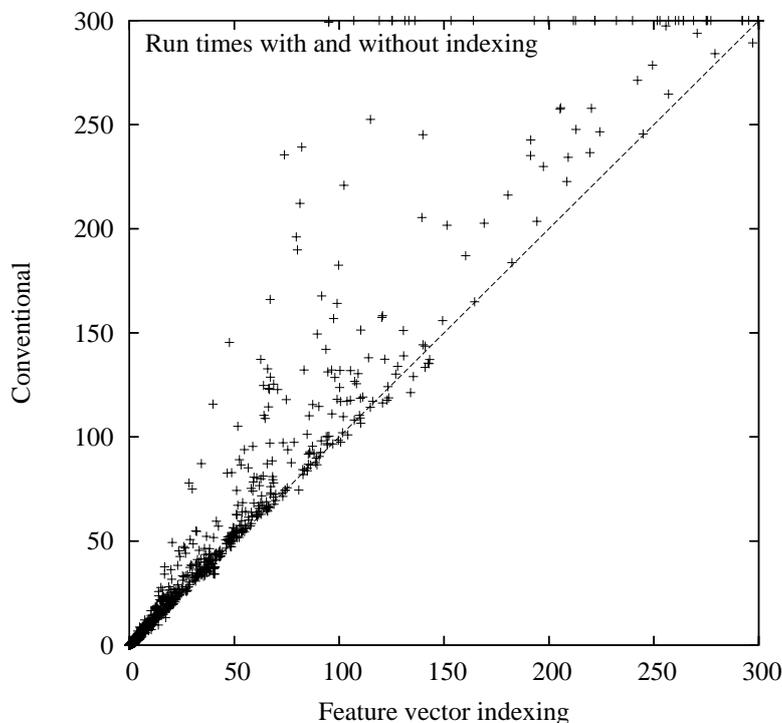


Fig. 6. Run times of indexed versus conventional implementation for E 0.8 automatic mode

The plot is similar to Figure 4, with many problems, especially for lower run times, showing similar performance for both versions, and another group showing significant improvement. Of course, since overall subsumption cost is lower for the automatic mode, the gains are not as pronounced as for the case with aggressive contextual literal cutting.

7 Future Work

While we are very satisfied with the performance of our current implementation, there are a number of research problems we are actively working on.

First, with increased speed of hardware, it has become very hard to actually quantify the time spent in different parts of a high-performance prover using standard UNIX profiling tools (which only resolve to the 1/100th second level). Thus, in this paper we only compare overall performance of the system with and without feature vector indexing. We are in discussions with some of the authors of the COMPIT framework [8] to extend it to cover (unit and non-unit) subsumption, so that more detailed measurements and a comparison of different indexing techniques become easier.

Secondly, up to now, we have only experimented with some simple and obvious features. In particular, all of the features used with our currently best parameter settings are AC compatible, and hence will not differentiate between clauses that are equal modulo AC theories. Whereas this is desirable if subsumption modulo AC is used, it is a disadvantage for our calculus, which only allows us to use some limited AC redundancy elimination. We will investigate the effect of more complex features in these cases.

Finally, we are trying to develop similar simple, but effective algorithms for other operations in the system, in particular for backward simplification (where it might be possible to use a slightly modified version of feature vector indexing) and paramodulation.

8 Conclusion

Feature vector indexing has proved to be a simple, but effective answer to the subsumption problem for saturating first-order theorem provers. In our experiments, it is able to reduce the number of subsumption tests by, on average, about 97% compared to a naive sequential implementation, and thus reduces cost for subsumption in our prover to a level that makes it hard to measure using standard UNIX profiling tools.

In addition to the direct benefit, this gain in efficiency has enabled us to implement otherwise relatively expensive subsumption-based simplification techniques (like contextual literal cutting), further improving overall performance of our prover.

References

- [1] Denzinger, J., M. Kronenburg and S. Schulz, *DISCOUNT: A Distributed and Learning Equational Prover*, Journal of Automated Reasoning **18** (1997), pp. 189–198, special Issue on the CADE 13 ATP System Competition.
- [2] Graf, P., “Term Indexing,” LNAI **1053**, Springer, 1995.

- [3] Graf, P. and D. Fehrer, *Term Indexing*, in: W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, Applied Logic Series **9 (2)**, Kluwer Academic Publishers, 1998 pp. 125–147.
- [4] Jaynes, E., “Probability Theory: The Logic of Science,” Cambridge University Press, 2003.
- [5] Kapur, D. and P. Narendran, *NP-Completeness of the Set Unification and Matching Problems*, in: J. Siekmann, editor, *Proc. of the 8th CADE, Oxford*, LNCS **230** (1986), pp. 489–495.
- [6] Löchner, B. and T. Hillenbrand, *A Phytophagy of Waldmeister*, Journal of AI Communications **15** (2002), pp. 127–133.
- [7] McCune, W., *Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval*, Journal of Automated Reasoning **9** (1992), pp. 147–167.
- [8] Nieuwenhuis, R., T. Hillenbrand, A. Riazanov and A. Voronkov, *On the Evaluation of Indexing Techniques for Theorem Proving*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, LNAI **2083** (2001), pp. 257–271.
- [9] Riazanov, A. and A. Voronkov, *The Design and Implementation of VAMPIRE*, Journal of AI Communications **15** (2002), pp. 91–110.
- [10] Riazanov, A. and A. Voronkov, *Efficient Instance Retrieval With Standard and Relational Path Indexing*, in: F. Bader, editor, *Proc. of the 19th CADE, Miami*, LNAI **2741** (2003), pp. 380–396.
- [11] Schulz, S., *Information-Based Selection of Abstraction Levels*, in: I. Russel and J. Kolen, editors, *Proc. of the 14th FLAIRS, Key West* (2001), pp. 402–406.
- [12] Schulz, S., *E – A Brainiac Theorem Prover*, Journal of AI Communications **15** (2002), pp. 111–126.
- [13] Schulz, S., *System Description: E 0.81*, in: *Proc. of the 2nd IJCAR, Cork, Ireland*, LNAI, 2004, (to be published).
- [14] Sekar, R., I. Ramakrishnan and A. Voronkov, *Term Indexing*, in: A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, II*, Elsevier Science and MIT Press, 2001 pp. 1853–1961.
- [15] Shannon, C. and W. Weaver, “The Mathematical Theory of Communication,” University of Illinois Press, 1949.
- [16] Sleator, D. and R. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM **32** (1985), pp. 652–686.
- [17] Tammet, T., *Towards Efficient Subsumption*, in: C. Kirchner and H. Kirchner, editors, *Proc. of the 15th CADE, Lindau*, LNAI **1421** (1998), pp. 427–441.
- [18] Voronkov, A., *The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees*, Journal of Automated Reasoning **15** (1995), pp. 238–265.

- [19] Weidenbach, C., *SPASS: Combining Superposition, Sorts and Splitting*, in: A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, II*, Elsevier Science and MIT Press, 2001 pp. 1965–2013.

Things to know when implementing LPO

Bernd Löchner¹

*FB Informatik
Technische Universität Kaiserslautern
Kaiserslautern, Germany*

Abstract

The Lexicographic Path Ordering (LPO) poses an interesting problem to the implementor: How to achieve a version that is both efficient and correct? The method of program transformation helps us to develop an efficient version step-by-step, making clear the essential ideas, while retaining correctness. By theoretical analysis we show that the worst-case behavior is thereby changed from exponential to polynomial. Detailed measurements show the practical improvements of the different variants. They allow us to assess experimentally various optimizations suggested for LPO.

Key words: Lexicographic Path Ordering, implementation, program transformation.

1 Introduction

For the implementation of a practically successful prover it is important to have efficient implementations of the most time-consuming subtasks. Whereas much research has been spent, for example, into the development of efficient indexing techniques (see [RSV01] for an overview), the implementation of orderings has received far less attention (a recent exception is [RV04]). However, the time spent on determining ordering relations between terms can amount to a significant part of the prover's overall running time. For example, WALDMEISTER [LH02] needs up to 50 % of the total running time for ordering comparisons, although much effort has been spent into the optimization of the corresponding routines and many ordering comparisons are avoided by using unorientable equations only in a very restricted form for rewriting [BH96].

The aim of this paper is the development of an efficient version of the *Lexicographic Path Ordering* (LPO) [KL80], which is one of the orderings in widespread use. Several ideas lead from the direct translation of the usual definition, which has an exponential behavior, to a non-obvious implementation

¹ Email: loechner@informatik.uni-kl.de

with polynomial requirements. Our approach is based on the methodology of *program transformations*, which amounts in this case to the following: We formulate the programs in a language that is close to functional programming or algebraic specification. This concise and abstract notation allows us to focus on the essential ideas in the development of the efficient version. We start with some “obviously correct” implementation, which is as close as possible to the original definition of the LPO. Then we refine the implementation in several small steps. The equivalence of two successive versions can be shown by proof with induction. As the used language can easily be translated into the input language of some inductive theorem prover, these proofs can essentially be performed within such a system. This is more elaborate than manual proofs, but prevents the introduction of bugs by avoiding oversights.

To measure the progress between the different versions we translate them in a straightforward way into the programming language C and integrate them into WALDMEISTER. This allows us to test and compare them on thousands of test cases occurring in real proof-attempts and shows the impact of the different optimizations on a real prover running on real hardware.

2 Preliminaries

We use standard concepts from term rewriting (see e. g. [DP01]). The set $\text{Term}(\mathcal{F}, \mathcal{V})$ denotes the set of (first-order) terms built over the set of function symbols \mathcal{F} and the set of variables \mathcal{V} . The length $|t|$ of a term t is the number of function symbols and variables it contains. The set of variables occurring in term t is written as $\text{Var}(t)$.

We describe the different implementations in a small algebraic specification language. For boolean values it has a sort **Bool** with the two constructors **true**, **false** $:\rightarrow \text{Bool}$. With their help we define disjunction and conjunction:

$$\begin{array}{ll} \vee : \text{Bool Bool} \rightarrow \text{Bool} & \wedge : \text{Bool Bool} \rightarrow \text{Bool} \\ b_1 \vee b_2 = \mathbf{if } b_1 \mathbf{ then true else } b_2 & b_1 \wedge b_2 = \mathbf{if } b_1 \mathbf{ then } b_2 \mathbf{ else false} \end{array}$$

This asymmetric definition expresses that the second argument is not evaluated if the result is already determined by the first. If all arguments are defined, both operations have the AC-property, i. e., the associativity and commutativity laws hold:

$$\begin{array}{ll} (b_1 \vee b_2) \vee b_3 = b_1 \vee (b_2 \vee b_3) & b_1 \vee b_2 = b_2 \vee b_1 \\ (b_1 \wedge b_2) \wedge b_3 = b_1 \wedge (b_2 \wedge b_3) & b_1 \wedge b_2 = b_2 \wedge b_1 \end{array}$$

We assume the availability of the sorts **Vid** and **Fid** to represent variables \mathcal{V} and function symbols \mathcal{F} . Then we define the datatype **Term** via two constructors using an additional sort **Termlist** for lists of terms.

$$\begin{array}{ll} \text{V} : \text{Vid} \rightarrow \text{Term} & [] : \rightarrow \text{Termlist} \\ \text{F} : \text{Fid Termlist} \rightarrow \text{Term} & \bullet : \text{Term Termlist} \rightarrow \text{Termlist} \end{array}$$

Note that for a term $F(f, ts)$ there is no relationship encoded in the datatype between the arity of f and the length of ts . It is therefore possible to construct **Terms** that are not well-formed, i. e., do not represent elements of $\text{Term}(\mathcal{F}, \mathcal{V})$. To distinguish well-formed terms, we define the following predicates (or rather boolean valued functions). We assume the availability of the functions $\text{arity} : \text{Fid} \rightarrow \text{Nat}$ and $\text{length}_{\text{tl}} : \text{Termlist} \rightarrow \text{Nat}$.

$$\begin{array}{ll} \text{well} : \text{Term} \rightarrow \text{Bool} & \text{well}_{\text{tl}} : \text{Termlist} \rightarrow \text{Bool} \\ \text{well}(V(x)) = \text{true} & \text{well}_{\text{tl}}([\]) = \text{true} \\ \text{well}(F(f, ts)) = \text{arity}(f) = \text{length}_{\text{tl}}(ts) & \text{well}_{\text{tl}}(t \bullet ts) = \text{well}(t) \wedge \text{well}_{\text{tl}}(ts) \\ & \wedge \text{well}_{\text{tl}}(ts) \end{array}$$

The explicit use of the predicates is necessary in an inductive prover, which does not allow implicit assumptions, such as the requirement that all terms are well-formed. The definition of the helper function well_{tl} lifts well from **Term** to **Termlist**. The alternative is the use of higher-order functions which would complicate the translation into the input language of some (first-order) inductive prover. This pattern of some function over **Term** and some helper function over **Termlist**, which call each other recursively, can also be observed in the next definition.

The function **contains** tests whether a variable symbol occurs in a term.

$$\begin{array}{ll} \text{contains} : \text{Term Vid} \rightarrow \text{Bool} & \text{contains}_{\text{tl}} : \text{Termlist Vid} \rightarrow \text{Bool} \\ \text{contains}(V(x), y) = x = y & \text{contains}_{\text{tl}}([\], y) = \text{false} \\ \text{contains}(F(f, ts), y) = \text{contains}_{\text{tl}}(ts, y) & \text{contains}_{\text{tl}}(t \bullet ts, y) = \text{contains}(t, y) \\ & \vee \text{contains}_{\text{tl}}(ts, y) \end{array}$$

Obviously, the worst-case running time of **contains** and $\text{contains}_{\text{tl}}$ is linear in the size of the first argument.

3 The reference implementation

Let us first recall the definition of the LPO which requires \mathcal{F} to contain function symbols of fixed arity only.

Definition 3.1 Let $>_{\mathcal{F}}$ be an ordering on \mathcal{F} and $s, t \in \text{Term}(\mathcal{F}, \mathcal{V})$. Then $s \succ_{\text{lpo}} t$ iff either $s \equiv f(s_1, \dots, s_n)$, $t \equiv g(t_1, \dots, t_m)$, and

- (α) $s_i \succ_{\text{lpo}} t$ for some $i \in \{1, \dots, n\}$ or
- (β) $f >_{\mathcal{F}} g$ and $s \succ_{\text{lpo}} t_k$ for all $k \in \{1, \dots, m\}$ or
- (γ) $f = g$, there exists some $i \in \{1, \dots, m\}$ such that $s_j = t_j$ for all $j \in \{1, \dots, i-1\}$ and $s_i \succ_{\text{lpo}} t_i$, and $s \succ_{\text{lpo}} t_k$ for all $k \in \{1, \dots, m\}$

or $s \equiv f(s_1, \dots, s_n)$, $t \equiv x$, and

- (δ) $x \in \text{Var}(s)$,

where $u \succ_{\text{lpo}} v$ iff $u \equiv v$ or $u \succ_{\text{lpo}} v$ for $u, v \in \text{Term}(\mathcal{F}, \mathcal{V})$.

Proposition 3.2 *The LPO \succ_{lpo} for precedence $>_{\mathcal{F}}$ is a reduction ordering on $\text{Term}(\mathcal{F}, \mathcal{V})$. The subterm relation \succ_{st} is embedded in \succ_{lpo} , i. e., $\succ_{\text{st}} \subset \succ_{\text{lpo}}$. \square*

The following program tries to follow this definition as close as possible. We call it the *reference implementation*.² Function $>_{\text{Fid}}$ represents the precedence.

```

lpo1 : Term Term → Bool
lpo1(F(f, ss), F(g, ts)) = alpha1(ss, F(g, ts)) ∨ beta1(F(f, ss), F(g, ts))
                               ∨ gamma1(F(f, ss), F(g, ts))
lpo1(F(f, ss), V(y)) = delta1(F(f, ss), V(y))
lpo1(V(x), t) = false

alpha1 : Termlist Term → Bool
alpha1([], t) = false
alpha1(s . ss, t) = s = t ∨ lpo1(s, t) ∨ alpha1(ss, t)

beta1 : Term Term → Bool
beta1(F(f, ss), F(g, ts)) = f >Fid g ∧ majo1(F(f, ss), ts)

gamma1 : Term Term → Bool
gamma1(F(f, ss), F(g, ts)) = f = g ∧ lex1(ss, ts) ∧ majo1(F(f, ss), ts)

delta1 : Term Term → Bool
delta1(F(f, ss), V(y)) = containstl(ss, y)

majo1 : Term Termlist → Bool
majo1(s, []) = true
majo1(s, t . ts) = lpo1(s, t) ∧ majo1(s, ts)

lex1 : Termlist Termlist → Bool
lex1([], []) = false
lex1(s . ss, t . ts) = if s = t then lex1(ss, ts) else lpo1(s, t)

```

The functions beta_1 , gamma_1 , and delta_1 are not strictly necessary; they are introduced for clarity reasons. This is in contrast to the function alpha_1 , which calls itself tail-recursively. The function majo_1 implements the common part of the cases β and γ in the definition, namely $s \succ_{\text{lpo}} t_k$ for all $k \in \{1, \dots, m\}$. The lexicographic comparison is provided by lex_1 , which is only defined for lists of the same length. This partiality reflects that LPO requires function symbols of fixed arity. Therefore, lpo_1 is only defined for well-formed terms, i. e., arguments fulfilling the *well* predicate.

4 First optimizations

A quick analysis reveals that it is not advisable to evaluate the case α before the cases β or γ . The intuitive explanation is that case α reduces the size of the left argument, while the right argument remains the same. This seems disadvantageous for showing the left argument to be greater than the right

² The subscript 1 denotes that this is the first version of the implementation. It will increase subsequently.

argument. Hence, reducing the size of the left argument should be delayed.³ Our second version therefore swaps the order of the cases α , β , and γ :

$$\begin{aligned} \text{lpo}_2 & : \text{Term Term} \rightarrow \text{Bool} \\ \text{lpo}_2(\text{F}(f, ss), \text{F}(g, ts)) & = \text{beta}_2(\text{F}(f, ss), \text{F}(g, ts)) \vee \text{gamma}_2(\text{F}(f, ss), \text{F}(g, ts)) \\ & \quad \vee \text{alpha}_2(ss, \text{F}(g, ts)) \\ \text{lpo}_2(\text{F}(f, ss), \text{V}(y)) & = \text{delta}_2(\text{F}(f, ss), \text{V}(y)) \\ \text{lpo}_2(\text{V}(x), t) & = \text{false} \end{aligned}$$

The functions alpha_2, \dots , are identical to alpha_1, \dots , except that they call the $(\cdot)_2$ -versions, in particular lpo_2 .

Lemma 4.1 *If $s, t \in \text{Term}$ are well-formed then $\text{lpo}_1(s, t) = \text{lpo}_2(s, t)$.*

Proof. As under these conditions all functions are defined for their arguments this is a consequence of the AC-property of \vee . \square

A second observation concerns case γ . If there is some i such that $s_i \succ_{\text{lpo}} t_i$ and $s_j = t_j$ for all $j < i$ the comparisons $s \succ_{\text{lpo}} t_k$ for $1 \leq k \leq i$ are superfluous. This is a corollary of the following lemma.

Lemma 4.2 *If $s \succ_{st} s'$ and $s' \succ_{\text{lpo}} t$ then $s \succ_{\text{lpo}} t$.*

Proof. The LPO contains the subterm relation and is transitive. \square

To take advantage of this observation we modify the function gamma to use the newly defined function lexM which combines the lexicographic comparison with the call to majo : $\text{lexM}(s, ss, ts) = \text{lex}(ss, ts) \wedge \text{majo}(s, ts)$ for $s = \text{F}(f, ss)$. We get⁴

$$\begin{aligned} \text{gamma}_3 & : \text{Term Term} \rightarrow \text{Bool} \\ \text{gamma}_3(\text{F}(f, ss), \text{F}(g, ts)) & = f = g \wedge \text{lexM}_3(\text{F}(f, ss), ss, ts) \\ \text{lexM}_3 & : \text{Term Termlist Termlist} \rightarrow \text{Bool} \\ \text{lexM}_3(s, [], []) & = \text{false} \\ \text{lexM}_3(s, s_i \cdot ss, t_i \cdot ts) & = \text{if } s_i = t_i \text{ then } \text{lexM}_3(s, ss, ts) \\ & \quad \text{else } \text{lpo}_3(s_i, t_i) \wedge \text{majo}_3(s, ts) \end{aligned}$$

Corollary 4.3 *If $s, t \in \text{Term}$ are well-formed then $\text{lpo}_2(s, t) = \text{lpo}_3(s, t)$.* \square

5 First comparisons

A performance analysis shows that the two optimizations have a profound effect on the running time. The three versions of LPO are translated into

³ Of course, case γ reduces also the size of the left argument, but, in contrast to case α , it reduces in addition the size of the right argument.

⁴ As above, the remaining functions alpha_3, \dots , are identical to the $(\cdot)_2$ -counterparts except for calling the new versions of the functions.

problem	number of calls to ordering	time needed by ordering			time needed by other operations
		lpo_1	lpo_2	lpo_3	
GRP180-1	161 446	896.000	255.600	226.800	0.840
LAT020-1	829 439	9.170	6.232	4.994	12.230
LCL109-6	141 266	6.539	3.532	2.355	0.930
RNG027-5	148 787	335.700	161.500	129.800	2.350
<i>z22</i>	2 249	12.260	0.495	0.349	0.060
TPTP ₁₀	138 399 908	2 450.000	1 922.000	1 631.000	3 032.000

Table 1
Time needed (in seconds) for ordering comparisons

C functions in a straightforward way⁵ and are integrated into the theorem prover WALDMEISTER. As test examples we choose proof tasks with different characteristics. The first four examples are taken from TPTP [SS97]. The example *z22* is an encoding of a string-rewriting system as a term-rewriting system. Therefore, there are only unary function symbols and the terms are very deep and narrow with a variable at the leaf. Such examples seem to be absent in TPTP.

Furthermore, we make measurements for all 776 UEQ-problems of TPTP-2.6.0. For that, we first run WALDMEISTER using lpo_1 with a time limit of 10 seconds and record how often the ordering routines are called. We then perform the real measurements by running the prover with the different versions of LPO and aborting the run after the corresponding number of calls is reached. We use TPTP₁₀ to refer to the summarized results.

Table 1 contains running times measured on a machine with 1 GHz Pentium III and 4 GByte RAM. For each example we give the number of calls the prover makes to the ordering routines. The next three columns contain the time spent in deciding the ordering relations. The last column gives the time needed by the system *without* the time needed for the orderings.⁶ Note the diversity of the examples which can be seen for example in the large differences of the average time needed for one call to the ordering. The running times document that especially the first optimization leads to substantial improvements over the reference implementation. Nevertheless, the time spent in the ordering is in most cases considerably larger than the time for the rest of the system. TPTP₁₀ shows different results for two reasons. First, the start over-

⁵ The tail-recursions used in some functions, such as in `alpha`, `lex`, etc., are turned into `while`-loops.

⁶ This was determined by running the prover twice. In the first run each result of a call to the ordering was written to a file. In the second run instead of calling the ordering procedure the value of the protocol file was returned. By careful low-level coding we could achieve this without any noticeable overhead, as profiles show.

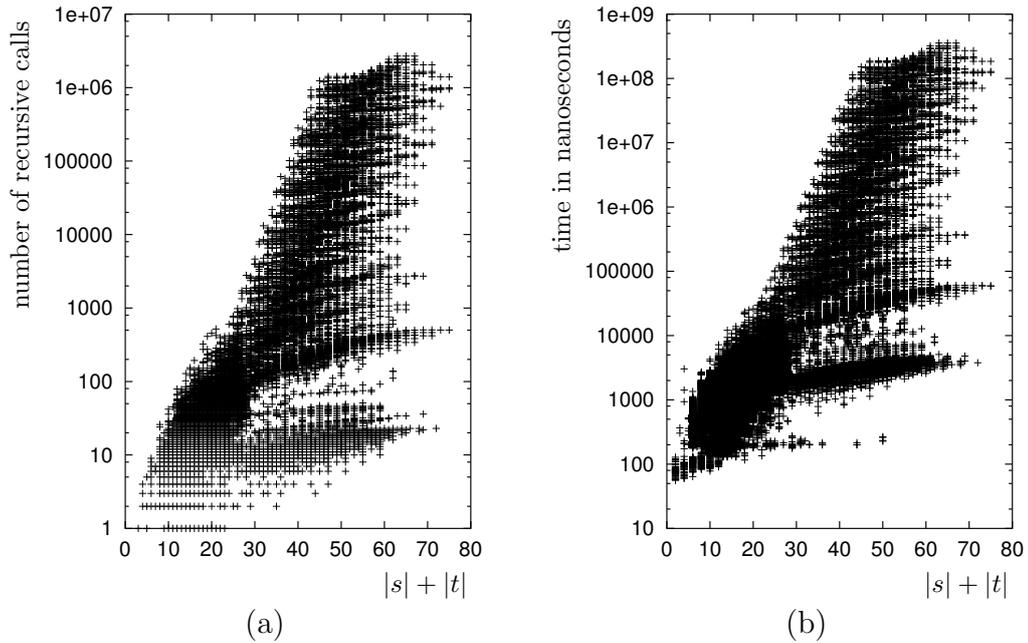


Fig. 1. Distribution of (a) number of recursive calls and (b) time needed to compute $\text{lpo}_3(s, t)$ depicted over $|s| + |t|$ for GRP180-1.

head for initializing the prover is emphasized by running at most 10 seconds, which shows in the last column. Second, the size of terms tends to increase during the completion, hence larger input sizes occur more rarely in TPTP_{10} than in completed runs. This reduces the differences observed between the variants of LPO.

Why does the prover spend so much time in LPO? Fine-grained measurements reveal the following: A single call to the ordering may require a number of recursive calls that is exponential in the size of the input. This can be seen in Fig. 1a. The distribution of running times of single calls reflects the pattern, cf. Fig. 1b. The following example shows the reason for this behavior, and that the behavior is independent of the used term data structure.

Example 5.1 Consider two constants $a, b \in \mathcal{F}$ and two unary function symbols $f, g \in \mathcal{F}$ with precedence $a >_{\mathcal{F}} b >_{\mathcal{F}} f >_{\mathcal{F}} g$. Let $u_n \equiv f^n(b)$ and $v_m \equiv g^m(a)$. Then the implementations lpo_1 , lpo_2 , and lpo_3 all show exponential behavior for the test whether $u_n \succ_{\text{lpo}}^? v_n$. As $v_n \succ_{\text{lpo}} u_n$ holds, all recursive calls return **false**. Therefore, in each invocation the cases α and β are considered both. The situation for $n = 3$ is illustrated⁷ in Fig. 2.

Now the source for the exponential behavior becomes clear: For $u_{n+1} \succ_{\text{lpo}}^? v_{m+1}$ the subproblem $u_n \succ_{\text{lpo}}^? v_m$ is considered twice. Figure 3a shows how often each subproblem is considered starting from $n = m = 3$. Obviously, this is the top part of Pascal's triangle of the binomial coefficients.

⁷ Strictly speaking one has to consider $s \succ_{\text{lpo}}^? t$ in the recursive calls via α . But this reduces essentially to $s \succ_{\text{lpo}}^? t$, because $s \neq t$ can be determined immediately.

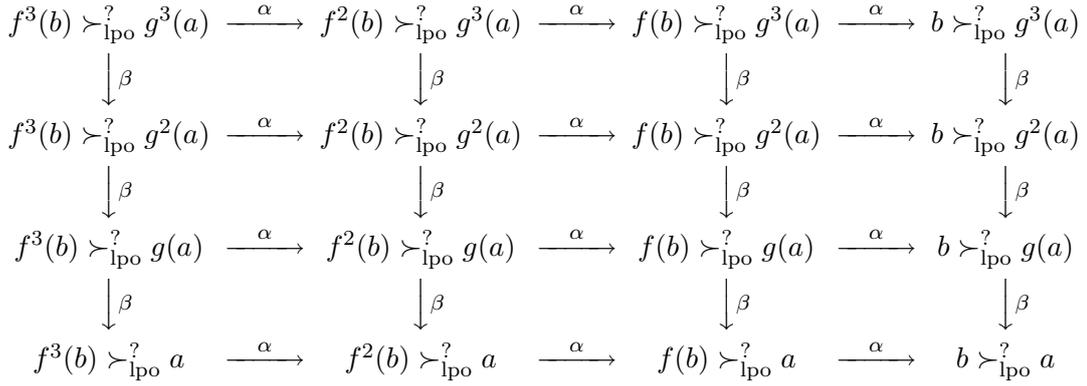


Fig. 2. Calling graph for test of $f^3(b) \succ_{\text{lpo}}^? g^3(a)$.

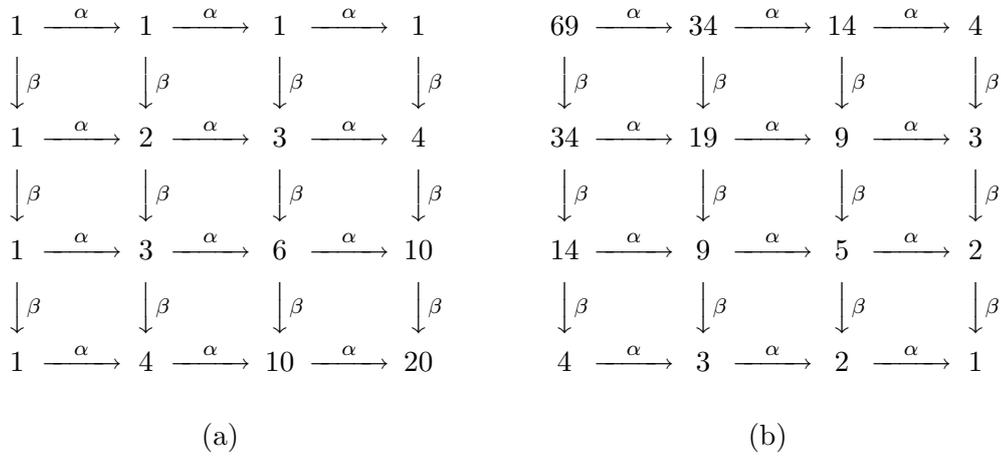


Fig. 3. (a) Number of invocations of subproblems during the test $f^3(b) \succ_{\text{lpo}}^? g^3(a)$. (b) Costs $T(n, m)$ associated with each subproblem of $f^3(b) \succ_{\text{lpo}}^? g^3(a)$.

To determine the costs $T(n, m)$ of testing $u_n \succ_{\text{lpo}}^? v_m$ we count the number of recursive calls. For $n, m \in \{0, \dots, 3\}$ the values of $T(n, m)$ are depicted in Fig. 3b. We get the following recurrences:

$$\begin{aligned}
 T(n, 0) &= n + 1 \\
 T(0, m) &= m + 1 \\
 T(n + 1, m + 1) &= 1 + T(n + 1, m) + T(n, m + 1)
 \end{aligned}$$

Here too, we have a connection to the binomial coefficients:

$$T(n, m) = \binom{n + m + 2}{m + 1} - 1$$

This can be shown via induction on $n + m$:

$$\begin{aligned}
 T(n, 0) &= n + 1 = \binom{n + 2}{1} - 1 \\
 T(0, m) &= m + 1 = \binom{m + 2}{m + 1} - 1 \\
 T(n + 1, m + 1) &= 1 + T(n + 1, m) + T(n, m + 1) \\
 &= 1 + \binom{(n + 1) + m + 2}{m + 1} - 1 + \binom{n + (m + 1) + 2}{(m + 1) + 1} - 1 \\
 &= \binom{(n + 1) + (m + 1) + 2}{(m + 1) + 1} - 1
 \end{aligned}$$

Using Stirling's formula we get the asymptotic expression

$$T(n, n) \approx \frac{4^{n+1}}{\sqrt{\pi(n+1)}}$$

which clearly describes an exponential growth. \square

For such situations there are two standard techniques to avoid the exponential behavior. The first is *dynamic programming* [Bel57]. Here a table is introduced which contains one entry for each subproblem. The top-down recursion is replaced by a bottom-up computation, which fills each table entry with the help of already determined ones. In our case, we assign in a leftmost-outermost way each subterm of a term t a unique number from $1, \dots, |t|$. Hence, a table of size $|s| \cdot |t|$ is sufficient for $\mathbf{lpo}(s, t)$. For computing the entries from large indices downwards to small indices, the work for each entry is determined by the arity of the leading function symbols – the ordering relationships between subterms are already known. This leads to an algorithm with running time in $\Theta(|s| \cdot |t|)$. But the structure of the algorithm is changed in a nontrivial way.

For the second technique, *memoization* [Mic68], the top-down recursion is retained. The result of each function call is stored in a so-called memo-table. When the function is called again with the same arguments, the value from the table is returned and so the recomputation is avoided. This can be thought of as using the table of dynamic programming, but filling in the values of the table lazily. A more typical implementation is with a hash table. The running time of the resulting algorithm is $O(|s| \cdot |t|)$ in the worst case, recomputations do not occur. If only a fraction of the $|s| \cdot |t|$ subproblems is needed, this method is faster than dynamic programming. If (nearly) all subproblems are considered, dynamic programming is probably faster; the code is simpler and avoids the overhead of hashing, especially the equality tests for the arguments.

In the literature [Sny93] the use of dynamic programming is recommended to get a polynomial algorithm. Our aim is to achieve the same worst-case running time without the use of additional data structures.

6 A polynomial version

As a preparation step we first replace the conjunctions in **beta** and **gamma** by **if**-expressions using the comparisons of function symbols as guards:

$$\begin{aligned} \text{beta}(F(f, ss), F(g, ts)) &= \mathbf{if} \ f >_{\text{Fid}} \ g \ \mathbf{then} \ \text{majo}(F(f, ss), ts) \ \mathbf{else} \ \text{false} \\ \text{gamma}(F(f, ss), F(g, ts)) &= \mathbf{if} \ f = g \ \mathbf{then} \ \text{lexM}(F(f, ss), ss, ts) \ \mathbf{else} \ \text{false} \end{aligned}$$

Then they are inlined into **lpo** and combined into one **if-elif**-expression as the guards are mutually exclusive:

$$\text{lpo}(F(f, ss), F(g, ts)) = \left(\begin{array}{l} \mathbf{if} \ f >_{\text{Fid}} \ g \ \mathbf{then} \ \text{majo}(F(f, ss), ts) \\ \mathbf{elif} \ f = g \ \mathbf{then} \ \text{lexM}(F(f, ss), ss, ts) \\ \mathbf{else} \ \text{false} \end{array} \right) \vee \text{alpha}(ss, F(g, ts))$$

Finally, we distribute **alpha** into the different branches and simplify:

$$\begin{aligned} \text{lpo}(F(f, ss), F(g, ts)) &= \mathbf{if} \ f >_{\text{Fid}} \ g \ \mathbf{then} \ \text{majo}(F(f, ss), ts) \vee \text{alpha}(ss, F(g, ts)) \\ &\quad \mathbf{elif} \ f = g \ \mathbf{then} \ \text{lexM}(F(f, ss), ss, ts) \vee \text{alpha}(ss, F(g, ts)) \\ &\quad \mathbf{else} \ \text{alpha}(ss, F(g, ts)) \end{aligned}$$

The performed transformations are typical for an optimizing compiler and do not change the time consumption of the implementation significantly.

The first insight leading to real improvements is that if $\text{majo}(F(f, ss), ts) = \text{false}$, then $\text{alpha}(ss, F(g, ts)) = \text{false}$ as well. This means that in contrast to the previous optimizations we can make use of *negative* results from recursive calls. The optimization is justified by the following lemma, which is formulated positively (i. e., as contraposition of the observation):

Lemma 6.1 *Let $s \equiv f(s_1, \dots, s_n)$ and $t \equiv g(t_1, \dots, t_m)$. If there is some $i \in \{1, \dots, n\}$ such that $s_i \not\geq_{\text{lpo}} t$ then $s \succ_{\text{lpo}} t_j$ for all $j \in \{1, \dots, m\}$.*

Proof. Because LPO contains the subterm relation, this is a consequence of transitivity. \square

This enables us to omit the call to **alpha** in the first branch of the **if**-expression. For the second branch we combine **lexM** and **alpha** into **lexMA** for $s = F(f, ss)$ and $t = F(f, ts)$:

$$\text{lexMA}(s, t, ss, ts) = \text{lexM}(s, ss, ts) \vee \text{alpha}(ss, t)$$

Performing the **alpha**-test is useless for subterms shown equal by the lexicographic test:

Lemma 6.2 *Let $s \equiv f(s_1, \dots, s_n)$ and $t \equiv f(t_1, \dots, t_n)$. If $s_i \equiv t_i$ for some $i \in \{1, \dots, n\}$, then $s_i \not\geq_{\text{lpo}} t$.*

Proof. Clear, as we have $t \succ_{\text{lpo}} t_i \equiv s_i$. \square

This leads to the following version of **lexMA**:

$$\begin{aligned} \text{lexMA}(s, t, [], []) &= \text{false} \\ \text{lexMA}(s, t, s_i \cdot ss, t_i \cdot ts) &= \mathbf{if} \ s_i = t_i \ \mathbf{then} \ \text{lexMA}(s, t, ss, ts) \\ &\quad \mathbf{else} \ (\text{lpo}(s_i, t_i) \wedge \text{majo}(s, ts)) \vee \text{alpha}(s_i \cdot ss, t) \end{aligned}$$

We turn the conjunction into an **if**-expression and distribute the call to **alpha** into both branches. After some small simplifications we get:

$$\begin{aligned} \text{lexMA}(s, t, s_i \cdot ss, t_i \cdot ts) = & \mathbf{if} \quad s_i = t_i \quad \mathbf{then} \text{lexMA}(s, t, ss, ts) \\ & \mathbf{elif} \text{lpo}(s_i, t_i) \mathbf{then} \text{majo}(s, ts) \vee \mathbf{alpha}(s_i \cdot ss, t) \\ & \mathbf{else} \mathbf{alpha}(s_i \cdot ss, t) \end{aligned}$$

Now, two further optimizations are possible. Using Lemma 6.1 the first call to **alpha** can be completely eliminated. For the second call to **alpha** we already know from the evaluation of the guard that $s_i \not\prec_{\text{lpo}} t_i$. Hence $s_i \not\prec_{\text{lpo}} t$, and s_i can be dropped from the argument list.

Summing up, we get the following version of LPO:

$$\begin{aligned} \text{lpo}_4 & : \text{Term Term} \rightarrow \text{Bool} \\ \text{lpo}_4(\text{F}(f, ss), \text{F}(g, ts)) = & \mathbf{if} \quad f >_{\text{Fid}} g \mathbf{then} \text{majo}_4(\text{F}(f, ss), ts) \\ & \mathbf{elif} \quad f = g \quad \mathbf{then} \text{lexMA}_4(\text{F}(f, ss), \text{F}(g, ts), ss, ts) \\ & \mathbf{else} \mathbf{alpha}_4(ss, \text{F}(g, ts)) \\ \text{lpo}_4(\text{F}(f, ss), \text{V}(y)) = & \text{delta}_4(\text{F}(f, ss), \text{V}(y)) \\ \text{lpo}_4(\text{V}(x), t) = & \text{false} \\ \text{lexMA}_4 & : \text{Term Term Termlist Termlist} \rightarrow \text{Bool} \\ \text{lexMA}_4(s, t, [], []) = & \text{false} \\ \text{lexMA}_4(s, t, s_i \cdot ss, t_i \cdot ts) = & \mathbf{if} \quad s_i = t_i \quad \mathbf{then} \text{lexMA}_4(s, t, ss, ts) \\ & \mathbf{elif} \text{lpo}_4(s_i, t_i) \mathbf{then} \text{majo}_4(s, ts) \\ & \mathbf{else} \mathbf{alpha}_4(ss, t) \end{aligned}$$

Corollary 6.3 *If $s, t \in \text{Term}$ are well-formed then $\text{lpo}_3(s, t) = \text{lpo}_4(s, t)$. \square*

* * *

It remains to show that this version needs time that is polynomial bound in the size of the arguments. We assume, that the comparison of two function symbols can be performed in constant time.

Lemma 6.4 *Evaluating $\text{lpo}_4(s, t)$ needs $O(|s| \cdot |t|)$ running time in the worst case.*

Proof. Induction on $|s| + |t|$. If t is a variable, the time needed is $O(|s|)$ as **contains** and **contains_{tl}** run in linear time. Otherwise, let $s \equiv f(s_1, \dots, s_n)$ and $t \equiv g(t_1, \dots, t_m)$. The time needed without the recursive calls is mainly determined by the arity of f and g and can therefore be bound by a constant c . Thus, it is sufficient to analyze the recursive calls. We proceed by case analysis.

(i) $f >_{\mathcal{F}} g$. Then s will be compared with each t_j , $j \in \{1, \dots, m\}$. By induction hypothesis, each of them will need $O(|s| \cdot |t_j|)$ time in the worst case, which gives

$$c + \sum_{j=1}^m O(|s| \cdot |t_j|) = O(|s| \cdot |t|) .$$

(ii) $f = g$. If $s \equiv t$, this will be determined in $O(|s| + |t|)$ time. Otherwise, there is some i with $s_i \neq t_i$. For each $j \leq i$ the test for $s_j = t_j$ can be performed in $O(|s_j| + |t_j|)$ time. If $s_i \succ_{\text{lpo}} t_i$, then s is compared against t_k for $k > i$. By induction hypothesis we get

$$c + \sum_{j=1}^i O(|s_j| + |t_j|) + O(|s_i| \cdot |t_i|) + \sum_{k=i+1}^n O(|s| \cdot |t_k|) \leq \sum_{j=1}^n O(|s| \cdot |t_j|) = O(|s| \cdot |t|).$$

In the other case, we have $s_i \not\succeq_{\text{lpo}} t_i$ and the remaining subterms of s are compared to t . By induction hypothesis we have

$$c + \sum_{j=1}^i O(|s_j| + |t_j|) + O(|s_i| \cdot |t_i|) + \sum_{k=i+1}^n O(|s_k| \cdot |t|) \leq \sum_{j=1}^n O(|s_j| \cdot |t|) = O(|s| \cdot |t|).$$

(iii) $f \not\prec_{\mathcal{F}} g$. In the worst case each subterm s_j of s is compared to t both for $s_j = t$ and $s_j \succ_{\text{lpo}} t$. Applying induction hypothesis leads to

$$c + \sum_{j=1}^n \left(O(|s_j| + |t|) + O(|s_j| \cdot |t|) \right) = O(|s| \cdot |t|).$$

□

The following example shows that for certain inputs $\text{lpo}_4(s, t)$ needs time proportional to $|s| \cdot |t|$.

Example 6.5 Consider two constants $a, b \in \mathcal{F}$, a unary function symbol $f \in \mathcal{F}$, and a binary one $g \in \mathcal{F}$ with precedence $a \succ_{\mathcal{F}} b \succ_{\mathcal{F}} f \succ_{\mathcal{F}} g$. Then the terms u_i and v_i are recursively defined by $u_0 \equiv a$ and $u_{i+1} \equiv f(u_i)$, resp. $v_0 \equiv b$ and $v_{i+1} \equiv g(v_0, v_i)$. Therefore, $|u_i| = i + 1$ and $|v_i| = 2i + 1$, and $u_i \not\equiv v_j$ for all $i, j \geq 0$. The evaluation of $\text{lpo}_4(u_i, v_j)$ makes $(i + 1)(j + 1) + j$ recursive calls. This can be shown by induction on $i + j$. Let $j = 0$. The comparison of u_0 with v_0 needs one call – both terms have no subterms. As $f \not\prec_{\mathcal{F}} b$ the call of $\text{lpo}_4(u_{i+1}, v_0)$ results via alpha_4 in one call to $\text{lpo}_4(u_i, v_0)$; the number of recursive calls for $\text{lpo}_4(u_i, v_0)$ is therefore $i + 1$. For the comparison of u_i with $v_{j+1} \equiv g(v_0, v_j)$ two recursive calls to $\text{lpo}_4(u_i, v_0)$ and $\text{lpo}_4(u_i, v_j)$ occur, because g is in the precedence smaller than f and than a . This makes $1 + (i + 1) + (i + 1)(j + 1) + j + 1 = (i + 1)(j + 2) + j + 2$ calls by use of induction hypothesis.

It can be seen in Figure 4 that the measured running times reflect quite well the number of recursive calls giving experimental support to the analysis above. □

Corollary 6.6 *The worst-case running time of $\text{lpo}_4(s, t)$ is $\Theta(|s| \cdot |t|)$.* □

In some situations we get a better bound:

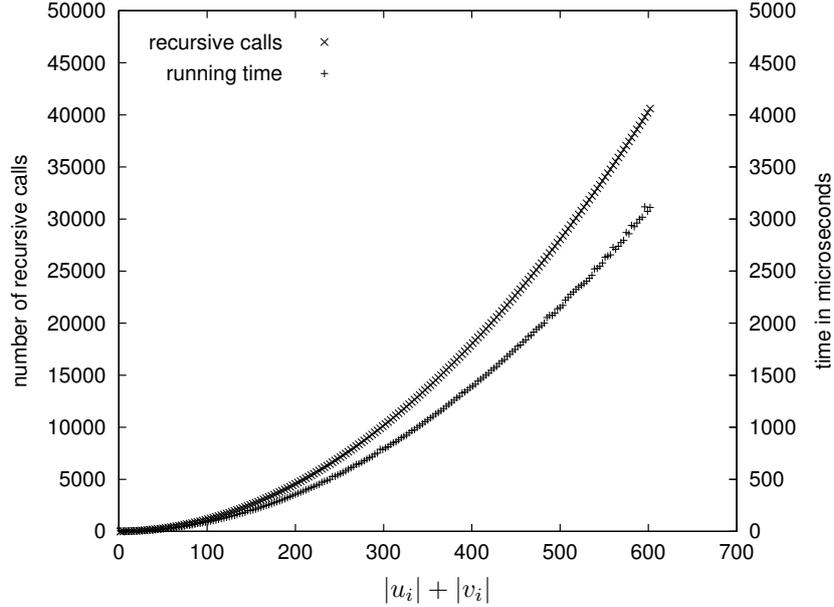


Fig. 4. Number of recursive calls and running times for $\text{lpo}_4(u_i, v_i)$, $i = 0, \dots, 200$, (cf. Example 6.5).

Lemma 6.7 *If the maximal arity of function symbols is one, the running time of $\text{lpo}_4(s, t)$ is $O(|s| + |t|)$ in the worst case.*

Proof. Depending on the relationship of the top symbols at most one recursive call occurs, for which the size of the arguments is one or two symbols smaller. \square

Therefore, lpo_4 can handle Example 5.1 within linear time.

7 Bidirectional comparisons

During theorem proving we are often not only interested if a term is greater than an other with respect to LPO, but in which ordering relation the two terms are. As LPO establishes on terms a partial ordering, we can have for terms s and t either $s \equiv t$, $s \succ_{\text{lpo}} t$, $t \succ_{\text{lpo}} s$, or that s and t are incomparable. To capture this information we introduce a new sort **Res** with constructors **Eq**, **Gt**, **Lt**, **Nc** $:\rightarrow$ **Res** for each of the situations above. The function **clpo** determines the ordering relationship of two terms. For $i = 1, \dots, 4$ its definition is the following:

$$\begin{aligned} \text{clpo}_i &: \text{Term Term} \rightarrow \text{Res} \\ \text{clpo}_i(s, t) &= \text{if } s = t \quad \text{then Eq} \\ &\quad \text{elif } \text{lpo}_i(s, t) \quad \text{then Gt} \\ &\quad \text{elif } \text{lpo}_i(t, s) \quad \text{then Lt} \\ &\quad \text{else Nc} \end{aligned}$$

With this definition it may happen that both invocations of lpo_4 will traverse both terms comparing the same subterms – only in different directions.

Consider the (pathologic) example $s \equiv f^n(x)$ and $t \equiv f^n(y)$, which are incomparable. To determine this, the function clpo_4 makes $3(n + 1)$ steps. Our aim is to develop a variant that avoids duplicated work. Our transformations will combine the traversals into one in most cases, so the resulting function clpo_5 will make $n + 1$ steps for this example.

We use clpo_4 as starting point. First, we do some case splitting. If one or both arguments are variables, it is sufficient to unfold the definitions and to do some simplifications:

$$\begin{aligned} \text{clpo}_5(\mathbf{F}(f, ss), \mathbf{V}(y)) &= \mathbf{if\ contains}_{\text{el}}(ss, y) \mathbf{then\ Gt\ else\ Nc} \\ \text{clpo}_5(\mathbf{V}(x), \mathbf{F}(g, ts)) &= \mathbf{if\ contains}_{\text{el}}(ts, x) \mathbf{then\ Lt\ else\ Nc} \\ \text{clpo}_5(\mathbf{V}(x), \mathbf{V}(y)) &= \mathbf{if\ } x = y \mathbf{then\ Eq\ else\ Nc} \end{aligned}$$

The remaining case is more interesting. Here too, we start by unfolding the definitions:

$$\begin{aligned} \text{clpo}(\mathbf{F}(f, ss), \mathbf{F}(g, ts)) &= \\ \mathbf{if\ } f = g \wedge ss = ts & \mathbf{then\ Eq} \\ \mathbf{elif\ } \left(\begin{array}{l} \mathbf{if\ } f >_{\text{Fid}} g \mathbf{then\ } \text{majo}(\mathbf{F}(f, ss), ts) \\ \mathbf{elif\ } f = g \mathbf{then\ } \text{lexMA}(\mathbf{F}(f, ss), \mathbf{F}(g, ts), ss, ts) \\ \mathbf{else\ } \text{alpha}(ss, \mathbf{F}(g, ts)) \end{array} \right) & \mathbf{then\ Gt} \\ \mathbf{elif\ } \left(\begin{array}{l} \mathbf{if\ } g >_{\text{Fid}} f \mathbf{then\ } \text{majo}(\mathbf{F}(g, ts), ss) \\ \mathbf{elif\ } g = f \mathbf{then\ } \text{lexMA}(\mathbf{F}(g, ts), \mathbf{F}(f, ss), ts, ss) \\ \mathbf{else\ } \text{alpha}(ts, \mathbf{F}(f, ss)) \end{array} \right) & \mathbf{then\ Lt} \\ \mathbf{else\ Nc} & \end{aligned}$$

Then we reorganize the nested **if**-expressions by using the comparisons of the leading function symbols as the main guards:

$$\begin{aligned} \text{clpo}(\mathbf{F}(f, ss), \mathbf{F}(g, ts)) &= \\ \mathbf{if\ } f = g \mathbf{then\ } & \mathbf{if\ } ss = ts \mathbf{then\ Eq} \\ & \mathbf{elif\ } \text{lexMA}(\mathbf{F}(f, ss), \mathbf{F}(g, ts), ss, ts) \mathbf{then\ Gt} \\ & \mathbf{elif\ } \text{lexMA}(\mathbf{F}(g, ts), \mathbf{F}(f, ss), ts, ss) \mathbf{then\ Lt} \\ & \mathbf{else\ Nc} \\ \mathbf{elif\ } f >_{\text{Fid}} g \mathbf{then\ } & \mathbf{if\ } \text{majo}(\mathbf{F}(f, ss), ts) \mathbf{then\ Gt} \\ & \mathbf{elif\ } \text{alpha}(ts, \mathbf{F}(f, ss)) \mathbf{then\ Lt} \\ & \mathbf{else\ Nc} \\ \mathbf{elif\ } g >_{\text{Fid}} f \mathbf{then\ } & \mathbf{if\ } \text{majo}(\mathbf{F}(g, ts), ss) \mathbf{then\ Lt} \\ & \mathbf{elif\ } \text{alpha}(ss, \mathbf{F}(g, ts)) \mathbf{then\ Gt} \\ & \mathbf{else\ Nc} \\ \mathbf{else\ } & \mathbf{if\ } \text{alpha}(ss, \mathbf{F}(g, ts)) \mathbf{then\ Gt} \\ & \mathbf{elif\ } \text{alpha}(ts, \mathbf{F}(f, ss)) \mathbf{then\ Lt} \\ & \mathbf{else\ Nc} \end{aligned}$$

As can be seen, the second and the third branch are nearly symmetric – except return values. This can be remedied with the following function $\text{flip} : \text{Res} \rightarrow \text{Res}$:

$$\begin{aligned} \text{flip}(\text{Eq}) &= \text{Eq} & \text{flip}(\text{Lt}) &= \text{Gt} \\ \text{flip}(\text{Nc}) &= \text{Nc} & \text{flip}(\text{Gt}) &= \text{Lt} \end{aligned}$$

Now it suffices to introduce three helper functions to get a concise version of `clpo`:

$$\begin{aligned} \text{clpo}(F(f, ss), F(g, ts)) &= \text{if } f = g \text{ then } \text{cLMA}(F(f, ss), F(g, ts), ss, ts) \\ &\quad \text{elif } f >_{\text{Fid}} g \text{ then } \text{cMA}(F(f, ss), ts) \\ &\quad \text{elif } g >_{\text{Fid}} f \text{ then } \text{flip}(\text{cMA}(F(g, ts), ss)) \\ &\quad \text{else } \text{cAA}(F(f, ss), F(g, ts), ss, ts) \end{aligned}$$

The initial specifications of these functions are as follows:

$$\begin{aligned} \text{cMA} &: \text{Term Termlist} \rightarrow \text{Res} \\ \text{cMA}(s, ts) &= \text{if } \text{majo}(s, ts) \text{ then } \text{Gt} \\ &\quad \text{elif } \text{alpha}(ts, s) \text{ then } \text{Lt} \\ &\quad \text{else } \text{Nc} \\ \text{cLMA} &: \text{Term Term Termlist Termlist} \rightarrow \text{Res} \\ \text{cLMA}(s, t, ss, ts) &= \text{if } ss = ts \text{ then } \text{Eq} \\ &\quad \text{elif } \text{lexMA}(s, t, ss, ts) \text{ then } \text{Gt} \\ &\quad \text{elif } \text{lexMA}(t, s, ts, ss) \text{ then } \text{Lt} \\ &\quad \text{else } \text{Nc} \\ \text{cAA} &: \text{Term Term Termlist Termlist} \rightarrow \text{Res} \\ \text{cAA}(s, t, ss, ts) &= \text{if } \text{alpha}(ss, t) \text{ then } \text{Gt} \\ &\quad \text{elif } \text{alpha}(ts, s) \text{ then } \text{Lt} \\ &\quad \text{else } \text{Nc} \end{aligned}$$

It remains to optimize them in turn. We start with `cMA` by case splitting on ts . For $ts = []$ we get after some simplifications $\text{cMA}(s, []) = \text{Gt}$. The other case is:

$$\begin{aligned} \text{cMA}(s, t \cdot ts) &= \text{if } \text{lpo}(s, t) \wedge \text{majo}(s, ts) \text{ then } \text{Gt} \\ &\quad \text{elif } s = t \vee \text{lpo}(t, s) \vee \text{alpha}(ts, s) \text{ then } \text{Lt} \\ &\quad \text{else } \text{Nc} \end{aligned}$$

The two calls to `lpo` and the equality test of s and t can be covered by a single call to `clpo`:

$$\begin{aligned} \text{cMA}(s, t \cdot ts) &= \text{case } \text{clpo}(s, t) \\ \text{Gt} &: \text{if } \text{majo}(s, ts) \text{ then } \text{Gt} \\ &\quad \text{elif } \text{alpha}(ts, s) \text{ then } \text{Lt} \\ &\quad \text{else } \text{Nc} \\ \text{Eq, Lt} &: \text{Lt} \\ \text{Nc} &: \text{if } \text{alpha}(ts, s) \text{ then } \text{Lt} \text{ else } \text{Nc} \end{aligned}$$

We can now fold back the definition of `cMA`, i. e., call $\text{cMA}(s, ts)$ in the case $\text{clpo}(s, t) = \text{Gt}$.

For space reasons we omit the transformations of `cLMA`. We finally get the following definition of `clpo5`. Note that the definitions of `lpo5`, `alpha5`, etc., are identical to their `lpo4`-counterparts.

$$\begin{aligned}
& \text{clpo}_5 : \text{Term Term} \rightarrow \text{Res} \\
& \text{clpo}_5(F(f, ss), V(y)) = \text{if contains}_{\text{tl}}(ss, y) \text{ then Gt else Nc} \\
& \text{clpo}_5(V(x), F(g, ts)) = \text{if contains}_{\text{tl}}(ts, x) \text{ then Lt else Nc} \\
& \text{clpo}_5(V(x), V(y)) = \text{if } x = y \text{ then Eq else Nc} \\
& \text{clpo}_5(F(f, ss), F(g, ts)) = \text{if } f = g \text{ then cLMA}_5(F(f, ss), F(g, ts), ss, ts) \\
& \quad \text{elif } f >_{\text{Fid}} g \text{ then cMA}_5(F(f, ss), ts) \\
& \quad \text{elif } g >_{\text{Fid}} f \text{ then flip(cMA}_5(F(g, ts), ss)) \\
& \quad \text{else cAA}_5(F(f, ss), F(g, ts), ss, ts) \\
& \\
& \text{cMA}_5 : \text{Term Termlist} \rightarrow \text{Res} \\
& \text{cMA}_5(s, []) = \text{Gt} \\
& \text{cMA}_5(s, t \cdot ts) = \text{case clpo}_5(s, t) \\
& \quad \text{Gt} : \text{cMA}_5(s, ts) \\
& \quad \text{Eq, Lt} : \text{Lt} \\
& \quad \text{Nc} : \text{if alpha}_5(ts, s) \text{ then Lt else Nc} \\
& \\
& \text{cLMA}_5 : \text{Term Term Termlist Termlist} \rightarrow \text{Res} \\
& \text{cLMA}_5(s, t, [], []) = \text{Eq} \\
& \text{cLMA}_5(s, t, s_i \cdot ss, t_i \cdot ts) = \text{case clpo}_5(s_i, t_i) \\
& \quad \text{Eq} : \text{cLMA}_5(s, t, ss, ts) \\
& \quad \text{Gt} : \text{cMA}_5(s, ts) \\
& \quad \text{Lt} : \text{flip(cMA}_5(t, ss)) \\
& \quad \text{Nc} : \text{cAA}_5(s, t, ss, ts) \\
& \\
& \text{cAA}_5 : \text{Term Term Termlist Termlist} \rightarrow \text{Res} \\
& \text{cAA}_5(s, t, ss, ts) = \text{if } \text{alpha}_5(ss, t) \text{ then Gt} \\
& \quad \text{elif } \text{alpha}_5(ts, s) \text{ then Lt} \\
& \quad \text{else Nc}
\end{aligned}$$

If we compare the transformations in this section with the optimizations in Sections 4 and 6, we see that no further domain-specific knowledge about LPO is needed. The transformations in this section rely solely on some standard Unfold/Fold-calculus [BD77].

Corollary 7.1 *If $s, t \in \text{Term}$ are well-formed then $\text{clpo}_4(s, t) = \text{clpo}_5(s, t)$. \square*

With an analysis that is similar to the analysis of lpo_4 we can show that the running time of clpo_5 is $\Theta(|s| \cdot |t|)$ in the worst case.

8 Further measurements and further variants

The optimizations of Section 6 lead to a dramatic improvement in the time consumption of LPO. Table 2 shows that lpo_4 is significantly faster than lpo_3 . Depending on the example the speedup is between 5 and more than 1000. When we compare the time needed by lpo_4 with that for the rest we see that the overall running time of the prover is no longer dominated by the time for the ordering. The optimizations for bidirectional comparisons show,

problem	time needed by ordering							
	lpo_4	lpo_5	lpo_6	lpo_7	lpo_8	lpo_9	lpo_{10}	lpo_{11}
GRP180-1	0.149	0.128	0.148	0.252	0.189	0.148	0.171	0.290
LAT020-1	0.809	0.769	0.828	1.284	1.136	0.859	0.867	1.338
LCL109-6	0.169	0.142	0.184	0.335	0.227	0.185	0.188	0.329
RNG027-5	0.451	0.443	0.491	0.788	0.594	0.509	0.526	0.909
z22	0.004	0.003	0.004	0.006	0.005	0.004	0.005	0.012
TPTP ₁₀	186.100	168.800	190.700	289.700	263.100	194.700	184.300	281.300

Table 2
Time needed (in seconds) for ordering comparisons

contrary to our expectations, only moderate improvements. Columns lpo_4 and lpo_5 differ in using clpo_4 and clpo_5 for bidirectional comparisons. Depending on the example we get improvements of up to 15 per cent. Measurements indicate that the multiple traversals are successfully avoided. It seems that the more complicated code reduces these benefits.

The running time of lpo_4 is $\Theta(|s| \cdot |t|)$ in the worst case, as shown in Section 6. The dynamic-programming algorithm sketched in Section 5 has to determine the ordering relationship between each subterm of s with each subterm of t , i. e. exactly $M = |s| \cdot |t|$ pairs. It is therefore interesting to determine the actual number N of pairs that are compared by the optimized top-down algorithm lpo_4 . Figure 5 contains the quotient N/M for each invocation of the ordering of example RNG027-5. The quotient is only for a small number of invocations higher than 0.4, the larger the input, the smaller the quotient. It is therefore highly unlikely that an implementation of the dynamic programming algorithm can be competitive with lpo_4 .

* * *

Some further variations are worth discussion. We use as starting point lpo_4 , because it is simpler in structure than lpo_5 .

The variant lpo_6 . We want to avoid the possibly double traversal of subterms for the tests $s = t$ and $s \succ_{\text{lpo}} t$ which occur in the case α and the lexicographic tests. Similar to the modifications in Section 7 we combine both tests into one. The function lpo_6 returns then the values Eq , Gt , and Nc . Measurements show, however, that this has a small negative effect on the time requirements, it does not lead to a further improvement. A detailed analysis reveals the reason. We have to distinguish two cases: If $s = t$, which accounts for 10 to 20% of the cases, the new version is slower, because the code is more complicated. If $s \neq t$, the equality test detects that pretty fast. On the average,

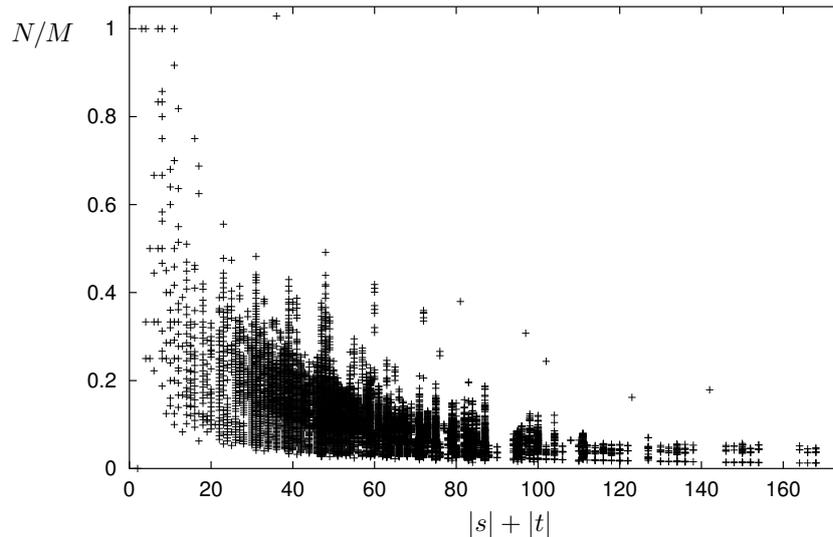


Fig. 5. Let N be the number of compared pairs of subterms of s and t performed by lpo_4 and let $M = |s| \cdot |t|$, the number of compared pairs a dynamic programming version would need. Depicted is N/M over the combined length of s and t for RNG027-5.

it compares less than 2 symbols to find $s \neq t$.⁸ So the possible gain is rather small for the analyzed examples, which explains why the transformation does not pay off.

The next two variations exploit the following property of LPO: If $s \succ_{\text{lpo}} t$ then $\text{Var}(s) \supseteq \text{Var}(t)$. The idea is to use the contraposition – $\text{Var}(s) \not\supseteq \text{Var}(t)$ implies $s \not\succeq_{\text{lpo}} t$ – as a sufficient pretest to restrict or to stop the computation. Bit-oriented machine-instructions allow us to handle a fixed subset $\mathcal{V}_0 \subset \mathcal{V}$ very efficiently. With $|\mathcal{V}_0| = 32$ it fits well the characteristics of the machines we work on (Pentium III). The time to compute $\text{Var}_0(s) = \text{Var}(s) \cap \mathcal{V}_0$ is $\Theta(|s|)$; the time to compare two such variable sets is $O(1)$. Limiting the pretest to $\text{Var}_0(s) \not\supseteq \text{Var}_0(t)$ reduces the power as prefilter. However, WALDMEISTER minimizes for various reasons the number of different variables in the proof state, hence terms with variables not in \mathcal{V}_0 occur rarely in practice.

The variant lpo_7 . We decorate in a preprocessing step each term (including all subterms) with the accompanying variable sets. This is linear in the size of s and t . After that we use in each recursive call the comparison of the variable sets as a pretest. For variables in \mathcal{V}_0 the case δ becomes trivial to decide and needs $O(1)$ time, which is an additional motivation for this variant. The experimental results show that the pretests speed up the computation of LPO, the overhead for the precomputation is not compensated though.

The variant lpo_8 . This variant uses variable sets in a weaker form. It is suggested in [Wei01]. Only at the top-level the sets $\text{Var}(s)$ and $\text{Var}(t)$ are determined and used as a pretest for lpo_8 or as a restriction for clpo_8 . An

⁸ For most examples the number of symbol comparisons shows a negative-exponential distribution.

analysis shows that the preprocessing step is now cheaper, because the costly write-operations for each variable set into the subterm are avoided. But the benefits are smaller as well. In our measurements, the use of variable sets does not pay off.

The last three variants try to use additional knowledge that is available in special cases.

The variant lpo_9 . This variant uses the so-called “Pacman-Lemma”:⁹ For unary function symbols f we have $f(s) \succ_{\text{lpo}} f(t)$ iff $s \succ_{\text{lpo}} t$. Then it is possible in lpo to skip the call to `lexMA` and perform a direct tail-call to itself. We can implement this in an imperative setting by two assignments and a `goto`. Our measurements show that the benefits are tiny, mostly encoded string-rewriting systems profit a little bit, other examples may suffer a slight slow-down.

The variant lpo_{10} . This variant has specialized code for cases where at least one argument is a constant. Then the rather complex recursion can be replaced by simpler loops. Nevertheless, it does not seem to pay off, the single examples show a small slow-down, TPTP_{10} indicates a small speed-up.

The variant lpo_{11} . Because LPO contains the subterm relation, this variant uses a check for $s \succ_{\text{st}} t$ as a sufficient pretest for $s \succ_{\text{lpo}} t$. This can be a win, because $s \succ_{\text{st}} t$ is cheaper to evaluate. However, our measurements show that the subterm-pretest applies too rarely in practice to justify the extra costs.

When we summarize the measurements we see that lpo_4 shows the best combination of efficiency and coding complexity. Only lpo_5 gives noticeable running-time improvements, but these are moderate (about 10% for TPTP_{10}) and the implementation has roughly twice the size. Of the other variants, some show no marked improvements, some are actually disimprovements. But why does folklore suggest them? Our guess is: because they help to improve nonoptimal versions. For instance, the “Pacman-Lemma” applied to lpo_3 gives for some examples dramatic improvements over lpo_3 . It captures some of the insights that lead from lpo_3 to lpo_4 by avoiding in special situations the call to `alpha` and thus recomputations that lead to exponential behavior.¹⁰ Nevertheless, the improvements are far from uniform. Applied to the polynomial lpo_4 the potential of these optimizations is much smaller and the overhead of the more complicated code may dominate.

There is one final topic we want to discuss briefly. Up to now we have made no assumption about the term data structure. We conjecture that underlying

⁹ This colloquial term stems from the visualization of what happens to common prefixes of the terms under consideration. Take as examples $f(g(f(g(s)))) \succ_{\text{lpo}}^? f(g(f(g(t))))$ – the prefixes are efficiently nibbled away and $s \succ_{\text{lpo}}^? t$ remains.

¹⁰ Consider a variant of Example 5.1: Let $a \succ_{\mathcal{F}} b \succ_{\mathcal{F}} f$ and test $f^n(b) \succ_{\text{lpo}}^? f^n(a)$. The function lpo_3 needs exponential time – the analysis of Example 5.1 applies with small modifications. With the “Pacman-Lemma” this test becomes linear. However, for Example 5.1 the lemma is not applicable.

ing term representations that keep subterms individual, such as tree-terms, flat-terms, or Prolog-terms (see e. g. [RSV01]), behave quite similar for LPO comparisons. Our algorithms require access to the top-symbol and to the subterms from left to right, which is supported by all three term representations at similar costs. However, if we have a data structure that supports sharing of identical subterms, we can take profit. First, the test for term equality is now a simple pointer comparison. Second, it is then rather cheap to augment the terms with additional information. So we expect that lpo_7 should be an improvement on lpo_4 in such an environment, as the costs for the explicit computation of the variable sets disappear. Such sets are computed once at term-construction time, the costs are then amortized over many ordering comparisons. Third, it may even become feasible to keep a global history in form of a memo-table of the results of LPO. Then it is possible to share results *between* different invocations of the ordering.

9 Related work and implementation status

With Snyder [Sny93] we share the finding that the number of publications about the efficient implementation of orderings is really small compared to the number of publications about the definition of new orderings. The thesis of Steinbach contains a short overview about the time complexities of orderings [Ste94, Chap. 6.2]. It seems to be agreed in the literature that the standard way to achieve a polynomial version of LPO and related orderings is via dynamic programming [KNS85]. Snyder devised an $O(n \cdot \log n)$ algorithm for comparing ground terms with LPO based on total precedences [Sny93].

In [Wei01] Weidenbach acknowledges that the top-down implementation of LPO may result in exponential behavior. He then goes on and compares the top-down implementation with an implementation of [Sny93] and concludes that the latter gives no advantages in practice. Inspection of the source code of SPASS [W⁺02] shows that it contains a top-down implementation of the *Recursive Path Ordering with Status* (RPOS). This is a well-known generalization of LPO, where each function symbol has assigned a status (left-to-right, right-to-left, or multi-set). If all function symbols have status left-to-right, we get LPO as a special case. Then the RPOS-implementation of SPASS behaves like lpo_6 . That this is actually a polynomial version is not mentioned.

Analyzing the source code of other systems that participated in CASC [S⁺] we find no implementation of the dynamic programming algorithm. Snyder's method seems to be implemented in SPASS only. In Otter [McC] we find an implementation of RPOS which specializes essentially to lpo_4 if all functions symbols have status left-to-right. WALDMEISTER's original LPO is an elaborate version of lpo_5 . Several costly pretest are used in fear of exponential behavior. The straightforward implementation of lpo_4 is more than three times faster. The other systems that we analyzed use a version similar to lpo_1 or lpo_2 , sometimes enriched with optimizations for special cases. Often

some form of caching (of different implementation quality) is used to avoid exponential behavior.¹¹

10 Conclusions

In this work we described and compared several implementations of LPO. Simple translations of the definition show an exponential behavior, because they consider subproblems repeatedly. To achieve a polynomial version the use of dynamic programming is recommended in the literature, which incurs the use of an additional data structure. However, we can achieve the same worst-case behavior by modifying the recursive calls. These optimizations rely on simple properties of LPO and avoid the use of additional data structures. A further variant performs bidirectional comparisons, which lead to additional speed-ups. We find these optimizations in several provers, but we know of no reference in the literature. It is unclear to us, if it was known before that with these optimizations the top-down version of LPO is polynomial.

Experiments show that the optimizations lead to tremendous speed-ups for our test cases. Nevertheless, as the best version to compare the terms s and t requires $O(|s| \cdot |t|)$ time in the worst case, the time spent in LPO can still not be neglected for large terms. Therefore, one should be sparing with calls to LPO – if possible. It remains open, if for nonground terms a sub-quadratic version exists.

In this work we concentrated on the basic LPO test $s \succ_{\text{lpo}} t$. An important extension is to determine for terms s and t and substitution σ whether $\sigma(s) \succ_{\text{lpo}} \sigma(t)$ holds. By passing σ as an additional parameter in the recursive calls and looking up bindings of variables if necessary, we can improve considerably on explicitly constructing $\sigma(s)$ and $\sigma(t)$ before calling the basic ordering test. In some situations s and t are fixed and σ is varying. Then, we can achieve further improvements by preprocessing s and t . In [RV04] Riazanov and Voronkov consider this and similar problems for KBO.

To describe the different variants of LPO we used a small algebraic specification language to keep the presentation concise. The translation of the different specifications into the input language of the inductive theorem prover QUODLIBET [A⁺03] was straightforward. We later realized that other inductive provers have problems with mutual recursive functions and partiality (e. g. lex_1 is only partially defined). Trying to prove properties of the different versions with QUODLIBET revealed implicit assumptions in the code. For example, to show $\text{lpo}_1(s, t) = \text{lpo}_2(s, t)$ we need that $\text{well}(s) = \text{well}(t) = \text{true}$, i. e., that the arguments are well-formed terms. Our examples spawned some interest into the development of tactics that are specialized in the support of mutual recursive functions. This is ongoing work.

¹¹After reading a preliminary version of this work, Schulz replaced his old LPO-implementation with lpo_4 in his theorem prover E [Sch02]. Some TPTP-problems show an overall speedup of 20. (Personal communication, May 2004.)

For developing the different versions we used the paradigm of program transformations. This helped not only to focus on the essential ideas, but also to prevent errors in the implementation. In the C translations of all considered variants we discovered *three* bugs, all of which were caught easily by comparing the different versions during test-runs. One bug occurred because of a copying error, the other two by replacing recursive calls with iteration constructs. This low number of errors is far better than what is usually achieved by traditional coding practice. This experience suggests to use this two-level development approach for other subtasks in a prover, especially, if they need a significant amount of the running time and an efficient implementation is not obvious.

Acknowledgments

The curiosity of Christian Schmidt renewed my interest into the topic. While trying to answer some of his questions, I discovered that lpo_4 runs in polynomial time, which was new to me. The numerous remarks of one of the anonymous reviewers helped to improve the presentation considerably.

References

- [A⁺03] J. Avenhaus, U. Kühler, T. Schmidt-Samoa, C.-P. Wirth. How to Prove Inductive Theorems? QUODLIBET! In F. Baader, ed., *Proc. of the 19th Intern. Conference on Automated Deduction*, vol. 2741 of *LNCS*, pp. 328–333. Springer, 2003. See <http://www-avenhaus.informatik.uni-kl.de/quodlibet.html>.
- [BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
- [Bel57] R. E. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.
- [BH96] A. Buch and Th. Hillenbrand. WALDMEISTER: Development of a High Performance Completion-Based Theorem Prover. SEKI-Report 96-01, Univ. Kaiserslautern, 1996.
- [DP01] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. I, chapter 9, pp. 535–610. Elsevier Science, 2001.
- [KL80] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Department of Computer Science, University of Illinois, Urbana, IL, 1980.
- [KNS85] D. Kapur, P. Narendran, and G. Sivakumar. A path ordering for proving termination of term rewriting systems. In H. Ehrig et al., eds., *Coll. on Trees in Algebra and Programming*, vol. 185 of *LNCS*, pp. 173–187. Springer, 1985.

- [LH02] B. Löchner and Th. Hillenbrand. A Phytography of WALDMEISTER. *AI Communications*, 15(2–3):127–133, 2002. See <http://www.waldmeister.org>
- [McC] W. McCune. Otter: An Automated Deduction System. See <http://www-unix.mcs.anl.gov/AR/otter/>
- [Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.
- [RSV01] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. II, chapter 26, pp. 1853–1964. Elsevier Science, 2001.
- [RV04] A. Riazanov and A. Voronkov. Efficient checking of term ordering constraints. In D. Basin and M. Rusinowitch, eds. *Proc. 2nd IJCAR*, LNCS, Springer, 2004.
- [S⁺] G. Sutcliffe et al. The CADE ATP System Competition (CASC). See <http://www.tptp.org/CASC>.
- [Sch02] S. Schulz. E – A Brainiac Theorem Prover. *J. of AI Communications*, 15:111–126, 2002. See <http://www.eprover.org>
- [Sny93] W. Snyder. On the complexity of recursive path orderings. *Information Processing Letters*, 46:257–262, 1993.
- [SS97] C. B. Suttner and G. Sutcliffe. The TPTP problem library (TPTP v2.1.0). Technical Report 97/08, Department of Computer Science, James Cook University, Townsville, Australia, 1997. See <http://www.tptp.org>.
- [Ste94] J. Steinbach. *Termination of Rewriting*. PhD thesis, Universität Kaiserslautern, 1994. See <http://www-madlener.informatik.uni-kl.de/seki/1994/Steinbach.PhDThesis.ps.Z>.
- [W⁺02] Ch. Weidenbach, U. Brahm, Th. Hillenbrand, E. Keen, Ch. Theobald, and D. Topic. SPASS Version 2.0. In A. Voronkov, ed., *Proc. 18th Intern. Conference on Automated Deduction*, vol. 2392 of LNCS. Springer, 2002. See <http://spass.mpi-sb.mpg.de/>.
- [Wei01] Ch. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. II, chapter 27, pp. 1965–2013. Elsevier Science, 2001.

An Empirical Evaluation of Automated Theorem Provers in Software Certification

Ewen Denney¹

*QSS / NASA Ames Research Center
Moffett Field, CA, USA*

Bernd Fischer² Johann Schumann³

*RIACS / NASA Ames Research Center
Moffett Field, CA, USA*

Abstract

We describe a system for the automated certification of safety properties of NASA software. The system uses Hoare-style program verification technology to generate proof obligations which are then processed by an automated first-order theorem prover (ATP). We discuss the unique requirements this application places on the ATPs, focusing on automation, proof checking, and usability. For full automation, however, the obligations must be aggressively preprocessed and simplified, and we demonstrate how the individual simplification stages, which are implemented by rewriting, influence the ability of the ATPs to solve the proof tasks. Our results are based on 13 certification experiments that lead to 366 top-level safety obligations and ultimately to more than 25,000 proof tasks which have each been attempted by Vampire, Spass, e-setheo, and Otter. The proofs found by Otter have been proof-checked by IVY.

Key words: software certification, automated theorem proving, program synthesis, proof checking, verification condition generator, Hoare logic

1 Introduction

Software certification aims to show that the software in question achieves a certain level of quality, safety, or security. Its result is a *certificate*, i.e., independently checkable evidence of the properties claimed. Certification approaches vary widely, ranging from code reviews to full formal verification, but the highest degree of

¹ Email: edenney@email.arc.nasa.gov

² Email: fisch@email.arc.nasa.gov

³ Email: schumann@email.arc.nasa.gov

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

confidence is achieved with approaches that are based on formal methods and use logic and theorem proving to construct the certificates.

We have developed a certification approach which uses Hoare-style techniques to demonstrate the safety of aerospace software which has been automatically generated from high-level specifications. Our core idea is to extend the code generator so that it simultaneously generates code *and* the detailed annotations, e.g., loop invariants, that enable a safety proof. A verification condition generator (VCG) processes the annotated code and produces a set of *safety obligations*, which are provable if and only if the code is safe. An automated theorem prover (ATP) then discharges these obligations and the proofs, which can be verified by an independent proof checker, serve as certificates. This approach largely decouples code generation and certification and is thus more scalable than, e.g., verifying the generator or generating code and complete safety proofs in parallel. The aim of this work is to increase trust in the code generator.

In this paper, we evaluate the extent to which the current generation of ATPs is capable of supporting the formal certification of software. In our view, this covers three main aspects. First, full *automation* is crucial since the practicability of our approach hinges on it. Second, the ability to generate proof objects and to carry out *proof checking* is essential to create explicit certificates. Third, there are a range of *traceability* issues which have a significant bearing on the ability of an ATP to create meaningful certificates.

Program certification is a demanding application for ATPs because the number of proof obligations is potentially very large and program verification is generally a hard problem domain. However, in our case there are several factors which make a successful ATP application possible. First, we certify separate aspects of safety and not full functional correctness. This separation of concerns allows us to show non-trivial properties like matrix symmetry but results in more tractable obligations. Second, the extensions of the code generator are specific to the safety properties to be certified and to the algorithms used in the generated programs. This allows us to fine-tune the annotations which, in turn, also results in more tractable obligations. Third, we aggressively simplify the obligations before they are handed over to the prover, taking advantage of domain-specific knowledge.

In this paper, we evaluate three hypotheses. The first hypothesis is that the current generation of high-performance ATPs is—in principle—already powerful enough for practical application in program certification. The second hypothesis is that ATPs can still not be considered entirely as black boxes but require careful integration with the application at hand; in particular, the application must carefully preprocess the proof tasks to make them more tractable. The final hypothesis is that proof checkers for first-order logic have not yet reached the same level of maturity as the ATPs themselves, despite the fact that proof checking is, *prima facie*, conceptually simpler than proof finding.

We have tested our hypotheses by running five high-performance provers on seven different versions of the 366 safety obligations resulting from certifying five different safety policies for four different programs—in total more than 25,000

proof tasks per prover. In Section 2 we give an overview of the system architecture, describing the safety policies as well as the generation and preprocessing of the proof tasks. In Section 3, we outline the experimental set-up used to evaluate the theorem provers over a range of different preprocessing levels. The detailed results are given in Section 4; they confirm our first two hypotheses: the provers are generally able to solve the emerging obligations but only after substantial preprocessing. However, for almost all programs and all policies, a few hard obligations remain, and a successful certification (i.e., proof of *all* obligations) can be achieved only after even more tuning. Section 5 then discusses the proof checking experiments, and Section 6 looks at traceability issues. Finally, Section 7 draws some conclusions.

Conceptually, this paper continues the work described in [35,36] but the actual implementation of the certification system has been completely revised and substantially extended. We have expanded the range of both algorithms and safety properties which can be certified; in particular, our approach is now fully integrated with the `AUTOFILTER` system [37] as well as with the `AUTOBAYES` system [12] and the certification process is now completely automated. We have also implemented a new generic VCG which can be customized for a given safety policy and which directly processes the internal code representation instead of `Modula-2` as in the previous version. All these improvements and extensions to the underlying framework result in a substantially larger experimental basis than reported before. A shorter version of this paper appears as [5].

Related Work Our approach is related to proof-carrying code (PCC) [22]. PCC works on the machine-code level instead of the source-code level (as we do) and concentrates on very simple safety policies (mainly array-bounds safety) which leads to comparatively simple proof obligations. Hence, PCC is complementary to our approach, and a certifying compiler [23] could be used to ensure that the compilation step does not compromise the demonstrated safety policies. PCC also spawned an entire cottage industry of proof checkers, e.g., [1]; however, these use various higher-order logics and so are not applicable for our purposes.

Program verification is a popular application domain for theorem provers; we mention only a few systems here. `KIV` [26,28] is an interactive verification environment which can use different ATPs but relies heavily on term rewriting and user guidance. `Sunrise` [15] is a fully automatic system but uses custom-designed tactics in `HOL` to discharge the obligations. `ESC/Java` [10] is an automatic verification system but relies on the user to provide additional information on the program, e.g., loop invariants. `Houdini` [9] is an automatic annotation assistant which guesses invariants, but a significant amount of user interaction remains.

2 System Architecture

Our certification tool is built as an extension to the `AUTOBAYES` and `AUTOFILTER` program synthesis systems. `AUTOBAYES` works in the statistical data analysis domain and generates parameter learning programs while `AUTOFILTER` generates state estimation code based on variants of the Kalman filter algorithm. Figure 1 gives

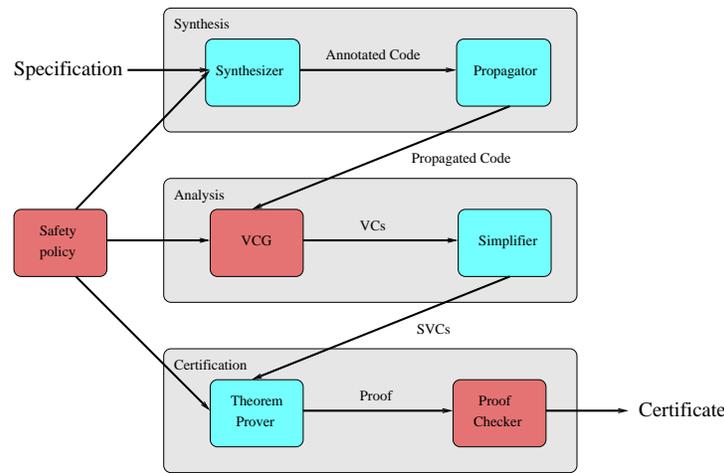


Fig. 1. Certification system architecture

an overview of the overall system architecture. Both underlying synthesis systems take as input a high-level problem specification and generate code that implements the specification. This process is based on the repeated application of schemas. *Schemas* are generic algorithms which are instantiated in a problem-specific way after their applicability conditions have been proven to hold for the given problem specification. The synthesizers first generate C++-style intermediate code which is then compiled down into any of the different supported languages and runtime environments.

For the certification tool, we extended the schemas such that the synthesis systems generate code that is marked up with annotations relevant to the chosen safety policy. These annotations encode local safety information which is then propagated throughout the program. In the next stage, the *analysis* is carried out by a VCG applying rules from the safety policy to generate verification conditions which are then simplified by a rewrite system. Finally, *certification* is achieved by sending these simplified verification conditions to an automated theorem prover and checking the resulting proofs.

The individual components are described in some detail in the subsequent sections. We distinguish *trusted* and *untrusted* components, shown in red (dark grey) and blue (light grey), respectively. In particular, the correctness of our certification system does not depend on the correctness of the two largest subsystems: the synthesizer, and the theorem prover; instead, we need only trust the safety policy, the VCG, and the proof checker.

This lets us adopt an approach to certification which we call *product-oriented certification*, in contrast to process-oriented approaches, which rely on the qualification (i.e., verification) of the tools being used. A product-oriented approach is more feasible when using complex tools like theorem provers and hence is more scalable.

safety policy	safety condition	domain theory
<i>array</i>	$\forall a[i] \in c. a_{lo} \leq i \leq a_{hi}$	arithmetic
<i>init</i>	$\forall read\text{-}var x \in c. init(x)$	propositional
<i>in-use</i>	$\forall input\text{-}var x \in c. use(x)$	propositional
<i>symm</i>	$\forall matrix\text{-}exp m \in c. \forall i, j. m[i, j] = m[j, i]$	matrices
<i>norm</i>	$\forall vector v \in c. \sum_{i=1}^{size(v)} v[i] = 1$	arithmetic, summations

Table 1

Safety conditions for different policies

2.1 Safety Properties and Safety Policies

The certification tool automatically certifies that a program satisfies a given *safety property*, i.e., an operational characterization that the program “does not go wrong”. It uses a corresponding *safety policy*, i.e., a set of Hoare-style proof rules and auxiliary definitions which are specifically designed to show that programs satisfy the safety property of interest. The distinction between safety properties and policies is explored in [3].

We further distinguish between *language-specific* and *domain-specific* properties and policies. Language-specific properties can be expressed in the constructs of the underlying programming language itself (e.g., array accesses), and are sensible for any given program written in the language. Domain-specific properties typically relate to high-level concepts outside the language (e.g., matrix multiplication), and must thus be expressed in terms of program fragments. Since these properties are specific to a particular application domain, the corresponding policies are not applicable to all programs.

We have defined five different safety properties and implemented the corresponding safety policies. Array-bounds safety (*array*) requires each access to an array element to be within the specified upper and lower bounds of the array. Variable initialization-before-use (*init*) ensures that each variable or individual array element has been assigned a defined value before it is used. Both are typical examples of language-specific properties. Matrix symmetry (*symm*) requires certain two-dimensional arrays to be symmetric. Sensor input usage (*in-use*) is a variation of the general *init*-property which guarantees that each sensor reading passed as an input to the Kalman filter algorithm is actually used during the computation of the output estimate. These two examples are specific to the Kalman filter domain. The final example (*norm*) ensures that certain one-dimensional arrays represent normalized vectors, i.e., that their contents add up to one; it is specific to the data analysis domain.

The safety policies can be expressed in terms of two families of definitions. For each command the policy defines a safety condition and a substitution, which captures how the command changes the environmental information relevant to the safety policy. The rules of the safety policy can then be derived systematically from the standard Hoare rules of the underlying programming language [3].

From our perspective, the safety conditions are the most interesting aspect since they have the greatest bearing on the form of the proof obligations. Table 1 summarizes the different conditions and the domain theories needed to reason about

them. Both variable initialization and usage as well as array bounds certification are logically simple and rely just on propositional and simple arithmetic reasoning, respectively, but can require a lot of information to be propagated throughout the program. The symmetry policy needs reasoning about matrix expressions expressed as a first-order quantification over all matrix entries. The vector norm policy is formalized in terms of the summation over entries in a one-dimensional array, and involves symbolic reasoning over finite sums.

2.2 *Generating Proof Obligations*

For certification purposes, the synthesis system *annotates* the code with mark-up information relevant to the selected safety policy. These annotations are part of the schema and thus are instantiated in parallel with the code fragments. The annotations contain local information in the form of logical pre- and post-conditions and loop invariants, which is then propagated through the code. The fully annotated code is then processed by the VCG, which applies the rules of the safety policy to the annotated code in order to generate the safety conditions. As usual, the VCG works backwards through the code. At each line, the safety conditions are generated and the safety substitutions are applied. The VCG has been designed to be “correct-by-inspection”, i.e., to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic. Hence, the VCG does not implement any optimizations, such as structure sharing on verification conditions (VCs) or even apply any simplifications; in particular, it does not actually apply the substitutions but maintains explicit formal substitution terms. Consequently, the generated VCs tend to be large and must be simplified separately; the more manageable simplified verification conditions (SVCs) which are produced are then processed by a first order theorem prover. The resulting proofs can be sent to a proof checker, e.g., Ivy [19].

The structure of a typical safety obligation (after substitution reduction and simplification) is given in Figure 3. It corresponds to the initialization safety of an assignment within a nested loop (given in Figure 2, including the generated invariants but omitting the postconditions). Most of the hypotheses consist of annotations which have been propagated through the code and are, in the best case, merely irrelevant to the line at hand but, in the worst case, prevent the prover from finding a proof. The proof obligation also contains the local loop invariants together with bounds on `for`-loops. Finally, the conclusion is generated from the safety conditions for the statement given by the corresponding safety policy. Although safety obligations with more complex conclusions can arise with the *symm* and *norm* policies, they always have this general form.

2.3 *Processing Proof Obligations and Connecting the Prover*

The simplified safety obligations are exported as a number of individual proof obligations using TPTP first order logic syntax. A small script then adds the axioms of the domain theory, before the completed proof task is processed by the theorem

```

for(i = 0; i <= 5; i++)
  /*{ inv forall x,y:int . 0<=x<=i-1 && 0<=y<=5 =>
    tmp2_init[x][y]==init
  }*/
for(j = 0; j <= 5; j++)
  /*{ inv forall x,y:int . 0<=x<=5 && 0<=y<=5 =>
    (x<i => tmp2_init[x][y]==init) &&
    (x==i && y<j => tmp2_init[x][y]==init)
  }*/
tmp2[i][j] = id[i][j] - tmp1[i][j];

```

Fig. 2. Generated Code with Annotations

$$\begin{array}{l}
\dots \forall x, y \cdot 0 \leq x \leq 5 \wedge 0 \leq y \leq 5 \Rightarrow \text{sel}(\text{id_init}, x, y) = \text{init} \\
\wedge \forall x, y \cdot 0 \leq x \leq 5 \wedge 0 \leq y \leq 5 \Rightarrow \text{sel}(\text{tmp1_init}, x, y) = \text{init} \\
\dots \forall x, y \cdot 0 \leq x \leq i - 1 \wedge 0 \leq y \leq 5 \Rightarrow \text{sel}(\text{tmp2_init}, x, y) = \text{init} \\
\wedge \forall x, y \cdot 0 \leq y \leq 5 \wedge 0 \leq x \leq 5 \Rightarrow \\
\quad (x < i \Rightarrow \text{sel}(\text{tmp2_init}, x, y) = \text{init} \wedge \\
\quad (y < j \wedge x = i \Rightarrow \text{sel}(\text{tmp2_init}, x, y) = \text{init})) \\
\dots 0 \leq i \leq 5 \wedge 0 \leq j \leq 5 \\
\Rightarrow (\text{sel}(\text{id_init}, i, j) = \text{init} \wedge \text{sel}(\text{tmp1_init}, i, j) = \text{init})
\end{array}
\begin{array}{l}
\left. \begin{array}{l} \\ \\ \end{array} \right\} \text{environmental} \\
\left. \begin{array}{l} \\ \\ \end{array} \right\} \text{information} \\
\left. \begin{array}{l} \\ \\ \end{array} \right\} \text{invariants} \\
\left. \begin{array}{l} \\ \end{array} \right\} \text{index bounds} \\
\left. \begin{array}{l} \end{array} \right\} \text{safety condition}
\end{array}$$

Fig. 3. Structure of a safety obligation

prover. Parts of the domain theory are generated dynamically in order to facilitate reasoning with (small) integers. The domain theory is described in more detail in Section 3.3.

The connection to a theorem prover is straightforward. For provers that do not accept the TPTP syntax, the appropriate TPTP2X-converter is used before invoking the theorem prover. In the experiments, run-time measurement and prover control (e.g., aborting provers) were performed with the same TPTP tools as in the CASC competition [32].

3 Experimental Setup

3.1 Program Corpus

As a basis for the certification experiments we generated annotated programs from four different specifications which were written prior to and independently of the experiments. The size of the generated programs ranges from 431 to 1157 lines of commented C-code, including the annotations. Table 3 in Section 4 gives a more detailed breakdown. The first two examples are `AUTOFILTER` specifications. `ds1` is taken from the attitude control system of NASA's Deep Space One mission [37]. `iss` specifies a component in a simulation environment for the Space Shuttle docking procedure at the International Space Station. In both cases, the

generated code is based on Kalman filter algorithms, which make extensive use of matrix operations. The other two examples are `AUTOBAYES` specifications which are part of a more comprehensive analysis of planetary nebula images taken by the Hubble Space Telescope (see [7,11] for more details). Although these data analysis applications are not safety-critical, they can run onboard a spacecraft thus making the software subject to qualification. `seg` describes an image segmentation problem for which an iterative (numerical) statistical clustering algorithm is synthesized. Finally, `gau` fits an image against a two-dimensional Gaussian curve. This requires a multivariate optimization which is implemented by the Nelder-Mead simplex method. The code generated for these two examples has a substantially different structure from the state estimation examples. First, the numerical optimization code contains many deeply nested loops. Also, some of the loops are convergence loops which have no fixed upper bounds but are executed until a dynamically calculated error value gets small enough. In contrast, in the Kalman filter code, all loops are executed a fixed (i.e., known at synthesis time) number of times. Second, the numerical optimization code accesses all arrays element by element and contains no operations on entire matrices (e.g., matrix multiplication). The example specifications and all generated proof obligations can be found at <http://ase.arc.nasa.gov/autobayes/ijcar>.

3.2 Simplification

Proof task simplification is an important and integral part of our overall architecture. However, as observed before [13,8,30], simplifications—even on the purely propositional level—can have a significant impact on the performance of a theorem prover. In order to evaluate this impact, we used six different rewrite-based simplifiers to generate multiple versions of the safety obligations. We focus on rewrite-based simplifications rather than decision procedures because rewriting is easier to certify: each individual rewrite step $T \rightsquigarrow S$ can be traced and checked independently, e.g., by using an ATP to prove that $S \Rightarrow T$ holds.

Baseline The baseline for all simplifications is given by the rewrite system \mathcal{T}_\emptyset which eliminates the extra-logical constructs (including explicit formal substitutions) which the VCG employs during the construction of the safety obligations. Our original intention was to axiomatize these constructs in first-order logic and then (ab-) use the provers for this elimination step, but that turned out to be infeasible. The main problem is that the combination with equality reasoning produces tremendous search spaces.

Propositional Structure The first two proper simplification levels only work on the propositional structure of the obligations. $\mathcal{T}_{\forall, \Rightarrow}$ splits the few but large obligations generated by the VCG into a large number of smaller obligations. It consists of two rewrite rules $\forall x \cdot P \wedge Q \rightsquigarrow (\forall x \cdot P) \wedge (\forall x \cdot Q)$ and $P \Rightarrow (Q \wedge R) \rightsquigarrow (P \Rightarrow Q) \wedge (P \Rightarrow R)$ which distribute universal quantification and implication, respectively, over conjunction. Each of the resulting conjuncts is then treated as an independent proof task. $\mathcal{T}_{\text{prop}}$ simplifies the propositional structure of the obligations

more aggressively. It uses the rewrite rules

$$\begin{array}{ll}
\neg true \rightsquigarrow false & \neg false \rightsquigarrow true \\
true \wedge P \rightsquigarrow P & false \wedge P \rightsquigarrow false \\
true \vee P \rightsquigarrow true & false \vee P \rightsquigarrow P \\
P \Rightarrow true \rightsquigarrow true & P \Rightarrow false \rightsquigarrow \neg P \\
true \Rightarrow P \rightsquigarrow P & false \Rightarrow P \rightsquigarrow true \\
P \Rightarrow P \rightsquigarrow true & (P \wedge Q) \Rightarrow P \rightsquigarrow true \\
P \Rightarrow (Q \Rightarrow R) \rightsquigarrow (P \wedge Q) \Rightarrow R & \forall x \cdot true \rightsquigarrow true
\end{array}$$

in addition to the two rules in $\mathcal{T}_{\forall, \Rightarrow}$. The rules have been chosen so that they preserve the overall structure of the obligations as far as possible; in particular, conjunction and disjunction are not distributed over each other and implications are not eliminated. Their impact on the clausifier should thus be minimal.

Ground Arithmetic This simplification level additionally handles common extensions of plain first-order logic, i.e., equality, orders, and arithmetic. The rewrite system $\mathcal{T}_{\text{eval}}$ contains rules for the reflexivity of equality and total orders as well as the irreflexivity of strict (total) orders, although the latter rules are not invoked on the example obligations. In addition, it normalizes orders into \leq and $>$ using the rules

$$\begin{array}{ll}
x \geq y \rightsquigarrow y \leq x & \neg x > y \rightsquigarrow x \leq y \\
x < y \rightsquigarrow y > x & \neg x \leq y \rightsquigarrow x > y
\end{array}$$

Since the programs and thus the generated safety obligations contain occurrences of the different symbols, these eliminations have to be applied explicitly by the simplifier. However, the choice of the specific symbols is to some extent arbitrary; choosing for example $<$ instead of $>$ makes no difference. We could even replace the two rules on the right with a single rule $x > y \rightsquigarrow \neg x \leq y$ and thus eliminate all but one ordering symbol but instead decided to minimize the term size rather than the signature size.

$\mathcal{T}_{\text{eval}}$ also contains rules to evaluate ground integer operations (i.e., addition, subtraction, and multiplication), equalities, and partial and strict orders. Moreover, it converts addition and subtraction with one small integer argument (i.e., $n \leq 5$) into Pressburger notation, using rules of the form $n + 1 \rightsquigarrow \text{succ}(n)$ and $n - 1 \rightsquigarrow \text{pred}(n)$. For many safety policies (e.g., *init*), terms of this form are introduced by relativized bounded quantifiers (e.g., $\forall x \cdot 0 \leq x \leq n - 1 \Rightarrow P(x)$) and contain the only occurrences of arithmetic operators. A final group of rules handles the interaction between *succ* and *pred*, as well as with the orders.

$$\begin{array}{ll}
\text{succ}(\text{pred}(x)) \rightsquigarrow x & \text{pred}(\text{succ}(x)) \rightsquigarrow x \\
\text{succ}(x) \leq y \rightsquigarrow x < y & \text{succ}(x) > y \rightsquigarrow x \geq y \\
x \leq \text{pred}(y) \rightsquigarrow x < y & x > \text{pred}(y) \rightsquigarrow x \geq y
\end{array}$$

Language-Specific Simplification The next level handles constructs which are specific to the program verification domain, in particular array-expressions and

conditional expressions, encoding the necessary parts of the language semantics. The rewrite system $\mathcal{T}_{\text{array}}$ adds rewrite formulations of McCarthy's array axioms [18], i.e., $\text{sel}(\text{upd}(a, i, v), j) \rightsquigarrow i = j ? v : \text{sel}(a, j)$ for one-dimensional arrays and similar forms for higher-dimensional arrays. Some safety policies are formulated using arrays of a given dimensionality which are uniformly initialized with a specific value. These are represented by a *constarray*-term, for which similar rules are required, e.g., $\text{sel}(\text{constarray}(v, d), i) \rightsquigarrow v$.

Nested *sel/upd*-terms, which result from sequences of individual assignments to the same array, lead to nested conditionals which in turn lead to an exponential blow-up during the subsequent language normalization step. $\mathcal{T}_{\text{array}}$ thus also contains two rules $\text{true} ? x : y \rightsquigarrow x$ and $\text{false} ? x : y \rightsquigarrow y$ to evaluate conditionals.

In order to properly assess the effect of these domain-specific simplifications, we also experimented with a rewrite system $\mathcal{T}_{\text{array}^*}$, which applies the two *sel*-rules in isolation.

Policy-Specific Simplification The most aggressive simplification level $\mathcal{T}_{\text{policy}}$ uses a number of rules which are fine-tuned to handle specific situations that frequently arise with the individual safety policies. The *init*-policy uses a rule

$$\forall x \cdot 0 \leq x \leq n \Rightarrow (x \neq 0 \wedge \dots \wedge x \neq n \Rightarrow P) \rightsquigarrow \text{true}$$

which is derived from the finite induction axiom to handle the result of simplifying nested *sel/upd*-terms. For *in-use*, we need a single rule $\text{def} = \text{use} \rightsquigarrow \text{false}$, which follows from the fact that the two tokens *def* and *use* used by the policy are distinct. For *symm*, we make use of a lemma about the symmetry of specific matrix expressions: $A + BCB^T$ is already symmetric if (but not only if) the two matrices A and C are symmetric, regardless of the symmetry of B . The rewrite rule

$$\begin{aligned} \text{sel}(A + BCB^T, i, j) &= \text{sel}(A + BCB^T, j, i) \\ &\rightsquigarrow \text{sel}(A, i, j) = \text{sel}(A, j, i) \wedge \text{sel}(C, i, j) = \text{sel}(C, j, i) \end{aligned}$$

formulates this lemma in an element-wise fashion.

For the *norm*-policy, the rules become a lot more specialized and complicated. Two rules are added to handle the inductive nature of finite sums:

$$\begin{aligned} \sum_{i=0}^{\text{pred}(0)} x &\rightsquigarrow 0 \\ P \wedge x = \sum_{i=0}^{\text{pred}(n)} Q(i) &\Rightarrow x + Q(n) = \sum_{i'=0}^n Q(i') \\ &\rightsquigarrow P \wedge x = \sum_{i=0}^{\text{pred}(n)} Q(i) \Rightarrow \sum_{i=0}^n Q(i) = \sum_{i=0}^n Q(i) \end{aligned}$$

The first rule directly implements the base case of the induction; the second rule, which implements the step case, is more complicated. It requires alpha-conversion for the summations as well as higher-order matching for the body expressions, both of which are, however, under explicit control of this specific rewrite rule and not the general rewrite engine, and are implemented directly as Prolog predicates. Note that the right hand side can easily be simplified into *true* by the application of

further rules. A similar rule is required in a very specific situation to substitute an equality into a summation:

$$\begin{aligned} P \wedge (\forall i \cdot 0 \leq i \leq n \Rightarrow x = \text{sel}(f, i)) &\Rightarrow \sum_{i=0}^n \text{sel}(f, i) = 1 \\ \rightsquigarrow P \wedge (\forall i \cdot 0 \leq i \leq n \Rightarrow x = \text{sel}(f, i)) &\Rightarrow \sum_{i=0}^n x = 1 \end{aligned}$$

The above rules capture the central steps of some of the proofs for the *norm*-policy and mirror the fact that these are essentially higher-order inferences.

Another set of rewrite rules handles all occurrences of the random number generator by asserting that the number is within its given range, i.e., $l \leq \text{rand}(l, u) \leq u$. Since no other property of random numbers is used, *rand* is treated as an uninterpreted function symbol.

Normalization The final preprocessing step transforms the obligations into pure first-order logic. It eliminates conditional expressions which occur as top-level arguments of predicate symbols, using rules of the form $P ? T : F = R \rightsquigarrow (P \Rightarrow T = R) \wedge (\neg P \Rightarrow F = R)$ and similarly for partial and strict orders. A number of congruence rules move nested occurrences of conditional expressions into the required positions. Finite sums, which only occur in obligations for the *norm*-policy, are represented with a de Bruijn-style variable-free notation.

Control The simplifications are performed by a small but reasonably efficient rewrite engine implemented in Prolog (cf. Table 3 for runtime information). This engine does not support full AC-rewriting but flattens and orders the arguments of AC-operators. The rewrite rules, which are implemented as Prolog clauses, then do their own list matching but can take the list ordering into account. The rules within each system are applied exhaustively. However, the two most aggressive simplification levels $\mathcal{T}_{\text{array}}$ and $\mathcal{T}_{\text{policy}}$ are followed by a structural “clean-up” phase. This consists of the language normalization followed by the propositional simplifications $\mathcal{T}_{\text{prop}}$ and the finite induction rule. Similarly, $\mathcal{T}_{\text{array}^*}$ is followed by the language normalization and then by $\mathcal{T}_{\forall, \Rightarrow}$ to split the obligations. Table 2 shows the number of rewrite rules for each simplification level, as well as for language normalization and clean-up.

	\mathcal{T}_{\emptyset}	$\mathcal{T}_{\forall, \Rightarrow}$	$\mathcal{T}_{\text{prop}}$	$\mathcal{T}_{\text{eval}}$	$\mathcal{T}_{\text{array}}$	$\mathcal{T}_{\text{array}^*}$	$\mathcal{T}_{\text{policy}}$
simplification	N/A	3	17	42	42	2	61
language norm.	8	8	8	8	8	8	8
clean-up	N/A	N/A	N/A	N/A	31	3	31

Table 2

Number of rewrite rules used in consecutive phases of different simplifications

3.3 Domain Theory

Each safety obligation is supplied with a first-order domain theory. In our case, the domain theory consists of a fixed part which contains 44 axioms, and a set of axioms which is generated dynamically for each proof task. The static axioms define the usual properties of equality and the order relations, as well as axioms

for Pressburger arithmetic and for the domain-specific operators (e.g., array accesses and matrix operations). This part axiomatizes 22 different predicate and function symbols. The dynamic axioms are added because most theorem provers cannot calculate with integers, and to avoid the generation of large terms of the form $\text{succ}(\dots \text{succ}(0) \dots)$. For all integer literals n, m in the proof task, we generate the corresponding axioms of the form $m > n$. For small integers (i.e., $n \leq 5$), we also generate axioms for explicit successor-terms, i.e., $n = \text{succ}^n(0)$ and add a finite induction schema of the form $\forall x. 0 \leq x \leq n \Rightarrow (x = 0 \vee x = 1 \vee \dots \vee x = n)$. In our application domain, these axioms are needed for some of the matrix operations; thus n can be limited to the statically known maximal size of the matrices. The default set of axioms contains all the formulas required for each of the safety policies.

3.4 Theorem Provers

For the experiments, we selected several high-performance theorem provers for untyped first-order formulas with equality. Most of the provers participated in the CASC-19 [31] prover competition in the FOL category. We used two versions of e-setheo [21] which were both derived from the CASC version. For e-setheo-csp03F, Flotter V2.1 [33,34] was used to convert the formulas into a set of clauses instead of the clausifier provided by the TPTP toolset [32]. e-setheo-new is a recent development version with several improvements over the original e-setheo-csp03 version. However, neither of the two versions of e-setheo was tuned in any way for this set of proof tasks. Both versions of Vampire [29] have been taken directly “out of the box”—they are the versions which were used at CASC-19. Spass 2.1 was obtained from the developer’s website [33]. For comparison purposes, we also used Otter V3.2 [20], which has been essentially unchanged since 1996.

In the experiments, we used the default parameter settings and none of the special features of the provers. The only exception is Otter, where the developer provided an alternative parameter setting since the defaults proved unsuitable. For each proof obligation, we limited the run-time to 60 seconds; the CPU time actually used was measured with the TPTP-tools on a 2.4GHz standard Linux PC with 4GB memory.

4 Empirical Results

4.1 Generating and Simplifying Obligations

Table 3 summarizes the results of generating the different versions of the safety obligations. For each of the example specifications, it lists the size of the generated programs (without annotations), the applicable safety policies, the size of the generated annotations (before propagation), and then, for each simplifier, the elapsed time T and the number N of generated obligations. The elapsed times include synthesis of the programs as well as generation, simplification, and file output of the safety obligations; synthesis alone accounts for approximately 90% of the times

				\mathcal{T}_\emptyset		$\mathcal{T}_{\forall, \Rightarrow}$		$\mathcal{T}_{\text{prop}}$		$\mathcal{T}_{\text{eval}}$		$\mathcal{T}_{\text{array}}$		$\mathcal{T}_{\text{array}^*}$		$\mathcal{T}_{\text{policy}}$	
ex	loc	P	loa	T	N	T	N	T	N	T	N	T	N	T	N	T	N
ds1	431	array	0	5.5	11	5.3	103	5.4	55	5.5	1	5.5	1	5.6	103	5.5	1
		init	87	9.5	21	14.1	339	11.3	150	11.0	142	10.5	74	20.1	543	11.4	74
		in-use	61	7.3	19	12.9	453	7.7	59	7.6	57	7.4	21	16.2	682	8.1	21
		symm	75	4.8	17	5.7	101	4.7	21	4.9	21	66.7	858	245.6	2969	70.8	865
iss	755	array	0	24.6	1	28.1	582	24.8	114	24.2	4	24.0	4	27.9	582	24.7	4
		init	88	39.5	2	65.9	957	42.3	202	41.8	194	39.2	71	82.6	1378	39.7	71
		in-use	60	33.4	2	68.1	672	36.7	120	35.7	117	32.6	28	79.1	2409	31.6	1
		symm	87	33.0	1	34.9	185	28.1	35	27.9	35	71.0	479	396.8	3434	66.2	480
seg	517	array	0	3.0	29	3.3	85	2.9	8	2.9	3	3.0	3	3.3	85	3.0	1
		init	171	6.5	56	12.1	464	7.8	172	7.7	130	7.6	121	12.8	470	7.6	121
		norm	195	3.8	54	5.0	155	3.8	41	3.6	30	3.8	32	5.2	157	3.6	14
gau	1039	array	20	21.0	69	24.9	687	21.2	98	21.0	20	20.9	20	24.3	687	21.3	20
		init	118	49.8	85	65.5	1417	54.1	395	53.2	324	53.9	316	66.2	1434	54.3	316

Table 3
Generation of safety obligations

listed under the *array* safety policy. In general, the times for generating and simplifying the obligations are moderate compared to both generating the programs and discharging the obligations. All times are CPU times and have been measured in seconds using the Unix `time` command.

Almost all of the generated obligations are valid, i.e., the generated programs are safe. The only exception is the *in-use*-policy which produces one invalid obligation for each of the `ds1` and `iss` examples. This is a consequence of the original specifications which do not use all elements of the initial state vectors. The invalidity is confined to a single conjunct in one of the original obligations, and since none of the rewrite systems contains a distributive law, the number of invalid obligations does not change with simplification.

The first four simplification levels show the expected results. The baseline \mathcal{T}_\emptyset yields relatively few but large obligations which are then split up by $\mathcal{T}_{\forall, \Rightarrow}$ into a much larger (on average more than an order of magnitude) number of smaller obligations. The next two levels then eliminate a large fraction of the obligations. Here, the propositional simplifier $\mathcal{T}_{\text{prop}}$ alone already discharges between 50% and 90% of the obligations while the additional effect of evaluating ground arithmetic ($\mathcal{T}_{\text{eval}}$) is much smaller and generally well below 25%. The only significant difference occurs for the *array*-policy where more than 80% (and in the case of `ds1` all) of the remaining obligations are reduced to true. This is a consequence of the large number of obligations which have the form $\neg n \leq n \Rightarrow P$ for an integer constant n representing the (lower or upper) bound of an array. The effect of the domain-specific simplifications is at first glance less clear. Using the array-rules ($\mathcal{T}_{\text{array}^*}$) only generally leads to an increase over $\mathcal{T}_{\forall, \Rightarrow}$ in the number of obligations; this even surpasses an order of magnitude for the *symm*-policy. However, in combination with the other simplifications ($\mathcal{T}_{\text{array}}$), most of these obligations can be discharged again, and we generally end up with fewer obligations than before; again, the *symm*-policy is the only exception. The effect of the final policy-specific simplifications is, as

should be expected, highly dependent on the policy. For *in-use* and *norm* a further reduction is achieved, while the rules for *init* and *symm* only reduce the size of the obligations.

4.2 Running the Theorem Provers

Table 4 summarizes the results obtained from running the theorem provers on all proof obligations (except for the invalid obligations from the *in-use*-policy), grouped by the different simplification levels. Each line in the table corresponds to the proof tasks originating from a specific safety policy (*array*, *init*, *in-use*, *symm*, and *norm*). Then, for each prover, the percentage of solved proof obligations and the total CPU time are given. Note that T_{ATP} also includes the actual CPU times for failed proof attempts.

For the fully simplified version ($\mathcal{T}_{\text{policy}}$), all provers are able to find proofs for all tasks originating from at least one safety policy; e-setheo-csp03F can even discharge *all* the emerging safety obligations. This result is central for our application since it shows that current ATPs can in fact be applied to certify the safety of synthesized code, confirming our first hypothesis.

For the unsimplified safety obligations, however, the picture is quite different. Here, the provers can only solve a relatively small fraction of the tasks and leave an unacceptably large number of obligations to the user. The only exception is the *array*-policy, which produces by far the simplest safety obligations. This confirms our second hypothesis: aggressive preprocessing is absolutely necessary to yield reasonable results.

Let us now look more closely at the different simplification stages. Breaking the large original formulas into a large number of smaller but independent proof tasks ($\mathcal{T}_{\vee, \Rightarrow}$) boosts the relative performance considerably. However, due to the large absolute number of tasks, the absolute number of failed tasks also increases. With each additional simplification step, the percentage of solved proof obligations increases further. Interestingly, however, $\mathcal{T}_{\vee, \Rightarrow}$ and $\mathcal{T}_{\text{array}}$ seem to have the biggest impact on performance. The reason seems to be that equality reasoning on deeply nested terms and formula structures can then be avoided, albeit at the cost of the substantial increase in the number of proof tasks. The results with the simplification strategy $\mathcal{T}_{\text{array}^*}$, which only contains the language-specific rules, also illustrates this behavior. The *norm*-policy clearly produces the most difficult safety obligations, requiring essentially inductive and higher-order reasoning. Here, all simplification steps are required to make the obligations go through the first-order ATPs.

The results in Table 4 also indicate there is no single best theorem prover. Even variants of the “same” prover can differ widely in their results. For some proof obligations, the choice of the clausification module makes a big difference. The TPTP-converter implements a straightforward algorithm similar to the one described in [17]. Flotter uses a highly elaborate conversion algorithm which performs many simplifications and avoids exponential increase in the number of generated clauses. This effect is most visible on the unsimplified obligations (e.g., \mathcal{T}_\emptyset under *init*),

		e-setheo03F		e-setheo-new		SPASS		Vampire6.0		Vampire5.0		Otter	
P	N	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}
T_{\emptyset}													
<i>ar</i>	110	96.4	192.4	94.5	284.9	96.4	73.4	95.5	178.1	95.5	102.1	83.6	870.3
<i>in</i>	164	76.8	3000.8	13.1	1759.8	75.0	2898.3	8.5	9224.9	8.5	8251.0	6.6	6534.7
<i>iu</i>	19	57.9	610.8	44.4	612.2	68.4	512.8	57.9	773.1	47.4	645.5	35.7	16.0
<i>sy</i>	18	50.0	387.7	8.3	266.1	38.9	555.3	16.7	744.9	16.7	723.6	16.7	847.9
<i>no</i>	54	51.9	1282.4	51.9	1341.0	51.9	1224.2	50.0	1316.5	48.1	1327.1	31.5	1537.7
$T_{V, \Rightarrow}$													
<i>ar</i>	1457	99.0	903.4	94.2	5925.0	99.8	217.0	99.9	240.5	99.8	152.4	99.5	714.5
<i>in</i>	3177	88.4	3969.4	91.7	20784.8	97.4	8732.2	95.0	14482.2	93.5	14203.4	92.0	19310.5
<i>iu</i>	1123	59.3	819.1	96.4	4100.3	99.1	1733.5	95.3	4183.7	94.3	4206.8	96.6	3014.2
<i>sy</i>	286	93.4	1785.9	90.6	2341.0	88.5	3638.7	90.2	3315.8	91.3	1789.2	80.8	2160.1
<i>no</i>	155	85.8	1422.1	73.5	2552.5	84.5	1572.0	87.7	1359.9	87.1	1276.0	76.1	2051.6
T_{prop}													
<i>ar</i>	275	99.3	278.2	76.4	4080.8	99.3	157.5	99.3	187.5	99.3	132.6	97.1	621.4
<i>in</i>	919	94.7	4239.4	73.0	17472.2	92.8	5469.7	84.9	10598.0	83.2	10546.8	78.8	13461.6
<i>iu</i>	177	86.4	1854.0	77.4	2768.2	94.9	1008.3	70.1	3806.2	65.0	3960.6	78.5	2729.2
<i>sy</i>	56	66.1	1476.2	51.8	1944.4	48.2	1911.3	58.9	1596.7	58.9	1424.8	19.6	2002.2
<i>no</i>	41	46.3	1361.2	41.5	1484.6	41.5	1478.2	53.7	1286.7	51.2	1275.3	9.8	2036.2
T_{eval}													
<i>ar</i>	28	100.0	16.2	100.0	19.7	100.0	10.4	100.0	12.7	100.0	1.7	100.0	20.7
<i>in</i>	790	94.6	3944.2	94.1	8288.0	93.3	4380.1	82.5	10239.0	82.0	9040.2	85.7	7983.6
<i>iu</i>	172	86.0	1852.2	83.1	2305.2	94.8	1023.1	69.8	3718.1	67.4	3561.1	64.0	3715.3
<i>sy</i>	56	66.1	1451.1	66.1	1500.4	51.8	1716.0	62.5	1455.5	58.9	1389.8	26.8	1828.2
<i>no</i>	30	53.3	859.4	13.3	1575.8	50.0	940.5	66.7	736.7	53.3	858.0	50.0	1007.7
T_{array}													
<i>ar</i>	28	100.0	15.4	100.0	19.8	100.0	10.4	100.0	12.7	100.0	1.7	100.0	20.2
<i>in</i>	582	100.0	527.6	100.0	823.9	99.7	875.8	100.0	1401.3	99.0	785.1	95.7	2468.7
<i>iu</i>	47	100.0	323.9	100.0	343.2	100.0	171.3	100.0	262.6	87.2	525.2	85.1	613.7
<i>sy</i>	1337	100.0	1104.3	99.9	1629.3	99.4	746.4	99.1	963.9	99.0	922.7	98.2	872.9
<i>no</i>	32	59.4	678.4	18.8	1583.1	59.4	709.7	62.5	791.7	50.0	858.6	59.4	896.2
T_{array}^*													
<i>ar</i>	1457	99.9	916.4	94.2	5918.0	99.9	210.8	99.9	240.6	99.9	153.1	99.5	711.9
<i>in</i>	3825	99.7	3412.3	96.3	13536.1	99.5	4574.9	99.8	4952.1	98.4	6000.1	95.5	13680.4
<i>iu</i>	3089	99.8	2438.4	99.4	5139.0	99.8	889.2	99.8	793.5	99.6	925.9	99.5	1427.8
<i>sy</i>	6403	99.9	5317.4	99.7	11787.7	99.7	3385.1	99.6	3277.3	99.6	1807.0	83.5	1682.8
<i>no</i>	157	86.0	1306.8	72.6	2670.8	86.0	1351.3	86.6	1449.9	86.0	1276.2	76.4	2078.3
T_{policy}													
<i>ar</i>	26	100.0	15.0	100.0	17.7	100.0	9.9	100.0	12.0	100.0	1.6	100.0	19.7
<i>in</i>	582	100.0	529.2	100.0	827.9	99.5	875.2	100.0	1418.9	99.0	782.5	95.7	2456.7
<i>iu</i>	20	100.0	281.7	100.0	329.7	100.0	170.7	100.0	262.6	70.0	524.8	65.0	601.1
<i>sy</i>	1345	100.0	1104.6	99.9	1640.5	99.4	760.0	99.1	1048.8	99.0	926.9	99.3	501.1
<i>no</i>	14	100.0	9.0	57.1	375.8	100.0	26.2	100.0	108.0	71.4	241.8	100.0	69.7

Table 4

Results and times for *array* (*ar*), *init* (*in*), *in-use* (*iu*), *symm* (*sy*), and *norm* (*no*) policy.

where Spass and e-setheo-csp03F—which both use the Flotter classifier—perform substantially better than the other provers.

Since our proof tasks are generated directly by a real application and are not

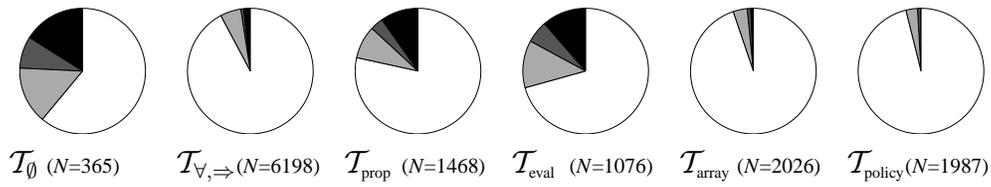


Fig. 4. Distribution of easy ($T_{proof} < 1s$, white), medium ($T_{proof} < 10s$, light grey), difficult ($T_{proof} < 60s$, dark grey) proofs, and failing proof tasks (black) for the different simplification stages (prover: e-setheo-csp03F). N is the total number of proof tasks at each stage.

hand-picked for certain properties, many of them are (almost) trivial—even in the unsimplified case. Figure 4 shows the resources required for the proof tasks as a series of pie charts for the different simplification stages. All numbers are obtained with e-setheo-csp03F; the figures for the other provers look similar. Overall, the charts reflect the expected behavior: with additional preprocessing and simplification of the proof obligations, the number of easy proofs increases substantially and the number of failing proof tasks decreases sharply from approximately 16% to zero. The relative decrease of easy proofs from $T_{V,=>}$ to T_{prop} and T_{eval} is a consequence of the large number of easy proof tasks already discharged by the respective simplifications.

4.3 Difficult Proof Tasks

Since all proof tasks are generated in a uniform manner through the application of a safety policy by the VCG, it is obvious that many of the difficult proof tasks share some structural similarities. We have identified three classes of hard examples; these classes are directly addressed by the rewrite rules of the policy-specific simplifications.

Most safety obligations generated by the VCG are of the form $\mathcal{A} \Rightarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$ where the \mathcal{B}_i are variable disjoint. These obligations can be split up into n smaller proof obligations of the form $\mathcal{A} \Rightarrow \mathcal{B}_i$ and most theorem provers can then handle these smaller independent obligations much more easily than the large original.

The second class contains formulas of the form $\text{symm}(r) \Rightarrow \text{symm}(\text{diag-updates}(r))$. Here, r is a matrix variable which is updated along its diagonal, and we need to show that r remains symmetric after the updates. For a 2x2 matrix and two updates (i.e., $r_{00} = x$ and $r_{11} = y$), we obtain the following simplified version of an actual proof task:

$$\begin{aligned} \forall i, j \cdot (0 \leq i, j \leq 1 \Rightarrow \text{sel}(r, i, j) = \text{sel}(r, j, i)) \Rightarrow \\ (\forall k, l \cdot (0 \leq k, l \leq 1 \Rightarrow \\ \text{sel}(\text{upd}(\text{upd}(r, 1, 1, y), 0, 0, x), k, l) = \text{sel}(\text{upd}(\text{upd}(r, 1, 1, y), 0, 0, x), l, k))). \end{aligned}$$

This already pushes the provers to their limits—e-setheo cannot prove this while Spass succeeds here but fails if the dimensions are increased to 3x3, or if three updates are made. In our examples, matrix dimensions up to 6x6 with 36 updates occur, yielding large proof obligations of this specific form which are not provable by current ATPs without further preprocessing.

Another class of seemingly trivial but hard examples, which frequently shows up in the *init*-policy, results from the expansion of deeply nested *sel/upd*-terms. These problems have the form

$$\forall i, j. 0 \leq i \leq n \wedge 0 \leq j \leq n \Rightarrow (\neg(i = 0 \wedge j = 0) \wedge \dots \wedge \neg(i = n \wedge j = n)) \Rightarrow \text{false}$$

and soon become intractable for the classifier, even for small n ($n = 2$ or $n = 3$), although the proof would be easy after a successful classification.

4.4 Policy-Specific Domain Theories

The domain theory described in Section 3.3 and used in the experiments summarized in Table 4 contains *all* axioms required to prove *any* of the obligations; in particular, it also contains axioms which are specific to the symbols used only in one policy and which should thus not be required for any obligation from the other policies. However, experience shows that the ATPs have problems detecting such redundant axioms [8,27,30].

		e-setheo03F						Vampire 5.0					
		reduced theory			full theory			reduced theory			full theory		
simp.	policy	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}
\mathcal{T}_{\emptyset}	<i>array</i>	96.4	61.5	0.56	96.4	131.2	1.19	96.6	0.4	0.01	95.5	3.1	0.03
	<i>init</i>	86.6	868.8	5.30	76.8	928.9	5.66	4.5	0.0	0.00	8.5	22.0	0.13
	<i>in-use</i>	57.9	32.0	1.68	57.9	62.4	3.28	47.4	7.4	0.4	47.4	7.5	0.40
$\mathcal{T}_{\forall, \Rightarrow}$	<i>array</i>	99.9	633.2	0.43	99.0	782.1	0.54	99.9	22.6	0.02	99.8	31.9	0.02
	<i>init</i>	98.6	7259.1	2.28	88.4	2155.1	0.68	93.5	1730.5	0.54	93.5	1845.2	0.58
	<i>in-use</i>	98.0	686.5	0.61	59.3	456.6	0.41	94.4	216.0	0.19	94.3	228.3	0.20
$\mathcal{T}_{\text{prop}}$	<i>array</i>	99.3	125.6	0.46	99.3	156.8	0.57	99.3	8.4	0.03	99.3	12.1	0.04
	<i>init</i>	95.2	5467.7	5.95	94.7	1274.4	1.39	83.0	1107.5	1.21	83.2	1258.5	1.37
	<i>in-use</i>	87.0	179.0	1.01	86.4	283.4	1.60	65.5	100.0	0.57	65.0	101.6	0.57
$\mathcal{T}_{\text{eval}}$	<i>array</i>	100.0	12.7	0.45	100.0	16.2	0.58	100.0	1.5	0.05	100.0	1.7	0.06
	<i>init</i>	94.7	5240.3	6.63	94.6	1342.1	1.70	82.3	491.4	0.62	82.0	478.3	0.61
	<i>in-use</i>	86.6	244.8	1.42	86.0	281.7	1.64	66.9	93.0	0.54	67.4	123.4	0.72
$\mathcal{T}_{\text{array}}$	<i>array</i>	100.0	12.4	0.44	100.0	15.4	0.55	100.0	1.4	0.05	100.0	1.7	0.06
	<i>init</i>	100.0	354.5	0.61	100.0	527.6	0.91	99.3	443.1	0.76	99.0	423.3	0.73
	<i>in-use</i>	100.0	31.4	0.67	100.0	203.4	4.33	87.2	39.5	0.84	87.2	42.9	0.91
$\mathcal{T}_{\text{array}^*}$	<i>array</i>	99.9	616.3	0.42	99.9	795.4	0.55	99.9	23.0	0.02	99.9	32.5	0.02
	<i>init</i>	99.8	2353.4	0.62	99.7	2807.3	0.73	98.2	1923.4	0.50	98.4	2200.5	0.58
	<i>in-use</i>	99.8	1485.6	0.48	99.8	2015.9	0.65	99.6	65.7	0.02	99.6	81.9	0.03
$\mathcal{T}_{\text{policy}}$	<i>array</i>	100.0	11.7	0.45	100.0	15.0	0.58	100.0	1.4	0.05	100.0	1.6	0.06
	<i>init</i>	100.0	363.3	0.62	100.0	529.2	0.91	99.3	443.2	0.76	99.0	420.7	0.72
	<i>in-use</i>	100.0	19.4	0.97	100.0	187.9	9.39	70.0	39.0	1.95	70.0	42.5	2.13

Table 5
Proof results and times—policy-specific domain theories

In order to evaluate the effect of redundant axioms in our case, we used a reduced domain theory for the *array*, *init*, and *in-use* safety policies and then re-ran e-setheo-csp03F and Vampire5.0. The reduced domain theory uses the same dynamic axiom generator as the full theory but omits seven axioms that specify the

behavior of matrix operations (i.e., addition, subtraction, multiplication, transposition, and inversion) which do not occur in the obligations resulting from the above policies. The reduced set thus contains 37 axioms and 17 symbols.

Table 5 summarizes this experiment and gives the results and times for both the reduced and the original full domain theory. Note that T_{proof} only includes the CPU times for successful proof attempts; T_{mean} is the average CPU time for these cases. There is no uniform trend, however—depending on the ATP, the applied simplifications, and the safety policy, either more or less tasks are proven while the proofs can become faster or slower. This non-uniform behavior is likely to be a consequence of the internal architecture of the provers. Both e-setheo and Vampire implement multiple search strategies and then derive a schedule from the proof task. However, e-setheo’s scheduling algorithm seems to be more sensitive to the changes than Vampire’s. e-setheo never fails to prove proof tasks by switching to the reduced domain theory and sometimes finds a substantial number of additional proofs, in particular for unsimplified or almost unsimplified tasks. The average proof times are usually slightly better but they can vary widely—up to one order of magnitude in both directions (e.g., *init* with $\mathcal{T}_{\text{prop}}$ and *in-use* with $\mathcal{T}_{\text{policy}}$). In contrast, the variation in Vampire’s results and proof times is a lot smaller and appears to be statistically insignificant.

5 Proof Checking

For certification purposes, explicit evidence must be provided that none of the individual tool components can yield incorrect results. The VCG is designed so that it can be manually inspected for correctness and, similarly, the rewrite rules used for simplification can be inspected and even individually proven correct. However, the state-of-the-art high performance ATPs in our system use complicated calculi, elaborate data structures, and optimized implementations to increase their deductive power and obtain fast results. This makes a formal verification of their correctness impossible in practice. Although they have been extensively validated by the theorem proving community (using the TPTP benchmark library), the ATPs remain the weakest link in the certification chain.

As an alternative to formally verifying the ATPs, they can be extended to generate sufficiently detailed proofs which can then be independently checked by a small and thus verifiable algorithm. This is the same approach we have taken in extending the synthesis system to generate annotated code, rather than directly verifying the synthesizer. However, although this idea is very simple in theory, there are currently (as of 2004) almost no proof checkers for high-performance ATPs. This has a number of practical reasons:

- Many ATPs simply do not generate the required detailed proofs, mainly due to implementation effort and run-time requirements.
- On-going changes in the ATP require frequent updates and re-verification of the proof checker.

- Most ATPs contain a large number of high-level inference rules (such as *splitting*) which cannot easily be expanded into sequences of low-level inferences, making the proof checker more complicated and thus hard to verify.
- Almost all ATPs work on problems in CNF, so the proof checking can only be done on that level, and not on the FOL level. Since clausification is often a large part of a proof, this reduces the confidence that proof checking can bring.

The notable exception is the IVY system [19] that we used in our experiments. IVY combines a clausifier and the Otter theorem prover with a proof checker. Because IVY is implemented within the ACL2 logic [16], both the clausifier and the proof checker have been verified. IVY thus provides the same functionality as a verified prover for first-order logic, but achieves relatively good performance by using Otter to find the proofs. However, the formal verification of the IVY clausifier and proof checker are based up finite domains [19] but since the implementation of IVY does not actually rely on the finiteness, the system can be used for arbitrary proof tasks.

Another limitation of IVY is shared by all existing clausification algorithms. Clausifiers usually take a first-order formula apart and reorganize the pieces using non-logical graph-based techniques. Thus, establishing traceability between the clauses (or literals) and the positions they had in the original formula would require substantial effort and has not yet been attempted in practical implementations. While this restriction makes it impossible to translate the clausal proof back into a first-order representation, it also has a negative influence on the prover's behavior. Many ATPs can be sped up considerably if it is known which parts of the formula are axioms and which belong to the conjecture. This distinction allows the prover to apply goal-oriented rules. Our application naturally provides this information, but this is ignored by IVY. Thus, the Otter prover used within IVY can only use Otter's auto-mode which is rather weak for our proof obligations. Experiments also revealed that IVY has problems in handling the full axiom set. With the policy-specific domain theory of Section 4.4, we obtained the following results for the fully simplified tasks: 100% in 34.8s for the *array* property, 89.2% in 4929.2s for *init*, and 65.0% in 657.5s for *in-use*.

6 Traceability

The successful application of an automated theorem prover to verification and, in particular, to certification problems such as we have described here, places more requirements on an ATP than just raw deductive power. Since the aim of certification is to provide explicit evidence that software meets a specified standard of safety, it is important that domain experts can assess the evidence for successful checks of the safety properties and any places where it is violated.

Safety checks are typically carried out during code reviews [24], where reviewers look in detail at each line of the code and check the individual safety properties statement by statement. The successful outcome of a code review, therefore,

consists of the code, where each statement is labelled with either “complies with property P ”, or with information about the violation. This requires two things: (i) tracing information which links the safety obligations (or their proofs) to specific lines of code in the program being certified, and (ii) a summary which relates this detailed information back to the specification and the safety policy, while drawing attention to specific areas of concern.

Existing techniques for addressing the tracing problem [14], however, need to be extended for our purposes. The required information about code locations needs to be threaded through all stages of our certification architecture (cf. Figure 1). Only then can the tracing information be obtained and displayed in the appropriate way. Even if we just want to know if a certain line in the code fulfills a safety property, the location information still needs to be threaded through the VCG and the simplifier.

To get more detailed information, however, the tracing has to be threaded through the ATP and into the proof it generates. For example, the analysis needs to reveal which other lines of code are actually required to satisfy a property. For variable initialization safety this can mean computing on which line the variable that is accessed in the current statement was initialized. The extraction of this information requires knowledge of which parts of the formula contributed to the proof, as well as their location information. This problem is aggravated by the fact that most theorem provers work on clausal normal form, which usually loses the important location information.

In general, useful information extracted from an ATP can be used for purposes of auto-generating documentation. In [6], we describe a *safety documentation* tool, which generates a natural language description explaining the safety of a program, by converting the VCs into text. This could be extended by carrying out some symbolic evaluation from the simplifier as an intermediate step to using the full proofs.

7 Conclusions

We have described a system for the automated certification of safety properties of NASA state estimation and data analysis software. The system uses a generic VCG together with explicit safety policies to generate policy-specific safety obligations which are then automatically processed by a first-order ATP. We have evaluated several state-of-the-art ATPs on more than 25,000 proof tasks generated by our system. With “out-of-the-box” provers, only about two-thirds of the tasks could be proven but after aggressive simplification, most of the provers could solve almost all emerging tasks. In order to see the effects of simplification more clearly, we experimented with several preprocessing stages. Figure 5 shows (on the left) the overall results for the different stages and provers.

However, the percentage of solved *proof* tasks is a very ATP-centric metric; it is also somewhat artificial because it can easily be boosted by splitting the original obligations into a larger number of small proof tasks (cf. the results for \mathcal{T}_0

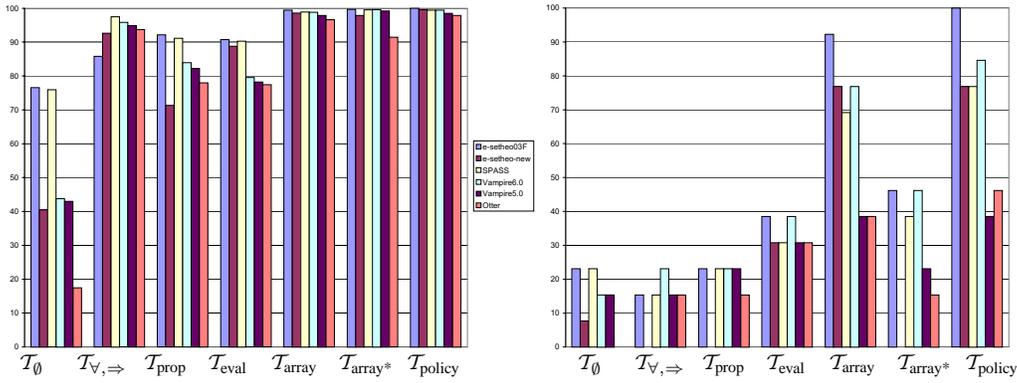


Fig. 5. Comparison of proof results and certification results

and $\mathcal{T}_{V, \Rightarrow}$). An empirically more meaningful metric for the success of this ATP-application is the percentage of solved *certification* tasks, i.e., the relative number of cases in which the ATP solves *all* safety obligations resulting from the application of a safety policy to an individual program. Figure 5 shows this metric (on the right) for the different simplification stages and provers. This change in perspective leads to a dramatic change in the interpretation of the same results. The two major differences are: (i) the numbers go down and (ii) the variation between the provers becomes larger. Both differences result from a few hard proof tasks which are distributed evenly over the different certification tasks. Consequently, empirical success is a lot harder to come by if it is defined in terms of the application rather than in terms of the TPTP corpus. However, as our experiments show it is clearly not impossible.

It is well-known that, in contrast to traditional mathematics, software verification hinges on large numbers of mathematically shallow (in terms of the concepts involved) but structurally complex proof tasks, yet current provers are not well suited to this. Since the propositional structure of a formula is of great importance, we believe that clausification algorithms should integrate more simplification and split goal tasks into independent subtasks. Likewise, certain application-specific constructs (e.g., *sel/upd*) can easily lead to proof tasks which cannot be handled by current ATPs. The reason is that simple manipulations on deep terms, when combined with equational reasoning, can result in a huge search space.

Our certification approach relies on proof checking to ensure that the proofs are correct. However, the ATPs fare less well when assessed in these terms and more research efforts should go into the development of proof checkers for high-performance provers. Moreover, it is very difficult to get useful information from the ATPs which can then be used as a basis for documentation. Since we believe that software certification is potentially one of the main application areas for automated theorem proving, this is clearly another area in need of further work.

With our approach to certification of auto-generated code, we are able to automatically produce safety certificates for code of considerable length and structural complexity. By combining rewriting with state-of-the-art automated theorem

proving, we obtain a safety certification tool which compares favorably with tools based on static analysis (see [4] for a comparison). Our current efforts focus on extending the certification system in a number of areas. One aim is to develop a certificate management system, along the lines of the Programatica project [25]. In another thread of future work we will experiment with other reasoning systems and decision-based tools (such as PVS, Vampire, and Simplify) to process our verification conditions. We also plan to combine our work on certification with automated safety and design document generation [6] tools that we are developing. Finally, we continue to integrate additional safety properties.

References

- [1] A. Appel, N. Michael, A. Stump, and R. Virga. “A Trustworthy Proof Checker”. *JAR*, 31(3–4):191–229, 2003.
- [2] W. Bibel and P. H. Schmitt, (eds.). *Automated Deduction — A Basis for Applications*. Kluwer, 1998.
- [3] E. Denney and B. Fischer. “Correctness of Source-Level Safety Policies”. In *Proc. FM 2003: Formal Methods, LNCS 2805*, pp. 894–913. Springer, 2003.
- [4] E. Denney, B. Fischer, and J. Schumann. “Adding Assurance to Automatically Generated Code”. In *Proc. 8th IEEE Intl. Symp. High Assurance System Engineering*, pp. 297–299. IEEE Comp. Soc. Press, 2004.
- [5] E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software, 2004. In *Proc. IJCAR’04*. To appear.
- [6] E. Denney and R. P. Venkatesan. “A generic software safety document generator”. In *Proc. 10th AMAST*. To appear, 2004.
- [7] B. Fischer, A. Hajian, K. Knuth, and J. Schumann. Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond. In *Proc. 23rd MaxEnt*, 2004. <http://ase.arc.nasa.gov/people/fischer/>.
- [8] B. Fischer. *Deduction-Based Software Component Retrieval*. PhD thesis, U. Passau, 2001. <http://elib.ub.uni-passau.de/opus/volltexte/2002/23/>.
- [9] C. Flanagan and K. R. M. Leino. “Houdini, an Annotation Assistant for ESC/Java”. In *Proc. FME 2001: Formal Methods for Increasing Software Productivity, LNCS 2021*, pp. 500–517. Springer, 2001.
- [10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. “Extended static checking for Java”. In: *PLDI*, pp. 234–245. ACM, 2002.
- [11] B. Fischer and J. Schumann. “Applying AutoBayes to the Analysis of Planetary Nebulae Images”. In *Proc. 18th ASE*, pp. 337–342. IEEE Comp. Soc. Press, 2003.
- [12] B. Fischer and J. Schumann. “AutoBayes: A System for Generating Data Analysis Programs from Statistical Models”. *J. Functional Programming*, 13(3):483–508, 2003.
- [13] B. Fischer, J. Schumann, and G. Snelling. “Deduction-Based Software Component Retrieval”. Volume III of Bibel and Schmitt [2], pp. 265–292. 1998.
- [14] R. Fraer. “Tracing the Origins of Verification Conditions”. In *Proc. 5th AMAST*, pp. 241–255, 1996.

- [15] P. Homeier and D. Martin. “Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator”. In *Proc. TPHOLS 94*, pp. 269–284. Springer, 1994.
- [16] M. Kaufmann and J S. Moore. “An Industrial Strength Theorem Prover for a Logic Based on Common Lisp”. *Software Engineering*, 23(4):203–213, 1997.
- [17] D. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, 1978.
- [18] J. McCarthy. “Towards a Mathematical Science of Computation”. In *Proc. IFIP Congress 62*, pp. 21–28. North-Holland, 1962.
- [19] W. McCune and O. Shumsky. “System description: IVY”. In *Proc. 17th CADE, LNAI 1831*, pp. 401–405. Springer, 2000.
- [20] W. McCune and L. Vos. “Otter—The CADE-13 Competition Incarnations”. *JAR*, 18(2):211–220, April 1997.
- [21] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann. and K. Mayr. “The Model Elimination Provers SETHEO and E-SETHEO”. *JAR*, 18:237–246, 1997.
- [22] G. C. Necula. “Proof-Carrying Code”. In *Proc. 24th POPL*, pp. 106–19. ACM, 1997.
- [23] G. C. Necula and P. Lee. “The Design and Implementation of a Certifying Compiler”. In: *PLDI*, pp. 333–344. ACM, 1998.
- [24] S. Nelson and J. Schumann. “What makes a Code Review Trustworthy?”. In *Proc. 37th Annual Hawaii International Conference on System Sciences*. IEEE, 2004.
- [25] The Programatica Team. “Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell”. In *Proc. High Confidence Software and Systems Conf.*, Baltimore, MD, April 2003.
- [26] W. Reif. “The KIV Approach to Software Verification”. In *KORSO: Methods, Languages and Tools for the Construction of Correct Software, LNCS 1009*, pp. 339–370. Springer, 1995.
- [27] W. Reif and G. Schellhorn. “Theorem Proving in Large Theories”. Volume III of Bibel and Schmitt [2], pp. 265–292. 1998.
- [28] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. *Structured Specifications and Interactive Proofs with KIV*. Volume II of Bibel and Schmitt [2], pp. 225–241, 1998.
- [29] A. Riazanov and A. Voronkov. “The Design and Implementation of Vampire”. *AI Communications*, 15(2–3):91–110, 2002.
- [30] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.
- [31] G. Sutcliffe and C. Suttner. CASC Home Page. <http://www.tptp.org/CASC>.
- [32] G. Sutcliffe and C. Suttner. TPTP Home Page. <http://www.tptp.org>.
- [33] C. Weidenbach. SPASS Home Page. <http://spass.mpi-sb.mpg.de>.
- [34] C. Weidenbach, B. Gaede, and G. Rock. “Spass and Flotter version 0.42”. In *Proc. 13th CADE, LNAI 1104*, pp. 141–145. Springer, 1996.
- [35] M. Whalen, J. Schumann, and B. Fischer. “AutoBayes/CC — Combining Program Synthesis with Automatic Code Certification (System Description)”. In *Proc. 18th CADE, LNAI 2392*, pp. 290–294. Springer, 2002.
- [36] M. Whalen, J. Schumann, and B. Fischer. “Synthesizing Certified Code”. In *Proc. FME 2002, LNCS 2391*, pp. 431–450. Springer, 2002.
- [37] J. Whittle and J. Schumann. Automating the Implementation of Kalman Filter Algorithms, 2004. In review.

MoMM - Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics

Josef Urban¹

*Dept. of Theoretical Computer Science
Charles University
Malostranské nám. 25, Praha, Czech Republic*

Abstract

MoMM (in the narrower sense) is a tool allowing fast interreduction of a high number of clauses, dumping and fast-loading of the interreduced clause sets, and their use for real-time retrieval of matching clauses in an interactive mode. MoMM's main task is now providing these services for the world's largest body of formalized mathematics - the Mizar Mathematical Library (MML), which uses a richer formalism than just pure predicate logic. This task leads to a number of features (strength, speed, memory efficiency, dealing with the richer Mizar logic, etc.) required from MoMM, and we describe the choices taken in its implementation corresponding to these requirements.

An important part of MoMM (in the wider sense) are the tools exporting the richer logic of MML into the clause-like format suitable for fast interreduction, and the tools allowing the use of MoMM as an interactive advisor for the authors of Mizar articles. These tools and choices taken in their implementation are also described here. Next we present some results of the interreduction of MML, which provide an interesting information about subsumption and repetition in the MML and can be used for its refactoring. This interreduction reveals that more than 2 percent of the main MML theorems are subsumed by others, and that for more than 50 percent of the internal lemmas proved by Mizar authors MoMM can provide a useful advice for their justification. Finally some problems and possible future work are discussed.

1 Motivation, Basic Ideas and Related Work

1.1 Motivation

The primary goal in the design of MoMM was to have a fast tool for fetching matching theorems from the Mizar Mathematical Library (MML) [Rudnicki 1992], [Rudnicki and Trybulec 1999], which already now contains some 40000 theorems. The typical usage of such a tool is assistance during authoring Mizar articles. A

¹ Email: urban@kti.ms.mff.cuni.cz

user writes a formula which he wants to justify, possibly with a partial justification, like

A3: ψ by A1,A2;

and the tool tries to find Mizar theorems which subsume it (here we would look for a theorem subsuming formula “(A1 & A2) implies ψ ”) and thus complete the justification. E.g. if in our example theorem Th1 was found, the following inference

A3: ψ by A1,A2,Th1;

should then usually be accepted by the Mizar checker [Wiedijk 2000].

This goal is closely related to that of the interreduction of a set of clauses. We do not want to load redundant theorems into our tool, and once it has the subsumption functionality, we can use it to find the subsumed theorems first and not to load them for the normal interactive usage. Detecting subsumed theorems is not only useful for our tool, it is generally useful for the maintenance of a large body of mathematics like the MML.

Having a good interreduction tool in turn leads to even more interesting applications. The main (exported) Mizar theorems usually have longer proofs written in the Jaskowski [Jaskowski 1934] style containing a number of lemmas with various degrees of difficulty. While the main Mizar theorems are often used (and thus also searched for) by Mizar authors and the probability of an unspotted redundant theorem should be quite low, the various lemmas used for proving the main theorems are usually forgotten once they served their purpose in the proof. If such lemmas are correctly exported from the context provided by the layered supposition structure of the Jaskowski-style proofs, they become universally valid formulas, which can be interreduced and used for subsumption in the same way as the main Mizar theorems (and together with them). Such large-scale (there are now almost 900000 clauses generated from the Mizar lemmas) interreduction can give us

- the strongest versions of such internal lemmas
- the usage counts of these strongest versions, which give a general “usefulness” criterion for classification of all lemmas (as well as theorems), and can be used e.g. for upgrading the most frequent lemmas into regular Mizar theorems
- the possibility of “refactoring” of the Jaskowski-style proofs by replacing the proofs of repeated or subsumed lemmas with just a single reference to the strongest versions
- an interesting general statistics about subsumption and repetition in a large body of formal mathematics

As we show later in this article, the last item already now really justifies one of the interpretations of the acronym **MoMM**:

Most of Mizar Matches

I.e. more than a half of the lemmas written by the Mizar authors are subsumed by another lemma or theorem. This does not necessarily mean that the subsumed

lemmas are redundant at the particular point of a Mizar proof. Replacing one big proof step with two smaller proof steps, typically with an explicit instantiations of the universal formulas that justify them, can be useful both for the limited (but fast) Mizar checker and for the human readers. This statistics just means that for more than a half of the lemmas a useful advice can be given by MoMM. This is a strong evidence that formalization together with quite simple “brute force” methods (“store and index everything, unless it is redundant”) can give some very real benefits to mathematicians. We can only speculate about similar statistics for all of mathematics, if we ever succeed in providing some common formal language for it or at least for a large part of it. The author’s personal opinion is that it is quite a safe bet to replace ‘Mizar’ with ‘Mathematics’ in the above given acronym interpretation. Again, that does not have to mean that only less than a half of e.g. the “original” results published in mathematical journals is really original, but we conjecture that for many lemmas a useful advice could be given in exactly the same way as we do for Mizar and that the repetition rate might be even higher due to the limitation of the human memory and text processing, the lack of a common formal language and the lack of a semantically searchable central repository.

1.2 *Related Work and Basic Ideas*

Mizar substantially differs from many other proof assistants [[Wiedijk 2003](#)] in the following aspects important for the work presented here:

- It is practically a classical first-order system², which makes it suitable for cooperation with the first-order theorem proving technology. Many other proof assistants use some kind of higher-order logic, some use non-classical logics.
- It has a large, maintained and internally consistent repository of formalized mathematics written mostly by humans, which can be used as a source for all kinds of data-mining experiments.

Especially the second aspect forces us to look for solutions which are as efficient as possible in terms of memory consumption and speed, while systems with hundreds or at most thousands of theorems can afford to neglect such issues to a great extent. Closer to the MoMM’s area of application in the given aspects could actually be large ontologies and common sense systems like OpenCyc [[OpenCycUrl](#)] or Cyc, where first-order representation seems to be used (at least at some level).

Related is also the normal database technology that is being used e.g. in the MMLQuery [[Bancerek 2003](#)] system for Mizar and also in corresponding projects done for other systems. This technology is good for various symbol-based queries, like “Give me all theorems containing all symbols from theorem T1”, but it is hard to extend to queries involving the term structure.

² Its axiomatics is the Tarski-Grothendieck set theory [[Tarski 1939](#), [TARSKI](#)] which is very close to ZFC. Therefore the language allows infinite schemes of axioms (like Replacement) and theorems parameterized by second-order variables. The usage of such second-order features is however quite rare and limited by the language.

On the other side, there is the full automated theorem proving, used e.g. in the MPTP [Urban 2004] project. Initial experiments with selecting suitable hints and attempting a proof of an arbitrary theorem from the MML are described in [Urban 2004], but at the time of writing this article, MPTP is not yet an interactive tool available to Mizar authors. This functionality is planned in some short time, but even when it is implemented, a full theorem proving attempt will typically take a longer time, and will be able to use only a (well chosen) fraction of MML theorems, because of the high sensitivity of full theorem proving to the initial number of formulas.

What we want for a system like MoMM are thus just the very efficient indexing methods developed for automated theorem provers (ATPs), which have already enabled ATPs to quickly find matchings or unifications in clause sets containing millions of clauses, without those parts of ATPs which generate new clauses. MoMM is based on such indexing methods implemented in the E prover [Schulz 2002], and it started as a modified version of the CSSCPA subsumption tool written by Stephan Schulz and Geoff Sutcliffe.

2 Basic Description

The MoMM system is available at

<http://kti.ms.mff.cuni.cz/~urban/MoMM/MoMM.tar.gz> and it now consists of the following parts

- (i) The main matching and interreduction tool (also called MoMM), based on version 0.7 of the E prover [Schulz 2002], particularly on the CSSCPA tool written by Stephan Schulz and Geoff Sutcliffe.
- (ii) The programs exporting the MML into a clause-like (TPTP-like) format suitable for MoMM.
- (iii) The interreduced clausebases created from MML. Some of them are complemented with termbanks which speed up their fast-loading.

This distribution is tailored to the real-time interaction during authoring Mizar articles, therefore the clausebases are already suitably interreduced and for some of them terms are dumped into termbanks, which accelerates their fast-loading for the real-time interaction. The “raw” (i.e. noninterreduced) clausebases corresponding to the MML version 4.04.834 can be downloaded from http://kti.ms.mff.cuni.cz/~urban/MoMM/MoMM834_raw.tar.gz. The export programs (relcprem and tptpexp available in the distribution) can be used to build the “raw” clausebases from any compatible Mizar distribution.

An important part of the MoMM’s functionality is implemented in the Mizar mode for Emacs [Urban 2002], which is available in the standard Mizar distribution and therefore does not have to be included in the MoMM distribution. This part provides a user interface allowing real-time interaction with MoMM during authoring Mizar articles.

The MoMM production and usage stages can be described as follows:

- (i) Exporting MML to the clause-like format. Currently both the main Mizar theorems (about 64000 clauses) and all Mizar internal lemmas (about 860000 clauses) are exported.
- (ii) Interreducing the exported clausebases. This has many options and can also provide useful statistics and hints for refactoring of the MML.
- (iii) Fast-loading MoMM with suitable clausebases during authoring new Mizar articles.
- (iv) Using a modified Mizar verifier (very similar to the exporting tool) which generates MoMM queries from the currently authored article (typically from the parts which are not accepted by the Mizar checker, i.e. lacking sufficient justification).
- (v) The generated queries are associated to their counterparts in the Mizar article, and the author can use the interactive Emacs functions for sending the queries to the MoMM process.
- (vi) If such a query is successful (i.e. a match was found by MoMM) and it is a MML theorem, it can be used for direct justification of the corresponding Mizar formula. If the match is an unexported Mizar lemma, the author is presented with its exact position in the MML and can copy its justification into his article.

3 Implementation of MoMM

As noted, the main matching and interreduction tool is based on the E prover and derived from the CSSCPA tool. The E prover is a clausal first order theorem prover, the main data structures are clauses made of literals. E is an equational prover, so predicates and functors are treated almost in the same way by the implementation. Atomic formulas are represented as pairs of terms, expressing their equality, and a special term TRUE accompanies in these pairs all the “terms” which have a non-equality predicate in the top position. E’s main indexing data structure is the perfect discrimination tree, described e.g. in [Ramakrishnan et al 2001]. There are other efficient indexing techniques, e.g. the code trees or context trees [Nieuwenhuis et al 2001], however perfect discrimination trees are still among the most efficient, and apart from being used in E, e.g. the very efficient Waldmeister equational prover [Hillenbrand 2003] has been using them for several years.

One of the first things that had to be changed in E’s implementation of the perfect discrimination trees was its usage of dynamic array indexed by the functor codes at each node. We want to load all MML theorems into MoMM, and want to make it possible for normal Mizar users, without any special hardware. The problem is that MML has a very large signature - about 7000 constructors. Having an array 7000 integers long at each tree node consumes memory very quickly. After some experimenting the dynamic array was replaced with a splay tree of functor codes at each node, and the memory consumption dropped very significantly. The

expected drop in speed actually did not come (about 1 percent), which was quite surprising³.

To speed up the subsumption a bit further, a signature pretest for pairs of clauses was implemented. Each term keeps an array of its symbols, which is cheap thanks to the banks of shared terms used in E. Such array is also created for each clause, and we reject the subsumption, if the subsumer contains symbols that are not in the candidate for subsumption. The speed up was about 50 percent. Recently, similar functionality has been implemented by Stephan Schulz in a more advanced way in E 0.8, so the plan is to switch to his implementation.

Quite significant modifications of the E’s algorithms and data structures have been done in order to efficiently implement a fast, type-aware subsumption algorithm that would resemble that of Mizar. These modifications are explained later in this article, after the explanation of the Mizar type system which motivates them. Despite all these modifications, there still are parts of mathematics formalized in Mizar, which are very hard to deal with in MoMM, and the worst-case complexity of subsumption can be observed on them. Such parts are e.g. the formalizations of various geometric configurations and properties (e.g. the AFF series of Mizar articles and its relatives), where there often occur formulas with many literals based on just one predicate, with either variables or very shallow terms, and with very few type constraints. Since computing subsumption between two such formulas can be very time consuming, we have implemented a hard last-resort limit, which is currently set to 1000 literal matchings before we give up with the particular subsumption attempt. This setting has been adjusted experimentally and (depending on the MML version) it causes roughly hundreds of subsumption attempts to be aborted. If this should become a serious problem, more “targeted” versions of such a hard limit are easy to implement.

4 Exporting Mizar for MoMM

4.1 *Export of the Mizar language*

Mizar is a system for formalizing mathematics by humans, consisting of several parts. The main “product” of Mizar is its large and growing mathematical library, containing more than 800 articles from various fields of mathematics. One of the main concerns of the Mizar designers is the suitability of the Mizar language for such large scale formalization, which leads to its quite complicated structure. On the proof level, Mizar is based on the Jaskowski’s system of suppositions [Jaskowski 1934], which is often put among natural deduction systems. The language of formulas is basically first order logic, but since MML is built on a variant

³ The measurement of the slowdown caused by this was not exhaustive, but was done on problems with “normal” small signature. The option of using splay trees instead of dynamic arrays is now controlled by a compiler directive and can thus be used for the whole E prover. Getting some large-scale statistics e.g. by running the two versions on the complete TPTP should not be a problem. Stephan Schulz has pointed out that the good performance can be due to the fact that the splay trees rebalance themselves for frequent queries.

of ZFC (Tarski-Grothendieck set theory), a language expressing infinite schemes of axioms is sometimes needed. This means that some second-order constructs are also allowed, though they appear only in quite a small part of MML. The symbols in Mizar can be overloaded in different ways (some frequently used mathematical symbols like “*” or “+” have more than 100 (re)definitions in the MML) which is indispensable for human authoring. It is important to know, that beneath this (sometimes complicated) notation there is a semantic layer, in which all symbols are disambiguated into the so called “constructors”. This constructor representation is then used for proof checking. There is a default naming scheme for this constructor representation used already now e.g. by MMLQuery or by MPTP. It is based on numbering of the constructors as they are defined in the Mizar articles, so e.g. the first mode (type constructor) in the article SUBSET_1 (with the user symbol “Element”) gets the name “m1_subset_1”. The transformation from the constructor format back into the user format is generally difficult, not unique, and sometimes probably impossible. Being a semantic tool, MoMM obviously has to use this constructor representation too⁴.

4.2 *The Mizar type system, Mizar-like Horn theories and their implementation in MoMM*

Mizar employs a number of automations, that can be generally called “the Mizar type system”. Examples and explanation of these type rules are presented in [Urban 2003], we will just try to explain the main ideas here.

The Mizar types are formed by clusters of attributes (e.g. *empty*, *finite*, *real*, *measurable*, etc.) that play the linguistic role of adjectives, and by type radices (e.g. *set*, *Function*, *Lattice*), which provide the main inheritance relationship. Both attributes and types are semantically predicates, however they differ from Mizar predicates by the existence of the type and cluster hierarchies. E.g. every type has to define its parent type, and obviously, if a formula is quantified with a variable of type T, it can be correctly applied only to variables of type T or more special. Functors also have to define their result types, and similar mechanisms are also used for attributes.

The Mizar type system (without the attributive part) as exported in MoMM can actually be thought of as a Horn theory with some strong “stratification” properties allowing fast traversal of the type hierarchies. We will call it here a *Mizar-like Horn theory* and try to give a bit abstract description that should be useful for further understanding of the type representation employed in MoMM. It might be useful also as an example of one particular way of dealing with *dependent*

⁴ Having more user-friendly names for the Mizar constructors would be useful for all the tools working with the semantic layer of Mizar and also for communication with other formalization projects. However keeping such names in an external table outside the MML would quickly make such a table outdated, since the MML is very often revised. The proper solution seems to be keeping unique user-friendly names inside Mizar articles. This would however require a simple addition to the Mizar language, and some additional effort either from the Mizar authors or from the Library Committee, and it is therefore hard to negotiate.

type hierarchies which are preferred in several proof assistants [Wiedijk 2003] to the simple non-dependent type hierarchies for their expressivity, but are generally more difficult to implement in a decidable and fast way.

Definition 1 (Mizar-like Horn theory M)

- (M1) *The signature of a Mizar-like Horn theory M is finite and consists of the disjoint sets of functor symbols F_M and type symbols T_M . There is a given linear ordering $<_M$ of the set of all symbols (i.e. $F_M \cup T_M$). The motivation for this is the order in which the symbols are defined in the MML.*
- (M2) *A Mizar-like Horn theory M is a union of its type hierarchy part TH_M and functor types part FT_M , i.e. $M = TH_M \cup FT_M$. These parts are explained in the following conditions.*
- (M3) *The type hierarchy part of M is a finite set of Horn clauses of the form:*
 $t_1(X, T_1, \dots, T_m) :- t_2(X, Y_1, \dots, Y_n)$.
Such clauses mean that if X has a (parameterized) type t_2 , then it also has its parent type t_1 , and they must satisfy the following two conditions:
- (M3.1) *$t_1, t_2 \in T_M$ and $t_1 <_M t_2$ (i.e. the parent type t_1 has to be introduced earlier in the MML).*
- (M3.2) *T_1, \dots, T_m are terms over $\{f_l \in F_M : f_l <_M t_2\} \cup \{Y_1, \dots, Y_n\}$, i.e. they use only the signature defined earlier than t_2 , cannot introduce new variables and the “typed variable” X is not allowed in them.*
- (M4) *The functor types part of M is a finite set of Horn clauses of the form:*
 $t(f(X_1, \dots, X_n), T_1, \dots, T_m)$.
Such clauses mean that t is the (parameterized) result type of the functor f , and they must satisfy the following two conditions (similar to (M3.1) and (M3.2)):
- (M4.1) *$t \in T_M$, $f \in F_M$ and $t <_M f$ (i.e. the result type t has to be introduced earlier in the MML).*
- (M4.2) *T_1, \dots, T_m are terms over $\{f_l \in F_M : f_l <_M f\} \cup \{X_1, \dots, X_n\}$, i.e. they use only the signature defined earlier than f , and cannot introduce new variables.*

Note that in this definition nothing prevents multiple inheritance both for the types and for the functors. The usage of multiple inheritance in Mizar is however quite limited (to the Mizar *structures*). Also note that the parent and result type clauses are completely insensitive to the types of arguments. This may feel counterintuitive to Mizar users, who are accustomed to the vast parametric polymorphism in Mizar (e.g. a more special result type can be given for a functor, when it has more special argument types), however this is really a faithful description of how Mizar behaves at the constructor level. The parametric polymorphism is dealt with in Mizar simply by having formally different constructors (i.e. functors and types) with different parent or result types when polymorphism occurs. The bottom line of this approach is that equality of the various polymorphic variants is internally used in many places in Mizar, while it is not captured by the notion of a *Mizar-like Horn theory*, and actually neither by the current MoMM’s subsumption algorithm, which is based on that notion. This makes the matching in MoMM

a bit weaker than in Mizar and it could probably be improved in the future by similar techniques as in Mizar, i.e. by keeping tables of the same polymorphic variants and doing the matching “modulo” them.

We want that the MoMM’s subsumption algorithm (described below) was aware of the Mizar type hierarchies. For that, we need to be able to determine the types of terms during the matchings and quickly traverse the parent type hierarchy. For the relatively simple *Mizar-like Horn theory* of the nonattributive part of the Mizar type system it is implemented in MoMM in the following way. The Horn theory of the Mizar types is exported as a special `typetable`⁵, which for each functor or type tells how to obtain its result or parent type (if it is nontrivial). This information is then used by MoMM, for proper computing of the complete set of types for each (nonvariable) term. The types of variables obviously have to be kept in the clauses. Hence e.g. for the functor `k4_relset_1` (user symbol `dom`) the exported `typetable` entry is:

```
type(k4_relset_1(A1,A2,A3),m1_subset_1(k4_relset_1(A1,A2,A3), k1_zfmisc_1(A1))).
```

To get a type of a term with the top-level functor `k4_relset_1`, MoMM will first match its arguments against the variables `A1, A2, A3`, and then instantiate the parent type

```
m1_subset_1(k4_relset_1(A1,A2,A3),k1_zfmisc_1(A1))
```

with the resulting substitution. Thanks to the usage of banks of shared terms in E, we do this only once for each term, and remember the beginning of its type hierarchy in an added “type” slot of the term structure. The type hierarchy is always finite, thanks to the strong “stratification” properties of *Mizar-like Horn theories*, and the type literals forming it can be normally shared in the common E’s termbanks, thanks to the above mentioned insensitivity of the *Mizar-like Horn theory* clauses to the types of its argument terms⁶. This mechanism allows us to access the complete (nonattributive) type hierarchy for a given nonvariable term when it is needed, i.e. not only its immediate parent type, but all of its ancestor types. This implementation has currently one drawback: it pretends that there is no multiple inheritance in Mizar. Having multiple inheritance requires generally an array of parent types instead of just one “type” slot of the term structure and some modifications of the type hierarchy traversing algorithms, and its implementation is not yet fully debugged. So for the rare cases (some *structure* types) when Mizar uses multiple inheritance, we export only the first parent type to the `typetable`. This is a limitation which may make the typed subsumption fail in cases when it succeeds in Mizar, but it is not too serious thanks to the low frequency of such cases.

To handle the types of variables, we use the fact that they are normalized in clauses and keep their types in a fixed array associated to the clause, where look-up can be done according to their numbers. Variables can be given only one initial type in Mizar formulas, and its ancestor hierarchy is again normally

⁵ This is the file `a11.typ` in the MoMM distribution.

⁶ Note that this would be hard to do, if the parent type of some term could vary e.g. with varying of the types of variables contained in it.

accessible through its “type” slot. The situation is more complicated with Mizar attributes, we explain it in the next subsection.

4.3 *Attributive extensions of the Mizar-like Horn theories and their implementation in MoMM*

The attributive part of the Mizar type system is a bit more complicated and it is no longer a Horn theory, because negated versions of attributes (e.g. “non empty”) are allowed in its formulation⁷. It is also more relaxed than the strongly stratified *Mizar-like Horn theory* and while direct and fast inheritance algorithms are sufficient for the non-attributive part, graph-based or fixpoint algorithms are needed for the attributive part. It differs also by generally taking into account the types of arguments⁸.

Definition 2 (Mizar-like type theory with attributes MA)

(MA1) *MA extends the notion of a Mizar-like Horn theory M by extending M 's signature with a finite set of attribute symbols A_{MA} (disjoint from F_{MA} and T_{MA}). Although in practice the MML ordering $<_M$ applies also to A_{MA} , it has no significance for the conditions given here.*

(MA2) *MA adds to the type hierarchy part TH_{MA} and functor types part FT_{MA} the conditional clusters part CC_{MA} and functor clusters part FC_{MA} , i.e. $MA = TH_{MA} \cup FT_{MA} \cup CC_{MA} \cup FC_{MA}$. The conditions for TH_{MA} and FT_{MA} are the same as above.*

(MA3) *The conditional clusters part of MA is a finite set of (generally non-Horn) clauses (or rather “rules”) of the form:*

$$(not)a_1(X) :- (not)a_2(X), \dots, (not)a_n(X), t(X, T_1(\bar{Y}), \dots, T_m(\bar{Y})), \tau(\bar{Y}).$$

Such clauses mean that if the variable X has attributes $(not)a_2, \dots, (not)a_n$ and a (parameterized) type t (with the types of its parameters' variables \bar{Y} specified by the set of type and attribute declarations $\tau(\bar{Y})$), then it also has the attribute $(not)a_1$. The reason for writing the clause in an implicative form is that the Mizar implementation really makes use of only this implicative version of the clause, other implicative variants have to be stated explicitly if the user wants that Mizar used them too. The additional formal specifications are:

(MA3.1) *$a_1, \dots, a_n \in A_{MA}$, $t \in T_{MA}$, $T_1(\bar{Y}), \dots, T_m(\bar{Y})$ are terms not containing X with all variables in the set $\bar{Y} = \{Y_1, \dots, Y_l\}$ and $\tau(\bar{Y})$ consists of type declarations for the variables (i.e. atoms $t^i(Y_i, T_1^i(\bar{Y}^i), \dots, T_{m_i}^i(\bar{Y}^i))$ where $i \in 1, \dots, l$, $\bar{Y}^i \subseteq \bar{Y} \setminus \{Y_1, \dots, Y_i\}$) and attribute declarations for the variables (i.e. literals*

⁷ It can be trivially changed into a Horn theory by creating new symbols for the negated attributes and adding their dependencies, and this is actually used in some parts of the Mizar implementation, but such renaming is not currently used in the MoMM export. The Horn-like presentation is more elucidating as it suggests one-directional (rule-like) usage of the clauses, which is actually used in Mizar. In this sense, the formal clause-like presentation is a bit incorrect, since adding new permutations of existing rules really changes Mizar's behavior.

⁸ If the following definition is too formal to understand, the reader is encouraged to have a look at the large number of examples in the MML, which usually give the intuition quite quickly.

$(not)a_1^i(Y_i), \dots, (not)a_{n_i}^i(Y_i)$ where $i \in 1, \dots, l$.

(MA4) The functor clusters part of MA is a finite set of (generally non-Horn) clauses (or rather “rules”) of the form:

$(not)a(T) :- \tau(\bar{Y})$.

Such clauses add the attribute $(not)a$ ($a \in A_{MA}$) to the (nonvariable) term T (with the types of its parameters’ variables \bar{Y} specified by the set of type and attribute declarations $\tau(\bar{Y})$). The restrictions on $\tau(\bar{Y})$ are as in (MA3.1), the implicative form is used for the same reason as in (MA3).

Later in the article we will need to point out to the fact that any M and MA satisfy the following natural “monotonicity” condition and its extension to the application of any term context. For its justification, just have a look at the kinds of clauses present in M and MA .

(Monot) Let Y be a variable declared with the (parameterized) type t and attributes $(not)a_i$, and let T be a term having t (with the same parameters) as its (not necessarily most special) result type, and also (at least) the attributes $(not)a_i$. Let $\theta_{M,MA}(Y)$ be the complete set of type and attribute literals generated by M and MA for Y , and let $\theta_{M,MA}(T)$ be the same thing generated for T . Then $\theta_{M,MA}(Y)$ subsumes $\theta_{M,MA}(T)$ with the substitution $\{Y/T\}$ (i.e. all types and attributes generated for Y will also be generated for T). Similarly, for any term context $\alpha(\cdot)$ holds that $\theta_{M,MA}(\alpha(Y))$ subsumes $\theta_{M,MA}(\alpha(T))$.

Apart from the problem with generally slower (“graph-chasing”) algorithms for collecting attributes of a given Mizar term, the most problematic aspect of the attributive part of Mizar for a tool like MoMM is that unlike the simple result and parent type hierarchies which are *absolute across MML* (i.e. one functor constructor always gets the same result type, no matter what MML article it is used in), the attributive part can vary in different articles. The attributive part is controlled by a special “clusters” directive in each Mizar article and it is a modular, “information-hiding” method for controlling the context in which Mizar does the verification. For dealing with the attributive theory, using some “global” method similar to the global typetable described in the previous subsection is therefore not only significantly more difficult, but could also give some quite unexpected results to the Mizar authors. That’s why we now use Mizar for collecting the terms’ attributes as they are used in the particular articles and for passing them to MoMM, without any further attribute computation in MoMM. Unlike in *Mizar-like Horn theories*, the attributive theories are in general sensitive to the arguments’ types. This (together with the article-locality of the attributive theories) prevents us from storing terms’ attributes at some special “attributes” slot of the shared term structure (which would be similar to the way we store types). The complete attribute information is therefore kept locally for each clause. For variables, this is again done in a fixed array indexed by their number in the clause. The attributive literals of nonvariable terms are kept in a splay tree.

4.4 A simple example of the export

We will now explain with an example the format used by MoMM, and how the Mizar articles is exported into it. Consider the Mizar theorem PARTFUN2:17 (17-th theorem in article PARTFUN2):

f is one-to-one & x in dom f & y in dom f & $f/.x = f/.y$ implies $x = y$;

The variables used in this theorem have been earlier in the article (context) reserved with the following types:

```
reserve C,D for non empty set;
reserve f for PartFunc of C,D;
reserve x,y for set;
```

The meaning of this theorem is that for one-to-one partial functions from C to D , an element of its domain is determined by its value. The first step done by Mizar is to add the quantifications explicitly:

```
for C,D being non empty set,
for f being PartFunc of C,D,
for x,y being set holds
f is one-to-one & x in dom f & y in dom f & f/.x = f/.y implies x = y;
```

The second step is translating the user symbols into the constructors:

```
for C,D being non v1_xboole_0 m1_hidden,
for f being v1_funct_1 v1_relat_1 m2_relset_1 of C,D,
for x,y being m1_hidden holds
f is v2_funct_1 & r2_hidden(x, k4_relset_1(f) &
r2_hidden(y, k4_relset_1(f) &
r1_hidden(k4_finseq_4(f,x), k4_finseq_4(f,y)) implies
r1_hidden(x,y)
```

This transformation to absolute notation alone suffices to make the theorem very difficult to read. As noted above, attributes and types are just specially handled predicates, so a formula of the form “for x being T holds $P(x)$ ” is translated to “for x holds $T(x)$ implies $P(x)$ ”. Hence our formula becomes:

```
for C,D,f,x,y holds
( not v1_xboole_0(C) & m1_hidden(C) &
not v1_xboole_0(D) & m1_hidden(D) &
v1_funct_1(f) & v1_relat_1(f) & m2_relset_1(f,C,D) &
m1_hidden(x) & m1_hidden(y)
)
implies
( v2_funct_1(f) & r2_hidden(x, k4_relset_1(f) &
r2_hidden(y, k4_relset_1(f) &
r1_hidden(k4_finseq_4(f,x), k4_finseq_4(f,y)) implies
r1_hidden(x,y)
)
```

The type “set” (`m1_hidden`) has no semantic content, everything in Mizar is “set”, so such atomic formulas can be completely eliminated. Additionally, we clausify the formula (which is completely successful here, as there are only universally quantified variables), replace the Mizar equality (`r1_hidden`) with E’s equality (`equal`), and put the “context information” (i.e. the type and attribute literals) in the end. The result is in file `partfun2.ths` in the MoMM distribution:

```
accept: pos(partfun2, 1, 17, 219, 69, 0)
input_clause(th,axiom,
[--v2_funct_1(C5)
, --r2_hidden(C1,k4_relset_1(C3,C4,C5))
, --r2_hidden(C2,k4_relset_1(C3,C4,C5))
, --equal(k4_finseq_4(C3,C4,C5,C1),k4_finseq_4(C3,C4,C5,C2))
, ++equal(C1,C2)
, --v1_relat_1(C5)
, --v1_funct_1(C5)
, --m2_relset_1(C5,C3,C4)
, ++v1_xboole_0(C3)
, ++v1_xboole_0(C4)
, --$true], 6 ).
```

Note that `k4_relset_1` (`dom`) has arity 3 here. This is the effect of the hidden arguments `C` and `D` in the type of the variable `f`. Since “`dom`” is defined for such types, it uses these arguments explicitly on the constructor level.

The number of literals obtained in this way is usually quite high, mainly because of the context literals. Clauses with 40 context literals are no exceptions. As we explained above, the context information is handled specially in MoMM. The number 6 following the literal list tells MoMM that the context literals start at that position.

The initial line gives the status (`accept`) of the clause, and its Mizar position. It contains

- (i) The article name
- (ii) The kind of the exported clause (1 for theorems, 2 for definitional theorems, 3 for functor property formulas, etc.).
- (iii) Theorem or definition number if the kind is 1 or 2, otherwise 0.
- (iv) Line number in the Mizar article.
- (v) Column number in the Mizar article.
- (vi) The serial number of the clause generated from this Mizar formula - sometimes there are hundreds or thousands of clauses created from a single formula.

The status is used to tell MoMM what it should do with the clause. Status “`accept`” e.g. means that tautology check and both forward and backward subsumptions should be tried. There are several other statuses, documented in the MoMM help, e.g. the status “`nsaccept`” is used for fast loading of a clause base,

without trying any reductions.

4.5 Export of the proof lemmas

We not only export the main Mizar theorems, as e.g. the PARTFUN2:17 given above, but the great majority of the MoMM clauses is created by exporting the internal lemmas introduced inside the proofs of the main Mizar theorems. There are two kinds of these lemmas, those with a *Simple Justification* and those with a *Proof*. The *Simple Justification* lemmas generally look as follows in Mizar:

A3: ψ by A1,A2,PARTFUN2:17;

This tells Mizar, that the formula ψ should be provable by the local references A1,A2 and the MML theorem PARTFUN2:17. Both the formula ψ and the formulas denoted by A1 and A2 can apart from the standard Mizar signature contain also local constants, created on various levels of the Jaskowski-style proofs. The Mizar checker is presented with the types of these local constants, the negation of ψ and the formulas A1, A2 and PARTFUN2:17, and tries to infer a contradiction. The *Simple Justification* lemmas are now exported as implications

for *local_constants* **holds** \bigwedge *references* **implies** *conjecture*;

This would be in our case

for *local_constants* **holds** $A1 \wedge A2 \wedge PARTFUN2:17$ **implies** ψ ;

The changing of the local constants to the universally quantified variables (with corresponding types) is justified by the standard theorem about constants and the fact that the only knowledge the checker has about them are their types. The reference PARTFUN2:17 is a Mizar theorem, whose validity does not depend on any possible suppositions done along the proof path to ψ . Therefore it can be removed from the exported lemma, which then contracts just to

for *local_constants* **holds** $A1 \wedge A2$ **implies** ψ ;

Note that in this kind of export only the universally valid theorems can be removed in this way, the references A1 and A2 might be proved with the use of some local supposition, and thus not be generally removable. Another kind of export of the internal lemmas could be implemented e.g. by collecting all of the suppositions made along the proof path to ψ . Such other kinds of export are not yet implemented.

The export of the internal lemmas with a *Proof* is similar. Each *Proof* can be thought of as a block of justification steps. If we collect all the references used inside the block, which are not introduced in the block (i.e. are external to it), it gives as the set of formulas from which the proved lemma logically follows. Again, the universally valid theorems can be removed from that set and we can generalize over all of the local constants.

5 Typed Subsumption in MoMM

We have described above several performance improvements to the basic indexing and subsumption mechanism in E, however the crucial point was implementation of a typed subsumption, which treats the context (i.e. type and attribute) literals

specially, in accordance with a given *Mizar-like Horn theory* M and its (rule-like) attributive extension MA .

First, let us realize that the semantics of a clause $C = \{L^1, \dots, L^{n_1}\}$ with variables \bar{Y} with initial types given by context literals $\Theta = \{\theta^1, \dots, \theta^l\}$ is simply $\bigwedge \Theta \rightarrow \bigvee C$. The normal subsumption in E takes two lists of literals - *subsum_List* and *sub_cand_List* and tries to find a substitution σ such that $\sigma(\text{subsum_List}) \subseteq \text{sub_cand_List}$. This works in theory for typed clauses too, if we put all of the context literals generated by the given *Mizar-like Horn theory* M and its (rule-like) attributive extension MA directly into them and treat them as normal clauses. In more detail, suppose that we have two clauses $C_1 = \{L_1^1, \dots, L_1^{n_1}, \tau_1^1, \dots, \tau_1^{m_1}\}$ and $C_2 = \{L_2^1, \dots, L_2^{n_2}, \tau_2^1, \dots, \tau_2^{m_2}\}$, where L_i^j are the normal literals and τ_i^j give the full context informations (i.e. full type hierarchies and full attribute information for all terms in C_i). The sets $\{\tau_i^1, \dots, \tau_i^{m_i}\}$ are generated by the exhaustive application of M and MA to the terms appearing in $\mathcal{L}_i = \{L_i^1, \dots, L_i^{n_i}\}$ using the initial types and attributes $\Theta_i = \{\theta_i^1, \dots, \theta_i^{l_i}\}$ of the sets of variables \bar{Y}_i of C_i . Note that in this “explicit context generating” process new terms (but not new variables) can appear, due to the general parametric form of the clauses in M and MA , but this process is finite, due to the stratification and finiteness properties of M and MA and their signature, i.e. given finite initial Θ_i and \mathcal{L}_i , the number of the full explicit context literals τ_i^j is also finite.

Now suppose that we have found a substitution $\sigma = \{Y_1^1/T_2^1, \dots, Y_1^{l_1}/T_2^{l_1}\}$ such that $\sigma(C_1) \subseteq C_2$. That particularly means that all L_1^i matched. It also means that all of the initial types of variables from C_1 , e.g. $\theta_1^i = t_1^i(Y_1^i, S_1^1, \dots, S_1^{n_1})$ matched, i.e. that there was a type literal $\tau_2^i = t_2^i(T_2^i, S_2^1, \dots, S_2^{n_2}) \in C_2$ such that $\sigma(\theta_1^i) = \tau_2^i$, and similarly for the initial attributes of these variables. In other words, all the bindings Y_1^i/T_2^i are correctly typed, since the bound terms have at least as special types (with attributes) as required by the variable declarations for Y_1^i . It is easy to see that the converse holds too, i.e. if there is a correctly typed substitution giving subsumption on the “normal” literal parts of C_1 and C_2 , it already gives complete subsumption of C_2 by C_1 (i.e. also on all the context literals). Just observe (best by looking at the kinds of clauses (or rules) forming M and MA) that all of the context literals generated by M and MA for C_1 will be generated for the corresponding terms of C_2 too (the (Monot) property given above).

An inspection of the previous argument can confirm the intuitive requirement, that adding only the initial variable types and attributes Θ_1 to \mathcal{L}_1 is enough, if we are looking for a subsumption of C_2 by C_1 . This again follows from the (Monot) property, i.e. once we know that the initial variable types and attributes matched, we also know that their full context matched, and also any term context applied to them. This improves the efficiency of the typed subsumption significantly, since we do not have to include all the additional context literals into the subsumers. This fact can also be used to decrease the sizes (both in the memory and in the filesystem) occupied by the clausebanks, if we know that they will only be used for forward subsumption of other formulas and no backward subsumption will be applied to them. In that case, the attributive context of all nonvariable terms can

be removed from them.

The current implementation of the algorithm additionally organizes the literals in such a way, that the “type checking” was done as early as possible, cutting off the ill-typed bindings. Note however that the types are parameterized, so the “type checking” can bind new variables. Therefore our implementation should be thought of as a special version of a normal subsumption, steered by the additional type information, rather than thinking of it as of two strictly divided “normal” and “type checking” parts. The sketch of the subsumption algorithm that we implement is following:

- (i) Start with *subsum_list* equal to the normal literals of the subsumer, and *sub_cand_list* to those of the candidate.
- (ii) Each time a new variable V is bound, collect its initial types and attributes into $\Theta(V)$ (they are kept in the fixed arrays associated with the clause, as explained above). Also collect the complete types and attributes of the corresponding term T_V into $\tau(T_V)$ (the types are obtained by traversing the type hierarchy starting at the T_V 's “type” slot, the attributes are fetched from the splay tree associated with the clause, as explained above).
- (iii) With the current partial substitution, try to find a subsumption between $\Theta(V)$ and $\tau(T_V)$, this may extend the current partial subsumption.
- (iv) If success, continue the suspended subsumption job, with the extended substitution, otherwise try backtracking the type subsumption, and if no success the previous matchings.

Note that the type subsumptions are not special in any way in this, and particularly, they may also trigger further type subsumptions, when they really instantiate some variable parameterizing the types. To implement this ‘eager’ type checking, we need a stack of ‘subsumption jobs’, that we postponed because of the newly arrived type subsumption jobs.

6 Processing Modes

MoMM has two basic purposes: interreduction of a set of clauses, and loading a (possibly interreduced) set of clauses quickly for giving hints interactively. The processing mode is influenced both by command line options, and by the clause statuses. The status can tell MoMM e.g. that a clause should be accepted unconditionally, or that all checks should be used, and the clause accepted if not redundant, or that the clause should just be checked and never accepted. The clauses are capable of keeping the “subsumption” information, i.e. the positions of clauses which they subsumed (recursively). This can be used after the interreduction for all kinds of data-mining applications.

For the 65702 theorem clauses generated from MML (version 7.0.04) we use a complete mutual interreduction, with a complete Mizar typetable loaded. The interreduced clause base is then dumped using numerical abbreviations for terms, into the file `all.ths.cb`, and the corresponding termbank into the file `all.ths.tb`.

These files are then typically fast-loaded without any interreductions in a “read-only” mode, and used for printing theorems that subsume the interactive queries.

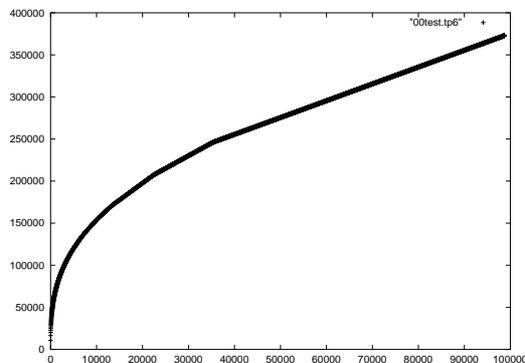
The clauses generated from the internal lemmas are first interreduced by the theorems from their articles, and then all the lemmas generated from one article are mutually interreduced. This first fast interreduction usually removes a very large portion of the internal lemmas (cca 40 percent). Such (partially) interreduced clausebanks are then compressed and available for the interactive use in the same way as the theorems.

7 Results

The complete interreduction of the 65702 typed theorem clauses from MML 7.0.04 takes about 9 minutes on Pentium 4 3GHz, and about 180M RAM. The subsequent fast loading of the interreduced theorem clause bank takes 14 seconds and 140M RAM on the same machine. It is possible to load MoMM only with a part of the MML theorems, but it seems that these time and space requirements are today within the reach of most Mizar users.

Recently we have also tried an interreduction of the 860000 clauses generated from the internal Mizar lemmas, but without the backward subsumption. This interreduction took 36 minutes on Pentium 4 3GHz, and 1.4 GB RAM. More than a half of the clauses was subsumed, and the strongest clauses have subsumed a very high number (thousands) of others and thus suggest many useful changes to the structure of the MML. The graph of the cumulative subsumption counts is given below.

Fig. 1. Cumulative subsumption counts of the internal lemmas



After the interreduction of the MML theorems, we are left with 64009 clauses, so 1695 clauses generated from MML theorems are redundant, usually subsumed. This finding alone confirms the necessity of tools like MoMM for management of large libraries of formalized mathematics. There is quite a lot of attention paid to the compactness and integrity of MML, however even the best informed maintainers cannot keep precise track of some 40000 theorems. The informations about redundant theorems is available for further analysis in the interreduced clause bank all.ths.cb in the distribution, because each time a clause C_1 subsumes

clause C_2 , the position information of C_2 is added to the “subsumed” list of C_1 . The subsumption theorem pairs found by MoMM can be also viewed at <http://ktiml.mff.cuni.cz/~urban/MoMM/pos1.cb2>.

8 Problems and Future Work

There are still number of possibilities for improving the basic type subsumption algorithm, both in the Mizar export and in MoMM. As noted, clusters are implemented differently than types, because their hierarchy in Mizar is more complicated. Also the export of Mizar structures pretends that they only have one ancestor in the type hierarchy, while they usually inherit from more than one parent structure. This is rather a problem of the export, and the appropriate tables in MoMM, the subsumption algorithm itself can deal already now with multiple inheritance.

The second order features of Mizar are not exported now, providing a solution for this part of Mizar would be useful too. It is hard to say, how difficult that would be. Generally, a certain kind of a “brute force” principle works for us even if there are now such holes in the implementation: If a second order feature is often used, it will be often formulated in its most frequently used first order instances, and these we will capture. This is true, because we not only export theorems, but also other (correctly generalized) parts of Mizar proofs, e.g. subproofs and simple justifications.

Stronger mechanisms than just subsumption can be probably quite easily added, going in the direction of a limited theorem prover. We might print not just the exact match if it is found, but also try to select the “best” matches (e.g. in terms of the number of literals subsumed, etc.), and show them to users.

The dealing with types now differs from the MPTP export. Some compatibility between the two would be useful, to be able to use MoMM as a fast postfilter e.g. when using the MPTP translation with some prover as a theorem discovery tool.

9 Acknowledgments

Obviously my thanks go to Stephan Schulz, for implementing and GPL-ing the E prover, which is not just very efficient, but also extremely cleanly written and documented. Thanks also to Geoff Sutcliffe, who is (at least) a co-author of the idea of the CSSCPA filter, which is a real Father of MoMM.

References

- [Bancerek 2003] Bancerek G. [2003], Information Retrieval in MML, In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), Mathematical Knowledge Management, Proceedings of MKM 2003, LNCS 2594.
- [Fellbaum 1998] Christiane Fellbaum (editor), WordNet: An Electronic Lexical Database. The MIT Press, May 1998, ISBN 0-262-06197-X.

- [Hillenbrand 2003] Thomas Hillenbrand: Citius altius fortius: Lessons learned from the Theorem Prover WALDMEISTER. *Electr. Notes Theor. Comput. Sci.* 86(1): (2003)
- [Jaskowski 1934] Jaskowski, S. (1934) On the Rules of Suppositions in Formal Logic” *Studia Logica* v.1.
- [Nieuwenhuis et al 2001] Robert Nieuwenhuis, Thomas Hillenbrand, Alexander Riazanov and Andrei Voronkov On the Evaluation of Indexing Techniques for Theorem Proving *Int. Joint Conf. On Automated Reasoning (IJCAR)*, Siena, Italy, 2001.
- [OpenCycUrl] OpenCyc home page at <http://www.opencyc.org/>
- [Ramakrishnan et al 2001] Ramakrishnan I. V. R. C. Sekar, Andrei Voronkov [2001]: Term Indexing. *Handbook of Automated Reasoning 2001: 1853-1964*
- [Rudnicki 1992] Rudnicki P. [1992], An Overview of the Mizar Project, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, Chalmers University of Technology, Bastad.
- [Rudnicki and Trybulec 1999] Rudnicki, P. and Trybulec, A. [1999], On Equivalents of Well-foundedness. An experiment in MIZAR, *Journal of Automated Reasoning*, Vol. 23, pp. 197 - 234, Kluwer Academic Publishers, 1999.
- [Schulz 2002] Schulz S. [2002], E – A Brainiac Theorem Prover, *Journal of AI Communications*, Vol. 15, pp. 111-126.
- [Schulz 2001] Schulz S. [2001], Learning Search Control Knowledge for Equational Theorem Proving, In F. Baader and G. Brewka and T. Eiter (Eds.), *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, LNAI Vol. 2174, pp. 320–334, Springer.
- [Sutcliffe and Suttner 1998] Sutcliffe G. and Suttner C.B. [1998], The TPTP Problem Library: CNF Release v1.2.1, *Journal of Automated Reasoning*, Vol. 21/2, pp. 177-203.
- [Tarski 1939] Tarski A. [1939], On Well-ordered Subsets of any Set, *Fundamenta Mathematicae*, vol.32 (1939), pp.176-183
- [TARSKI] Trybulec A., Tarski Grothendieck Set Theory, *Journal of Formalized Mathematics*, 1, 1989.
- [Urban 2003] Urban J. [2003], Translating Mizar for First Order Theorem Provers. In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), *Mathematical Knowledge Management, Proceedings of MKM 2003*, LNCS 2594.
- [Urban 2002] Urban J. [2002], MizarMode: Emacs Authoring Environment for Mizar, available online at <http://kti.mff.cuni.cz/~urban/MizarModeDoc/html/>
- [Urban 2004] Josef Urban. MPTP - Motivation, Implementation, First Experiments. Accepted to editors Ingo Dahn, Deepak Kapur and Laurent Vigneron - *Journal of Automated Reasoning, First-Order Theorem Proving Special Issue*. Kluwer Academic Publishers (supposed publication: end of 2004). Available online at <http://kti.ms.mff.cuni.cz/~urban/MPTP/mptp-jar.ps.gz>.

- [Wiedijk 2000] Wiedijk F. [2000], CHECKER - notes on the basic inference step in Mizar. available at <http://www.cs.kun.nl/~freek/mizar/by.dvi>
- [Wiedijk 2003] Wiedijk F. [2003] Comparing mathematical provers, In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), Mathematical Knowledge Management, Proceedings of MKM 2003, LNCS 2594, pp. 188-202.

Milestones for Automated Reasoning

Larry Wos^{1,2}

*Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439-4801*

Abstract

In the beginning (the early 1960s), the long-term goal of automated deduction was the design and implementation of a program whose use would lead to “real” and significant contributions to mathematics by offering sufficient power for the discovery of proofs. The realization of that goal appeared to be at least six decades in the future. However, with amazement and satisfaction, we can report that less than four decades were required. In this article, we present evidence for this claim, thanks to W. McCune’s program OTTER. Our focus is on various landmarks, or milestones, of two types. One type concerns the formulation of new strategies and methodologies whose use greatly enhances the power of a reasoning program. A second type focuses on actual contributions to mathematics and (although not initially envisioned) to logic. We give examples of each type of milestone, and, perhaps of equal importance, demonstrate that advances are far more likely to occur if the two classes are indeed intertwined. We draw heavily on material presented in great detail in the new book *Automated Reasoning and the Discovery of Missing and Elegant Proofs*, published by Rinton Press.

Key words: automated reasoning, OTTER, missing proofs

1 An Effective Template for Research

In this article, we focus on the pursuit of a single goal: the design and implementation of a general-purpose program whose power is sufficient to prove deep theorems. One obvious template for pursuing such a goal is to consider what might be done in the abstract, with no specific theorem or class of theorems in mind, to increase the power of a reasoning program. Our approach,

¹ This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.

² Email: wos@mcs.anl.gov

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

on the other hand, is to attempt to prove a *specific theorem*—one never before proved with any reasoning program—by devising a *general strategy or methodology* that can be applied to various domains. Indeed, we consider the most effective template for research to be an *experimental* template, where the wellspring for the particular study is a single, apparently-out-of-reach theorem and where the objective is the discovery of a proof of that theorem by means of a new strategy or methodology that can be added to the arsenal offered by an automated reasoning program.

Even at our entrance into the field in the early 1960s, we applied this approach. For example, our attempt to solve a simple classroom exercise—prove that groups in which the square of every x is the identity e are commutative—led to the formulation of the powerful, and general, set of support strategy.

In this article we feature numerous other examples of such successes with W. McCune’s automated reasoning program OTTER[4]. We discuss three strategies and various methodologies whose use greatly enhances the power of a reasoning program. And we relate our experiences with practical applications in mathematics and in logic. Some of the successes featured here have answered questions that had been open and had resisted the minds of the masters for decades.

2 The Resonance Strategy

We begin *in medias res* in the early 1990s when, after a visit to Argonne, Dana Scott sent us 68 theorems to prove. The theorems are numbered by Lukasiewicz theses 04 through 71. Theses 1, 2, and 3 form Lukasiewicz’s axiom system for two-valued (classical propositional) calculus, studied in terms of the functions for implication and negation and expressed in the following OTTER notation.

$$P(i(i(x,y),i(i(y,z),i(x,z))))).$$

$$P(i(i(n(x),x),x)).$$

$$P(i(x,i(n(x),y))).$$

Scott challenged us to prove with OTTER these 68 theses, using as inference rule *condensed detachment*, expressed in OTTER notation, with “|” denoting logical **or** and “-” denoting logical **not**.

$$-P(i(x,y)) \mid -P(x) \mid P(y).$$

Despite our years of experience with OTTER, we were able to obtain only thirty-three proofs; thirty-five theses remained unproven. Even with ROO, a parallel version of OTTER, we could prove only forty-eight of the theorems, leaving twenty unconquered. Clearly, to meet Scott’s challenge—and consistent with our recommended experimental template—we needed to formulate a new technique. The *resonance strategy* was born[6].

For the problem in hand, we placed correspondents (*resonators*) of all sixty-eight theses in a `pick_and_purge` `weight_list`, assigning to each the same small value. A formula or equation placed in that list is assigned a priority, or weight, based on the assigned value, rather than assigning a priority or weight based on symbol count. By placing a formula or equation in that list and assigning it a value strictly greater than that assigned the `max_weight`, the user of OTTER enables the program to discard any deduced conclusion that matches said formula, where the variables are treated as indistinguishable. On the other hand, by assigning a value is less than or equal to that assigned the `max_weight`, the user provides guidance for directing the program's reasoning. The smaller the value, the more preferred is a matching retained conclusion, ignoring the individual variables. A resonator is used to direct a program's reasoning; it does not have a **true** or **false** value. The pattern of the resonator is the key, where all variables are considered as indistinguishable, alike. Any retained conclusion that matches a resonator, ignoring specific variables, is assigned the value that is assigned to the corresponding resonator. With resonators, the user can, by assigning different values, express preference for one class of conclusions over another.

Our intention was to have OTTER prefer over all others any deduced conclusion that matched at the functional level; that is, the preferred conclusion(s) would be used before all others for inference rule initiation. Phrased more generally, we were informing the program that the pattern of any of the sixty-eight formulas was most attractive, that, if one treats all variables as indistinguishable, a deduced conclusion with such a pattern is attractive. The functional shape, ignoring the specific names of the variables, is what counts.

And it worked. Within less than 16 CPU-minutes on a computer of the 1991 type, all 68 theorems of interest were proved.

One could, of course, argue that the set of sixty-eight actually form the outline of a so-called master proof and, therefore, the effort was more in the spirit of proofchecking than of proof finding. Indeed, since the sixty-eight theorems had essentially been proved by Lukasiewicz, one could argue that no new proof of any type had been found. But as the following shows, the resonance strategy does have both the desired generality and the desired power. Not only has it found proofs that resisted full automation for years, but, more important, it has found proofs previously absent from the literature.

3 The Methodology of Lemma Adjunction

We present here a vivid example of a principle that has dominated much of our research since the early 1960s. The principle asserts that advances are likely to occur if one chooses a theorem whose proof is fully detailed and attempt to find a means for a reasoning program to complete a proof of that theorem *without* guidance. If successful, almost certainly the unaided proof will be sharply different from that in hand before the study was undertaken.

The specific theorem to be proved asserts the deducibility of the three-axiom system of Lukasiewicz for classical propositional calculus from the Meredith single axiom, the following.

$$P(i(i(i(i(x,y),i(n(z),n(u))),z),v),i(i(v,x),i(u,x))))).$$

We again used condensed detachment; we also used hyperresolution.

Meredith's proof is (in effect) of length 41. Attempts at discovering a fully automated proof with OTTER were spread over eight years, each, obviously, unsuccessful. Many diverse attacks were tried. Eventually we did formulate a methodology, called *lemma adjunction*, for finally enabling (in 1999) a reasoning program to prove Meredith's theorem without relying on his proof.

Briefly, lemma adjunction has the researcher choose some set of lemmas to be proved, with no certainty that any or all are relevant to the target theorem. Correspondents of those lemmas are placed as resonators in `weight_list(pick_and_purge)`, each with the same small value assigned to it. (A layered-resonator approach has also proved quite useful, where the set of resonators is partitioned into subsets with members of a given subset assigned a common value.) The denials of the lemmas are placed in `list(passive)`. Members of that list are used mainly to detect the completion of a proof by noting that a unit conflict has been reached, and they are also used for forward subsumption; they do not participate in the reasoning process for drawing conclusions.

The lemma-adjunction methodology is iterative, each experiment (after the first) building on the results of its predecessor. In run $n + 1$, one adjoins to `list(sos)` the lemmas proved in run n . (A powerful variation has one adjoin all of the proof steps of the proved lemmas; a proof step is a line of an OTTER proof, for example, deduced with condensed detachment or with paramodulation.) We note that resonators themselves do not have a **true** or a **false** value; they are merely included to direct a program's reasoning. In other words, resonators are not themselves lemmas. Also, we note that the lemmas that are adjoined in a succeeding run may have no value to completing a proof of the target theorem.

The use of lemma adjunction did succeed in four runs, discovering a 160-step proof of level 74; Meredith's 41-step proof has level 30[8]. For the experiments, we used Lukasiewicz theses 4 through 71 as resonators.

Our next step was to test the effectiveness and generality of the new lemma-adjunction methodology. The goal was to find a proof where the literature offered none. We turned to a 23-letter formula (the following), presented in the mid-1930s by Lukasiewicz as the first single axiom for classical propositional calculus.

$$P(i(i(i(x,y),i(i(i(n(z),n(u)),v),z)),i(w,i(i(z,x),i(u,x)))))).$$

Lukasiewicz offered no guidance regarding the nature of his proof, just noting that three years of study had been required to obtain a proof. We conjectured

that we might complete a proof (perhaps his) by deducing a known axiom system. We chose as target various axiom systems, including the Lukasiewicz three-axiom system and a six-axiom system of Hilbert. We again used these 4 through 71 as resonators and as so-called intermediate lemmas. Four runs sufficed, requiring 4.5 CPU-hours, discovering a 200-step proof of level 68, completing with the deduction of the Lukasiewicz three-axiom system, among others. Hilbert's axiom system was deduced with a proof of length 194 and level 68, before the Lukasiewicz proof was completed. We view our successes as evidence of the generality of the resonance strategy.

4 The Hot List Strategy

Another application domain we have addressed with great success is algebra. In algebra, the proof being examined often requires repeated visits to the special hypothesis. For example, when proving commutativity for rings in which the cube of x is x (for all x), many crucial steps have as a parent the special hypothesis, namely, $xxx = x$. The *hot list strategy* was formulated in response to this observation[7].

With OTTER, a hot list is provided, in which the researcher places clauses that the program can visit, revisit, and the like. Let us say that *heat* is assigned the value 1; then, when a new clause is retained, immediately it is considered with each of the members of the initial hot list for drawing additional conclusions for possible retention. If heat is assigned the value 2, then the clauses with heat = 1 (just described) are immediately considered with members of the initial hot list.

McCune generalized the hot list strategy, which was originally formulated in the context of paramodulation only, to apply to all inference rules. He also generalized it to the *dynamic hot list strategy*, whose use permits the program to adjoin during a run new clauses to the hot list.

Our first experiment with McCune's generalization to other inference rules, testing the OTTER version for the first time with hyperresolution employed for condensed detachment, yielded a new result. The field was classical propositional calculus. The theorem to prove asked for a derivation, from the Lukasiewicz three-axiom system, of a different three-axiom system of Lukasiewicz, sometimes referred to as a system of Church. OTTER, using the hot list strategy, found a 21-step proof, whereas the best in hand before the experiment was a 22-step proof. More recently, the hot list strategy was put to most profitable use by Z. Ernst, B. Fitelson, and K. Harris in the study of various areas of logic[1]. They succeeded in finding six new single axioms for $C5$, the implicational fragment of the modal logic $S5$. Even more impressive was their success with $C4$, the implicational fragment of the modal logic $S4$. Indeed, where (apparently) Meredith had sought and failed to find a single axiom for this field, the trio did find one.

5 The Strategy of Cramming

The formulation of the *cramming strategy*[10] was motivated by a query from one of our colleagues, B. Fitelson. Impressed with many successes with OTTER, he asked about the existence of a proof shorter than Meredith and Prior’s 33-step proof for the sufficiency of the Lukasiewicz shortest single axiom for the implicational fragment of propositional calculus. None of the approaches we had devised in the preceding years for proof refinement yielded the prize. However, one of the experiments yielded a 30-step proof of the most complicated Tarski-Bernays axiom, the target three-axiom system, whose denial is the following.

$$\neg P(i(p,i(q,p))) \mid \neg P(i(i(i(p,q),p),p)) \mid \neg P(i(i(p,q),i(i(q,r),i(p,r)))) \mid \\ \$ANS(TARSKI_BERNAYS).$$

The ANSWER literal can be used to (so to speak) capture a construction found by OTTER but is also useful for identifying what has been proved, especially in the case where the program is instructed to prove a number of theorems.

We therefore hypothesized that, if we could formulate a technique that would enable OTTER to extend the 30-step proof with exactly two additional steps, one deriving each of the two remaining formulas to be proved, we would win: we would surpass Meredith and Prior’s marvelous contribution. Put a bit differently, if we could find a way to “cram” the 30-step proof into a 32-step proof of the Tarski-Bernays system, we would succeed in the goal set by Fitelson. And the cramming strategy was born.

One begins with extending the set of support by placing the thirty formulas in it. One next includes as resonators the two formulas to be proved, each assigned a very small value, and one then assigns that value to max_weight. Then one has OTTER rely on level saturation. The basic idea is to see whether the two formulas can be derived and little or nothing else. The desired 32-step proof was discovered.

Cramming has played an important role in proof refinement with respect to length, a topic we now address.

6 Hilbert’s New Problem

For many mathematicians and logicians, the knowledge that some implication holds—is a theorem—suffices; the proof itself offers little interest. Others are satisfied with seeing some proof; its “elegance” is not a concern. Hilbert was a member of neither group.

Indeed, as his recently discovered twenty-fourth problem reveals, Hilbert was greatly interested in actual proofs and in proof simplification. A proof can be simplified by shortening it, by finding a proof with fewer deduced steps than that in hand. Simplification can instead focus on the complexity of the

deduced steps, on the avoidance of thought-to-be indispensable lemmas, on the avoidance of some type of term, and more. As it turns out, an automated reasoning program can play a vital role in the discovery of more elegant proofs, of proofs simpler than those offered by the literature. (For a piquant aside, we note that the discovery by R. Thiele in 2002 of Hilbert's twenty-fourth problem delighted us[5]; we have been studying proof refinements of various types since perhaps 1992.)

In our pursuit of such proofs, we have formulated various methodologies. For example, we often use demodulation in a nonstandard way, use it to block the retention of one or more unwanted steps of a proof under study. Sometimes we use cramming. We choose a type of term, for example so-called double-negation terms, and use demodulation to avoid retaining any deduced clause that contains such a term. A double-negation term is of the form $n(n(t))$ for some term t . We seek proofs that avoid lemmas the literature suggests are key to finding any proof.

Among our successes, we have—actually, OTTER has—discovered a 38-step proof for the Meredith single axiom, a proof three steps shorter than that of the cited master who himself was clearly interested in finding shorter proofs. Although we cannot make an appropriate comparison because of what is absent from the literature, we have used (OTTER and) cramming to complete a 50-step proof for the Lukasiewicz 23-letter formula. Also in part with that strategy, we now have in hand a 5-variable proof for a 19-letter single axiom of Meredith for propositional calculus in which **false** is part of the language. Meredith's proof includes a deduced formula that relies on eight distinct variables.

We have found double-negation-free proofs showing that one of the Lukasiewicz five axioms for his many-valued sentential calculus is in fact dependent. Meredith was the first to prove the corresponding theorem, but his proof does rely on double negation. Our find coupled with many others of its type led to M. Beeson (with colleagues) proving that the Lukasiewicz three-axiom system for classical propositional calculus has the following charming property. If the target is a theorem free of double negation, then there must exist a double-negation-free proof of the theorem with the Lukasiewicz three-axiom system as hypothesis. Beeson's success answered a question posed by the logician D. Ulrich. We thus have yet one more example of how automated reasoning is now affecting the research of those whose primary interest is indeed outside of our field.

7 A Bright Future

The applications of OTTER are truly wide ranging. Powerful single axioms have been found for Boolean algebra[3]. McCune's monograph with R. Padmanabhan provides a most visible and exceedingly satisfying milestone, with its proof after proof and its answers to open questions[2]. And just two years

ago, the last possible shortest single axiom, XCB , was found for equivalential calculus[9]. the corresponding question had remained open for seven decades. Without automated reasoning, almost certainly the question would still be open.

If this material has inspired some researcher to consider additional challenges and open questions that have been identified, the book *Automated Reasoning and the Discovery of Missing and Elegant Proofs* provides a beginning[11]. That book discusses in far more detail than we are able to include here the various strategies, methodologies, and contributions to mathematics and logic that have been witnessed in the past few years. For a question not offered by the cited book, the following might prove of interest. Does there exist the analogue to the double-negation-free theorem for, say, group theory, where negation is replaced by inverse?

In summary, we have mined some treasure—but by no means all of it. Much remains, and we now have some of the means for mining far more.

References

- [1] Ernst, Z., B. Fitelson, K. Harris, and L. Wos, *Shortest Axiomatizations of Implicational S_4 and S_5* , Notre Dame J. Formal Logic **43**, no. 3 (2002), 169–179.
- [2] McCune, W., and R. Padmanabhan, “Automated Deduction in Equational Logic and Cubic Curves,” in *Lecture Notes in Computer Science 1095*, Springer-Verlag, New York, 1996.
- [3] McCune, W., R. Veroff, B. Fitelson, K. Harris, A. Feist, and L. Wos, *Short Single Axioms for Boolean Algebra*, J. Automated Reasoning **29**, no. 1 (2002), 1–16.
- [4] McCune, W., *OTTER 3.3 Reference Manual and Guide*, Argonne National Laboratory, Argonne, Illinois, technical memorandum ANL/MCS-TM-263, 2003.
- [5] Thiele, R., and L. Wos, *Hilbert’s Twenty-Fourth Problem*, J. Automated Reasoning **29**, no. 1 (2002), 67–89.
- [6] Wos, L., *The Resonance Strategy*, Computers and Math. with Apps. **29**, no. 2 (1995), 133–178.
- [7] Wos, L., and G. W. Pieper, *The Hot List Strategy*, J. Automated Reasoning **22**, no. 1 (1999), 1–44.
- [8] Wos, L., *Conquering the Meredith Single Axiom*, J. Automated Reasoning **27**, no. 2 (2001), 175–199.
- [9] Wos, L., D. Ulrich, and B. Fitelson, *Vanquishing the XCB Question: The Methodological Discovery of the Last Shortest Single Axiom for the Equivalential Calculus*, J. Automated Reasoning **29**, no. 2 (2002), 107–124.

- [10] Wos, L., *The Strategy of Cramming*, J. Automated Reasoning **30**, no. 2 (2003), 179–204.
- [11] Wos, L., and G. W. Pieper, “Automated Reasoning and the Discovery of Missing and Elegant Proofs,” Rinton Press, Paramus, N.J., 2003.

The submitted manuscript has been created National Laboratory (“Argonne”) under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

Otter-lambda, a theorem prover with untyped lambda-unification

Michael Beeson^{1,2}

*Computer Science Department
San José State University
San José, Calif., USA*

Abstract

Support for lambda calculus and an algorithm for untyped lambda-unification has been implemented, starting from the source code for Otter. The result is a new theorem prover called Otter- λ . This is the first time that a resolution-based, clause-language prover (that accumulates deduced clauses and uses strategies to control the deduction and retention of clauses) has been combined with a lambda-unification algorithm to assist in the deductions. The resulting prover combines the advantages of the proof-search algorithm of Otter and the power of higher-order unification. We describe the untyped lambda unification algorithm used by Otter- λ and give several example theorems.

Key words: unification, lambda calculus, automated deduction.

1 Introduction

Our purpose in this paper is to demonstrate the successful combination of lambda unification with the well-known resolution-based theorem prover Otter [19]. What we call “untyped lambda unification” is similar to second-order or higher-order unification, but those algorithms are designed for use in typed theories, and we have chosen to use a new name for this algorithm, since it is a new algorithm and is applied in a different context.

The advantages of clause-based provers like Otter lie in the retention of deduced conclusions and the development of effective search strategies to generate and/or retain desired, rather than undesired, new conclusions. The advantages of type-based systems are the greater ease of describing mathematical concepts at a “higher level” and the availability of higher-order unification.

¹ Research supported by NSF grant number CCR-0204362.

² Email: beeson@cs.sjsu.edu

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

By adding lambda unification to Otter, we have created a system offering a combination of both advantages.

The implementation of lambda calculus and an algorithm for lambda unification in Otter, is described in the last sections of this paper. A soundness proof for the algorithm is given in [6]. That paper answers the question, *In what logic does Otter- λ find proofs?*. This paper reports on several examples intended to demonstrate some of the capabilities of Otter- λ , as we call our enhanced version of Otter (or when typography dictates, Otter-lambda). The main point is to explore the combination of lambda unification with a clause-based theorem prover using search strategies to generate new conclusions, as opposed to either a type-based prover (unaided by retention of many clauses, demodulation and paramodulation, and a variety of inference strategies) or a first-order, clause-based prover (unaided by higher-order unification and types).

First, we show that our enhancements give Otter- λ the ability to manipulate quantifiers at the clausal level, so that definitions involving quantifiers can be conveniently used in proofs. Second, we take up the algebraic part of Lagrange's theorem, to demonstrate how lambda unification enables the automatic construction of maps needed in algebraic proofs. Third, we give an example to show how Otter- λ can find a proof in algebra that uses mathematical induction, namely, there are no nilpotents in an integral domain. Fourth, we reproduce the well-known automatic deduction of Cantor's theorem, which appears to depend heavily on a typed logical system, to show that it can be done in Otter- λ ; and that a similar approach in Otter- λ can yield Russell's paradox and fixed-point constructions, which cannot be formalized in type theory. Our fifth and last example is a proof by mathematical induction, directly from the axioms of Peano arithmetic, of the commutativity of addition. Both the basis case and the induction step of the main induction must in turn be proved by mathematical induction. Otter- λ is able to find all three instances of induction on its own, and complete the proof.

Input files and the proofs produced by Otter- λ for all these examples can be found on the Otter- λ web site [7].³

Acknowledgements. I am grateful to Bill McCune for prompt and lucid answers to questions about Otter, and to Larry Wos and Freek Wiedijk for discussions and suggestions.

³ To clear up some misconceptions about Otter- λ : It is a theorem-prover, built on Otter, but it is not a "library" that can be plugged into other provers, not even Otter, since it was necessary to modify Otter at several points to connect the new capabilities. On the other hand, it performs on first-order problems exactly like Otter—the new code is never invoked unless appropriate flags are set in the input file. Thus Otter files run in Otter- λ , and at the same speed, though they may use a few percent more memory since one data structure was slightly expanded.

Lambda unification

We focus on the attempt to extend unification to instantiate variables for functions by means of λ -terms. All the past research and implementation in this subject has involved typed formalisms. In that context, it is known as *higher-order unification*. What is novel in this paper is the implementation of (a version of) higher-order unification in an untyped formalism. We call our unification “untyped lambda-unification”. In [23,24], Pietrzykowski gave an algorithm for higher-order unification, which we here call typed λ -unification. Huet [17] did not invent this algorithm, but that paper contains another important contribution and has become the classic reference. An encyclopedic treatment of type theory and second-order unification can be found in [1,14]; we shall say no more about the thirty years of research covered there. We note the following points about our implementation:

- We do not implement the full (typed) higher-order λ -unification algorithm.
- Our algorithm can solve problems which are not solvable in a typed setting, e.g. finding a term X such that $X(y) = f(X(y))$.

Otter- λ recognizes the reserved words `lambda` and `Ap` (or synonymously `ap`). One uses `lambda(x,Ap(f,x))` to enter the term $\lambda x.f(x)$. Beta-reduction has been implemented in the framework Otter uses for demodulation. To understand this paper, the reader will need to read both the examples and the algorithm for untyped lambda-unification. Either one is difficult to understand without the other, so it may be best to read them in parallel. Logically the algorithm definition should precede the examples, and the paper is organized that way, but it should also work to skip the rest of this section for now and return to it later.

A substitution σ is a λ -unifier of terms t and s if λ -logic [6] proves $t = s$. A λ -unification algorithm is an algorithm that finds one or more lambda unifiers, when given as input two terms t and s . We have (so far) implemented a single-valued version of second order unification. It therefore misses some unifiers that would be returned by a multiple-valued version, but it also solves some problems that cannot be solved in a typed theory, and it works efficiently, and it works in Otter. (We are in the process as of June 2004 of extending the implementation to return multiple unifiers.)

In this paper, as in Otter and Otter- λ , we distinguish between variables (which can be instantiated by unification) and constants (which cannot be instantiated) by a typographic convention: variable names begin with upper case or lower case x,y,z,u,v,w . Names beginning with other letters, such as f or c , are constants. The scope of a free variable is the clause in which it occurs. The scope of a λ -bound variable is the λ -term in which it occurs. The same variable will never occur both free and λ -bound in the same clause. (Nested λ -bindings of the same variable are allowed during unification, but for technical reasons are renamed when kept clauses are stored.) Technically,

a *variable* refers to an integer *varnum* and a *context*, which is a table of pairs of the form $(term, context)$ whose entries describe a substitution. Unification algorithms take as input two pairs (t, c) where t is a term and c is a context. In this section we refer to such a pair as a term, leaving the context implicit, or referring to it as *the context of t* . A term t depends on a variable y if t contains y or if t contains a variable that has been assigned, in the context of t , a value depending on y . When we say in the pseudocode below that $x := t$ we mean that an entry is made under the varnum of x in the context of x assigning x the value t . This setup is exactly as in first-order unification; see [19] for details.

The definition of λ -unification makes use of the concept of a variable y being *forbidden* to a variable x . This means that unification is not allowed to assign x a value depending on y . The context data structure is expanded to allow for storing, for each variable x , a list of variables forbidden to x . To forbid t to x means to add to the forbidden list of x , all free variables of t that have no values in the context of t , and all variables forbidden to those free variables of t that do have values assigned in the context of t .

Three functions are involved in our λ -unification algorithm: *unify2*, *unify_lambda*, and *subst*. The first two are called from Otter's unification algorithm, *unify*; *subst* is mutually recursive with *unify*. When *unify* has to unify two terms s and t , if either one has functor Ap , then *unify2* is called, and if both have functor λ , then *unify_lambda* is called. We will give commented pseudocode for each of these three functions.

```
unify_lambda(lambda(x,t), lambda(y,s))
{ saveit = term currently assigned to x; // NULL unless x is a nested bound variable
  x:=y; // assign y to x in the context of t
  forbid y to all free variables in t or s;
  unify(t,s);
  x:= saveit;
}
```

Example: $\lambda x. f(x)$ unifies with $\lambda y. f(y)$. No (net) variable assignment is made in either context.

The substitution algorithm *subst* is mutually recursive with unification. It takes three terms r , s , and t and returns a term (and a context). It is assumed r is a variable that does not occur in the contexts of s or t . The term is the result of substituting r for s in t . Here is pseudocode:

```
subst(r,s,t)
  { if(unify(r,s))
    { z = nextvar(t); // get unused variable z in the context of t
      z:= r; // assign z the value r in the context of t
      return t[s:=z]; // return t with s replaced by z
    }
  }
  if(t is a variable and not  $\lambda$ -bound, and already assigned in the context of t to q)
```

```

    return subst(r,s,q);
  if(t is a variable or name)
    return t;
  n = arity of t;
  ans = new term with arity n and same symbol as t;
  for(i=0;i<n;i++)
    i-th argument of ans = subst(r,s,q);
  return ans;
}

```

Example: substituting z for $g(X)$ in $g(Y)$, we return the substitution $X = Y$ and the term z . Below we write $t[s ::= r]$ for the term returned by *subst*.

We now give pseudocode for *unify2*. The code assumes that t and s are already dereferenced, i.e. are not variables that have values in their respective contexts.

```

int unify2(t,s)    // return value is 1 for success, 0 for failure
{
  if(t has the form Ap(t1,t2) and s has the form Ap(s1,s2))
    if(unify(t1,s1) and unify(t2,s2))
      return 1;
  if(t has the form Ap(lambda(x,r),q))
    { z = a new variable;
      rename x in (a copy of) r to z;
      z:=q // in the context of t
      return unify(r,s); // perform a beta-reduction before unifying
      // em if unification fails, destroy the copy of r before returning
      // no memory used for this beta-reduction as context keeps track of it
    }
  if(s has the form Ap(lambda(x,r),q))
    return unify(s,t);
  if(t is a variable not occurring in s)
    { t:= s;
      return 1; // but don't fail if occurs check fails
    }
  if(s is a variable not occurring in t)
    { s:= t;
      return 1;
    }
  if(t has the form Ap(X,r))
    return unify2(s,t);
  if(s has the form Ap(X,r) and X is already assigned a value lambda(z,q))
    { // for simplicity omitting some details about clash of bound variables
      label z as "not lambda bound";
      z:= r;
      rval = unify(q,t);
    }
}

```

```

    label z as "bound";
    return rval;
  }
  if(s does not have the form Ap(X,r) with X a variable)
    return 0;    unification fails
  fix X and w with s = Ap(X,w);
  z = new variable not occurring in s or t;
  if(w is a variable and X does not occur in t)
    { b = getConstant(t);    explained below
      rr = subst5(z,b,t);    explained below
      X:= lambda(z,rr);
      return 1;    success
    }
  else if(w is a variable and X does occur in t)
    { b = getMaskingSubterm(t); explained below
      rr = subst(z,b,t);
      X:= lambda(z,rr);
      return 1;    success
    }
  else if w is not a variable
    { rr = subst(z,w,t);
      if(X occurs in rr)
        return 0;    failure
      X:= lambda(z,rr);
      return 1;    success
    }
}

```

The above pseudocode outlines the algorithm, but leaves the functions *getConstant* and *getMaskingTerm* still to explain. The two cases in question are unifying $Ap(X, w)$ with t , where w is a variable, and X either does occur in t (in which case *getMaskingTerm* is used) or does not occur in t (in which case *getConstant* is used). We illustrate with examples of each.

We first explain *getConstant*. Suppose t is $(a+b = b+a)$, as occurs when we try to prove the commutativity of addition by induction. Then *getConstant*(t) returns b , and then *subst5* substitutes z for both occurrences of b (in this case), resulting in $X := \lambda z a + z = z + a$, the right propositional function for this inductive proof. In general, however, there will be many choices of a possible constant, and then *subst5* has to choose which occurrences of the constant to substitute for. For example, if t is $(b + b) + b = b + (b + b)$ there are 65 possible choices of X . As of June, 2004, we are working to allow *unify2* to return multiple unifiers, but we have had good success with the existing code, that just picks the rightmost constant and substitutes for all occurrences of it that do not occur inside Skolem terms from the induction axioms, if any.

(The user indicates the Skolem terms to be avoided by using reserved symbols in the statement of the induction axioms.) This definition of `subst5` allows Otter- λ to find the correct instances of induction for some nested inductive proofs, such as the commutativity of multiplication, where the basis case and induction step must themselves be proved by induction.

Now we describe *getMaskingSubterm*, which is used when unifying $Ap(X, w)$ with t where t does contain X . A *masking subterm* of t is a subterm r of t such that (i) r contains all the occurrences of X or variables forbidden to X in t , and (ii) r is not forbidden to w , and (iii) r does not contain any variables bound in t . We call such a term r a “masking subterm”, because it masks the forbidden values of X . Any masking subterm could in principle be used to form a unifying substitution, as follows: We substitute the new variable z for r in t , obtaining rr , and take $X := \lambda z. rr$. Let σ be the substitution assigning this value to X and assigning r to w . Then we verify that σ is a lambda unifier of $X(w)$ and t as follows:

$$\begin{aligned} (1) \quad & Ap(X, w)\sigma = Ap((\lambda z. rr)\sigma, w\sigma) \\ (2) \quad & = Ap(\lambda z. rr)\sigma, r) \\ (3) \quad & = rr[z := r] \\ (4) \quad & = t \end{aligned}$$

The present implementation always chooses (if possible) a maximal masking subterm which respects any typing constraints the user has supplied in the input file under `list(types)`; or if no `list(types)` is provided, a maximal masking subterm which is a second argument of an Ap term; that is, a masking subterm which occurs as a second argument of Ap and contains no masking subterm occurring as a second argument of Ap . In general we would get different potential unifiers for every masking subterm. There will sometimes be many of them. In the future the algorithm *getMaskingSubterm* can be altered or even made to return multiple values without affecting the soundness of the algorithm as proved in [6]. In this way the tradeoff between efficiency and completeness can be further explored.

Finally, we give an example to illustrate the case where *subst* is called. To unify $Ap(X, s)$ with t , where s is a compound term: We create a new variable z and use *subst* to substitute z for s in t . If the result successfully eliminates the variables in t that are forbidden to X (that is, if $t[s ::= z]$ is not forbidden to X), then unification succeeds, giving X the value $\lambda z. t[s ::= z]$. *Example:* in attempting to unify $Ap(X, g(X))$ with $g(Y)$, we substitute z for $g(X)$ in $g(Y)$, so $t[x ::= z]$ is z and the substitution is $X = Y$, and the substitution returned by *unify2* is $X = \lambda z. z$.⁴

⁴ In the example, we have missed the *other* solution, $X = \lambda z. g(Y)$. But we already knew that we are missing some of the unifiers produced by typed λ -unification. The question is, do we get the unifiers that we need to prove theorems of interest, and do we get them fast enough? Yes, we do, as shown by examples.

Regarding the implementation details: It was possible to make use of Otter’s “Layer 1” tools, so that we had relatively little low-level implementation to do. Most of the code we wrote was directly related to the algorithm. We were able to allow Otter to manage memory, restoring trails after unification failures, etc. Otter uses the data structures described in [18] to perform unification without needing to allocate and deallocate memory for substitutions; we were also able to make good use of these data structures. The result was that we were able to add lambda-calculus, beta reduction, and second order (or “higher-order” if you like) unification to Otter by adding fewer than 2000 lines of C code. However, we did have to modify the indexing code that keeps track of possible subsuming terms, to take care of difficulties caused by λ -bound variables having scope less than the entire clause.

Example. We give an example in which untyped λ -unification succeeds, while typed λ -unification fails. The example is this:

$$Ap(X, y) = f(Ap(X, y))$$

See [14], p. 1035 for a discussion of why this fails, and how the failure is detected by one algorithm for typed λ -unification but not by another. Here we show how it works in untyped λ -unification. First we identify a masking subterm on the right: it is not $Ap(X, y)$ since this contains y , while a masking subterm on the right must not be forbidden to y . For the same reason $f(Ap(X, y))$ is not a masking term. The only masking term is X itself. The unification algorithm therefore delivers the values

$$\begin{aligned} X &= \lambda z. f(Ap(z, z)) \\ y &= X \end{aligned}$$

Checking that this is a solution we find, using these equations,

$$\begin{aligned} Ap(X, y) &= Ap(\lambda z. f(Ap(z, z)), X) \\ &= f(Ap(X, X)) \\ &= f(Ap(X, y)) \end{aligned}$$

Of course, this is the standard fixed-point construction in λ -calculus; the solution cannot be typed in simple type theory, and hence in a typed setting there is no solution to this unification problem. But in an untyped setting, untyped λ -unification can find fixed points.

Implementation of λ -calculus in Otter

Since second order unification involves lambda terms, it is first necessary to implement the lambda-calculus. It is the *untyped* lambda-calculus that is implemented. See [6] for the theoretical background and a soundness proof of the algorithm whose implementation is reported here. Otter- λ recognizes the reserved words `lambda` and `ap` (or synonymously `Ap`). One uses `lambda(x, Ap(f, x))` to enter the term $\lambda x.f(x)$. Beta-reduction has been im-

plemented in the framework Otter uses for demodulation. Technicalities necessary to deal with the more or less successfully dealt with. The following Otter proof shows a beta-reduction combined with an ordinary demodulation, given the demodulator $x * x = x$. The theorem proved is

$$(\lambda x.x * x)c = c.$$

Of course the proof is trivial; it is only meant to demonstrate the successful implementation of β -reduction working more or less the same way as ordinary demodulation in a first-order theorem-prover. Line 3 of the proof is the negation of the goal. The first two lines are axioms. The proof completes with the derivation of a contradiction.

```

1 □ x=x.
2 □ x*x=x.
3 □ ap(ap(lambda(x,lambda(y,x)),c),lambda(x,ap(c,ap(x,x)))) !=
  lambda(x,ap(c,ap(x,x))). This verifies
  that qq = lambda(x,xx) is a fixed point of q, i.e. q(q) = qq.
  This example demonstrates that we are not working in a typed environment.
  Otter-lambda is also capable of finding fixed points, not just verifying them
  as in this simple example.

```

Give several more examples that demonstrate the correct handling of the λ calculus. For brevity we give only the negated goal from the input file and a brief comment. These examples have short proofs—a single unification leads to a unit conflict with the input $x = x$.

```

ap(lambda(x,lambda(y,x)),c) != lambda(x,ap(c,ap(x,x))).
  The first x is renamed and then x = a.
ap(c,ap(lambda(x,ap(c,ap(x,x))),lambda(x,ap(c,ap(x,x)))) !=
  ap(lambda(x,ap(c,ap(x,x))),lambda(x,ap(c,ap(x,x)))).
  This verifies that qq = lambda(x,xx) is a fixed point of q, i.e. q(q) = qq.
  This example demonstrates that we are not working in a typed environment.
  Otter-lambda is also capable of finding fixed points, not just verifying them
  as in this simple example.

```

Twenty-one years ago, in [18], the Argonne group published a description of the layered architecture which still underpins *AMVA*. They *are* *collaboration* (p. 76), *“By Sata steatunav na ryyer s lamnot greater generality (including, for example, the λ -calculus), but for the time being Layer 1 contains no manipulative procedures other than the basic ones.”* The “time being” lasted twenty-one years.

Quantification in a clausal theorem-prover

Although it is normal in resolution-based theorem provers to remove quantifiers before submitting a problem to the prover, this makes it difficult to deal with definitions that involve quantifiers, such as continuity of a function, or the “divides” relation on integers. In this section we show how

this problem can be solved in what is known as the *naïve* approach to the clause level, based on the idea of *naïve* unification. An example proof will be worked through in detail.

One can regard \exists as a boolean-valued functional, whose arguments are boolean formulas, for simplicity, of the integers, *etc.* (i.e., $\text{Obj} \rightarrow \text{Bool}$). One can also regard \exists as a function from $\text{Obj} \rightarrow \text{Obj}$ to $\text{Obj} \rightarrow \text{Obj}$, introducing a constant symbol \exists this takes the explicit form $\text{Ap}(\exists, \lambda n.P(n))$. This definition of quantification allows one to work with quantified statements directly in Otter- λ . Since Otter- λ uses only characters on the keyboard, the constant \exists is defined to be `exists`. The `exists` function is defined as follows:

```
exists(lambda(x, Ap(Z,x)))
```

This works in Otter- λ as follows: if $Z(t)$ can be proved for any term t , then the literal $\neg \text{Ap}(Z,w)$ will be resolved away, using the substitution $w := t$. Then $\exists(\lambda x.Z(x))$ is deduced, which can be abbreviated as `exists(lambda(x, Z(x)))`. This is the definition of `exists` in Otter- λ .

This will permit the use of definitions that explicitly involve a quantified formula in the definition. For example, we could define

```
divides(u,v) = exists(lambda(x, u*x = v)).
```

Now, given the clause $2*3 = 6$, Otter can deduce `divides(2,6)` as follows. The clause `exists(lambda(x, 2*x = 6))` will rewrite to

```
exists(lambda(x, 2*x = 6)).
```

This will unify with `exists(lambda(x, Ap(Z,x)))` if $2 * x = 6$ will unify with $\text{Ap}(Z,x)$. Second-order unification will be called with x forbidden to Z , since the unification happens in the clause `exists(lambda(x, 2*x = 6))`. The resolution of `exists(lambda(x, 2*x = 6))` with `exists(lambda(x, Ap(Z,x)))` then generate a new clause containing the single literal $\neg \text{Ap}(Z,w)$ with this value of Z . Specifically,

$$\neg \text{Ap}(\lambda w.(2 * w = 6), w).$$

That will be β -reduced to $2 * w \neq 6$ so the clause finally generated will be $2 * w \neq 6$. That resolves with $2 * 3 = 6$, producing a resolution proof. ⁵

⁵ In the original paper, we told the program that $2*3 = 6$ and concluded that 2 divides 6, which is trivial as number theory. Our point is rather to demonstrate how the program expanded a definition that involved a quantifier, and then used the second-order definition of Skolemization and second-order unification to accomplish the proof. Of all the things that Otter incorporates a utility for Skolemizing first-order formulas, so that in some sense one can input quantifiers, it cannot do what is demonstrated here: use quantified formulas within the clause language while a resolution proof is being constructed.

If one wants to prove, say, the transitivity of the `divides` relation, then one needs the other law for the existential quantifier. Space does not permit the inclusion of this example.

The algebraic part of Lagrange's theorem

In this section we use Otter- λ to find the proof of the algebraic part of Lagrange's theorem: the existence of the map $\lambda z. z * a$ from a subgroup H of a group G to the coset Ha , where $a \in G$.

To prepare this for Otter- λ , we assume that the range of the variables is the group G , so we do not need a unary predicate $G(x)$. (See the section on "implicit typing" for justification.) We use a unary predicate $H(x)$ for the subgroup H , and include the axioms that assert that the universe is a group under $*$ with identity e and inverse $i(x)$, and that H is closed under $*$ and inverse. We also include a constant a for a fixed element of G , and the definition of "there exists":

`-Ap(Z,w) | exists(lambda(x,Ap(Z,x)))`.

Now consider how to formalize the coset Ha . We have

$$\begin{aligned} Ha &= \{ha : h \in H\} \\ &= \{w : \exists h.(h \in H \wedge h * a = w)\} \\ &= \lambda w.(\exists h.(h \in H \wedge h * a = w)) \\ &= \lambda w.(\exists(\lambda(h, H(h)) \wedge h * a = w)) \end{aligned}$$

We put this into Otter- λ using a function symbol `f`, as follows:

`f(w) = exists(lambda(h, and(H(h), h*a = w)))`.

We use this as a demodulator, so that it will be used to rewrite any literal `-f(t)` that arises. Now, for simplicity, we begin by proving that there is a function F from H to Ha , without worrying about the one-to-one and onto part yet. The goal is then

$$\exists F \forall x (x \in H \rightarrow Ap(F, x) \in Ha)$$

Since we are using f for the characteristic function of Ha this becomes

$$\exists X \forall x (x \in H \rightarrow f(Ap(X, x)))$$

Introducing a Skolem function g , and replacing x by $g(X)$, the negated goal becomes the two clauses

$$\begin{aligned} &H(g(X)). \\ &-f(Ap(X, g(X))). \end{aligned}$$

The literal `-f(Ap(X, g(X)))` demodulates and resolves with the existential axiom; the unification produces the substitution

`Z := lambda(x, and(H(x), x*a = Ap(X, g(X))))`

and the resolution produces the unit clause

$\text{-Ap}(\text{lambda}(x, \text{and}(\text{H}(x), x*a = \text{Ap}(X, g(X))))), w)$.

This clause however is not stored yet, because it β -reduces to
 $\text{-and}(\text{H}(w), w*a = \text{Ap}(X, g(X)))$.

We now require a special rule of inference that connects the function symbol ‘and’ to its logical meaning: from $\text{-and}(X, Y) \mid Z$ we can infer the clause $\text{-X} \mid \text{-Y} \mid Z$, where of course X or Y could be negative or positive literals, Z could be a list of literals, and X and Y do not necessarily come first in the clause. This rule of inference, along with a similar rule for the function symbol `or`, has been implemented in Otter- λ . These rules are sound in the sense that they are provable in λ -logic ([6]). Thus Otter- λ obtains the clause $\text{-H}(w) \mid w*a \neq \text{Ap}(X, g(X))$. Next the literal $\text{-H}(w)$ resolves with $\text{H}(g(Z))$. The inferred clause is $g(Z)*a \neq \text{Ap}(X, g(X))$. The single literal in this clause can be unified with the equality axiom $x = x$. This results in unifying $g(Z)*a$ with $\text{Ap}(X, g(X))$. Since $g(X)$ is not atomic, the only way this unification can take place is if $g(Z)*a$ can be written as a term in $g(X)$. More precisely, substituting a new variable z for $g(X)$ in $g(Z)*a$ gives us $z*a$, unifying X with Z , so the top level call to unify returns the result $X = \lambda z.z*a$. That is exactly the desired result for the proof of Lagrange’s theorem: $\lambda z.z*a$ is the desired map from H to the coset Ha .

The construction of the map $\lambda z.z*a$ required nothing but the definition of “there exists” and the existence of a binary operation. The next step is to prove the existence of an inverse map $Y = \lambda w.w*i(a)$, by giving Otter- λ the negated goal

$$(5) \quad \text{-}f(\text{Ap}(X, g(X))) \mid \text{-} \text{Ap}(X, \text{Ap}(Y, r(Y))) = r(Y)$$

instead of only the first half, which was the goal in the example just completed. Here $r(Y)$ represents an arbitrary “constant” forbidden to Y , using the usual Skolemization technique. After finding $X = \lambda z.z*a$ as described above, the program deduces

$$\text{Ap}(\lambda z.z*a, \text{Ap}(Y, r(Y))) \neq r(Y)$$

which beta-reduces to

$$\text{Ap}(Y, r(Y))*a \neq r(Y).$$

Now some group theory is finally needed. The program has to deduce

$$\text{Ap}(Y, r(Y)) \neq r(Y)*i(a).$$

using the axioms of group theory. Once that is done, this clause is resolved against $x = x$, which will cause the unification of $\text{Ap}(Y, r(Y))$ with $r(Y)*i(a)$. Since $r(Y)$ is forbidden to Y , the definition of unification causes a new variable w to be substituted for $r(Y)$ in $r(Y)*i(a)$, and the result is $Y = \lambda w.w*i(a)$ as desired. In order not to distract attention from the main issues of this paper, we put the group-theoretic lemma

$$u*x = v \mid u \neq v*i(x)$$

into the input file; but this is not necessary—one can comment that line out. Otter is generally good at this sort of thing, and unmodified Otter easily finds a four-line proof of this lemma.

This example, although it is only the algebraic part of Lagrange’s theorem, still shows the usefulness of second-order unification operating as part of a resolution-based prover. The rest of Lagrange’s theorem involves natural numbers, and the concepts of partition and counting. Otter-λ is also useful in those areas.

No nilpotents in an integral domain, solved in Otter

An *integral domain* is a ring satisfying $xy = 0 \rightarrow x = 0 \vee y = 0$. A *nilpotent* element is an x such that for some natural number n , $x^n = 0$. As a warming-up exercise in combining mathematical induction with algebra, we asked Otter-λ to prove that there are no nilpotents in an integral domain. It should discover the instance of induction required for itself. This experiment was successful, and there are several aspects of it worth discussing, before we move on to more difficult examples of mathematical induction.

First, the clausal formulation of induction. Experience has shown that this is neither familiar nor intuitive to many people, even experts in automated deduction, and therefore we spell it out. Second, the issue of “sorts” and its solution by “implicit typing”. Although this also arises in connection with ordinary Otter (or other first-order provers), again many people are not familiar with the concept. Third, the use of λ-unification to instantiate the variable property in the induction axiom, thus “finding the correct instance of induction” to use. This is the important point of the example—the other two points are just prerequisites.

We take up the clausal form of induction now. We start with the usual formulation of induction, using a variable y for a predicate of natural numbers:

$$Ap(y, 0) \wedge \forall x (Ap(y, x) \rightarrow Ap(y, s(x))) \rightarrow Ap(y, z)$$

We now reduce it to clausal form. The universally quantified x in the antecedent has to get Skolemized as $g(z, y)$, where g is a new Skolem function symbol. As an intermediate step we obtain

$$Ap(y, 0) \wedge Ap(y, g(z, y)) \rightarrow ap(y, s(g(z, y))) \rightarrow Ap(y, z).$$

Changing the first implication $P \rightarrow Q$ to $Q | - P$ (where as usual in the clause language the vertical bar means “or” and the dash means “not”), we have

$$Ap(y, 0) \wedge (-Ap(y, g(z, y)) | Ap(y, s(g(z, y)))) \rightarrow Ap(y, z)$$

and doing that again we get

$$-Ap(y, 0) | (Ap(y, g(z, y)) \wedge -ap(y, s(g(z, y)))) | Ap(y, z)$$

Because of the conjunction, this splits into two clauses:

$$\begin{aligned} & \neg Ap(y, 0) \mid Ap(y, g(z, y)) \mid Ap(y, z). \\ & \neg Ap(y, 0) \mid \neg Ap(y, s(g(z, y))) \mid Ap(y, z). \end{aligned}$$

Note that if we humans choose a particular formula $P(z)$ to replace $Ap(y, z)$, we can express induction on one particular P in ordinary first order logic. People say that Otter cannot do induction, but actually it can do induction, if you tell it what instance of induction to use. In the case of the problem of “no nilpotents in an integral domain”, obviously we should take $P(x, z)$ to be $x^z \neq 0$. If we do that, and give Otter the two resulting clauses to use and the negated goal $a^n = 0$, it is not difficult to set Otter’s inference rules and parameters so that it finds a proof. However, one may object that with this formulation we have only proved the theorem for integral domains whose underlying set is the natural numbers; if we want something more general we ought to have used unary predicates for the ring and for the natural numbers. We will see in the next section that this is not so.

Implicit Typing

In the “no nilpotents in an integral domain” problem, there are three kinds, or “sorts”, of things involved: ring elements, natural numbers, and predicates of natural numbers (because the induction axiom is stated with a variable for a predicate, and we expect Otter- λ to instantiate that variable). Should we use unary predicates to define these three “sorts”? Generally we wish to avoid using such unary predicates if possible. The method of “implicit typing” shows that under certain circumstances we can dispense with these unary predicates. One assigns a type to each predicate, function symbol, and constant symbol, telling what the sort of each argument is, and the sort of the value (in case of a function; predicates have Boolean value). Specifically each argument position of each function or predicate symbol is assigned a sort and the symbol is also assigned a “value type” or “return type”. For example, in this problem the ring operations $+$ and $*$ have type $R \times R \rightarrow R$, which we might express as $type(R, +(R, R))$. If we use N for the sort of natural numbers then we need to use a different symbol for addition on natural numbers, say $type(N, plus(N, N))$, and we need to use a different symbol for 0 in the ring and $zero$ in N . The symbol Ap in this problem satisfies $type(N, Ap(P, N))$, where P represents the sort of predicates. The Skolem symbol g in the induction axiom satisfies $type(N, g(N, P))$. We call a formula or term “correctly typed” if it is built up consistently with these type assignments. Note that variables are not typed; e.g. $x + y$ is correctly typed no matter what variables x and y are. But when a variable occurs in a formula, it inherits a type from the term in which it occurs, and if it occurs again in the same clause, it must have the same type at the other occurrence for the clause to be considered correctly

typed. Once all the function symbols, constants, and predicate symbols have been assigned types, one can check (manually for now) whether the clauses supplied in an input file are correctly typed. Then one observes that if the rules of inference preserve the typing, and if the axioms are correctly typed, and the prover finds a proof, then every step of the proof can be correctly typed. That means that it could be converted into a proof that used unary predicates for the sorts. Hence, if it assists the proof-finding process to omit these unary predicates, it is all right to do so. This technique was introduced long ago in [28], but McCune says it was already folklore at that time. It implies that the proof Otter finds using an input file corresponding to the above formulation actually is a valid proof of the theorem, rather than just of the special case where the ring elements are the natural numbers.

No nilpotents in an integral domain, solved in Otter- λ

Using Otter- λ instead of Otter, we just give the prover the general formulation of induction, with a variable for the predicate. Recall those two clauses:

$$\begin{aligned} & -Ap(y, 0) | Ap(y, g(z, y)) | Ap(y, z). \\ & -Ap(y, 0) | - Ap(y, s(g(z, y))) | Ap(y, z). \end{aligned}$$

Otter- λ will apply binary resolution to the negated goal $a^n = 0$ and the third literal in each of the two induction clauses. Untyped λ -unification will get the inputs $Ap(y, z)$ and $a^n \neq 0$ as terms to be unified. The result will be to instantiate y to $\lambda z. a^z \neq 0$. The term $-Ap(y, 0)$ will become $-Ap(\lambda z. a^z \neq 0, 0)$, which will beta-reduce to $a^0 \neq 0$, readily recognized as the basis case of the induction. This case can be solved (resolved away) using the axioms $x^0 = 1$ and $1 \neq 0$. There then remain two clauses with one literal each. Writing c to abbreviate $g(n, \lambda(z. a^z \neq 0))$, the remaining clauses can be written $a^c = 0$ and $a^{s(c)} \neq 0$. This is evidently the induction step, set up for proof by contradiction. The Skolem term c can be thought of as an arbitrary “constant”. Indeed Otter- λ finds a proof.

But does this proof actually prove the theorem? We ask whether implicit typing remains a valid technique in Otter- λ , where beta-reduction and lambda-unification are used in the inference process. A λ -term $\lambda x. t$ (which in Otter- λ is written `lambda(x, t)`) is correctly typed if t is correctly typed. In that case if x occurs in t it inherits a type from its parents in t (and since t is correctly typed it inherits the same type from all occurrences). If that type is R and the return type of t is S then $\lambda x. t$ has the return type $R \rightarrow S$ of functions from R to S . We can also ask whether the algorithm for untyped lambda-unification preserves correct typings. For now, suffice it to say that in the case of the “no nilpotents” problem, lambda-unification is used only to instantiate y to $\lambda z. a^z \neq 0$. (This is the 0 of the ring, not the *zero* of N , so that $Ap(y, z)$ is made equal to $a^z = 0$. In the induction clauses, Ap can be correctly typed

by $\text{type}(\text{Boolean}, \text{Ap}(\text{Pred}, N))$, where Pred is the type of predicates on N , i.e., $N \rightarrow \text{Bool}$. Thus y inherits the type Pred from its occurrences in (each of) those clauses. The term $\lambda z. a^z \neq 0$ also has type Pred , so at least in this proof, implicit typing does apply.

This argument illustrates the point that while we can sometimes guarantee in advance of running a prover (Otter or Otter- λ) that a proof obtained from a particular input file will certainly be correctly typable, it is also possible to run the prover first, and then inspect the resulting proof, verifying that it can be correctly typed. In the example at hand, it is possible to carry out an *a priori* argument, but we do not do that here and now. We simply want to illustrate how Otter- λ can find the correct instance of induction, and how implicit typing can be used. As of June 2004, Otter- λ can make use of an optional list of typings of function symbols and predicates supplied in the input file to ensure that it produces only unifiers that respect those typings.

Cantor's theorem and Russell's paradox

One of the most celebrated examples of the use of higher-order unification in theorem-proving is the automatic generation of the “diagonal” proof of Cantor's theorem (see e.g. [4]). Since this theorem and its proof seem at first glance to be heavily dependent, on type theory, we want to show that Otter- λ also can find the proof. We use the method of “implicit typing” discussed above to formulate this problem. Intuitively, we have a set (or type) α and its power set $P(\alpha)$, and we suppose for proof by contradiction that we have a map c from α onto $P(\alpha)$. Then the statement to be proved contradictory is

$$\forall x \in P(\alpha) \exists j \in \alpha \forall z \in \alpha (\text{Ap}(x, z) = \text{Ap}(\text{Ap}(c, j), z)).$$

As usual we introduce a Skolem function $j = J(x)$. Then the “negated goal” becomes

$$\text{Ap}(x, z) = \text{Ap}(\text{Ap}(c, J(x)), z).$$

We also need the following two axioms:

$$\begin{array}{ll} w \neq \text{not}(w) & \text{definition of negation} \\ u \neq v | v = u & \text{symmetry of equality} \end{array}$$

The (implicit) typings in this problem are as follows:

$$\begin{array}{ll} \alpha \ J(P(\alpha)) & J \text{ maps } P(\alpha) \text{ into } \alpha \\ P(\alpha) \ c(\alpha) & c \text{ maps } \alpha \text{ into } P(\alpha) \\ \text{Prop} \ \text{Ap}(P(\alpha), \alpha) & \text{Ap is proposition-valued on } P(\alpha) \\ P(\alpha) \ \text{Ap}(\beta, \alpha) & \text{where } \beta \text{ is the type of maps from } \alpha \text{ to } P(\alpha) \\ \text{Prop} \ \text{not}(\text{Prop}) & \text{the negation of a proposition is a proposition} \end{array}$$

These types are *implicit*, i.e. do not form part of the input to the prover. Note that Ap does not have a unique implicit type in this problem, unlike in the problem about nilpotents in integral domains. Nevertheless the second

argument of Ap always has type α . We do not, then, know *a priori* that an Otter- λ proof of contradiction from the goal given above will be correctly typable, but this can easily be verified by inspection if it is true for a given proof. Otter- λ finds such a proof instantaneously: It resolves the equation $Ap(x, z) = Ap(Ap(c, J(x)), z)$ with $u \neq v | v = u$, producing the new conclusion

$$Ap(Ap(c, J(x)), y) = Ap(x, y).$$

This is then resolved with $w \neq not(w)$; the first step assigns w the value $Ap(Ap(c, J(x)), y)$, and then we have the unification problem

$$Ap(x, y) = not(Ap(Ap(c, J(x)), y)).$$

The “masking term” on the right, i.e. the smallest term that contains all occurrences of x , and occurring as a second argument of Ap , is $J(x)$. Therefore our unification algorithm selects $J(x)$ and y to be replaced by the new λ -bound variable, and takes $x = \lambda u. not(Ap(Ap(c, u), u))$ and $y = J(x)$, or explicitly, $y = J(\lambda u. not(Ap(Ap(c, u), u)))$. We check this solution: $Ap(x, y)$ then beta-reduces to $not(Ap(Ap(c, y), y))$, but $y = J(x)$, so we have $not(Ap(Ap(c, J(x)), y))$, which is the right-hand side of the unification problem. Yes, they are equal. The Otter- λ input file and resulting proof are available online at [7]. It is apparent that the Otter- λ proof is correctly typable.

It is quite amusing that this very same proof is also a proof of Russell’s paradox. To see this, we use the usual representation of sets in λ -calculus, namely we regard $u \in v$ as an abbreviation for $Ap(v, u)$, where v is a *Prop*-valued mapping. Then if we add the axioms $Ap(c, x) = x$ and $Ap(J, x) = x$ to the input file, the term $x = \lambda u. not(Ap(Ap(c, u), u))$ is just another notation for $\{u : u \notin Ap(c, u)\} = \{u : u \notin u\}$. That is just the Russell set R , automatically defined by the magic of higher-order unification. But note that the Russell paradox cannot even be formulated in type theory—that was, after all, the point of Russell’s invention of type theory. Of course, we could remove c and J entirely, rather than adding extra axioms saying they are the identity, but leaving them in shows the relation between Cantor’s theorem and Russell’s paradox more clearly.

We can modify this input file in two different ways to escape Russell’s paradox; these modifications correspond directly to the traditional “solutions” of the paradox. First, we can simply regard the paradox as showing that there is no fixed point of *not*: it is the axiom $w \neq not(w)$ that is suspect. We can say that this holds only for propositions. Alternately we can say that the problem is that Ap is not total. We need to allow for propositions that “never converge” to true or false. We can do that by introducing a “logic of partial terms”, replacing the equality axiom $x = x$ by $x = x | E(x)$, where $E(x)$ means “ x is defined” or “ x exists”. Then we get no contradiction unless we also assume $E(R)$; in other words, the “paradox” shows that the expression “defining” the Russell set is actually undefined; or one may choose to express

this by saying the Russell set does not exist.

This latter fine distinction is reflected in different logics of partial terms. More than one such logic has been studied (see [5], p. 97.), and now we are able to deal with them in Otter- λ . Feferman's theories of classes and operations (see [5], Chapter X), which have been put forward as a competitor to type theory for the formalization of mathematics, are based on a logic of partial terms (or the equivalent). Now, higher-order unification can be used with these theories, not only with type theory. See [6] for more on this subject.

Commutativity of addition proved from Peano's axioms

We now give an example to illustrate search and untyped λ -unification working in concert. The example is, to prove the commutativity of addition from Peano's axioms. Of course, we don't need the axioms about multiplication—all we need is the definition of addition and the induction axioms. Specifically the axioms and the negated goal are shown in the first six lines of the proof:

```

1  □  $x+0=x$ .
10 □  $x=x$ .
13 □  $\neg \text{ap}(y,0) \mid \text{ap}(y,g(z,y)) \mid \text{ap}(y,z)$ .
14 □  $\neg \text{ap}(y,0) \mid \neg \text{ap}(y,s(g(z,y))) \mid \text{ap}(y,z)$ .
15 □  $x+s(y)=s(x+y)$ .
16 □  $a+n \neq n+a$ .

```

At the next step, Otter- λ finds the correct instance for the main induction:

$$a \neq 0+a \mid a+s(g(n,\text{lambda}(x,a+x=x+a))) \neq s(g(n,\text{lambda}(x,a+x=x+a))) + a.$$

The occurrence of this lambda term shows that Otter- λ is attempting to prove $a + x = x + a$ by induction on x . Later in the proof Otter- λ constructs and uses the terms $\text{lambda}(y,y=0+y)$, indicating that it will try to prove the basis case of the main induction by induction on the other variable, and

$$\text{lambda}(y,s(g(n,\text{lambda}(z,a+z=z+a))+y))=s(g(n,\text{lambda}(u,a+u=u+a)))+y).$$

Writing c to abbreviate the inner Skolem term, this is $\text{lambda}(y,s(c+y)=s(c)+y)$, showing that it has identified the crucial lemma: it will try to prove $s(c+y) = s(c)+y$ by induction on y . When I proved this theorem by hand, I chose to prove $c+s(y) = s(c)+y$ by induction on y , which is just one step away from Otter- λ 's choice, by the definition of addition. Here are the remaining twenty steps of Otter- λ 's proof of the theorem, edited for readability by substituting c , b , and d for Skolem terms involving the three instances of induction just mentioned. The unedited proof can be read (or regenerated) at [7]. The search was modest by Otter's standards: 8980 clauses were generated, and the proof took two seconds of CPU time to find. Even so, finding it unaided direct from Peano's axioms is an interesting achievement.

```

17 [binary,16.1,14.3,demod,beta,1,beta]  $a \neq 0+a \mid a+s(c) \neq s(c)+a$ .
18 [binary,16.1,13.3,demod,beta,1,beta]  $a \neq 0+a \mid a+c=c+a$ .

```

35 [binary,17.1,14.3,demod,beta,1,beta,unit_del,10] $a+s(c)\neq s(c)+a \mid s(b)\neq 0+s(b)$.
 36 [binary,17.1,13.3,demod,beta,1,beta,unit_del,10] $a+s(c)\neq s(c)+a \mid b=0+b$.
 56 [binary,18.1,14.3,demod,beta,1,beta,unit_del,10] $a+c=c+a \mid s(b)\neq 0+s(b)$.
 57 [binary,18.1,13.3,demod,beta,1,beta,unit_del,10] $a+c=c+a \mid b=0+b$.
 85 [para_from,57.2.2,15.1.2.1] $0+s(b)=s(b) \mid a+c=c+a$.
 102 [para_from,36.2.2,15.1.2.1] $0+s(b)=s(b) \mid a+s(c)\neq s(c)+a$.
 716 [para_from,85.1.1,56.2.2,unit_del,10,factor_simp] $a+c=c+a$.
 719 [para_from,716.1.1,15.1.2.1] $a+s(c)=s(c+a)$.
 735 [para_from,719.1.1,15.1.1] $s(c+a)=s(a+c)$.
 759 [para_into,735.1.2,15.1.2] $s(c+a)=a+s(c)$.
 1598 [para_from,102.1.1,35.2.2,unit_del,10,factor_simp] $a+s(c)\neq s(c)+a$.
 1612 [para_into,1598.1.1,759.1.2] $s(c+a)\neq s(c)+a$.
 1620 [binary,1612.1,14.3,demod,beta,1,1,beta,unit_del,10] $s(c+s(d))\neq s(c)+s(d)$.
 1621 [binary,1612.1,13.3,demod,beta,1,1,beta,unit_del,10] $s(c+d)=s(c)+d$.
 1634 [para_into,1621.1.1,15.1.2] $c+s(d)=s(c)+d$.
 1663 [para_from,1634.1.1,15.1.2.1] $c+s(s(d))=s(s(c)+d)$.
 1710 [para_into,1620.1.2,15.1.1] $s(c+s(d))\neq s(s(c)+d)$.
 1764 [para_into,1663.1.1,15.1.1] $s(c+s(d))=s(s(c)+d)$.
 1765 [binary,1764.1,1710.1] \$F\$.

Related work

For references to the hundreds of technical papers about higher-order unification in typed formalisms, see [1,14]. As far as I can determine, there has been no previous implementation of lambda-calculus and second-order unification in an untyped, resolution-based, first-order theorem prover. Therefore the related work consists mostly of implementations of higher-order logic. These include:

- PVS, developed at SRI under the direction of N. Shankar [26]
- HOL-Light, which was written by John Harrison [15,16], and is currently being used by him at Intel’s Portland facility
- NuPrl, developed at Cornell under the direction of Constable [11]
- Isabelle, developed by Paulson and Nipkow [22]
- Coq, developed at INRIA, and also in use in Nijmegen [12]
- λ -Prolog [21]
- TPS [20,2,4]

PVS, HOL-Light, NuPrl, Isabelle, Theorema, and Coq are primarily proof-checkers, not proof-finders, although all of them have some ability to fill in the small steps of a proof automatically. Isabelle uses higher-order unification (see [22], p. 77). Coq is based on the Calculus of Constructions [13]. A discussion of the theoretical basis of Coq and a history of its development can be found in the introduction to [12], which states that “the salient feature

which clearly distinguishes our proof assistant ... is its possibility [ability] to extract programs from the constructive content of proofs.” Coq uses at least some higher-order unification⁶, but has no significant proof-search capabilities. Theorema inherits lambda-calculus from Mathematica (*Function*[x, t] is Mathematica notation for $\lambda x. t$), but the deduction apparatus of Theorema is based on first-order natural deduction. The architecture of Theorema allows for multiple (special-purpose) provers, but no prover using higher-order unification has been designed or implemented. NuPrl does not use higher-order unification; its proof-checking method is based on term reduction of a proof term constructed by the user-directed application of tactics. That same description applies to HOL-Light. PVS is similar, but with emphasis placed on the inclusion of decision procedures, and more recently, model-checking [26]. PVS has mainly been applied to the verification of programs, e.g. microcode for a commercial chip. λ -Prolog extends Prolog to a larger class of formulae using higher-order techniques, but like Prolog, it is based on a depth-first search algorithm rather than an architecture that maintains a database of generated conclusions and uses strategies to control the retention of conclusions and the selection of the parents for the next generation of conclusions. It does make use of higher-order unification. Table 1 summarizes this information. ACL2 [8] and Theorema [10] are included, even though they do not implement higher-order logic.

Of the systems mentioned, the one most relevant to our work is TPS. In [3], the creators of TPS say,

TPS combines ideas from two fields which, regrettably, have not achieved much cross-fertilization. On the one hand, there is the traditional work in first-order theorem proving using such methods as resolution, model elimination, or connection graphs. On the other hand we find avant-garde proof-checkers and theorem provers for type theories of a variety of flavors, mostly centered around interactive proof construction with the aid of tactics.

The same could be said about Otter- λ . The difference is that TPS imports inference mechanisms from first-order provers into a typed system, while we import unification methods from higher-order logic into a first-order system. Is there any point in taking this different approach to the problem? After all the higher-order methods are complete for first-order logic, so it might be (and has been!) alleged that it is a waste of energy to try to incorporate higher-order methods in an existing first-order prover like Otter.⁷ Here is another

⁶ I could not find this stated in the Coq tutorial or manual, but searching the Coq source code one finds several mentions of it, e.g. the comment in the implementation of the *apply* tactic: “Last chance: if the head is a variable, apply may try second order unification”. On the other hand Coq often reports “Cannot solve a second-order unification problem,” but one cannot be certain what that implies about the implementation details, if anything.

⁷ I have been told that the interest of the creators of TPS was more in closing the gap between automatic and interactive theorem proving, than in closing the gap between first-order and higher-order theorem proving.

System	Higher order unification?	Accumulates Conclusions?	Proof Search?	Can vary rules and strategies?
PVS	matching	n	some	y
HOL-Light	y	n	MESON_TAC	n
Isabelle	y	n	n	y
NuPrl	n	n	n	n
Coq	y	n	n	n
Theorema	n	n	some	y
λ Prolog	y	n	HH formulas only	n
TPS	y	y	y	some
ACL2	n	n	y	n
Otter	n	y	y	y
Otter2	some	y	y	y

Table 1
Comparison of Systems

quote from [3]:

Naturally, TPS can be used to prove theorems of first-order logic, but we focus mainly on examples from higher-order logic. (Apart from the development of path-focused duplication, a relatively small part of the development effort for TPS has been devoted thus far to certain basic issues of search which are important for first-order logic.)

Specifically, those “issues of search” include having a variety of rules of inference useful for different problems, and having a variety of strategies available for controlling the search, such as pick-and-purge, weight templates, etc., as well as making use of the indexing and memory management techniques provided in the layered architecture on which Otter is based. It is, however, clear that the first-order capabilities of TPS, and even more of the other systems listed, do not at present compare to those of Otter, in part because they lack the features just mentioned. We do not mean to criticize TPS or any of the systems above; the facilities that TPS offers for higher-order unification are probably superior to what we have added to Otter. Decades of work by many very capable experts have gone into these systems and admirable results (commensurate with the efforts) have been obtained. We are only trying to make clear where Otter- λ fits in. In the conclusion of [4], the authors say

There is much to be done in the development of methods for higher-order theorem proving Some major areas where work is needed are: the basic mechanisms

of searching for matings; the efficiency of higher-order unification; the treatment of equality; the introduction of rewrite rules. . . .

Three of these four are addressed in the present work by using the existing solution to these problems in Otter.

Regarding the specific example of Lagrange's theorem: A proof of Lagrange's theorem has been checked by the use of Nqthm [29]. Curiously, this proof is not the usual proof, but a proof by induction on the order of the subgroup. Lagrange's theorem has also been formalized (using set theory) in the Mizar library [27].

References

- [1] Andrews, P., Classical type theory, Chapter 15 of [25].
- [2] Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning, The TPS Theorem Proving System, in: Stickel, Mark (ed.), *10th International Conference on Automated Deduction*, 641–642, Lecture Notes in Artificial Intelligence **449**, Springer-Verlag, (1990).
- [3] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, Hongwei Xi, TPS: A Theorem Proving System for Classical Type Theory, Carnegie Mellon University Department of Mathematics Research Report 94-166A, February, 1995. Available at <http://gtps.math.cmu.edu/tps94-report-94-166a.pdf>, while [4] is not available electronically.)
- [4] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, Hongwei Xi, TPS: A Theorem Proving System for Classical Type Theory, *Journal of Automated Reasoning* **16**, 1996, 321-353.
- [5] Beeson, M., *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin/Heidelberg/New York (1985).
- [6] Beeson, M., Lambda Logic, accepted for publication in the proceedings of IJCAR 2004. Also available for download near the bottom of www.cs.sjsu.edu/faculty/beeson/Papers/pubs.html, along with a supplement containing two proofs that are not included in the version for publication.
- [7] Beeson, M., the Otter- λ website, temporarily to be found at <http://mh215a.cs.sjsu.edu>.
- [8] Boyer, R. S., and Moore, J. S., *A Computational Logic Handbook*, Academic Press, Boston (1988).
- [9] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos, Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning* **2** 287–327, 1986.
- [10] Buchberger, B., *et. al.* A Survey of the *Theorema* Project, RISC Technical Report 97-15 (1997), available online at [ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1997/97-15.ps.gz](http://ftp.risc.uni-linz.ac.at/pub/techreports/1997/97-15.ps.gz).

- [11] Constable, R. L. *et. al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey (1986).
- [12] Coq development team, The Coq Proof Assistant Reference Manual: Version 7.2, Rapport Technique RT-0255 de l'INRIA (2002).
www.inria.fr/rrrt/rt-0255.html
- [13] Coquand, T. and Huet, G., The Calculus of Constructions, *Information and Computation* **76**, 1988, 95-120.
- [14] Dowek, G., Higher-order unification and matching, Chapter 16 of [25].
- [15] Harrison, J., and Théry, L.: Extending the HOL theorem prover with a computer algebra system to reason about the reals, in *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG '93*, pp. 174–184, Lecture Notes in Computer Science **780**, Springer-Verlag (1993).
- [16] Harrison, J., *Theorem Proving with the Real Numbers*, Springer-Verlag, Berlin/Heidelberg/New York (1998).
- [17] G. Huet, A unification algorithm for typed λ -calculus, *Theoretical Computer Science* **1** (1975) 27–52.
- [18] Lusk, E., McCune, W., Overbeek, R., Logic machine architecture: kernel functions, in: Loveland, D. W. (ed.) *6th Conference on Automated Deduction* 70–79, Springer-Verlag, Berlin/Heidelberg/New York (1982).
- [19] McCune, W., Otter 3.0 Reference Manual and Guide, Argonne National Laboratory Tech. Report ANL-94/6, 1994.
- [20] Miller, D., Cohen, E. L., and Andrews, P., A look at TPS, in: Loveland, D. (ed.), *6th Conference on Automated Deduction*, 50–69, Lecture Notes in Computer Science 138, Springer-Verlag, (1982).
- [21] G. Nadathur and D. Miller, An Overview of λ -Prolog, in: Bowen and Kowalski (eds.), *Proceedings of the Fifth International Symposium on Logic Programming, Seattle, August 1988*.
- [22] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science **2283**, Springer-Verlag, Berlin/Heidelberg/New York (2003).
- [23] Pietrzykowski, T., and Jensen, D., A complete mechanization of second order logic, *J. Assoc. Comp. Mach.* **20** (2) pp. 333-364, 1971. +
- [24] Pietrzykowski, T., and Jensen, D., A complete mechanization of ω -order type theory, *ASsoc. Comp. Math. Nat. Conf.* 1972, Vol. 1, 82–92.
- [25] Robinson, Alan, and Voronkov, A. (eds.) *Handbook of Automated Reasoning, Volume II*, Elsevier Science B. V. Amsterdam, 2001. Co-published in the U. S. and Canada by MIT Press, Cambridge, MA.

- [26] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas, PVS: Combining Specification, Proof Checking, and Model Checking, in Alur, R., and Henzinger, T. A. (eds.) *Computer-Aided Verification, CAV '96*, 411–414, Lecture Notes in Computer Science **1102**, Springer-Verlag, New Brunswick, N. J. (1996).
- [27] Trybulec, W. A. Subgroup and Cosets of Subgroups, *Journal of Formalized Mathematics* **12**, 1990.
<http://mizar.uwb.edu.pl/JFM/Vol2/group.2.miz.html>.
- [28] Wick, C., and McCune, W., Automated reasoning about elementary point-set topology, *J. Automated Reasoning* **5(2)** 239–255, 1989.
- [29] Yuan Yu, Computer proofs in group theory, *J. Automated Reasoning* **6(3)** pp. 251–286, 1990.

Active Logic for More Effective Commonsense Reasoning

Michael L. Anderson^{1,2}

*Institute for Advanced Computer Studies
University of Maryland, College Park, MD USA*

Darsana P. Josyula^{1,3}

*Department of Computer Science
University of Maryland, College Park, MD USA*

Don Perlis^{1,4}

*Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park, MD USA*

Khemdut Purang

Santa Clara, CA USA

Abstract

The demands of real-time commonsense reasoning—as evidenced for example in the pragmatics of human-computer dialog—put stringent requirements on the underlying logic, including those of (i) perturbation tolerance, (ii) contradiction tolerance and (iii) time situatedness. Active logic is an attempt to meet all of these needs. In this paper we present this work and its application to natural language dialog via time-sensitive meta-reasoning. We illustrate this with a description of ALFRED, a cooperative natural language interface to multiple task-oriented domains.

Key words: first order logic, reasoning, meta-reasoning,
human-computer dialog

¹ This research supported in part by grants from AFOSR and ONR

² Email: anderson@cs.umd.edu

³ Email: darsana@cs.umd.edu

⁴ Email: perlis@cs.umd.edu

1 Introduction

The demands of real-time commonsense reasoning—as evidenced for example in the pragmatics of human-computer dialog—put stringent requirements on the underlying logic, including those of (i) perturbation tolerance, (ii) contradiction tolerance and (iii) time situatedness. Perturbation tolerance is the ability to maintain effective reasoning in the face of unanticipated changes.⁵ Because in a logic-based system change (especially when it is unexpected) can result in contradictions between what is believed or expected and what is actually being experienced, perturbation tolerance implies a need for contradiction tolerance. Time situatedness is the capacity to track (and reason about) the passage of time as reasoning proceeds; not only is this ability useful when reasoning in light of approaching deadlines, but the simple awareness, and explicit representation, of the fact that beliefs change over time can be an important part of effectively managing that change. Active logic (and its implementation in ALMA—Active Logic MACHine) is an attempt to meet all of these needs. We have, in recent years, gathered together some key features of Active Logic into an algorithm we call MCL, that seems to provide a powerful and general tool for commonsense reasoning, as well as a unifying view of our various applications.

Our current view of commonsense reasoning is that it depends in large measure on a special skill in meta-reasoning, in the service of error-handling. This view is motivated by the thought that the ability to “get along” in a wide range of changing environments is a key feature of human cognition. Our suspicion is that this ability, which largely amounts to perturbation tolerance, is the result of robust self-monitoring, and the ability to notice and respond to novelty, error, unexpected outcomes, and the like, which are the inevitable result of interaction with a complex and dynamic environment. More specifically, we hypothesize a three-part process that (i) notes anomalies in overall system performance, (ii) assesses options for dealing with it, and (iii) guides a response into place. We call this note-assess-guide, the metacognitive loop (MCL). That humans are skilled at such a process is clear; our efforts are aimed at seeing to what extent the same process can be effectively incorporated into automated systems as well.

Note that suitably simple versions of MCL need not employ logic at all. For instance, we have performed a number of experiments in reinforcement learning, where an “agent” (CHIPPY) meanders in a two-dimensional grid with various levels of reward (positive or negative) at different locations. Standard learning algorithms (such as Q-learning) work very well; however, when the reward locations *change* after the learning has gotten far along, then the re-learning takes significantly *longer* than the initial learning. This is because

⁵ Perturbation tolerance is meant as an extension and generalization of John McCarthy’s notion of “elaboration tolerance”: the ability of a reasoning agent to appropriately adjust its beliefs and actions to respond to new information [19].

the agent must un-learn what it has learned, and (in non-MCL versions) the algorithm is unable to detect that what it has learned is now doing poorly; it simply keeps plugging along, adjusting its action policy little by little. But even the seemingly simple stratagem of throwing out all that it has learned will speed up the re-learning a great deal; and this is easily done by a trivial version of MCL, which, when it notices that (say) three mismatches have occurred between the rewards it actually received, and the rewards it expected, decides to forget its current action policy and start over. Of course, throwing out one's entire policy is drastic, and for many purposes unwise; we are currently working on more sophisticated versions of MCL that will reason about matters such as context and time, to widen the range of options that can be guided into place. We expect that as the demands on the meta-reasoning increase, logic-based implementations will become more attractive.

This is certainly the case when implementing MCL for use in natural language interfaces. As we will discuss in more detail below, natural language understanding is a difficult process full of the opportunity for error; thus the ability to notice and recover from errors is extremely important. But in the case of language understanding there is no single solution to apply in every case; to assess and decide how to fix an error requires careful considerations of its origin, and this in turn requires the ability to represent and reason about one's own past reasoning. To implement this ability in first-order logic, and in particular a contradiction-tolerant, and time-sensitive variation of FOL, is the natural choice.

In what follows, we will first discuss active logic in general, and its specific implementation in ALMA. Next we will discuss human-computer interfaces and their requirements, especially in conversation. This leads finally to our implementation of a natural-language based interface, ALFRED, which is presented in some detail.

2 Active Logic: An Introduction

Active logics are a family of formalisms that combine inference rules with a constantly evolving measure of time (a 'now') that itself can be referenced in those rules. An account of the basic concepts can be found in [11].

In active logic, aspects of the environment are represented as first order formulas in the knowledge base. Such formulas might represent perceptions of a user's utterance, observations about the state of the domain, or rules added by a system administrator. Inference rules provide the mechanism for "using" the knowledge for reasoning. One aspect of active logic especially important in the current context is its robust ability to continue to reason normally as formulas are added, changed or deleted from its knowledge base. In other words,

the evolving knowledge base is naturally integrated into the ongoing reasoning processes. This makes active logic a good candidate for a reasoning agent which is expected to observe and interact in real time with a continually changing world or domain.

One of the original motivations for active logics was that of designing formalisms for reasoning about an approaching deadline; for this use it is crucial that the reasoning takes into account the ongoing passage of time as that reasoning proceeds. Such a formalism has the ability to explicitly track the individual steps of a deduction, making it a natural mechanism for reasoning about contradictions and their causes.

Thus, in active logic, the formulas at step $i+1$ include the formulas obtained by applying the rules of inference *once* to the formulas in step i as illustrated below:

$$(1) \quad \begin{array}{l} i \quad : \quad \underline{A, A \rightarrow B, B \rightarrow C} \\ i + 1 : \quad \underline{A, B, A \rightarrow B, B \rightarrow C} \\ i + 2 : \quad A, B, C, A \rightarrow B, B \rightarrow C \end{array}$$

Here, at step $i + 1$ B is derived using the axioms A and $A \rightarrow B$ and at step $i + 2$ C is obtained from the axioms, B and $B \rightarrow C$.

Each “step” in an active logic proof itself takes one active logic time-step; thus inference always moves into the future at least one step and this fact can be recorded in the logic. The KB will at all times be finite since the finitely-many inference rules can produce only finitely-many conclusions in one time-step.⁶

In ongoing work begun in [23] we have been exploring ways to keep the KB size not merely finite but bounded, analogous to human short-term memory.

By default, all beliefs from one step that are not directly contradicting are inherited to the next step. However some beliefs like the ones related to the current time are not inherited to the next step. (See below). The inheritance of formulas from one step to the next is controlled by inheritance rules. One simple version of such an “inheritance rule”⁷, which also illustrates the use of firing

6

7

Inheritance and disinheritance are directly related to belief revision [12] and to the frame problem

conditions, is shown in 2:

$$(2) \quad \begin{array}{l} i : A \\ \hline i+1 : A \end{array}$$

[condition: $\neg A \notin \text{KB}$ at step i and $A \neq \text{Now}(i)$]

In addition to the formulas obtained from applying rules of inference to formulas at the previous step, new formulas can be added at each step. Step-wise reasoning coupled with this ability to add new formulas, ensure that the logic would not get stuck in a lengthy proof, oblivious of the other events that occur *during* the reasoning.

To achieve much of their reasoning, active logics employ a notion of “now” that is constantly updated by the “clock rule” shown in 3:

$$(3) \quad \begin{array}{l} i : \text{Now}(i) \\ \hline i+1 : \text{Now}(i+1) \end{array}$$

The clock rule states that from the fact that it is step i at the current step, the step number of the next step is $i + 1$. This step-wise tracking model of time is very different from the “time-frozen” characterization of time that temporal logic [1,29] has. The

notion of past, present and future, that temporal logics have do not change while theorems are being derived. This sharply contrasts with the special evolving-during-inference model of time that active logics have. When an agent is reasoning about its own ongoing activity, or about another agent whose activity is highly interdependent, traditional “time-frozen” reasoning is at a disadvantage, and “time-tracking” active logics can bring new power and flexibility to bear. For instance, theorems can be marked with their time (step-number) of being proven, i.e., the current value of “now”.

This step-number is itself something that further inferences can depend on, such as inferring that a given deadline is now too close to meet by means of a particular plan under refinement if its enactment is estimated to take longer than the (ever shrinking) time remaining before the deadline.

It is the time-sensitivity of active logic inference rules that provides

[18,6];
see [23] for further discussion.

the chief advantage over more traditional logics. Thus, an inference rule can refer the results of all inferences *up until now*—i.e. thru step i —as it computes the subsequent results (for step $i + 1$). This allows an active logic to reason, for example, about its own (past) reasoning; and in particular about what it has *not* yet concluded. Moreover, this can be performed quickly, since it involves little more than a lookup of the current knowledge base. In the sub-sections that follow, we will discuss some of the useful general features of active logics.

2.1 *Contradiction Tolerance*

As is well known, traditional formalisms, including most modal, temporal and nonmonotonic logics, suffer from the “swamping problem” (this is related to the “omniscience” problem of traditional logics of belief: all (infinitely-many) consequences of the axioms are theorems and hence are believed). As a result, in those logics, any possible clues as to how to proceed with reasoning when a contradiction is encountered are rendered ineffective by their own negations which are also derived from the contradiction.

There have been some attempts to overcome the swamping problem, but so far only in the propositional case, and even so the essential time-dependency for real-time capabilities is still missing there. Even though the problem of inconsistency is treated by some logics like paraconsistent logics, in reality most of the traditional logics do not note or repair inconsistencies, they just carry on with them. Nor in general do they provide for any special real-time status as needed by a real-world agent. On the other hand, active logics are intended for on-board use by an agent, not as an external specification of an agent.

In active logics, since the notion of inference is time-dependent, it follows that at any given time only those inferences that have actually been carried out so far can affect the present state of the agent’s knowledge. As a result, even if directly contradictory wffs, P and $\neg P$, are in the agent’s KB at time t ,

it need not be the case that those wffs have been used by time t to derive any other wff, Q . Indeed, it may be that t is the first moment at which both P and $\neg P$ have simultaneously been in KB.

By endowing an active logic with a “conflict-recognition” inference rule such as that in 4, *direct* contradictions can be recognized as soon as they occur, and further reasoning can be

initiated to repair the contradiction, or at least to adopt a strategy with respect to it, such as simply avoiding the use of either of the contradictands for the time being. Unlike in truth maintenance systems [9,10] where a separate process resolves contradictions using justification information, in an active logic the contradiction detection and handling [21] occur in the same reasoning process. The *Contra* predicate is a meta-predicate: it is about the course of reasoning itself (and yet is also part of that same evolving history).

$$(4) \quad \begin{array}{l} i \quad : \quad \frac{P, \neg P}{\text{Contra}(i, P, \neg P)} \\ i+1 : \quad \text{Contra}(i, P, \neg P) \end{array}$$

The idea then is that, although an indirect contradiction may lurk undetected in the knowledge base, it may be sufficient for many purposes to deal only with direct contradictions. Sooner or later, if an indirect contradiction causes trouble, it may reveal itself in the form of a direct contradiction. After all, a real agent has no choice but to reason only with whatever wffs it has been able to come up with *so far*, rather than with implicit but not yet performed inferences. Moreover, since consistency (i.e., the lack of direct or indirect contradictions) is, in general, undecidable, all agents with sufficiently expressive languages will be forced to make do with a hit-or-miss approach to contradiction detection. The best that can be hoped for, then, seems to be an ability to reason effectively in the presence of contradictions, taking action with respect to them only when they become revealed in the course of inference (which itself might be directed toward finding contradictions, to be sure).

But while disinheritng contradictands is a reasonable first step, it is often not enough even to “defuse” the contradiction for long. P and $\neg P$ may have come into KB for reasons that are still in force and the system may re-derive P and $\neg P$, or other similar conflicts, later on. Thus, in

[22,13,25] we

have investigated ways to allow an active logic-based reasoner to retrace its history of inferences, examine what led to the contradiction, and perform meta-reasoning concerning which of these warrants continued belief.

2.2 Non-monotonicity

Unlike most NMR formalisms, we do not attempt to capture the (usually undecidable) absolute truth about what is consistent with what is known; this is in general impossible for real agents. If nothing is *already* known that would prevent a default conclusion, then the agent has little choice except to draw that conclusion, and this is what an active logic does. If later (with more time) the agent discovers a consequence of its beliefs that in fact should have prevented that conclusion, it is only at that later time that it can be withdrawn, and this is what active logic makes possible. In principle, in the limit, active logic should, in special cases at least, provide the same default conclusions as standard NMR formalisms; this is a topic of current investigation.

2.3 Introspection

In Active logic, negative introspection – the ability to determine that one does *not* know something – is often encoded as the following inference rule (where the notation [B] means that B is not present):

$$(5) \quad \begin{array}{l} i \quad : \quad \dots[B] \\ \hline i+1 \quad : \quad \text{-Know}(i,B) \end{array}$$

This mandates the conclusion at time $i+1$ that statement B was not known to the logic at time i (that is, B does not appear among those beliefs).

2.4 Defaults

Defaults are given a straightforward representation in active logics:

$$(6) \quad \begin{array}{l} i \quad : \quad Q, \text{-Know}(\neg P, i), \text{Now}(i) \\ \hline i+1 \quad : \quad P \end{array}$$

Thus if $\neg P$ is not known at the current time, and if Q is known, then P is inferred by default at the next time step. This avoids the decidability issues of traditional

default mechanisms, since only a linear lookup in the belief set for time i is needed to tell that $\neg P$ is not there (and that Q is there). This does not in itself deal with problems arising from interacting defaults. However, since such cases tend to involve contradictory

conclusions (as when, evidence for $\neg P$ becomes known), they can be treated as any other contradictands. In our past work, we have applied a very simple expedient in such cases: disinherit the default conclusion and accept the non-default evidence.

2.5 *History Tracking*

Active logic maintains a temporal history of its reasoning process that can be used by the logic for further reasoning. The history enables the logic to determine when each formula was added or deleted in its past and thus provides a mechanism to reason about the past reasoning.

2.6 *Quotation*

Quotation mechanism names the different formulas in Active logic. This allows an individual formula to be referenced using its name. The quotation and the history mechanism together provides a mechanism for meta-reasoning within the reasoning process itself.

2.7 *Integration with non-logical processing*

Finally, we extend the logic with one special proposition, *call*, which, if it is ever proved, will initiate external action (that can be reasoned about and tracked through observation). Our current implementation of active logic, as represented in ALMA [26], already has this ability to initiate, observe and respond to external events and non-logical processes. Active logics can be seen either as formalisms per se, or as inference engines that implement formalisms. Unlike traditional logics, the notion of theoremhood in Active logic depends directly on (i) what is in the current evolving belief set, and (ii) what the current evolving time is.

The traditional markers of a logic are its syntax and its semantics. Active logics have both of these: the syntax is (usually) that of FOL; and the semantics can also be that of FOL with a few addenda such as that $\text{Now}(x)$ has the meaning that the current evolving time is x . (There are also alternative semantics available.) What is missing is a soundness and completeness theorem, and for good reason: active logics are not intended to be sound or complete but rather to reflect the step-by-step process of reasoning of a real agent. Thus

many true assertions will not be proven, and many things proven are not true. In fact, active logics are designed with inconsistent belief sets in mind; and these of course can never be true. It is best to avoid a mere terminological squabble over the word “logic”. However, in many important senses, active logics are formal specifications of notions of theoremhood appropriate to the study of real agents. If we are concerned about agents and their reasoning, rather than about an agent-independent notion of truth, then we should not expect or want a tight coupling between what is proven (or provable) and what is true. Agents can only do what they have the resources to do, and whatever logic an agent uses must therefore also have that property. Thus to the extent that logic is the study of reasoning, active logics are the study of reasoning as an active process.

3 Alma/Carne

Alma/Carne was designed to be the core reasoning engine for active logic based applications. Alma generates active logic inferences and Carne executes arbitrary processes and interfaces with the world on behalf of Alma. These processes can include non-logical reasoning procedures like probabilistic reasoners, parsers and enables close interaction between different kinds of reasoning. Alma differs from the description of active logic given above in two main ways. Firstly, additional features are required for interaction with external systems. Secondly, a practical system requires compromises to the ideal system so it can run effectively. We describe these differences below.

An application is specified by a set of logical sentences that are loaded into Alma and a set of procedures loaded into Carne. As the system runs, inferences are made in Alma, some of which trigger procedures in Carne. These procedures can perform some computation or cause effects in the world. Alma’s state is updated with the status of the procedures which enables reasoning about the processes Alma triggered. Failure of a procedure, for instance, can lead to reasoning that causes retraction of earlier assumptions. Carne can also monitor the world and assert formulas about the state of the world into Alma, implementing the observation functionality of active logic. This enables Alma to react to changes in the world.

3.1 Alma implementation highlights

Following are some of the more interesting aspects of Alma’s

implementation of active logic.

3.1.1 Representation

Formulas input to Alma are converted to clausal form and asserted in the database. They are indexed by the predicate and the first two arguments of the atom. For formulas of the form $\text{pred}(a, b, c)$ the first two arguments are a and b . The system maintains information about the formulas, for example, how they were derived, at what step, and which derivations they were involved in. An important such piece of information is the status of the formula: trusted or distrusted. Distrusted formulas are not used for further inference but remain present in the database. Formulas typically become distrusted when they are involved in a direct contradiction. All of that meta-information is available through reserved predicates and form the basis of the meta-reasoning engine. Other reserved predicates are used for the database (including meta-information) when asserted. For example, user-controlled change of the status of a formula can be used to remove from further inference potentially false formulas, or, on the other hand, to reinstate formulas that were previously discredited.

3.1.2 Computation

Alma is not complete in the forward direction. This requires more care on the part of the user in choosing formulas to assert in Alma. The procedural as well as logical aspects of Alma have to be considered when specifying strategies. Alma can perform arbitrary logic computation with the *eval_base* form. The computations performed should be limited to those that will complete fast. Longer running computations are to be done using Carne.

The *contradiction detection rule* verifies whether the negation of any newly added atom is present in the database. Inconsistencies are therefore only detected when complementary atoms are derived. The presence of an atom and its complement causes the status of the formula to change to being 'distrusted'. The fact that there is a contradiction and the identity of the contradictands are asserted in the database. Strategies for dealing with the contradiction can be expressed as user-specified formulas which depend on the contradiction assertions being present in the database. These can use the predicates in Alma to decide as a resolution to the contradiction. Alma stops the immediate spread of potential bad

facts by distrusting them and provides the means for the user to handle the contradiction but has no further built-in strategy.

The *time rule* deletes the previous $now(i)$ and asserts $now(i + 1)$ and thereby defines each step.

As alluded to above, there is no explicit *inheritance rule*. All formulas are inherited by default, except for those that are distrusted or explicitly removed.

Introspection is achieved by determining the truth value of $Know(P)$ by merely inspecting the database once P

is ground. This limits the expressivity somewhat but provides substantial gains in efficiency.

3.1.3 Control

The rules above (resolution, contradiction detection and time) are applied at each step. While contradiction detection is applied to all formulas, the application of resolution can be restricted by the user. Prolog filters can be specified that determine which formulas resolution is to be applied to. The user can also specify an ordering on pairs of formulas that are candidates for resolution application. This, together with a limit on the time to be spent on each step can lead to a bounded step duration which only does the most important resolutions. Here as in contradiction handling, Alma provides the means for the user to specify strategies. It is possible for user-specified strategies to indefinitely delay inferences.

A further device that Alma provides for controlling the inference is the availability of three kinds of conditional according to whether they are to be used for forward chaining, or backward chaining or both. This can be a crucial distinction practically. There may be implications that are sometimes required but will typically generate “useless” formulas in forward chaining which can then swamp the database. Using the backward chaining implication eliminates the generation of these useless inferences while allowing the intended inferences to be made.

Backward chaining is done step by step in a breadth first fashion. Breadth first search has the advantage of finding the shortest solution but the disadvantage of requiring more resources. The search tree is maintained as a set of formulas in the Alma database. This means that new information can inform proofs started earlier and sprout new branches in the interior of the tree. Alma provides several ways to control the proof, including a depth limit.

3.1.4 Implementation

The system is implemented in Prolog with a GUI written in Java which displays ongoing inferences in Alma. Alma can output a history of its inference. This history can then be replayed in the GUI. This feature has proved useful for debugging domain descriptions.

3.2 Carne

Carne is a process separate from Alma that communicates with Alma to run procedures that would take too long to run in the Alma process. *Carne* also serves as a link from Alma to other external processes. This allows Alma to be embedded in a larger system with Alma providing reasoning services, as in [30] where Alma/*Carne* implements the dialog manager in a larger planning system.

The following table presents the simple interface between Alma and *Carne*.

Formula	Asserted by	Effect
$do(P, ID)$	Alma reasoning	<i>Carne</i> invokes program P
$doing(P, ID)$	<i>Carne</i>	Informs Alma that P has been invoked.
$done(P, ID)$	<i>Carne</i>	Informs Alma that P has completed.
$error(P, ID)$	<i>Carne</i>	Informs Alma that P has failed.
ϕ	<i>Carne</i>	Informs Alma of ϕ

P is a program that is defined in *Carne*, and ID is a unique identifier for that invocation instance. This simple interface together with Alma's inference allows several types of behavior. For example, the user can specify that a program should be run and aborted if it does not complete within a time limit. This can be done by comparing the time at which *doing* is asserted with the current time and invoking another program to abort the first.

The last row in the table represents *Carne* asserting formulas in the Alma database. Notice that the *do/doing/done* terms do not allow complex results of computation to be asserted in the database.

Programs run by *Carne* can use a reserved term to assert formulas in the Alma database.

4 Some Aspects of Interfaces

Human-computer interfaces are Janus-faced, with one side facing outward and interacting with the user, and another side facing inward, interacting with and controlling some task-oriented system. This system can be limited

and specific, as with a digital video recorder or camera, or more general, as with a desktop computer. A typical interface design strategy is to let the shape of the interface be primarily determined by the function(s) of the device itself, relying on familiar controls for each function. Thus some simple microwaves have just a timer dial and a power-level knob. The advantage of this approach is two-fold. First, the capabilities of the device are displayed on its face, and thus the way to access these functions is obvious to the user. And second, because this is so, very little thought needs to be given to the inward-facing side of the interface; for the user is essentially controlling the device directly. However, the approach also has some important drawbacks. First, as devices get more complex, it is not generally possible to “show” all its capabilities without making them resemble airplane cockpits. Second, as users acquire more electronic devices (computers, cameras, dvd players, mp3 players, cell phones, PDAs, etc.) they may find learning each new interface increasingly burdensome. Thus, as consumer electronics become more complex and various, designers will need to consider the possibility of employing a single, shared, general and flexible interface which is easy for users to learn, and can control many different devices. For this purpose, it is hard to beat natural language.

Implementing a general purpose, natural-language based interface, however, offers some difficult challenges. First, and most obvious, is the difficulty and complexity of language itself. Second is the fact that, insofar as the interface is not specifically designed to control a particular device, the interface must itself be intelligent enough to translate what a user says into something a given device can do. Deeply related to this latter fact is that the user is no longer directly manipulating and controlling the target device; thus much of the agency and proactive responsiveness which would normally be the provenance of the user must be taken over by the interface.

In the subsections that follow, we will discuss each of these challenges in turn. Then in section 5 we outline our general approach to these challenges, and highlight our use of active logic. The remainder of the paper will discuss the various aspects of ALFRED, our flexible, natural-language based interface to several task-oriented domains.

4.1 The Nature of Conversation

The development of a truly viable natural language computer interface will require advances in computer reasoning, in particular in the ability to monitor for, reason about and manage the particular uncertainties inherent in interactive dialog. This, in fact, is what human dialog partners do. They manage the uncertainty inherent in dialog by continually monitoring their conversations, their own comprehension, and the apparent comprehension of their interlocutor. This ability is apparently very basic, and fundamental to language use. Clark [8] presents empirical evidence for the use of meta-linguistic skills by

young children. Among many other things, these skills include monitoring one's ongoing speech, checking the results of that speech, predicting the effects of specific utterances, and deliberately trying to learn.

These behaviors are not just necessary to children learning a language, but in fact pervade conversation between fully competent language users [3,27,28]. The ability to ask questions about what's been said and understood—that is, the ability to engage in meta-dialog (dialog about the dialog or its elements)—and to use the results of these meta-dialogic interchanges to help understand otherwise problematic utterances, is essential to conversation.⁸ We believe that one of the problems facing natural language HCI at the present time is that the computer partners in dialog are not generally equipped with this ability to elicit and utilize appropriate feedback; this can make these systems difficult and frustrating to use [5].

Thus, we have been inching closer (we hope) to a viable natural language computer interface by focusing our efforts on understanding the importance, role, and extent of human dialog monitoring and repair—as well as the meta-reasoning and meta-linguistic skills this monitoring and repair involves—and implementing these behaviors in a task-oriented natural language computer interface [4,31,16,2]. The specific goal we have set for ourselves is the design of a system modeled not on conversation with a fluent colleague, but rather, for example, on a task-oriented interaction with a stranger who doesn't speak much of a common language. Despite the difficulties of understanding the language, interactive dialog behaviors and ongoing repairs allow humans to overcome problems and communicate.

4.2 *The Requirements for a Flexible Domain Controller*

As mentioned earlier, as electronic devices become more complex and various, users find it increasingly frustrating to learn how to operate each one separately. Thus, it would be a great advantage for a single agent to be able to interface with the user, and with all these different task-oriented systems (TOSs). If a single agent can interface users with different TOSs, then the users would need not learn how to interact with each TOS separately; instead, they would just need to learn how to interact with one agent. In addition, if the agent is rational, users can more flexibly adapt the operation of the TOSs by interacting with the agent.

However, for such an interfacing agent to effectively control different TOSs it should have the capability not only to translate a user request into a TOS instruction, and to issue that instruction to the TOS at the appropriate time(s), but, because the user is insulated from direct control of (and may be unable to monitor) the TOS, the interfacing agent should also be able to track the effect of those commands and to detect any perturbations, such as contradic-

⁸ Indeed, as any teacher—and student—knows, it is important to natural language comprehension more generally.

tory information, or a difference between expected and actual outcomes. In addition, because the user may be unaware of the actual capabilities of any given device, the interfacing agent will itself have to be able to do such things as distinguish between achievable and unachievable goals, informing the user of the latter. Finally, to manage its own reasoning it will have to distinguish between these and achieved goals (desires/intentions/expectations) so it can attempt achievable goals, resist attempting unachievable goals (until they become achievable) and ignore achieved goals.

In what follows, we will detail our approach to these different challenges, and describe some of the behaviors we have been able to generate in our natural language HCI systems.

5 Active Logic and the Interfacing Agent

Insofar as the above analysis of the nature of conversation and the requirements for an effective, flexible, error-tolerant domain controller is correct, it suggests that any agent hoping to be a good human-device interface should be:

- (i) **Self-monitoring:** An interfacing agent should track its own language comprehension, maintain a history of its translations of user utterances into domain-specific instructions, and track its own beliefs, desires, intentions, expectations and achievements, so as to be able not just to notice errors (such as mistakes of translation, mismatches between expectations and observations, or between implications of past utterances and current ones) but also to be able to trace their origins and fix them.
- (ii) **Contradiction-tolerant:** Having encountered problems, which are inevitable not just in conversation, but whenever one is dealing with a complex, dynamic environment, an interfacing agent should be able to gracefully handle these situations.
- (iii) **Time-sensitive:** Dialog is governed by a number of time-based expectations. If someone makes a conversational overture, or asks a question, a response is expected in a certain amount of time.⁹ An interfacing agent should be sensitive to these demands. Likewise, it should have expectations about the time it will take its human user to respond, as deviations could indicate a problem. For instance, a long pause could signal a turn change, or that the user is confused, or even that the user is no longer engaged in the conversation. Further, when the interfacing agent gives a command to a domain, it should have specific expectations for what should result, when. These time-specific expectations can be used to monitor progress, and notice and address problems as they arise.

⁹ Exactly how much time is context dependent. For a conversational overture or a very easy question, a near immediate response is required. If a question is perceived to be difficult, or require complex thinking, a longer delay may well be tolerated.

Alternately, a command (or series of commands) may have to be given at certain times and/or in a specific order; time-sensitivity can help make this possible. Finally, time sensitivity can help with deadline-sensitive planning, allowing the reasoner to properly allocate resources or choose strategies for acting and reasoning in light of the time remaining before a looming deadline.

- (iv) Multi-modal: A dialog agent should ideally be able to monitor all the various aspects of speech, including affect, tone and gesture, as well as to match its own utterances with appropriate accompaniment. Unfortunately, monitoring, interpreting and producing these effectively and freely (without prior scripting) is currently beyond the state of the art.¹⁰ However, the ability to attend to multiple contexts is not, and is something which ought to be included in dialog systems whenever possible. For, to understand an utterance it may not be enough to attend to the dialog context (e.g. the subject under consideration, open questions under discussion, turn) but also to the larger environment. If one says “I guess he’s had enough,” the “he” might equally well refer to the fellow under discussion who quit his job, or the one who just fell from his barstool.

Our approach to implementing these abilities involves three conceptual planks. First, there is the issue of representation of aspects of the dialog processing as it moves from user utterance to domain instructions to the setting of expectations. For many object-level behaviors, it is not necessary to have an explicit representation of the processes that the system performs, other than the programming or mechanism that produces the behavior at appropriate times. However, to engage in meta-behavior, such as dialog about dialog, or reasoning about the reasoning process, rich representations are needed. In the case of meta-dialog we are influenced by work such as [14,15,32,24,20], that proposes detailed logical representations of a range of dialog phenomena. Likewise for the interfacing agent’s beliefs, expectations, and the like. To effectively reason about these, especially when faced with an anomaly such as contradictory information, it is important for these to be richly represented. Second, the system must be able to effectively use such representations in inference to be able to notice interesting phenomena, such as implications of what has been done, recognizing resulting anomalies, and deciding what can be done about it. For use in a real-time system, this reasoning cannot be off-line, but must be integrated within the normal behavior of such a system. This leads to the third plank, integration of reasoning with acting and non-logical processing. Reasoning about anomalies and meta-dialog is not enough. To be effective such reasoning should be integrated with normal functioning, being able to affect object level processes.

As detailed in section 2, active logic is a formalism which appears to have all the requirements for implementing the abilities listed above. Thus, we have

¹⁰ Although some impressive work is being done, see, e.g. [7]

been using active logic to implement the real-time meta-reasoning needed for natural-language based interfaces. Because active logic uses an inference mechanism that takes account of the passage of time as it performs inferences, it has both the expressive and the inferential power to monitor its own reasoning in a real-time fashion, as that very reasoning is going on, thus watching for errors (such as mismatches between conveyed and intended meanings); noting temporal conversational cues such as pauses that may signal a turn change; and re-examining its beliefs and altering them appropriately. In what follows we will describe in detail our progress so far.

6 ALFRED

The logical engine ALMA/CARNE has been used to create an interfacing agent ALFRED (Active Logic For Reason Enhanced Dialog) that can exhibit some of the meta-linguistic skills that are necessary for a truly viable natural language human-computer interface. To reduce the complexity of the dialog problem, ALFRED is always connected to some task-oriented system (TOS) and the communication between ALFRED and

a human user is related to the operation of that task-oriented system. The ultimate task of ALFRED is to communicate with a human user in natural language to control different task-oriented systems. The implementation details and the architecture for ALFRED are discussed in [17].

In general, ALFRED interprets the user requests and creates desires to accomplish the requests. Based on knowledge and availability of time and resources, ALFRED creates intentions to achieve the desires and thus issues the appropriate instruction to the TOS. In order to determine whether an instruction that it issued to a TOS has resulted in the desired action, ALFRED creates an expectation regarding the outcome, whenever it initiates an action. It then compares the actual result that it obtains from either the user or the TOS with the expected outcome.

The current implementation of ALFRED has been successfully interfaced with the following domains:

- Simulated Pool: ALFRED controls the temperature settings of a pool based on user needs.
- Movie Player: ALFRED plays different movies based on user requests.
- Toy Train: ALFRED moves various trains to different cities based on user requests.
- Simulated House: ALFRED controls different appliances in a house based

on user needs.

- Home Designer: ALFRED helps the user create and move different objects in a house model.
- Chess Player: ALFRED plays chess for the user by sending the user requested moves to a chess program.

Switching from one domain to another is achieved by loading the domain specific information which includes the names of the domain objects and the syntax and semantics of the valid domain instructions.

6.1 Capabilities due to Active logic

Some of the skills that the current version of ALFRED exhibit and

how the properties of active logic contribute towards implementing these skills are discussed below:

1 Recognizing a pause:

The time-sensitive, history-tracking Active logic provides ALFRED the capability to notice the passage of idle-time and thus recognize pauses in conversation. If no new formulas (observations or inferences) are added into the logic at step i then, the agent notes at step $i+1$ that it was idling in step i . ALFRED associates idle-time with pauses in the on-going conversation and makes use of such pauses to capture the turn in the conversation if or when it needs to do so. This is illustrated in the following example:

User: Send Bullet to Balty

ALFRED: I do not know the city Balty. Which city is Balty?

User: Send Metroliner to Baltimore.

ALFRED: Ok. [sends Metroliner to Baltimore]

ALFRED: Could you tell me which city is Balty?

Here, since ALFRED did not get a reply from the user for a previous question, when there is a pause in the conversation, it takes the opportunity to reiterate its question.

2 Recognizing the unknown:

Using the introspection rule (5), ALFRED can determine that it does not know something. This knowledge, that it does not know X can be used to acquire X where X could be the meaning or referent of a word. In the above example, once ALFRED determines that it does not know the referent for the city “Balty” it deliberately makes an effort to identify the referent for the city “Balty” by engaging in a meta-dialog with the human user.

3 Learning new words:

In active logic new observations can be introduced into the reasoning process at any step. This allows ALFRED to learn new words as in the following example.

User: Balty is Baltimore.

ALFRED: Ok. [learns the new word Balty]

User: Send Metroliner to Balty.

ALFRED: Ok. [sends Metroliner to Balty]

In this example, ALFRED learns the new word Balty and makes use of this new information in the ongoing conversation. In other words, the newly acquired information is used in the inference mechanism used to find the referent for the city Balty during the interpretation of the second utterance. Alma/Carne makes such onboard reactive reasoning feasible by allowing new formulas (observations) to be added at any step and using all formulas at one step to derive the formulas in the next step.

4 Interpreting elliptical utterances:

Active logic's history mechanism provides the framework for maintaining the context of the ongoing conversation. At any point in time, ALFRED has access to all the interpretations that it has assigned to past user utterances. In addition, it has access to its own beliefs, desires, intentions, expectations and achievements at all previous steps. This means that ALFRED can know the expectations that have been achieved and those that have not yet been achieved. ALFRED uses this information to interpret elliptical utterances. For instance, in the example below, ALFRED expects a response from the user that would provide the meaning of the word Richy. So, ALFRED is able to interpret the word "Richmond" in the context of the ongoing conversation.

User: Send Metroliner to Richy.

ALFRED: I do not know the city Richy. Which city is Richy?

User: Richmond.

ALFRED: Ok. [learns that Richy is Richmond]

5 Resolving the references:

Reference resolution involves identifying the

objects that are referenced in the ongoing conversation. ALFRED identifies the referent by introspection, interacting with the task-oriented system and/or getting help from the user. This is illustrated in the following dialog where initially introspection fails to provide the referent for the word “Metro” and hence the agent asks the user for help. Once it learns that Metro is Metroliner, the agent identifies the referent for the word “Metro” in the ongoing conversation, using introspection alone. In the last scenario, introspection provides the information that the Chicago train refers to the train at Chicago. The agent, then interacts with the task-oriented system to identify the train at Chicago.

User: Send Metro to Baltimore

ALFRED: I do not know the train Metro. Which train is Metro?

User: Metro is Metroliner

ALFRED: Ok. [sends Metroliner to Baltimore]

User: Send Metro to Richmond

ALFRED: Ok. [sends Metroliner to Richmond]

User: Send the Chicago train to Richmond

ALFRED: Ok. [sends Bullet to Richmond]

6 Identifying communication problems:

Active logic’s ability to detect contradictions provides the agent the capability to identify communication problems and hence attempt appropriate actions to correct these problems. For instance, in an example originally described in [31], the user

initially says:

“Send the Boston train to New York”

But the agent moves a train other than the one the user intended by the phrase “the Boston train”. Thus the user says:

“No, send the *Boston* train to New York”

This creates a contradiction—do X, don’t do X—which the agent must recognize and consider. In order for the agent to properly interpret the correction in the latter portion of the above utterance, it must come to recognize that “no” is not a change of mind on the user’s part (as it might have been), nor is it an incoherent self-contradiction by the user (don’t send it and do send it), but rather an implicit correction of the intervening action taken by the

agent (sending, say, the Burlington train instead of the Boston train). Recognizing this, the agent reconsiders its initial interpretation of the phrase “Boston train”, and chooses another appropriate candidate to move.

In the example given above, the interpretation of the first utterance causes the belief $move(train1, new_york)$ to be asserted into the logic at some step i . The user’s “No”, causes the agent to believe that the

interpretation that it originally assigned to the first utterance is not correct and hence

$not(move(train1, new_york))$ gets asserted at some step j where $j > i$. This causes the

contradiction rule (4) to be triggered and at step $j+1$, the agent recognizes that there is a

contradiction in its beliefs. The agent then initiates its contradiction-handling mechanism to

decide which of the contradictands can be trusted. In this particular example, the contradiction

handler causes the latter belief to be trusted over the earlier belief. (The assumption here is that a user’s later commands overrides the earlier commands). Thus, the agent comes to

believe $not(move(train1, new_york))$. It uses this belief to eliminate $train1$ from the candidate

list, when it attempts to resolve the reference “the Boston train” the second time.

8 Switching between domains:

Active logic allows new formulas to be added or existing formulas to be deleted at any step.

This means that the agent can easily be switched from one domain to the other at step $i + 1$ by doing the following operations at step $i + 1$

(i) delete all the formulas in the KB that exist at step i and (ii) load the set of domain-specific formulas for the new domain into the logic. As a result the following dialog is feasible.

User: Move Metroliner to Baltimore

ALFRED: I do not know the command ‘‘Move’’. What does ‘‘move’’ mean?

User: Change to chess domain

ALFRED: OK. [connects to chess]

User: Move the king from e2 to e3

ALFRED: Ok. [moves king from e2 to e3]

In this example, when ALFRED is connected to the simulated trains system, since “move” is not a valid command, it asks the user the meaning of the word “move”. However, when it is connected to the chess domain. “move” is a known command and hence it issues the appropriate instruction to the chess player to make the move.

7 Conclusion

Our primary claim is that the demands of real-time commonsense reasoning require a logic that is (i) perturbation tolerant, (ii) contradiction tolerant and (iii) time situated. We believe that active logic has the necessary features for use in commonsense reasoning applications.

More specifically, as illustrated by our dialog system ALFRED, we have shown that one can use active logic to enhance the interactive capability of a task-oriented computer system, by providing the ability to detect and recover from miscommunication problems, including ambiguous references, incompatible or contradictory user intentions, and the use of unknown words.

This technology can already be usefully applied to current application domains, such as home-control software, and we expect that the techniques employed can be refined and extended to handle more sophisticated domains.

We are working on methods of structuring and manipulating conceptual relations to allow ALFRED to learn not just new words for known objects or actions, but also genuinely new objects and concepts. We are also continuing to study human dialog behavior for clues as to what strategies can be employed by competent language users to ensure adequate communication even under difficult circumstances. Our long range goal is to be able to emulate the various behaviors and clarification devices employed by a human learning a foreign language or a novice learning a new subject.

References

- [1] James F. Allen and George Ferguson. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation*, 4(5), 1994.
- [2] C. Andersen, D. Traum, K. Purang, D. Purushothaman, and D. Perlis. Mixed initiative dialogue and intelligence via active logic,. In *Proceedings of the AAAI’99 Workshop on Mixed-Initiative Intelligence*, 1999.
- [3] Michael L. Anderson, Andrew Fister, Bryant Lee, and Danny Wang. On the frequency and types of meta-language in conversation: a preliminary report. In *Proceedings of the 14th Annual Conference of the Society for Text and Discourse*, 2004.

- [4] Michael L. Anderson, Darsana Josyula, and Don Perlis. Talking to computers. In *Proceedings of the Workshop on Mixed Initiative Intelligent Systems, IJCAI-03*, 2003.
- [5] Peter Bohlin, Johan Bos, Staffan Larsson, Ian Lewin, Colin Matheson, and David Milward. Survey of existing interactive systems. *TRINDI Technical Report*, D1.3:1–57, 1999.
- [6] F. Brown, editor. *The Frame Problem in Artificial Intelligence*. Morgan Kaufmann, 1987.
- [7] Justine Cassell, Joseph Sullivan, Scott Prevost, and Elizabeth Churchill. *Embodied Conversational Agents*. MIT Press, Cambridge, MA, 2000.
- [8] Herbert H. Clark and Edward F. Schaefer. Contributing to discourse. *Cognitive Science*, 13:259–294, 1989.
- [9] Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, 1979.
- [10] Jon Doyle. *A Model for Deliberation, Action, and Introspection*. PhD thesis, Massachusetts Institute of Technology, 1980.
- [11] J. Elgot-Drapkin and D. Perlis. Reasoning situated in time I: Basic concepts. *Journal of Experimental and Theoretical Artificial Intelligence*, 2(1):75–98, 1990.
- [12] P. Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. MIT Press, Cambridge, MA, 1988.
- [13] J. Gurney, D. Perlis, and K. Purang. Interpreting presuppositions using active logic: From contexts to utterances. *Computational Intelligence*, 1997.
- [14] Jerry Hobbs. Ontological promiscuity. In *Proceedings ACL-85*, pages 61–69, 1985.
- [15] C. H. Hwang and L. K. Schubert. Episodic Logic: A comprehensive, natural representation for language understanding. *Minds and Machines*, 3:381–419, 1993.
- [16] Darsana Josyula, Michael L. Anderson, and Don Perlis. Towards domain-independent, task-oriented, conversational adequacy. In *Proceedings of the Eighteenth international Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1637–8, 2003.
- [17] Darsana P. Josyula, Michael L. Anderson, and Don Perlis. Domain-Independent Reason-Enhanced Controller for Task-ORiented systems - DIRECTOR. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-04)*, 2004.
- [18] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

- [19] John McCarthy. Elaboration tolerance. In *Proceedings of the Fourth Symposium on Logical Formalizations of Commonsense Reasoning*, 1998.
- [20] Susan W. McRoy, Susan Haller, and Syed Ali. Uniform knowledge representation for language processing in the b2 system. *Journal of Natural Language Engineering*, 3(2/3):123–145, 1997.
- [21] M. Miller. *A View of One’s Past and Other Aspects of Reasoned Change in Belief*. PhD thesis, Department of Computer Science, University of Maryland, College Park, Maryland, 1993.
- [22] M. Miller and D. Perlis. Presentations and this and that: logic in action. In *Proceedings of the 15th Annual Conference of the Cognitive Science Society*, Boulder, Colorado, 1993.
- [23] M. Nirkhe, S. Kraus, M. Miller, and D. Perlis. How to (plan to) meet a deadline between *now* and *then*. *Journal of logic computation*, 7(1):109–156, 1997.
- [24] Massimo Poesio and David R. Traum. Conversational actions and discourse situations. *Computational Intelligence*, 13(3), 1997.
- [25] K. Purang. *Systems that detect and repair their own mistakes*. PhD thesis, Department of Computer Science, University of Maryland, College Park, Maryland, 2001.
- [26] K. Purang, D. Purushothaman, D. Traum, C. Andersen, D. Traum, and D. Perlis. Practical reasoning and plan execution with active logic. In *Proceedings of the IJCAI’99 Workshop on Practical Reasoning and Rationality*, 1999.
- [27] Matthew Purver. A clarification request markup scheme for the BNC. Technical Report TR-02-02, Department of Computer Science, King’s College London, February 2002.
- [28] Matthew Purver, Jonathan Ginzburg, and Patrick Healey. On the means for clarification in dialogue. In R. Smith and J. van Kuppevelt, editors, *Current and New Directions in Discourse and Dialogue*. Kluwer Academic Publishers, 2002.
- [29] N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, New York, 1971.
- [30] D. Traum and C. Andersen. Representations of dialogue state for domain and task independent meta-dialogue. In *Proceedings of the IJCAI99 workshop: Knowledge And Reasoning in Practical Dialogue Systems*, pages 113–120, 1999.
- [31] David Traum, Carl Andersen, Yuan Chong, Darsana Josyula, Michael O’Donovan-Anderson, Yoshi Okamoto, Khemdut Purang, and Don Perlis. Representations of dialogue state for domain and task independent meta-dialogue. *Electronic Transactions on Artificial Intelligence*, 3:125–52, 1999.
- [32] David R. Traum, L. K. Schubert, M. Poesio, N. G. Martin, M. Light, C. H. Hwang, P. Heeman, G. Ferguson, and J. F. Allen. Knowledge representation in the TRAINS-93 conversation system. *International Journal of Expert Systems*, 9(1):173–223, 1996.