
ESCAR
ESCAR
ESCAR
ESCAR
ESCAR
ESCAR

The CADE-20 Workshop on

Empericallly Successful Classical Automated Reasoning

Bernd Fischer[1], Stephan Schulz[2], Geoff Sutcliffe[3]
[1] USRA/RIACS, NASA Ames Research Center
fisch@email.arc.nasa.gov
[2] Technische Universität München
schulz@informatik.tu-muenchen.de
[3] University of Miami
geoff@cs.miami.edu

This workshop brings together practioners and researchers who are concerned with the implementation and deployment of working automated reasoning systems for classical logic (propositional, first order, and higher order). The focus is on classical logic because it has adequate expressive power for many applications, has well understood and manageable computational properties, and the automated reasoning community has much experience and success with the implementation and application of automated reasoning systems for classical logics. Consequently, there exist empirically successful classical automated reasoning systems and applications. The workshop discusses "really running" systems and applications, and not theoretical ideas that have not yet been translated into working software.

The workshop has two main topic areas:

- Systems
  - Implementation techniques and comparisons
  - Data structures and algorithms for the efficient representation of terms, formulae, search states, etc., e.g., new indexing techniques, efficient implemetation 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
  - System integration

Two invited papers, and eight research papers selected from the submissions (by the program committee listed below), will be presented. A panel discussion will enable all attendees to participate in open discussion. The schedule of activities is given below.

Program Committee	
System papers	Application papers
Matt Kaufmann	Bernhard Beckert
Bernd Löchner	Nikolaj Bjorner
Bill McCune	Koen Claessen
Monty Newborn	Bernd Fischer
Larry Paulson	Ulrich Furbach
Silvio Ranise	Greg Nelson
Alexandre Riazanov	Geoff Sutcliffe
Stephan Schulz	
Carsten Schürmann	
Laurent Simon	

Presentation Schedule	
Applications Session 1: Friday 22nd, 14:00-16:00	
14:00-15:00	Invited Talk (shared with the Workshop on Disproving): Automatic Theorem Proving for Program Verification Engines *Byron Cook* (Microsoft Research)
15:00-15:30	Practical Proof Checking for Program Certification *Geoff Sutcliffe, Ewen Denney, Bernd Fischer*
15:30-16:00	Automated Theorem Proving for Quality-checking Medical Guidelines *Arjen Hommersom, Peter Lucas, Patrick van Bommel*
Applications Session 2: Friday 22nd, 16:30-18:00	
16:30-17:00	What First Order Theorem Provers Do For Monodic Temporal Reasoning *Michael Fisher, Ullrich Hustadt, Boris Konev, Alexei Lisitsa*
17:00-17:30	MPTP 0.2: Design, Implementation, and First Cross-verification Experiments; *Josef Urban*
17:30-18:00	The Arrival of Automated Reasoning *Larry Wos, Matthew Spinks*
Systems Session 1: Saturday 23rd, 09:00-10:00	
09:00-10:00	Invited Talk: MiniSAT and some Applications of SAT *Niklas Sörensson* (Chalmers University of Technology)
Systems Session 2: Saturday 23rd, 10:30-12:45	
10:30-11:00	Tau: A Web-Deployed Hybrid Prover for First-Order Logic with Identity, with Optional Inductive Proof; *Jay Halcomb, Randall Schulz*
11:00-11:30	The Implementation of Logiweb *Klaus Grue*
11:30-12:00	Things to know when implementing KBO *Bernd Löchner*
12:00-12:30	Panel: What Tools and Structures are Needed for more Empirically Successful Automated Reasoning? Panelists: *Byron Cook, TBA, TBA*
12:30-12:45	Best Paper Award

Automatic Theorem Proving for Program Verification Engines

Byron Cook
Microsoft Research
bycook@microsoft.com

Abstract

Microsoft has made an extensive investment in the area of software model checking. Tools such as Slam, Zing, KIS, Terminator, and ESP are now being used both inside and outside of the company. In this talk I will describe some of the underlying automatic theorem proving infrastructure used by these verification tools. I will also describe some recent advances and findings in this area.

Dr. Byron Cook is a researcher at Microsoft's research laboratory in Cambridge, England. His research interest include automatic theorem proving, model checking and programming language theory.

Practical Proof Checking for Program Certification

Geoff Sutcliffe[1], Ewen Denney[2], Bernd Fischer[2]
[1]University of Miami
geoff@cs.miami.edu
[2] USRA/RIACS, NASA Ames Research Center
{edenney,fisch}@email.arc.nasa.gov

Abstract

Program certification aims to provide explicit evidence that a program meets a specified level of safety. This evidence must be independently reproducible and verifiable. We have developed a system, based on theorem proving, that generates proofs that auto-generated aerospace code adheres to a number of safety policies. For certification purposes, these proofs need to be verified by a proof checker. Here, we describe and evaluate a semantic derivation verification approach to proof checking. The evaluation is based on 109 safety obligations that are attempted by EP and SPASS. Our system is able to verify 129 out of the 131 proofs found by the two provers. The majority of the proofs are checked completely in less than 15 seconds wall clock time. This shows that the proof checking task arising from a substantial prover application is practically tractable.

1 Introduction

Program certification tries to show that a given program 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. The highest degree of confidence is achieved with approaches that are based on formal methods, and use logic and theorem proving to construct the certificates.

Over the last few years we have developed, implemented, and evaluated a certification approach that uses Hoare-style techniques to formally demonstrate the safety of aerospace programs that are automatically generated from high-level specifications [WSF02a, WSF02b, DFS04a, DFS04b, DFS05]. In that work, we have extended a code generator so that it simultaneously generates code *and* the detailed annotations, e.g., loop invariants, that enable fully automated safety proofs. A verification condition generator (VCG) processes the annotated code and produces a set of *safety obligations* that are provable if and only if the code is safe. An automated theorem prover (ATP) discharges these obligations and its proofs serve as certificates; we focus on *automated*—as opposed to interactive or (the auto-modes of) tactic-based—provers, since we are aiming at a fully automated "push-button" tool.

For certification purposes, users and certification authorities like the FAA must be assured—or better yet, given explicit evidence—that none of the individual tool components yield incorrect results and, hence, that the certificates are valid. The assurance can take a variety of different

forms, e.g., tool pedigree, code inspections, paper-and-pencil proofs, or result checking. In this paper, we focus on automatically checking the correctness of the proofs generated by the ATP, which are crucial elements in our certification chain.

Proof checking is of course not necessary if the applied ATP is known to be correct. However, program certification is a difficult task that requires substantial "deductive power": the longest proof found during experiments involved more than 8000 inference steps. Consequently, simple "correct-by-inspection" theorem provers like leanTAP [BP95], or tactic-based provers built on top of a trusted kernel like Isabelle [Pau89], are not powerful enough.[1] Instead, we need to employ high-performance ATPs, which use complicated calculi, elaborate data structures, and optimized implementations. This makes formal verification of their correctness infeasible [MSM00]. One could argue that these provers have been extensively validated by the theorem proving community (e.g., the soundness checks required for participation in the CADE ATP System Competition (CASC), [PSS02]), so that a formal verification is not necessary. However, this argument by tool pedigree is weak. Most ATPs are under continuous development and single versions are never subjected to enough validation to achieve sufficient "social validation."[2] Moreover, the validation is necessarily incomplete. There have been several published instances of (unintentional) unsoundness in ATPs participating in the CASC, which have been detected only afterwards [SS99, Sut00b, Sut05].

As an alternative to formally verifying or extensively validating the ATPs, they can be extended to generate sufficiently detailed proofs that can be independently verified by a *proof checker*. The checker's function is to verify that the ATP's output is really a proof in the logical system in use. There are several approaches to proof checking, including the syntactic validation of Otter proof steps by Ivy [MSM00], higher-order proof term reconstruction in Isabelle [BN00], higher-order proof step checking in HOL [Won99], reducing proof checking to type checking as in Coq [BC04], and semantic derivation verification [SB05]. Semantic derivation verification has been used in this work. In semantic derivation verification, the required semantic properties of each proof step are encoded in one or more *proof check obligations* (typically an implication from the premises of the applied inference rule to its conclusion), which are then discharged by trusted ATPs. This way, the trusted ATP verifies the proof output of the original ATP. This approach is tractable because the correctness proof for each individual step in the original proof is substantially easier than the original proof itself, and thus within reach of the trusted ATP. For certification purposes, all proofs found by the trusted ATP become part of the certificate that is delivered by the overall certification system.

This paper describes how a semantic derivation verifier has been used to check the proofs that are found by ATPs for the safety obligations generated in the program certification process. The success of ATPs in discharging the safety obligations has been described in [DFS04a]. The success of (trusted) ATPs in verifying the resultant proofs is demonstrated here. Section 2 provides the necessary background on the program certification process, and Section 3 describes the semantic verification technique. Sections 4 and 5 provide empirical data that illustrate the success of the approach. Section 6 concludes, and discusses directions for future work.

[1] See `http://www.cl.cam.ac.uk/users/jeh1004/software/metis/performance.html` for benchmark data.

[2] The notable exception is Otter [McC03b], which has been essentially unchanged since 1996. However, previous experiments have shown that its performance is not sufficient for discharging the safety obligations we generate [DFS05].

2 Formal Program Certification

Formal program certification is based on the idea that the mathematical proof of some program property can be regarded as an externally verifiable certificate of this property. It is a limited variant of full program verification because it proves only individual properties and not the complete behavior, but it uses the same underlying technology.

2.1 Safety Policies

Formal program certification ensures that a program complies with a given *safety policy*. This is a formal characterization that the program does not "go wrong", i.e., does not violate certain conditions. A safety policy is defined by a set of Hoare-style inference rules and auxiliary definitions. The formal basis of this approach is explored in [DF03].

Safety policies exist at two levels of granularity. *Language-specific* policies can be expressed in terms of the constructs of the underlying programming language itself. They are sensible for any given program written in the language, regardless of the application domain. Typical examples of language-specific policies are array-bounds safety (i.e., each access to an array element to be within the specified upper and lower bounds of the array) and variable initialization-before-use (i.e., each variable or individual array element has been assigned a defined value before it is used). Various coding standards (e.g., restrictions on the use of loop indices) also fall into this category. *Domain-specific* properties are, in contrast, specific to a particular application domain and not applicable to all programs. These typically relate to high-level concepts outside the language. In principle, they are independent of the target programming language although, in practice, they tend to be be expressed in terms of program fragments. A typical example is matrix symmetry which requires certain two-dimensional arrays to be symmetric.

2.2 Generating Safety Obligations

For certification purposes, code must be *annotated* with information relevant to the selected safety policy. 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 a verification condition generator (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, and safety conditions are generated at each line. Our 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 or even apply any simplifications. Consequently, the generated verification conditions tend to be large and must be simplified. The more manageable simplified verification conditions can then processed by an ATP.

2.3 Certifiable Program Synthesis

As usual in Hoare-based approaches, the annotation effort can quickly become overwhelming and constitute a barrier for the adoption of the technique. This can be overcome by a *certifiable program synthesis system* that automatically generates the code *and* the detailed annotations

from a high-level specification of the problem. The basic idea is to make the annotations part of the code templates so that they can be instantiated and refined in parallel with the code fragments. We have implemented this approach in two synthesis systems, AUTOFILTER [WS04], which generates state estimation code based on the Kalman filter algorithm, and AUTOBAYES [FS03], which generates statistical data analysis code.

Figure 1 shows the overall architecture of a certifiable program synthesis system. At its core is the original synthesis system that generates code for a given specification. The core system is extended for certification purposes (i.e., by the annotation templates), and augmented with a VCG, a simplifier, an ATP, and a proof checker. These components are described in more detail in [DF03, DFS04b, DFS04a].

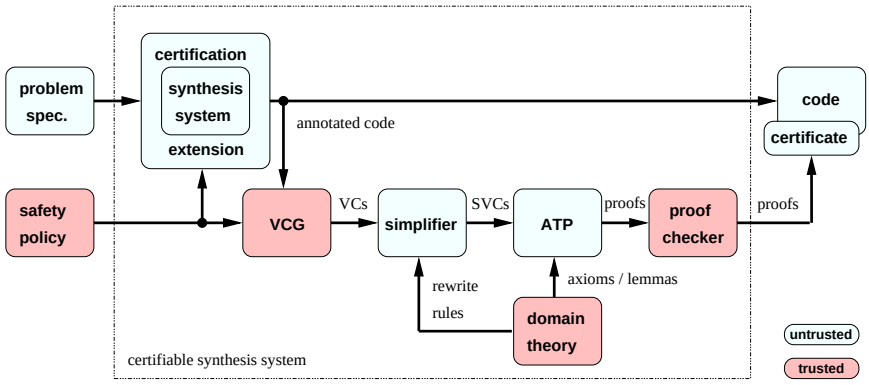


Figure 1: Certifiable program synthesis: System architecture

Similar to proof carrying code [NL98], the architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. Components are called *trusted*—and must thus be correct—if any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the correctness of the certifiable program synthesis system does not depend on the correctness of its two largest components: the original synthesis system (including the certification extensions), and the ATP; instead, we need only trust the safety policy, the VCG, and the proof checker.

3 Semantic Derivation Verification

The proofs produced by ATP systems can be considered more abstractly as derivations. For our purposes, a *derivation* is a directed acyclic graph (DAG), whose leaf nodes are formulae (possibly derived) from the input problem, whose interior nodes are formulae inferred from parent formulae, and whose unique root node is the final derived formula. In semantic derivation verification, the required semantic properties of each inference step in a derivation are encoded in one or more *proof check obligations*. These are then *discharged* by trusted ATPs.

Derivation verification involves three notionally distinct phases. First, it is necessary to check the overall structure of the derivation. This ensures that the ATP output actually is a well-formed derivation DAG. Second, it is necessary to check that each leaf node is a formula

that occurs in, or is derived from, the input problem. This ensures that the ATP actually solves the original problem. Third, it is necessary to check that each inferred formula has the required semantic relationship to its parents. This finally ensures that the proof is correct. The required semantic relationship of an inferred formula to its parents depends on the intent of the inference rule used. Most commonly an inferred formula is intended to be a logical consequence of its parents, but in other cases, e.g., Skolemization and splitting, the inferred formula has a weaker relation to its parents. A comprehensive list of inferred formula statuses is given in [SZS04]. Consequently, there are different forms of proof check obligations; currently GDV distinguishes between theorem obligations, satisfiability obligations, and leaf theorem obligations, which are explained in more detail in the following sections.

The main advantage of semantic derivation verification over other approaches is that it decouples proof checking from proof search—any ATP can serve as the trusted system that checks the output from the untrusted production system. Moreover, the approach is independent of the particular inference rules used in the production ATP, and is also robust with respect to any preprocessing of the input formulae that the production ATP might perform.

3.1 Logical Consequences and Relevance

The basic technique for verifying logical consequences is well known and quite simple. The earliest use appears to have been in the in-house verifier for SPASS [WB+02]. For each inference of a logical consequence in a derivation, a *theorem obligation* is formed; this formalizes that the inferred formula is a logical consequence of the parent formulae. If the inference rule implements any theory (e.g., paramodulation implements most of equality theory), then the corresponding axioms of the theory are added as axioms of the obligation. The obligation is then handed to the trusted ATP system. If the trusted system solves the problem (i.e., finds a proof), the obligation has been discharged.

In practice (see Section 3.5), each attempt to discharge an obligation is constrained by a CPU time limit. Thus the failure to prove a theorem obligation may be because it is actually invalid (indicating a fault in the original derivation), or because the obligation is too hard for the trusted ATP system to prove within the CPU time limit. In order to try to differentiate between these two situations, if the trusted ATP system fails to prove a theorem obligation, GDV generates a *satisfiability obligation* to show that the set consisting of the parents and the negation of the inferred formula is satisfiable, which is then attempted by the trusted ATP. If this is successful then it is known that the theorem obligation cannot be discharged.

The verification of logical consequences ensures the soundness of the inference steps, but does not check for *relevance*. As a contradiction in first order logic entails everything, an inference step with contradictory parents can soundly infer anything. An inference step with contradictory parents can thus always be the last in a derivation. If it is required that an inference step (that infers a formula other than a *false* formula) is not irrelevant, a satisfiability obligation consisting of the parents of the inference must be discharged. This verification step should not be implemented during conversion from FOF to CNF when there is a single parent formulae that is (derived from) the negation of the conjecture—such parent formulae are correctly unsatisfiable when the conjecture is a tautology.

Due to the semi-decidability of first order logic, satisfiability obligations cannot be guaranteed to be discharged. Three alternative techniques, described here in order of preference, may be used to show satisfiability. First, a finite model of the axioms may be found using a model

generation system such as MACE [McC03a] or Paradox [CS03]. Second, a saturation of the axioms may be found using a saturating ATP system such as SPASS or EP [Sch02b]. Third, an attempt to show the axioms to be contradictory can be made using a refutation system. If that succeeds then the satisfiability obligation cannot be discharged. If it fails it provides an incomplete assurance that the formulae are satisfiable.

3.2 Splitting

Many contemporary ATPs that build refutations for CNF problems use *splitting*. Splitting reduces a CNF problem to one or more potentially easier problems by dividing a clause into two subclauses. There are several variants of splitting that have been implemented in specific ATPs, including *explicit splitting* as implemented in SPASS, and forms of *pseudo-splitting* as implemented in Vampire [RV01] and E. Verification of splitting inferences requires several theorem obligations to be discharged.

Explicit splitting takes a CNF problem $S \cup \{L \vee R\}$, in which L and R do not share any variables, and replaces it by two subproblems $S \cup \{L\}$ and $S \cup \{R\}$. If both the subproblems have refutations (i.e., are unsatisfiable), then it is assured that the original problem is unsatisfiable. To verify an explicit splitting step's role in establishing the overall unsatisfiability of the original problem clauses, a theorem obligation to prove $\neg(L \vee R)$ from $\{\neg L, \neg R\}$ is discharged.

Pseudo-splitting takes a CNF problem $S \cup \{L \vee R\}$, in which L and R do not share any variables, and replaces $\{L \vee R\}$ by either (i) $\{L \vee t, \neg t \vee R\}$, or (ii) $\{L \vee t_1, R \vee t_2, \neg t_1 \vee \neg t_2\}$, where t and t_i are new propositional symbols. Vampire implements pseudo-splitting by (i) and E implements it by (ii). The replacement does not change the satisfiability of the clause set—any model of the original clause set can be extended to a model of the modified clause set, and any model of the modified clause set satisfies the original one [RV01, Sch02a]. The underlying justification for pseudo-splitting is that it is equivalent to inferring logical consequences of the split clause and new definitional axioms: for (i) $t \Leftrightarrow \neg\forall L$, and for (ii) $t_1 \Leftrightarrow \neg\forall L$ and $t_2 \Leftrightarrow \neg\forall R$. Pseudo-splitting steps are verified by discharging theorem obligations that prove each of the replacement clauses from the split clause and the new definitional axiom(s).

3.3 Leaf Formulae

The leaf formulae of a derivation must occur in or be derived from the original problem—otherwise, the ATP solves a different problem. To verify this, *leaf theorem obligations* to prove each leaf formula from the input formulae must be discharged. An advantage of the semantic technique for verifying leaf formulae is that it is robust to some of the preprocessing inferences that are performed by ATP systems. For example, Gandalf [Tam98] may factor and simplify input clauses before storing them in its clause data structure. The leaves of refutations output by Gandalf may thus be derived from input clauses, rather than directly being input clauses. These leaves are logical consequences of the original input clauses, and can be verified using this technique.

If the input problem is in FOF (i.e., first-order form including quantifiers, rather than CNF), and the derivation is a CNF refutation, the leaf clauses may have been formed with the use of Skolemization. Such leaf clauses are not logical consequences of the FOF input formulae. Skolemization steps can be incompletely verified by discharging a theorem obligation to prove the parent formula from the Skolemized formula. Although this is an incomplete verification

step (i.e., unsound Skolemization steps can pass this check), it catches simple "typographical" errors and thus provides some additional assurance.

3.4 Structural Verification

All forms of proof checking also include, at least implicitly, some *structural verification*. Structural verification checks that inferences have been used correctly in the context of the overall derivation.

For all derivations, two structural checks are necessary: First, the specified parents of each inference step must exist in the derivation. When semantic verification is used to verify each inference step then the formation of the obligation problems relies on the existence of the parents, and thus performs this check. The check can also be done explicitly. Second, there must not be any loops in the derivation. For derivations that claim to be CNF refutations, it is necessary to also check that the empty clause has been derived.

For refutations that use explicit splitting, two further structural checks are necessary. First, it is necessary to check that both subproblems have been refuted. Second, it is necessary to check that L (R) and its descendants are not used in the refutation of the R (L) subproblem. For refutations that use pseudo-splitting, a structural check is required to ensure that the "new propositional symbols" really are new, and not used elsewhere in the refutation.

3.5 Implementation

The semantic verification techniques described here have been implemented in the GDV system. GDV is implemented in C, using the JJParser library for input, output, and data structure support. The inputs to GDV are a derivation in TPTP format [SZS04], the original problem in TPTP format, a set of trusted ATPs to discharge the theorem obligations, and a CPU time limit for the trusted ATPs for each obligation. SystemOnTPTP [Sut00a] is used to run the trusted ATPs. Obligations that are successfully discharged are reported, and the output from the discharging is optionally retained for later inspection. If an obligation cannot be discharged, or a structural check fails, GDV reports the failure.

4 Experimental Setup

In [DFS05], we evaluate multiple ATPs on 366 safety obligations generated from the certification of programs generated by the AUTOBAYES and AUTOFILTER program synthesis systems. Of those 366 problems, 109 were selected for inclusion in the TPTP problem library [SS05], the standard library of test problem for testing and evaluating ATPs. The 109 problems were selected based on the results of evaluating several state-of-the-art ATPs against the problems, and were selected so as to be "difficult", i.e., with TPTP difficulty ratings strictly between 0.0 and 1.0 [SS01].

As a practical test and evaluation of the proof checking approach described in this paper, we scrutinized the proofs generated for these 109 problems by the ATPs EP (Version 0.82) [Sch02b][3] and SPASS (Version 2.1) [WB+02]. Both EP and SPASS work by converting the axioms and the negated conjecture to CNF, and then using clausal reasoning to find a refutation.

[3] EP is a simple extension of E that produces explicit proofs.

The proofs output by EP include details of the FOF-to-CNF conversion, and the subsequent CNF refutation. The proofs are natively output in TPTP format. The proofs output by SPASS document the CNF refutation, but not the FOF-to-CNF conversion. The SPASS proofs are natively in DFG format, which is translated to TPTP format prior to verification. Both systems are based on the superposition calculus, but differ in the specific inference rules used. A notable difference is EP's use of pseudo-splitting and SPASS's use of explicit splitting. Additionally, the systems have quite different control heuristics. As a result the proofs produced by the two systems have quite different characteristics.

For the verification of the EP proofs, GDV was configured to verify all aspects of each proof: the derivation was structurally verified, leaves were verified as being (possibly derived) from the input problem, all inferred formulae were semantically verified with relevance checking, and all splitting steps were verified. For the verification of the SPASS proofs, GDV was configured to verify selected aspects of each proof: leaves were not verified because SPASS does not document the FOF to CNF conversion, all inferred formulae were semantically verified but without relevance checking, all splitting steps were verified but the independence of the subproblems was not verified in the larger proofs because of the computational complexity, and the derivation was structurally verified (with the exception of the splitting aspect just mentioned). The trusted ATPs used were Otter 3.3 [McC03b] for discharging theorem obligations, Paradox 1.1 [CS03] for finding finite models, and SPASS 2.1 for finding saturations.[4] The outputs from Otter, Paradox, and SPASS were retained, and are available as part of any certificate. The verifications were done on Intel P4 2.8GHz computers with 1GB RAM, and running the Linux 2.4 operating system. The CPU time limit for each discharge was 10s.

5 Experimental Results

Out of the 109 problems, EP can solve 48 and SPASS can solve 83, thus giving a total of 131 proofs to check. The 48 problems solved by EP are a subset of those solved by SPASS, but the proofs are obviously different. Table 1 summarizes the results. The first column gives the overall values for the verification of the EP proofs, including the verification of the steps converting from FOF to CNF and the inferences in the refutation. The next two columns split these values into the two parts. The final column gives the values for the verification of the inferences in SPASS's refutations. The last two columns are thus directly comparable. The first row shows the number of problems solved out of the 109, and the second row shows how many of those were verified by GDV with the checks described above. The next row gives the numbers of theorem obligations that were generated for the verifications and discharged by Otter. The next row gives the average number of theorem obligations per proof, and then the next five rows give their distribution, thus giving an indication of the distribution of the proof sizes. The next block of four rows gives the distribution of the CPU times taken by Otter to discharge the theorem obligations. The final row gives the numbers of finite models found in the relevance checking done for EP proofs.

The table shows that 46 of the 48 problems solved by EP were fully verified. Both failure cases were caused by Otter's inability to discharge obligations arising from steps in the FOF-to-CNF conversion. In particular, the obligations to verify the step that negates the conjecture,

[4] Satisfiability tests, which employ saturation finding, are used only in the verification of leaves and relevance checking. As these checks were not done for the SPASS proofs, this is not a case of SPASS checking itself.

	EP	EP-CNF	EP-Ref	SPASS
Problems solved	48			83
Proofs verified	46			83
Obligations discharged	590	309	281	19737
Average obligations / proof	12.8	6.7	6.1	273.8
Theorem obligations / proof				
0	0	0	19	0
1-10	35	38	22	52
10-100	10	8	4	13
100-1000	1	0	1	12
>1000	0	0	0	6
Discharge time / obligation				
0.0-0.1s	208	123	85	19737
0.1-0.2s	362	172	190	0
0.2-0.3s	17	7	10	0
>0.3s	3	3	0	0
Models found	361	140	221	-

Table 1: Proof Checking Results

which entails proving the negation of the negation from the original, could not be discharged. All 83 of the SPASS proofs passed the verification checks chosen.

Most of the proofs require less than 10 obligations to be discharged, both for EP and SPASS. However, SPASS produces some very large proofs that consequently require a very large number of obligations to be discharged; the largest proof resulted in 3493 theorem obligations. This difference in distribution leads to a significant difference between the average numbers of obligations that had to be discharged per problem. At the same time, all of the SPASS obligations were discharged in almost no time. These figures indicate that SPASS proofs contain very many small, easily verified steps, while EP proofs have slightly larger steps. Note that 19 of the EP proofs were completed in the FOF-to-CNF conversion, and EP's largest proof steps, requiring the longest times for verification, over 0.3s, are within the FOF-to-CNF conversion. There is some overhead starting Otter for each theorem obligation, and this dominates the wall clock time taken (i.e., the time the user has to wait for a proof to be verified). It is thus preferable to have fewer but harder theorem obligations to discharge, as offered by EP.

Of the 590 theorem obligations discharged for EP, 361 had the parents verified as satisfiable, confirming the relevance of the parents to the inferred clause. The remaining 229 theorems were not relevance checked because one of the parent clauses was derived from the negation of the conjecture.

6 Conclusions and Future Work

In this paper, we have described and evaluated a semantic derivation verification approach to proof checking. The evaluation, which is the main contribution of the paper, is based on 109 safety obligations arising in the certification of auto-generated aerospace code.

The results are encouraging. Our system is able to verify 129 out of the 131 proofs found by EP and SPASS, showing that the proof checking task is practically tractable. The vast majority of the proof check obligations are discharged in less than 0.1 seconds. The majority of the proofs are checked completely in less than 15 seconds wall clock time, although some of the longer proofs cannot be verified completely and even the partial checks can take more than five minutes. Moreover, as a consequence of the substantial overheads our current implementation incurs for intermediate format transformation steps, proof checking often requires *more* wall clock time than the actual proof search. However, this is not a fundamental limitation and could easily be changed by an optimized implementation.

There is still a lot of room for improvement. The verification of some trivial proof steps in the FOF-to-CNF conversion failed. The corresponding obligations were of the form $L \models \neg\neg L$, where L is very large. The trusted ATP (i.e., Otter) does not recognize this form and produces a very difficult CNF obligation. Using SPASS as the trusted ATP, however, solves this problem. Similarly, some forms of structural verification are very expensive, in particular for the large proofs found by SPASS. Moreover, the approach relies on the production ATP generating well documented proofs. Currently, only EP satisfies this criterion. SPASS proofs are missing the FOF-to-CNF conversion, and Vampire does not record the negation of the original goal, which makes its proofs uncheckable. Finally, we have evaluated our techniques only for ATPs based on the superposition calculus. Future work will thus be concerned with systems based on other calculi, such as non-clausal resolution or model elimination.

Derivation verification does not provide absolute assurance. The biggest gap is the verification of Skolemization steps, which are only satisfiability-preserving. While the full verification of such steps (and clausification in general) requires further research and experimentation, the partial verification provided here already gives some additional assurance. Other potential gaps are that the construction of the proof check obligations is wrong, and that the trusted ATP contains errors. Moreover, derivation verification as described here only addresses errors in the proofs found by the ATPs but not any errors in the construction and, in particular, simplification of the original verification conditions. However, similar techniques can also be applied to double-check the rewrite engine and rules used for simplification.

Ultimately, however, in order to convince users of the validity of the overall certification process, there needs to be some explicit linking or tracing between the logical entities and the program being certified. In [DF05], we describe a browser which enables a two-way linking between the verification conditions and the individual statements of the annotated program. We are also developing an extension to the VCG which adds "semantic markup" to formulas in the form of labels which explain their origin and meaning. The accumulated labels can then be converted into text and used to interpret the generated verification conditions. We would like to combine tracing, textual rendering, and proof checking into an integrated environment for certification.

References

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. EATCS, 2004.

[BN00] S. Berghofer and T. Nipkow. “Proof Terms for Simply Typed Higher Order Logic”. In *Theorem Proving in Higher Order Logics*, pp. 38–52, 2000.

[BP95] B. Beckert and J. Posegga. “leanT^AP: Lean Tableau-based Deduction”. *J. Automated Reasoning*, **15**(3):339–358, 1995.

[CS03] K. Claessen and N. Sorensson. “New Techniques that Improve MACE-style Finite Model Finding”. In P. Baumgartner and C. Fermueller, (eds.), *Proc. CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[DF03] E. Denney and B. Fischer. “Correctness of Source-Level Safety Policies”. In K. Araki, S. Gnesi, and D. Mandrioli, (eds.), *Proc. FM 2003: Formal Methods*, *Lect. Notes Comp. Sci.* 2805, pp. 894–913. Springer, 2003.

[DF05] E. Denney and B. Fischer. “A Program Certification Assistant Based on Fully Automated Theorem Provers”. In *Proc. Intl. Workshop on User Interfaces for Theorem Provers*, Electronic Notes in Theoretical Computer Science, 2005.

[DFS04a] E. Denney, B. Fischer, and J. Schumann. “Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software”. In M. Rusinowitch and D. Basin, (eds.), *Proc. 2nd Intl. Joint Conf. Automated Reasoning*, *Lect. Notes Artificial Intelligence* 3097, pp. 198–212. Springer, 2004.

[DFS04b] E. Denney, B. Fischer, and J. Schumann. “An Empirical Evaluation of Automated Theorem Provers in Software Certification”. In *Proc. IJCAR 2004 Workshop on Empirically Successful First Order Reasoning*, 2004.

[DFS05] E. Denney, B. Fischer, and J. Schumann. “An Empirical Evaluation of Automated Theorem Provers in Software Certification”. *Intl. Journal of AI Tools*, 2005. To appear.

[FS03] B. Fischer and J. Schumann. “AutoBayes: A System for Generating Data Analysis Programs from Statistical Models”. *J. Functional Programming*, **13**(3):483–508, May 2003.

[McC03a] W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

[McC03b] W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[MSM00] W. McCune and O. Shumsky-Matlin. “Ivy: A Preprocessor and Proof Checker for First-Order Logic”. In M. Kaufmann, P. Manolios, and J. Strother Moore, (eds.), *Computer-Aided Reasoning: ACL2 Case Studies*, Advances in Formal Methods 4, pp. 265–282. Kluwer Academic Publishers, 2000.

[NL98] G. C. Necula and P. Lee. “The Design and Implementation of a Certifying Compiler”. In K. D. Cooper, (ed.), *Proc. ACM Conf. Programming Language Design and Implementation 1998*, pp. 333–344. ACM Press, 1998. Published as SIGPLAN Notices 33(5).

[Pau89] L. C. Paulson. “The Foundation of a Generic Theorem Prover”. *Journal of Automated Reasoning*, **5**(3):363–397, 1989.

[PSS02] F. J. Pelletier, G. Sutcliffe, and C. B. Suttner. "The Development of CASC". *AI Communications*, **15**(2-3):79–90, 2002.

[RV01] A. Riazanov and A. Voronkov. "Splitting without Backtracking". In B. Nebel, (ed.), *Proc. 17th Intl. Joint Conf. Artificial Intelligence*, pp. 611–617. Morgan Kaufmann, 2001.

[SB05] G. Sutcliffe and D. Belfiore. "Semantic Derivation Verification". In I. Russell and Z. Markov, (eds.), *Proc. 18th Florida Artificial Intelligence Research Symposium*. AAAI Press, 2005.

[Sch02a] S. Schulz. "A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae". In S. Haller and G. Simmons, (eds.), *Proc. 15th Florida Artificial Intelligence Research Symposium*, pp. 72–76. AAAI Press, 2002.

[Sch02b] S. Schulz. "E: A Brainiac Theorem Prover". *AI Communications*, **15**(2-3):111–126, 2002.

[SS99] G. Sutcliffe and C. Suttner. "The CADE-15 ATP System Competition". *Journal of Automated Reasoning*, **23**(1):1–23, 1999.

[SS01] G. Sutcliffe and C. Suttner. "Evaluating General Purpose Automated Theorem Proving Systems". *Artificial Intelligence*, **131**(1-2):39–54, 2001.

[SS05] G. Sutcliffe and C. Suttner. The TPTP Problem Library. `http://www.TPTP.org`.

[Sut00a] G. Sutcliffe. "SystemOnTPTP". In D. McAllester, (ed.), *Proc. 17th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 1831, pp. 406–410. Springer, 2000.

[Sut00b] G. Sutcliffe. "The CADE-16 ATP System Competition". *Journal of Automated Reasoning*, **24**(3):371–396, 2000.

[Sut05] G. Sutcliffe. "The IJCAR-2004 Automated Theorem Proving Competition". *AI Communications*, **18**(1), 2005.

[SZS04] G. Sutcliffe, J. Zimmer, and S. Schulz. "TSTP Data-Exchange Formats for Automated Theorem Proving Tools". In W. Zhang and V. Sorge, (eds.), *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, Frontiers in Artificial Intelligence and Applications 112, pp. 201–215. IOS Press, 2004.

[Tam98] T. Tammet. "Towards Efficient Subsumption". In C. Kirchner and H. Kirchner, (eds.), *Proc. 15th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 1421, pp. 427–440. Springer, 1998.

[WB+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. "SPASS Version 2.0". In A. Voronkov, (ed.), *Proc. 18th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 2392, pp. 275–279. Springer, 2002.

[Won99] W. Wong. "Validation of HOL Proofs by Proof Checking". *Formal Methods in System Design: An International Journal*, **14**(2):193–212, March 1999.

[WS04] J. Whittle and J. Schumann. "Automating the Implementation of Kalman Filter Algorithms". *ACM Transactions on Mathematical Software*, **30**(4):434–453, December 2004.

[WSF02a] M. Whalen, J. Schumann, and B. Fischer. “AutoBayes/CC — Combining Program Synthesis with Automatic Code Certification (System Description)”. In A. Voronkov, (ed.), *Proc. 18th Intl. Conf. Automated Deduction, Lect. Notes Artificial Intelligence* 2392, pp. 290–294. Springer, 2002.

[WSF02b] M. Whalen, J. Schumann, and B. Fischer. “Synthesizing Certified Code”. In L.-H. Eriksson and P. A. Lindsay, (eds.), *Proc. Intl. Symp. Formal Methods Europe 2002: Formal Methods—Getting IT Right, Lect. Notes Comp. Sci.* 2391, pp. 431–450. Springer, 2002.

Automated Theorem Proving for Quality-checking Medical Guidelines*

Arjen Hommersom, Peter Lucas, and Patrick van Bommel
Institute for Computing and Information Sciences
Radboud Universiteit Nijmegen
The Netherlands
{arjenh,peterl,pvb}@cs.ru.nl

Abstract

Requirements about the quality of medical guidelines can be represented using schemata borrowed from the theory of abductive diagnosis, using temporal logic to model the time-oriented aspects expressed in a guideline. Previously we have shown that these requirements can be verified using interactive theorem proving techniques [HLB04]. In this paper, we investigate how this approach can be mapped to the facilities of a resolution-based theorem prover, OTTER, and a complementary program that searches for finite models of first-order statements, MACE-2. It is shown that the reasoning that is required for checking the quality of a guideline can be mapped to such fully automated theorem-proving facilities. The medical quality of an actual guideline concerning diabetes mellitus 2 is investigated in this way.

1 Introduction

Health-care is becoming more and more complicated at an astonishing rate. On the one hand, the number of different patient management options has risen considerably during the last couple of decades, whereas, on the other hand, medical doctors are expected to take decisions balancing benefits for the patient against financial costs. There is a growing trend within the medical profession to believe that clinical decision-making should be based as much as possible on sound scientific evidence; this has become known as *evidence-based medicine* [Woo00]. Evidence-based medicine has given a major impetus to the development of guidelines, documents offering a detailed description of steps that must be taken and considerations that must be taken into account by health-care professionals in managing a disease in a patient, to avoid substandard practices or outcomes. Their general aim is to promote standards of medical care.

Researchers in artificial intelligence (AI) have picked up on these developments [FD00, OMGM98], and some of them, for example in the Asgaard project [SMJ98], are involved in the design of computer-oriented languages, tools and systems that support the design and deployment of medical guidelines. AI researchers see guidelines as

*This work has been partially supported by the European Commission's IST program, under contract number IST-FP6-508794 PROTOCURE II.

good real-world examples of highly structured, systematic documents that are amenable to formalisation. Previously, it was shown that for reasoning about models of medical knowledge, for example in the context of medical expert systems [Luc93], automated reasoning techniques (e.g., [Rob65, WOLB84]) are a practical option.

There are two approaches to checking the quality of medical guidelines: (1) the *object-level* approach amounts to translating a guideline to a formal language, such as Asbru [SMJ98], and next applying techniques from program verification to the resulting representation in establishing whether certain domain-specific properties hold; (2) the *meta-level* approach, which consists of formalising general properties to which a guideline should comply, and then investigating whether this is the case. Here we are concerned with the meta-level approach to guideline-quality checking. For example, a good-quality medical guideline regarding treatment of a disorder should preclude the prescription of redundant drugs, or advise against the prescription of treatment that is less effective than some alternative.

Such a meta-level approach corresponds to reasoning that occurs during the process of *designing* medical guidelines and therefore such checks could be valuable. The design of a guideline can be seen as a very complex process where formulation of knowledge and construction of conclusions and corresponding recommendations are intermingled. This makes it cumbersome to do *interactive* verification of hypotheses concerning the optimal recommendation during the construction of such a guideline, because guideline developers do not generally have the necessary background in formal methods to construct such proofs interactively. Automated theorem proving on a language could therefore be potentially more useful for supporting the guideline development process.

The goal of the research described here was to establish how feasible it is to implement such meta-reasoning techniques in existing tools for automated deduction. We will show that it is indeed possible to explore the route from informal medical knowledge to a logical formalisation and automated verification. Previously, we have shown that the theory of abductive diagnosis can be taken as a foundation for the formalisation of quality criteria of a medical guideline [Luc03] and that these can be verified using (interactive) program verification techniques [HLB04]. In this paper, we provide an alternative to this approach by translating this formalism, a restricted part of temporal logic, to standard first order logic. We will show that, because of the restricted language we used for the formalisation of the object knowledge, the translation is a relatively simple fragment of first-order logic and is therefore amenable to automated reasoning techniques.

The paper is organised as follows. In the next section, we start by explaining what medical guidelines are, and a method for formalising guidelines by temporal logic is briefly reviewed. In Section 3 the formalisation of guideline quality using a meta-level scheme which comes from the theory of abductive diagnosis is described. The guideline on the management of diabetes mellitus type 2 that has been used in the case study is given attention in Section 4, and a formalisation of this is given as well. An approach to checking the quality of this guideline using the deductive machinery offered by OTTER and MACE-2 is presented in Section 5. Finally, Section 6 discusses what has been achieved, the advantages and limitations of this approach are brought into perspective and future research plans are mentioned.

- Step 1: diet
- Step 2: if Quetelet Index (QI) $\leq$ 27, prescribe a sulfonylurea drug; otherwise, prescribe a biguanide drug
- Step 3: combine a sulfonylurea drug and biguanide (replace one of these by a α-glucosidase inhibitor if side-effects occur)
- Step 4: one of the following:
  - oral antidiabetics and insulin
  - only insulin

Figure 1: Tiny fragment of a clinical guideline on the management of diabetes mellitus type 2. If one of the steps $k = 1, 2, 3$ is ineffective, the management moves to step $k + 1$.

2 Preliminaries

2.1 The Design of Medical Guidelines

The design of a medical guideline is far from easy. Firstly, the gathering and classification of the scientific evidence underlying and justifying the recommendations mentioned in a guideline is time consuming, and requires considerable expertise in the medical field concerned. Secondly, medical guidelines are very detailed. Making sure that all the information contained in the guideline is complete for the guideline's purpose, and based on sound medical principles, is hard work. An example of a tiny portion of a guideline is shown in Figure 1; it is part of the guideline for general practitioners about the treatment of diabetes mellitus type 2. This guideline fragment is used in this paper as a running example.

One way to use formal methods in the context of guidelines is to automatically verify whether a medical guideline fulfills particular properties, such as whether it complies with quality *indicators* as proposed by health-care professionals [MBtTvH02]. For example, using particular patient assumptions such as that after treatment the levels of a substance are dangerously high or low, it is possible to check whether this situation does or does not violate the guideline. However, verifying the effects of treatment as well as examining whether a developed medical guideline complies with global criteria, such as that it avoids the prescription of redundant drugs, or the request of tests that are superfluous, is difficult to impossible if only the guideline text is available. Thus, the capability to check whether a guideline fulfills particular medical objectives may require the availability of more medical knowledge than is actually specified in a medical guideline, i.e., *background knowledge* is required.

Table 1: Used temporal operators; t stands for a time instance.

Notation	**Interpretation**	**Formal semantics**
$\mathsf{H}\varphi$	φ has always been true in the past	$t \vDash \mathsf{H}\varphi \Leftrightarrow \forall t' < t : t' \vDash \varphi$
$\mathsf{G}\varphi$	φ is true now and at all future times	$t \vDash \mathsf{G}\varphi \Leftrightarrow \forall t' \geq t : t' \vDash \varphi$

2.2 Using Temporal Logic for Guideline Representation

As medical management is a time-oriented process, diagnostic and treatment actions described in guidelines are performed in a temporal setting. It has been shown previously that the step-wise, possibly iterative, execution of a guideline, such as the example in Figure 1, can be described precisely by means of temporal logic [MBtTvH02]. This is a modal logic, where relationships between worlds in the usual possible-world semantics of modal logic is understood as time order, i.e., formulae are interpreted in a *temporal structure* $\mathcal{F} = (\mathbb{T}, <, I)$. We will assume that the progression in time is *linear*, i.e., $<$ is a strict linear order. For the representation of the medical knowledge involved it appeared to be sufficient to use rather abstract temporal operators, as proposed in literature [Luc03]. The language of standard logic, with equality and unique names assumption, is augmented with the modal operators G, H, P, and F, where the temporal semantics of the first two operators is defined in Table 1. The last two operators are simply defined in terms of the first two operators:

$$\vDash \mathsf{P}\varphi \leftrightarrow \neg\mathsf{H}\neg\varphi \quad \text{(some time in the past)}$$
$$\vDash \mathsf{F}\varphi \leftrightarrow \neg\mathsf{G}\neg\varphi \quad \text{(some time in the future)}$$

This logic offers the right abstraction level to cope with the nature of the temporal knowledge in medical guidelines required for our purposes. However, more fine-grained temporal operators can be added if needed. For a full axiomatisation of this logic, see Ref. [Tur85].

3 Application to Medical Knowledge

It is assumed that two types of knowledge are involved in detecting the violation of good medical practice:

- Knowledge concerning the (patho)physiological mechanisms underlying the disease, and the way treatment influences these mechanisms. The knowledge involved could be causal in nature, and is an example of *object-knowledge*.
- Knowledge concerning good practice in treatment selection; this is *meta-knowledge*.

Below we present some ideas on how such knowledge may be formalised using temporal logic (cf. [Luc95] for earlier work).

We are interested in the prescription of drugs, taking into account their mode of action. Abstracting from the dynamics of their pharmacokinetics, which are normally modelled using differential equations, this can be formalised in logic as follows:

$$(\mathsf{G}\, d \wedge r) \rightarrow \mathsf{G}(m_1 \wedge \cdots \wedge m_n)$$

where d is the name of a drug or possibly of a group of drugs indicated by a predicate symbol (e.g. SU(x), where x is universally quantified and 'SU' stands for sulfonylurea drugs, such as Tolbutamid), r is a (possibly negative or empty) *requirement* for the drug to take effect, and m_k is a mode of action, such as decrease of release of glucose from the liver, which holds at all future times.

The modes of action m_k can be combined, together with an *intention* n (achieving normoglycaemia, i.e., normal blood glucose levels, for example), a particular patient *condition* c, and *requirements* r_j for the modes of action to be effective:

$$(\mathsf{G}m_{i_1} \wedge \cdots \wedge \mathsf{G}m_{i_m} \wedge r_1 \wedge \cdots \wedge r_p \wedge \mathsf{H}c) \rightarrow \mathsf{G}n$$

In both formulas the antecedent is strong. For example, a drug should *always* be applied to conclude that a certain mode of actions occurs. In a strict sense, this formulation is unrealistic, but the idea is that the time points that the modalities refer to are finite and refers to the relevant information about the patient's disease. This imprecise information is enough to be able to verify a number of quality criteria, which will be shown below.

Good practice medicine can then be formalised as follows. Let $\mathcal{B}$ be background knowledge, $T \subseteq \{d_1, \ldots, d_p\}$ be a set of drugs, C a collection of patient conditions, R a collection of requirements, and N a collection of intentions which the physician has to achieve. A set of drugs T is a *treatment* according to the theory of abductive reasoning if [Poo90, Luc97]:

(M1) $\mathcal{B} \cup \mathsf{G}T \cup C \cup R \not\models \bot$ (the drugs do not have contradictory effects), and

(M2) $\mathcal{B} \cup \mathsf{G}T \cup C \cup R \models N$ (the drugs handle all the patient problems intended to be managed)

If in addition to (1) and (2) condition

(M3) $O_\varphi(T)$ holds, where O_φ is a meta-predicate standing for an optimality criterion or combination of optimality criteria φ,

then the treatment is said to be *in accordance with good-practice medicine*. A typical example of this is subset minimality $O_\subset$:

$$O_\subset(T) \equiv \forall T' \subset T : T' \text{ is not a treatment according to (1) and (2)}$$

i.e., the minimum number of effective drugs are being prescribed. For example, if $\{d_1, d_2, d_3\}$ is a treatment that satisfies condition (3) in addition to (1) and (2), then the subsets $\{d_1, d_2\}$, $\{d_2, d_3\}$, $\{d_1\}$, and so on, do not satisfy conditions (1) and (2). In the context of abductive reasoning, subset minimality is often used in order to distinguish between various solutions; it is also referred to in literature as *Occam's razor*. Another definition of the meta-predicate O_φ is in terms of minimal cost O_c:

$$O_c(T) \equiv \forall T', \text{with } T' \text{ a treatment: } c(T') \geq c(T)$$

where $c(T) = \sum_{d \in T} \mathit{cost}(d)$; combining the two definitions also makes sense. For example, one could come up with a definition of $O_{\subset,c}$ that among two subset-minimal treatments selects the one that is the cheapest in financial or ethical sense.

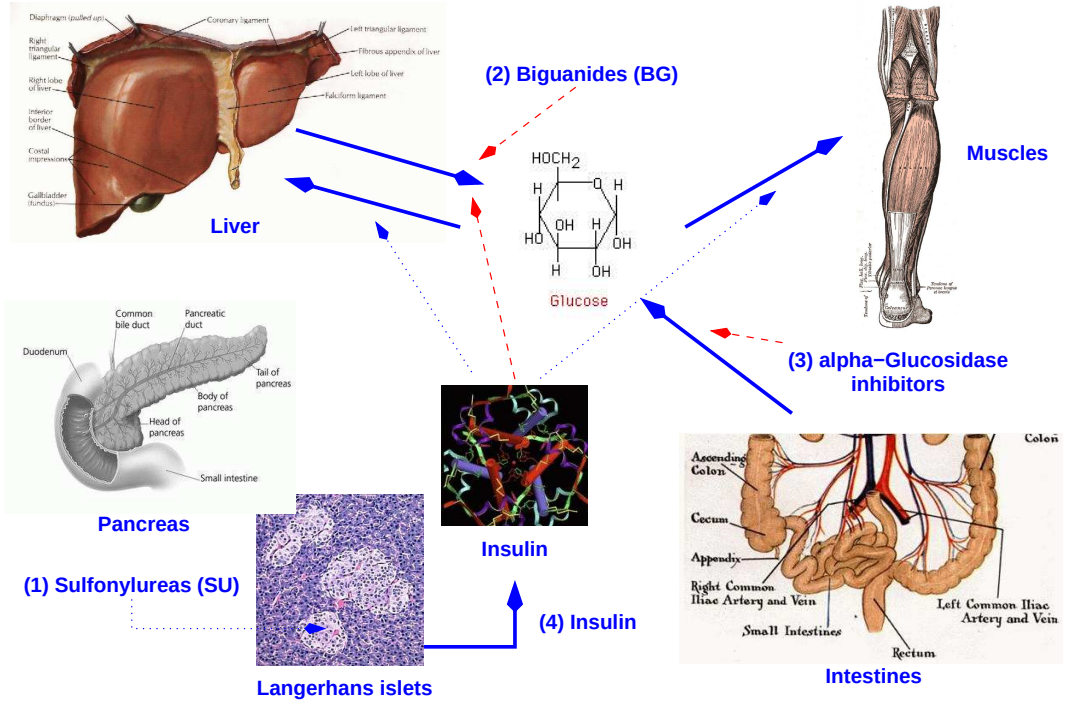


Figure 2: Summary of drugs and mechanisms controlling the blood level of glucose; − − →: inhibition, · · · · · →: stimulation.

4 Management of Diabetes Mellitus Type 2

4.1 Diabetes Type 2 Background Knowledge

It is well known that diabetes type 2 is a very complicated disease. Here we focus on the derangement of glucose metabolism in diabetic patients; however, even that is nontrivial. To support non-expert medical doctors in the management of this complicated disease in patients, access to a guideline is really essential.

One would expect that as this disorder is so complicated, the diabetes mellitus type 2 guideline is also complicated. This, however, is not the case, as may already be apparent from the guideline fragment shown in Figure 1. This indicates that much of the knowledge concerning diabetes mellitus type 2 is missing from the guideline, and that without this background knowledge it will be impossible to spot the sort of flaws we are after. Thus, the conclusion is that a deeper biological analysis is required, the results of which are presented below.

Figure 2 summarises the most important mechanisms and drugs involved in the control of the blood level of glucose. The protein hormone insulin, which is produced by the *B cells* in the Langerhans islets of the *pancreas*, has the following major effects:

- it increases the uptake of glucose by the liver, where it is stored as glycogen, and inhibits the release of glucose from the liver;
- it increases the uptake of glucose by insulin-dependent tissues, such as muscle and adipose tissue.

At some stage in the natural history of diabetes mellitus type 2, the level of glucose in the blood is too high (hyperglycaemia) due to the decreased production of insulin by the B cells.

Treatment of diabetes type 2 consists of:

- Use of *sulfonylurea* (SU) drugs, such as tolbutamid. These drugs stimulate the B cells in producing more insulin, and if the cells are not completely exhausted, the hyperglycaemia can thus be reverted to normoglycaemia (normal blood glucose levels).
- Use of *biguanides* (BG), such as metformin. These drugs inhibit the release of glucose from the liver.
- Use of *α-glucosidase inhibitors*. These drugs inhibit (or delay) the absorption of glucose from the intestines. We omit considering these drugs in the following, as they are only prescribed when treatment side-effects occur.
- Injection of *insulin*. This is the ultimate, causal treatment.

The background knowledge concerning the (patho)physiology of the glucose metabolism as summarised above is formalised using temporal logic, and kept as simple as possible. The specification is denoted by $\mathcal{B}_{\mathrm{DM2}}$:

(1) $\mathsf{G}\,\mathrm{Drug}(\mathit{insulin}) \rightarrow \mathsf{G}(\mathit{uptake}(\mathit{liver}, \mathit{glucose}) = \mathit{up} \wedge \mathit{uptake}(\mathit{peripheral\text{-}tissues}, \mathit{glucose}) = \mathit{up})$

(2) $\mathsf{G}(\mathit{uptake}(\mathit{liver}, \mathit{glucose}) = \mathit{up} \rightarrow \mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down})$

(3) $(\mathsf{G}\,\mathrm{Drug}(\mathrm{SU}) \wedge \neg \mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{exhausted}) \rightarrow \mathsf{G}\mathit{secretion}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{up}$

(4) $\mathsf{G}\,\mathrm{Drug}(\mathrm{BG}) \rightarrow \mathsf{G}\mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down}$

(5) $(\mathsf{G}\mathit{secretion}(\mathit{B\text{-}cell}, \mathit{insulin}) = \mathit{up} \wedge \mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{subnormal} \wedge \mathrm{QI} \leq 27 \wedge \mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})) \rightarrow \mathsf{G}\,\mathrm{Condition}(\mathit{normoglycaemia})$

(6) $(\mathsf{G}\mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down} \wedge \mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{subnormal} \wedge \mathrm{QI} > 27 \wedge \mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})) \rightarrow \mathsf{G}\,\mathrm{Condition}(\mathit{normoglycaemia})$

(7) $((\mathsf{G}\mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down} \vee \mathsf{G}\mathit{uptake}(\mathit{peripheral\text{-}tissues}, \mathit{glucose}) = \mathit{up}) \wedge \mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{nearly\text{-}exhausted} \wedge \mathsf{G}\mathit{secretion}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{up} \wedge \mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})) \rightarrow \mathsf{G}\,\mathrm{Condition}(\mathit{normoglycaemia})$

(8) $(\mathsf{G}\mathit{uptake}(\mathit{liver}, \mathit{glucose}) = \mathit{up} \wedge \mathsf{G}\mathit{uptake}(\mathit{peripheral\text{-}tissues}, \mathit{glucose}) = \mathit{up}) \wedge$

$$\begin{aligned}&\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{exhausted} \wedge \\ &\mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})) \\ &\rightarrow \mathsf{G}(\mathrm{Condition}(\mathit{normoglycaemia}) \vee \mathrm{Condition}(\mathit{hypoglycaemia}))\end{aligned}$$

(9) $(\mathrm{Condition}(\mathit{normoglycaemia}) \oplus \mathrm{Condition}(\mathit{hypoglycaemia}) \oplus \mathrm{Condition}(\mathit{hyperglycaemia})) \wedge \neg(\mathrm{Condition}(\mathit{normoglycaemia}) \wedge \mathrm{Condition}(\mathit{hypoglycaemia}) \wedge \mathrm{Condition}(\mathit{hyperglycaemia}))$

where $\oplus$ stands for the exclusive OR. Note that when the B-cells are exhausted, increased uptake of glucose by the tissues may result not only in normoglycaemia but also in hypoglycaemia (something not mentioned in the guideline).

4.2 Quality Check

As insulin can only be administered by injection, in contrast to the other drugs which are normally taken orally, doctors prefer to delay prescribing insulin as long as possible. Thus, the treatment part of the diabetes type 2 guideline mentions that one should start with prescribing oral antidiabetics (SU or BG, cf. Figure 1). Two of these can also be combined if taking only one has insufficient glucose-level lowering effect. If treatment is still unsatisfactory, the guideline suggests to: (1) either add insulin, or (2) stop with the oral antidiabetics entirely and to start with insulin.

The consequences of various treatment options were examined using the method introduced in Section 3. Hypothetical patients for whom it is the intention to reach a normal level of glucose in the blood (normoglycaemia) are considered, and treatment is selected according to the guideline fragments given in Figure 1:

- Consider a patient with hyperglycaemia due to nearly exhausted B-cells:

$$\begin{aligned}&\mathcal{B}_{\mathrm{DM2}} \cup \mathsf{G}\,T \cup \{\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{nearly\text{-}exhausted}\} \cup \\ &\quad \{\mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})\} \models \mathsf{G}\,\mathrm{Condition}(\mathit{normoglycaemia})\end{aligned}$$

  holds for $T = \{\mathrm{Drug(SU)}, \mathrm{Drug(BG)}\}$, which also satisfies the minimality condition $O_{\subset}(T)$.

- Prescription of treatment $T = \{\mathrm{Drug(SU)}, \mathrm{Drug(BG)}, \mathrm{Drug(insulin)}\}$ for a patient with exhausted B-cells, as is suggested by the guideline, yields:

$$\begin{aligned}&\mathcal{B}_{\mathrm{DM2}} \cup \mathsf{G}\,T \cup \{\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{exhausted}\} \cup \\ &\quad \{\mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})\} \models \\ &\qquad \mathsf{G}(\mathrm{Condition}(\mathit{normoglycaemia}) \vee \mathrm{Condition}(\mathit{hypoglycaemia}))\end{aligned}$$

In the last case, it appears that it is possible that a patient develops hypoglycaemia due to treatment; if this possibility is excluded from axiom (8) in the background knowledge, then the minimality condition $O_{\subset}(T)$, and also $O_{\subset,c}(T)$, do not hold since insulin by itself is enough to reach normoglycaemia. In either case, good practice medicine is violated, which is to prescribe as few drugs as possible, taking into account costs and side-effects of drugs. Here, three drugs are prescribed whereas only two should have been prescribed (BG and insulin, assuming that insulin alone is too costly), and the possible occurrence of hypoglycaemia should have been prevented.

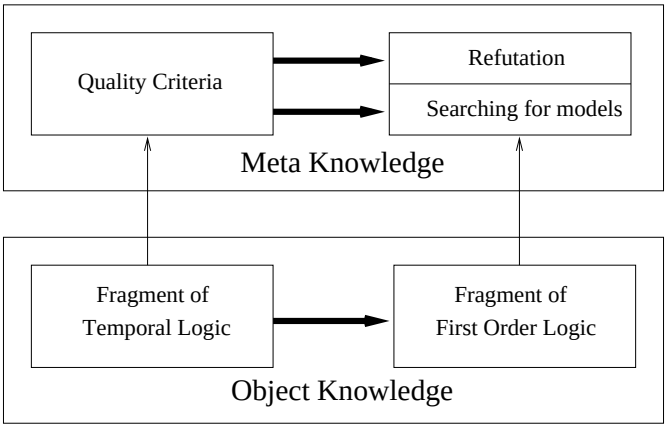


Figure 3: Translation of medical knowledge.

5 Automated Proving of Quality Requirements

As said in the introduction, we have explored the feasibility of using the automated reasoning tools OTTER and MACE-2 to check the quality of guidelines, in the sense as described above.

5.1 Motivation for the Theorem Proving Facilities

One of the most important application areas of model finders and theorem provers is program verification. Of course, with programs there is a clear beginning of the execution, which makes it intuitive to think about properties that occur after the start of the program. Therefore, it is not surprising much work has been done in the context of model finding and theorem proving with only the future time modality. However, it is more natural to model medical knowledge with past time operators, i.e., what happened to the patient in the past. It is well-known that formulas with a past-time modality can be mapped to a logical formula with only future time modalities such that both formulas are equivalent for some initial state [Gab89]. However, the main drawback to this approach is the fact that formulas will get much larger in size [Mar03] and as a consequence become much harder to verify in a theorem prover designed for modal logics.

For this reason, we have chosen to use an alternative approach which uses a *relational translation* to map the temporal logic formulas to first-order logic. As primary tools we use the resolution-based theorem prover OTTER and the finite model searcher MACE-2, which take first-order logic with equality as their input. There has been work done to improve the speed of resolution-based theorem provers on modal formulas [AGHdR00], but again, converse modalities such as the past-time operators are not considered. Nonetheless, we found that the general heuristics applicable to full first order logic are sufficient for our task

To clarify our approach, see Figure 3. We will first give a definition for translating the object knowledge to standard logic and then the translation of the meta-level knowledge will follow.

5.2 Translation

5.2.1 Translation of Object Knowledge

We assume that the formalisation is in propositional temporal logic. We do this by introducing a fresh proposition p for every equation that we find in the background knowledge. For functions with two elements in the co-domain, we have p and $\neg p$ and for the *capacity* function with three elements in its co-domain, we add a proposition p_x for each atom $\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = x$ and the appropriate axiomatisation such that exactly one p_x holds. Technically this is not required, since we could extend the translation below to full first-order temporal logic. In practice however, we would like to avoid additional complexity from first-order formulas during the automated reasoning.

The relational translation (e.g., [Moo79, AGHdR00, SH03]) $\mathsf{ST}_t(\varphi)$, also referred to as standard translation, translates a propositional temporal logical formula φ into a formula in a first-order logic with a (time-indexed) unary predicate symbols P for every propositional variable p and one binary predicate $>$. It is defined as follows, where t is an individual variable:

$$
\begin{array}{lcl}
\mathsf{ST}_t(p) & \Leftrightarrow & P(t) \\
\mathsf{ST}_t(\neg\varphi) & \Leftrightarrow & \neg\mathsf{ST}_t(\varphi) \\
\mathsf{ST}_t(\varphi \wedge \psi) & \Leftrightarrow & \mathsf{ST}_t(\varphi) \wedge \mathsf{ST}_t(\psi) \\
\mathsf{ST}_t(\mathsf{G}\varphi) & \Leftrightarrow & \forall_{t'}\colon (t \not> t' \rightarrow \mathsf{ST}_{t'}(\varphi)) \\
\mathsf{ST}_t(\mathsf{H}\varphi) & \Leftrightarrow & \forall_{t'}\colon (t > t' \rightarrow \mathsf{ST}_{t'}(\varphi))
\end{array}
$$

Note that in our notation $\cup$ is sometimes used instead of a conjunction, so $\mathsf{ST}_t(\Gamma \cup \Delta)$ is defined as $\mathsf{ST}_t(\Gamma) \cup \mathsf{ST}_t(\Delta)$. Note that the last two elements of the definition define the meaning of the G modality and its converse, the H modality. For example, the formula $\mathsf{G}(p \rightarrow \mathsf{P}p)$ translates to $\forall_{t_2}(t \not> t_2 \rightarrow (P(t_2) \rightarrow \exists_{t_3}(t_2 > t_3 \wedge P(t_3))))$. It is easy to show that a formula in temporal logic is satisfiable if and only if its relational translation is.

In the literature a functional approach to translating modal logic has appeared as well [Ohl88], which relies on a non-standard interpretation of modal logic and could be taken as an alternative to this translation.

5.2.2 Translation of Meta-level Knowledge

Again, we consider the criteria for good practice medicine and make them suitable for the automated reasoning tools. We say that a treatment T is a treatment complying with requirements of good practice medicine iff:

(M1′) $\mathsf{ST}_t(\mathcal{B} \cup \mathsf{G}T \cup C \cup R) \nvdash \bot$

(M2′) $\mathsf{ST}_t(\mathcal{B} \cup \mathsf{G}T \cup C \cup R \cup \neg N) \vdash \bot$

(M3′) $\forall T' \subset T : T'$ is not a treatment according to (1) and (2)

It is easy to see that, because the relational translation preserves satisfiability, these quality requirements are equivalent to their unprimed counterparts. To automate this reasoning process we use MACE-2 to verify **(M1′)**, OTTER to verify **(M2′)**, and **(M3′)** can be seen as a combination of both for all subsets of the given treatment.

5.3 Proofs

In this subsection we will discuss the actual implementation in OTTER [McC03] and some results by using various heuristics.

5.3.1 Resolution Strategies

The main advantage that one gains from using a standard theorem prover is the fact that a whole range of different resolution rules are available. Note that Otter uses the set-of-support strategy [WRC65] as a standard strategy. With this strategy the original set of clauses is divided into a *set-of-support* and a *usable* set such that in every resolution step at least one of the parent clauses has to be member of the set-of-support and each resulting resolvent is added to the set-of-support.

Looking at the structure of the formulas in Section 4, one can see that formulas are of the type $p_0 \wedge \ldots \wedge p_n \rightarrow q$, where $p_0 \wedge \ldots \wedge p_n$ and q are all positive literals. Hence, we expect mostly negative literals in our clauses, which was exploited by using negative hyperresolution in OTTER. With this strategy a clause with at least one positive literal is resolved with a number of clauses which only contain negative literals (i.e., negative clauses), provided that the resolvent is a negative clause. The parent clause with at least one positive literal is called the *nucleus*, and the other, negative, clauses are referred to as the *satellites*. Positive hyperresolution, which uses positive satellites and a nucleus with at least one negative literal, was also tried. However, this did not result in successful proofs, because the background knowledge contains few positive clauses.

5.3.2 Verification

The ordering predicate $>$ that was introduced in Section 5.2.1 was defined by anti-reflexivity, anti-symmetry, and transitivity. We did not find any cases where the axiom of transitivity was required to construct the proof, which can be explained by the low modal depth of our formulas. As a consequence, the axiom was omitted with the aim to improve the speed of theorem proving.

We used OTTER to perform the two proofs which are instantiations of **(M2′)**. First we, again, consider a patient with hyperglycaemia due to nearly exhausted B-cells and prove:

$$\begin{aligned}&\mathsf{ST}_0(\mathcal{B}_{\mathrm{DM2}} \cup \mathsf{G}\,T \cup \{\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{nearly\text{-}exhausted}\} \cup \\ &\quad \{\mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})\} \cup \{\neg\mathsf{G}\,\mathrm{Condition}(\mathit{normoglycaemia})\}) \vdash \bot\end{aligned}$$

where $T = \{\mathrm{Drug(SU)}, \mathrm{Drug(BG)}\}$,

This property was proven with OTTER in 62 resolution steps with the use of the negative hyperresolution strategy. As an example, we present a small snippet from the proof of this property. We will use the same syntax as we used in the previous sections, but each literal is augmented with a time-index. Note that $g(x, y) = \mathit{down}$ is implemented as a negative literal and functions f_1 and f_2 are Skolem functions introduced by OTTER. Both Skolem functions map a time point to a later time points. Consider the following clauses in the usable and set-of-support list. For example, assumption (53) models the capacity of the B-cells, i.e., nearly exhausted at time $t = 0$ where the property as shown above should be refuted:

2 $capacity(B\text{-}cells, insulin, t_0) \neq nearly\text{-}exhausted \vee$
$capacity(B\text{-}cells, insulin, t_0) \neq exhausted$

14 $t_0 \not> f_1(t_0) \vee capacity(B\text{-}cells, insulin, t_0) = exhausted \vee t_0 > t_1 \vee$
$secretion(B\text{-}cells, insulin, t_1) = up$

15 $\neg\text{Drug(SU)} \vee capacity(B\text{-}cells, insulin, t_0) = exhausted \vee t_0 > t_1 \vee$
$secretion(B\text{-}cells, insulin, t_1) = up\}$

51 $0 > t_0 \vee \text{Drug}(\text{SU}, t_0)$

53 $capacity(B\text{-}cells, insulin, 0) = nearly\text{-}exhausted$

Very early in the proof, OTTER deduces that if the capacity of insulin in B-cells is nearly-exhausted, then it is not completely exhausted:

56 [neg_hyper,53,2] $capacity(B\text{-}cells, insulin, 0) \neq exhausted$

Now we skip a part of the proof, which results in information about the relation between the capacity of insulin and the secretion of insulin in B-cells for a certain time point:

517 [neg_hyper,516,53] $0 \not> f_2(0)$

765 [neg_hyper,761,50,675] $capacity(B\text{-}cells, insulin, f_2(0)) \neq nearly\text{-}exhausted \vee$
$secretion(B\text{-}cells, insulin, f_2(0)) = down$

This information allows OTTER to quickly complete the proof, by combining it with the information about the effects of a sulfonylurea drug:

766 [neg_hyper,765,15,56,517] $capacity(B\text{-}cells, insulin, f_1(0)) \neq nearly\text{-}exhausted \vee$
$\neg\text{Drug(SU)}$

767 [neg_hyper,765,14,56,517] $capacity(B\text{-}cells, insulin, f_1(0)) \neq nearly\text{-}exhausted \vee$
$0 \not> f_1(0)$

after which (53) can be used as a nucleus to yield:

768 [neg_hyper,767,53] $0 \not> f_1(0)$

and consequently by taking (51) as a nucleus, we find that at time point 0 the capacity of insulin is not nearly exhausted:

769 [neg_hyper,768,51,766] $capacity(B\text{-}cells, insulin, 0) \neq nearly\text{-}exhausted$

This directly contradicts one of the assumptions and this results in an empty clause:

770 [binary,769.1,53.1] $\bot$

Similarly, we could prove that given a treatment $T = \{\text{Drug(SU)}, \text{Drug(BG)}, \text{Drug(insulin)}\}$ for a patient with exhausted B-cells, as is suggested by the guideline, it follows that:

$$\mathsf{ST}_0(\mathcal{B}_{\text{DM2}} \cup \mathsf{G}\,T \cup \{capacity(B\text{-}cells, insulin) = exhausted\} \cup$$
$$\{\mathsf{HCondition}(hyperglycaemia)\} \cup$$
$$\{\neg(\mathsf{G}(\text{Condition}(normoglycaemia) \vee \text{Condition}(hypoglycaemia)))\}) \vdash \bot$$

The proof of OTTER is omitted, but a similar magnitude of complexity in the proof can be observed, i.e., 52 resolution steps.

Weights	Clauses (binary res)	Clauses (negative hyper res)
$(0, 1)$	17729	6994
$(1, 0)$	13255	6805
$(1, 1)$	39444	7001
$(1, -1)$	13907	6836
$(2, -2)$	40548	7001
$(2, -3)$	16606	6805
$(3, -4)$	40356	7095
$(3, -5)$	27478	7001

Figure 4: Generated clauses to prove an instance of property **M2′** depending on weights (x, y) for the ordering relation on time.

5.3.3 Weighing the Clauses

In this section we consider the weighing facilities as offered by OTTER to improve the performance. Consider the example from [AGHdR00]. Suppose we have the formula $\mathsf{G}(p \rightarrow \mathsf{F}p)$. Proving this satisfiable amounts to proving that the following two clauses are satisfiable:

1. $c > t_1 \vee \neg P(t_1) \vee t_1 \not> f(t_1)$
2. $c > t_2 \vee \neg P(t_2) \vee P(f(t_2))$

The observation can be made, that although we have two possibilities to resolve these two clauses, for example on the P literal, this is useless because the negative P literal is only bound by the G-operator while the positive P literal comes from a formula at a deeper modal depth under the F-operator. Suppose we resolve these two P literals, which generates the clause:

$$c > f(t) \vee f(t) \not> f(f(t)) \vee c > t \vee \neg P(t)$$

and with (2) again we have:

$$c > f(f(t)) \vee f(f(t)) \not> f(f(f(t))) \vee c > f(t) \vee c > t \vee \neg P(t)$$

etc.

So we can see that we can generate a lot of new clauses, but clearly these nestings of the Skolem functions will not lead to a contradiction very quickly if the depth of the modalities in the formulas that we have is small.

In OTTER the weight of the clauses determines which clauses are chosen from the set-of-support and usable list to become parents in a resolution step. Clearly, because the goal of resolution is to find an empty clause, lighter clauses are preferred. By default, the weight of a clause is the sum of all occuring symbols (i.e., all symbols have weight 1), but as we have argued, the nesting of Skolem functions will not help to find such an empty clauses. Therefore it can be of use to manually change the weight of the ordering symbol, which is done in OTTER by a tuple (x, y) for each predicate, where x is multiplied with sum of the weight of its arguments and is added to y to calculate the new weight

```
> :                    Condition(hyperglycaemia) :
    | 0 1                          0 1
  --+----                        -------
  0 | F T                          T T
  1 | F F
```

Figure 5: Snippet from a MACE-2 generated model

of this predicate. For example, if $x = 2$ and $y = -3$, then $x > y$ has a total weight of $2+2-3 = 1$, while $f(f(x)) > f(y)$ has a weight of $2*3+2*2-3 = 7$. See Figure 4 where we show results when we applied this for some small values for x and y for both binary and negative hyperresolution. What these numbers tend to show (similar numbers were gained from the other property) is that the total weight of the ordering predicate should be smaller than the average weight of other, unary, predicates. Nonetheless, possibly somewhat suprisingly, the factor x should not be raised too much, although in the case of a negative hyperresolution strategy the effect is minimal. Furthermore, we can see that combining the resolution strategies with a weighing strategy does help, but the advantages are rather limited compared to the advantages of weighing in combination with binary resolution.

5.4 Disproofs

MACE-2 (Models And CounterExamples) [McC01] is a program that searches for small finite models of first-order statements using a David-Putman-Loveland-Logemann decision procedure [DP69, DLL62] as its core. Because of the relative simplicity of our temporal formulas, it is to be expected that counterexamples can be found with very few states. Hence, it can be expected that models are in the same magnitude as the propositional case and this is indeed the case. In fact, the countermodels that MACE-2 found consist of only 2 elements in the domain of the model.

The first property we check corresponds to checking if the background knowledge augmented with a patient and a therapy is consistent, i.e., criterium **(M1′)**. So, again consider a patient with hyperglycaemia due to nearly exhausted B-cells. We have used MACE-2 to verify:

$$\mathsf{ST}_0(\mathcal{B}_{\mathrm{DM2}} \cup \mathsf{G}\ T \cup \{\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{exhausted}\} \cup \{\mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})\}) \nvdash \bot$$

for $T = \{\mathrm{Drug(SU)}, \mathrm{Drug(BG)}, \mathrm{Drug(insulin)}\}$. From this, of course, it follows that there is a model if $T = \{\mathrm{Drug(SU)}, \mathrm{Drug(BG)}\}$ and consequently we have verified **(M1′)**.

Similarly, we find that for all $T \subset \{\mathrm{Drug(SU)}, \mathrm{Drug(BG)}\}$, it holds that:

$$\mathsf{ST}_0(\mathcal{B}_{\mathrm{DM2}} \cup \mathsf{G}\,T \cup \{\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{nearly\text{-}exhausted}\} \cup \{\mathsf{H}\,\mathrm{Condition}(\mathit{hyperglycaemia})\} \cup \{\neg\mathsf{G}\,\mathrm{Condition}(\mathit{normoglycaemia})\}) \nvdash \bot$$

So indeed the conclusion is that the treatment complies with **(M3′)** and thus complies with the criteria of good practice medicine. See for example Figure 5, which

contains a small sample of the output that MACE-2 generated. The output is a first-order model with two elements in the domain named '0' and '1' with an interpretation of all predicates and functions to this domain. It shows that it is consistent with the background knowledge to believe that the patient will continue to suffer from hyperglycaemia if one of the drugs is not applied. It is interesting to see that there is also a smaller model where the size of the domain is 1 for this set of formulas.

Finally, consider the treatment $T = \{\text{Drug(SU)}, \text{Drug(BG)}, \text{Drug(insulin)}\}$ for a patient with exhausted B-cells, and suppose we exclude the patient developing hypoglycaemia, we can show that:

$$\begin{aligned}&\mathsf{ST}_0(\mathcal{B}_{\text{DM2}} \cup \mathsf{G}\,T \cup \{\mathit{capacity}(\mathit{B\text{-}cells}, \mathit{insulin}) = \mathit{exhausted}\} \cup \\ &\quad \{\mathsf{H}\text{Condition}(\mathit{hyperglycaemia})\} \cup \\ &\quad \{\neg\mathsf{G}(\text{Condition}(\mathit{hyperglycaemia})\} \cup \{\neg\mathsf{G}(\text{Condition}(\mathit{hypoglycaemia}))\}) \nvdash \bot\end{aligned}$$

But, it is possible to prove the same property if $T = \{\text{Drug(insulin)}\}$ and thus **(M3′)** does not hold in this case and as a consequence the guideline does not comply with the quality requirements as discussed in the previous section.

6 Discussion

The quality of guideline design is for the largest part based on its compliance with specific treatment aims and global requirements. To this purpose, use was made of the theory of abductive, diagnostic reasoning, i.e., we diagnosed potential problems with a guideline using logical abduction [Luc97, Luc03, Poo90]. This is a meta-level characterisation of the quality of a medical guideline. What was diagnosed were problems in the relationship between medical knowledge, suggested treatment actions in the guideline text and treatment effects; this is different from traditional abductive diagnosis, where observed findings are explained in terms of diagnostic hypotheses. This method allows us to examine fragments of a guideline and to prove properties of those fragments.

In earlier work [HLB04], where we used a tool for interactive program verification named KIV [Rei95], we performed a similar exercise. The main advantage of using interactive theorem proving is that the resulting proof is relatively elegant compared to automated resolution-based solutions. This might be important if one wants to convince the medical community that a guideline complies with their medical quality requirements and to promote the implementation of such a guideline. However, to support the *design* of guidelines, this argument is of less importance. Moreover, the work that needs to be done to construct a proof in an interactive theorem prover would severely slow down the development process as people with specialised knowledge are required.

Even though guideline developers might not be interested in inspecting the full proof or disproof of a certain property, it is of importance for the process that if a certain proof fails, they have a method to find out *why* the proof failed. Thus in our future work we will focus on the question whether it is possible to give hints to guideline developers on how to improve their guidelines. Furthermore, our quality requirements are far from exhaustive and the last few years research has been done in this field (e.g. [FAB+04]). Our aim will be to extend our current work with these new insights.

In this paper, we have made use of tools designed for automated reasoning to actually quality check a medical guideline using the theory of quality of guidelines developed previously [Luc03]. This complements both the earlier work on object-level verification of medical guidelines using the interactive theorem prover designed for program verification KIV [MBtTvH02], but also our earlier work where we used KIV for meta-level reasoning [HLB04].

References

[AGHdR00] C. Areces, R. Gennari, J. Heguiabehere, and M. de Rijke. Tree-based Heuristics in Modal Theorem Proving. In *Proceedings of the ECAI'2000*, Berlin, Germany, 2000.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP69] M. Davis and H. Putman. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1969.

[FAB+04] J. Fox, A. Alabassi, E. Black, C. Hurt, and T. Rose. Modelling Clinical Guidelines: a Corpus of Examples and a Tentative Ontology. In K. Kaiser, S. Miksch, and S.W. Yu, editors, *Computer-based Support for Clinical Guidelines and Protocols. Proceedings of the Symposium on Computerized Guidelines and Protocols (CGP 2004)*, volume 101 of *Studies in Health Technology and Informatics*, pages 31–45, Amsterdam, 2004. IOS Press.

[FD00] J. Fox and S. Das. *Safe and Sound: Artificial Intelligence in Hazardous Applications*. MIT Press, 2000.

[Gab89] D.M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In H. Barringer, editor, *Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer-Verlag, Berlin, 1989.

[HLB04] A.J. Hommersom, P.J.F. Lucas, and M. Balser. Meta-level Verification of the Quality of Medical Guidelines using Interactive Theorem Proving. In J. J. Alferes and J. Leite, editors, *JELIA'04*, volume 3225 of *LNAI*, pages 654–666, Heidelberg, 2004. Springer-Verlag.

[Luc93] P.J.F. Lucas. The Representation of Medical Reasoning Models in Resolution-based Theorem Provers. *Artificial Intelligence in Medicine*, 5:395–419, 1993.

[Luc95] P.J.F. Lucas. Logic Engineering in Medicine. *The Knowledge Engineering Review*, 10(2):153–179, 1995.

[Luc97] P.J.F. Lucas. Symbolic Diagnosis and its Formalisation. *The Knowledge Engineering Review*, 12(2):109–146, 1997.

[Luc03] P.J.F. Lucas. Quality Checking of Medical Guidelines through Logical Abduction. In F. Coenen, A. Preece, and A.L. Mackintosh, editors, *Proceedings of AI-2003 (Research and Developments in Intelligent Systems XX)*, pages 309–321, London, 2003. Springer.

[Mar03] N. Markey. Temporal Logic with Past is Exponentially More Succinct. *EATCS Bulletin*, 79:122–128, 2003.

[MBtTvH02] M. Marcos, M. Balser, A. ten Teije, and F. van Harmelen. From Informal Knowledge to Formal Logic: a Realistic Case Study in Medical Protocols. In *Proceedings of the 12th EKAW-2002*, 2002.

[McC01] W. McCune. MACE 2.0 Reference Manual and Guide. Tech. Memo ANL/MCS-TM-249, Argonne National Laboratory, Argonne, IL, June 2001.

[McC03] W. McCune. Otter 3.3 Reference Manual. Tech. Memo ANL/MCS-TM-263, Argonne National Laboratory, Argonne, IL, August 2003.

[Moo79] R.C. Moore. *Reasoning about Knowledge and Action.* PhD thesis, MIT, 1979.

[Ohl88] H.J. Ohlbach. A Resolution Calculus for Modal Logics. In E. Lusk and R. Overbeek, editors, *Proceedings CADE-88: International Conference on Auomated Deduction*, volume 310 of *LNCS*, pages 500–516. Springer-Verlag, 1988.

[OMGM98] L. Ohno-Machado, J.H. Gennari, and S.N. Murphy. Guideline Interchange Format: a Model for Representing Guidelines. *Journal of the American Medical Informatics Association*, 5(4):357–372, 1998.

[Poo90] D. Poole. A Methodology for using a Default and Abductive Reasoning System. *International Journal of Intelligent System*, 5(5):521–548, 1990.

[Rei95] W. Reif. The KIV Approach to Sftware Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*. Springer-Verlag, Berlin, 1995.

[Rob65] J.A. Robinson. Automated Deduction with Hyperresolution. *International Journal of Computatational Mathematics*, 1:23–41, 1965.

[SH03] R.A. Schmidt and U. Hustadt. Mechanised Reasoning and Model Generation for Extended Modal Logics. In H.C.M. de Swart, E. Orlowska, G. Schmidt, and M. Roubens, editors, *Theory and Applications of Relational Structures as Knowledge Instrument*, volume 2929 of *LNCS*, pages 38–67. Springer, 2003.

[SMJ98] Y. Shahar, S. Miksch, and P. Johnson. The Asgaard Project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine*, 14:29–51, 1998.

[Tur85] R. Turner. *Logics for Artificial Intelligence*. Ellis Horwood, Chichester, 1985.

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

[Woo00] S.H. Woolf. Evidence-based Medicine and Practice Guidelines: an overview. *Cancer Control*, 7(4):362–367, 2000.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. *ACM Journal*, 12:536–541, October 1965.

What First Order Theorem Provers Do For Monodic Temporal Reasoning

Michael Fisher, Ullrich Hustadt, Boris Konev*, and Alexei Lisitsa
Department of Computer Science
The University of Liverpool, UK
{M.Fisher, U.Hustadt, B.Konev, A.Lisitsa}@csc.liv.ac.uk

Abstract

Monodic temporal logic is the most general fragment of first-order temporal logic for which sound and complete calculi have been developed so far, including resolution calculi. One such resolution calculus has been implemented in the system **TeMP** using the first-order theorem prover **Vampire** as a kernel for performing the main reasoning tasks of the system. In this paper, we describe the calculus underlying **TeMP**, its implementation, and our experiences with its application to the verification of parameterised cache coherence protocols. We present some preliminary observations about the use of a first-order theorem prover as kernel for a reasoning system for a more expressive logic.

1 Introduction

In [KDD+05] we have developed a sound and complete resolution calculus for *monodic temporal logic*—the most general known complete fragment of first-order temporal logic [HWZ00]. The calculus has been implemented in the system **TeMP** [HKRV04]. A distinctive feature of our calculus is the possibility to implement its inference rules using first-order ordered resolution. For **TeMP** we currently use **Vampire** [RV02] for this purpose. In this paper we describe our experiments with **TeMP** on a class of verification problems. While these verification efforts are interesting *per se*, they also allow us to study the behaviour of **Vampire** on non-standard problems—in the majority of cases the set of clauses fed to the **Vampire** kernel is satisfiable, and we are interested in this set being saturated. We believe that the results of our experiments and experience in complex temporal applications can encourage further research in the area of first-order provers.

The structure of this paper is as follows. In Section 2 we recall the syntax and semantics of first-order temporal logic over a linear flow of time, the definition of the monodic fragment of first-order temporal logic and of the normal form that we use in our approach. In Section 3 we describe the ordered monodic fine-grained temporal resolution calculus. Section 4 presents our approach to the implementation of this calculus in the system **TeMP** including the use that we make of **Vampire**. Section 5 demonstrates an application of **TeMP** to a particular class of verification problem, namely the verification of parameterised protocols, and states some observations we made during our experiments concerning the use of a first-order theorem prover as kernel of a theorem prover for a more expressive logic.

*On leave from Steklov Institute of Mathematics at St.Petersburg

2 First-order temporal logic

First-Order Temporal Logic, FOTL, is an extension of classical first-order logic by temporal operators for a discrete linear model of time (isomorphic to $\mathbb{N}$, that is, the most commonly used model of time). The signature of FOTL (without equality and function symbols) consists of a countably infinite set of *variables* $x_0, x_1, \ldots$, a countably infinite set of *constants* $c_0, c_1, \ldots$, a non-empty set of *predicate symbols* $P, P_0, \ldots$, each with a fixed arity ≥ 0, the *propositional operators* $\top$, $\neg$, $\vee$, the *quantifiers* $\exists x_i$ and $\forall x_i$, and the *temporal operators* $\Box$ ('always in the future'), $\Diamond$ ('eventually in the future'), $\bigcirc$ ('at the next moment'), and U ('until'). The set of formulae of FOTL is defined as follows: $\top$ is a FOTL formula; if P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are variables or constants, then $P(t_1, \ldots, t_n)$ is an *atomic* FOTL formula; if φ and ψ are FOTL formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\exists x\varphi$, $\forall x\varphi$, $\Box\varphi$, $\Diamond\varphi$, $\bigcirc\varphi$, and $\varphi \,\mathsf{U}\, \psi$. We also use $\bot$, $\wedge$, and $\Rightarrow$ as additional operators defined using $\top$, $\neg$, and $\vee$. Free and bound variables of a formula are defined in the standard way, as well as the notions of open and closed formulae. Given a formula φ, we write $\varphi(x_1, \ldots, x_n)$ to indicate that all the free variables of φ are among $x_1, \ldots, x_n$. As usual, a *literal* is either an atomic formula or its negation.

Formulae of this logic are interpreted over structures $\mathfrak{M} = (D_n, I_n)_{n\in\mathbb{N}}$ that associate with each element n of $\mathbb{N}$, representing a moment in time, a first-order structure $\mathfrak{M}_n = (D_n, I_n)$ with its own non-empty domain D_n and interpretation I_n. An *assignment* $\mathfrak{a}$ is a function from the set of variables to $\bigcup_{n\in\mathbb{N}} D_n$. The application of an assignment to terms is defined in the standard way, in particular, $\mathfrak{a}(c) = c$ for every constant c. The *truth relation* $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$ is defined, only for those $\mathfrak{a}$ such that $\mathfrak{a}(x) \in D_n$ for every variable x, as follows:

$\mathfrak{M}_n \models^{\mathfrak{a}} \top$	
$\mathfrak{M}_n \models^{\mathfrak{a}} P(t_1, \ldots, t_n)$	iff $(I_n(\mathfrak{a}(t_1)), \ldots, I_n(\mathfrak{a}(t_n))) \in I_n(P)$
$\mathfrak{M}_n \models^{\mathfrak{a}} \neg\varphi$	iff not $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \vee \psi$	iff $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$ or $\mathfrak{M}_n \models^{\mathfrak{a}} \psi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \exists x\varphi$	iff $\mathfrak{M}_n \models^{\mathfrak{b}} \varphi$ for some assignment $\mathfrak{b}$ that may differ from $\mathfrak{a}$ only in x and such that $\mathfrak{b}(x) \in D_n$
$\mathfrak{M}_n \models^{\mathfrak{a}} \forall x\varphi$	iff $\mathfrak{M}_n \models^{\mathfrak{b}} \varphi$ for every assignment $\mathfrak{b}$ that may differ from $\mathfrak{a}$ only in x and such that $\mathfrak{b}(x) \in D_n$
$\mathfrak{M}_n \models^{\mathfrak{a}} \bigcirc\varphi$	iff $\mathfrak{M}_{n+1} \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \Diamond\varphi$	iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \Box\varphi$	iff for all $m \geq n$, $\mathfrak{M}_m \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \,\mathsf{U}\, \psi$	iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \psi$ and $\mathfrak{M}_i \models^{\mathfrak{a}} \varphi$ for every $i, n \leq i < m$

In this paper we make the *expanding domain assumption*, that is, $D_n \subseteq D_m$ if $n < m$, and we assume that the interpretation of constants is *rigid*, that is, $I_n(c) = I_m(c)$ for all $n, m \in \mathbb{N}$.

The set of valid formulae of this logic is not recursively enumerable. However, the set of valid *monodic* formulae is known to be finitely axiomatisable [WZ02]. A formula φ of FOTL is called *monodic* if any subformula of φ of the form $\bigcirc\psi$, $\Box\psi$, $\Diamond\psi$, or $\psi_1 \,\mathsf{U}\, \psi_2$ contains at most one free variable. For example, the formulae $\forall x \Box \exists y P(x, y)$ and $\forall x \Box P(x, c)$ are monodic, while $\forall x \forall y (P(x, y) \Rightarrow \Box P(x, y))$ is not monodic.

Every monodic temporal formula can be transformed into an equi-satisfiable normal form, called *divided separated normal form (DSNF)* [KDD$^+$05].

Definition 1 *A* monodic temporal problem P in divided separated normal form (DSNF) *is a quadruple* $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$*, where:*

1. *the universal part,* $\mathcal{U}$*, and the initial part,* $\mathcal{I}$*, are finite sets of first-order formulae;*
2. *the step part,* $\mathcal{S}$*, is a finite set of* step clauses *of the form* $p \Rightarrow \bigcirc q$*, where* p *and* q *are propositions, and* $P(x) \Rightarrow \bigcirc Q(x)$*, where* P *and* Q *are unary predicate symbols and* x *is a variable; and*
3. *the eventuality part,* $\mathcal{E}$*, is a finite set of formulae of the form* $\Diamond L(x)$ *(a* non-ground eventuality clause*) and* $\Diamond l$ *(a* ground eventuality clause*), where* l *is a propositional literal and* $L(x)$ *is a unary non-ground literal with variable* x *as its only argument.*

With each monodic temporal problem $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ we associate the FOTL formula $\mathcal{I} \wedge \Box \mathcal{U} \wedge \Box \forall x \mathcal{S} \wedge \Box \forall x \mathcal{E}$. When we talk about particular properties of a temporal problem (e.g., satisfiability, validity, logical consequences, etc) we refer to properties of the associated formula. We often refer to a set of formulae as a formula, in this case the formula represents the conjunction of the formulae in the set.

The transformation to DSNF is based on a renaming and unwinding technique which substitutes non-atomic subformulae and replaces temporal operators by their fixed point definitions as described, for example, in [FDP01]. A step in this transformation which is of relevance for the results presented here is the following: We recursively rename each innermost open subformula $\xi(x)$, whose main connective is a temporal operator, by $P_\xi(x)$, where $P_\xi(x)$ is a new unary predicate symbol, and rename each innermost closed subformula ζ, whose main connective is a temporal operator, by p_ζ, where p_ζ is a new propositional variable. In the terminology of [HWZ00] $P_\xi(x)$ and p_ζ are called the *surrogates* of $\xi(x)$ and ζ, respectively. Renaming introduces formulae defining $P_\xi(x)$ and p_ζ of the following form (since we are only interested in satisfiability, we use implications instead of equivalences for renaming positive occurrences of subformulae, see also [NW01]):

$$(a) \quad \Box \forall x (P_\xi(x) \Rightarrow \xi(x)) \qquad \text{and} \qquad (b) \quad \Box (p_\zeta \Rightarrow \zeta).$$

If the main connective of $\xi(x)$ or ζ is either $\Box$ or U, then the formula will be replaced by its fixed point definition. If the main connective of $\xi(x)$ or ζ is either the $\bigcirc$ or $\Diamond$ operator, the defining formula will further be simplified to obtain step or eventuality clauses.

Theorem 1 (see [DFK], Theorem 1) *Any monodic first-order temporal formula can be transformed into an equi-satisfiable monodic temporal problem in DSNF with at most a linear increase in size of the problem.*

In the next section we briefly recall the temporal resolution calculus first developed in [DFK03] and later refined in [HKS05].

3 Monodic fine-grained temporal resolution

Our prover **TeMP** is based on *ordered monodic fine-grained temporal resolution calculus*, ordered fine-grained resolution for short, which we briefly describe in this section.

As is commonly the case for resolution calculi, the calculus operates on a clause normal form, which we obtain by clausifying monodic temporal problems in DSNF normal form.

Definition 2 *Let $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ be a monodic temporal problem. The* clausification $\mathrm{Cls}(\mathsf{P})$ of P *is a quadruple $\langle \mathcal{U}', \mathcal{I}', \mathcal{S}', \mathcal{E}' \rangle$ such that (i) $\mathcal{U}'$ is a set of clauses, called* universal clauses*, obtained by clausification of $\mathcal{U}$; (ii) $\mathcal{I}'$ is a set of clauses, called* initial clauses*, obtained by clausification of $\mathcal{I}$; (iii) $\mathcal{S}'$ is the smallest set of step clauses such that all step clauses from $\mathcal{S}$ are in $\mathcal{S}'$ and for every non-ground step clause $P(x) \Rightarrow \bigcirc L(x)$ in $\mathcal{S}$ and every constant c occurring P, the clause $P(c) \Rightarrow \bigcirc L(c)$ is in $\mathcal{S}'$; (iv) $\mathcal{E}' = \mathcal{E} \cup \{\Diamond L(c) \mid \Diamond L(x) \in \mathcal{E}, c$ is a constant in $\mathsf{P}\}$.*

Example 1 *Let $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ where $\mathcal{U} = \{\exists x Q(x)\}$, $\mathcal{I} = \{P(c)\}$, $\mathcal{S} = \{P(x) \Rightarrow \bigcirc Q(x)\}$, and $\mathcal{E} = \emptyset$. Then $\mathrm{Cls}(\mathsf{P}) = \langle \mathcal{U}', \mathcal{I}', \mathcal{S}', \mathcal{E} \rangle$ where $\mathcal{U}' = \{Q(d)\}$ with d a Skolem constant, $\mathcal{I}' = \{P(c)\}$, and $\mathcal{S}' = \{P(x) \Rightarrow \bigcirc Q(x), P(c) \Rightarrow \bigcirc Q(c)\}$.*

We consider initial and universal clauses as literal multisets, and step clauses as ordered pairs of literal multisets. We assume basic knowledge of classical first-order resolution (see, for example, [BG01, Lei97, CL71]).

During a derivation more general *step* clauses can be derived, which are of the form $C \Rightarrow \bigcirc D$, where C is a *conjunction* of propositions, atoms of the form $P(x)$ and ground formulae of the form $P(c)$, where P is a unary predicate symbol and c is a constant such that c occurs in the input formula, D is a *disjunction* of arbitrary literals.

In [HKS05] we have refined the original fine-grained temporal resolution of [DFK03] by restricting the applicability of our deduction rules based on an atom ordering and a selection function in analogy to the refinement of first-order resolution [BG01]. We assume that we are given an *atom ordering* $\succ$, that is, a total and well-founded ordering on ground first-order atoms which is stable under substitution, and a *selection function* S which maps any first-order clause C to a (possibly empty) subset of its negative literals. An atom ordering $\succ$ is extended to literals by $(\neg)A \succ (\neg)B$ if $A \succ B$ and $\neg A \succ A$. A literal L is called (strictly) maximal w.r.t. a clause C if, and only if, there exists a ground substitution σ such that for all $L' \in C$: $L\sigma \succeq L'\sigma$ ($L\sigma \succ L'\sigma$). A literal L is *eligible* in a clause $L \vee C$ if either it is selected in $L \vee C$, or no literal is selected in C and L is maximal w.r.t. C.

The ordered monodic fine-grained temporal resolution calculus consists of the following deduction rules (we assume that different premises and conclusions of the deduction rules have no variables in common; variables may be renamed if necessary):

(1) *First-order ordered resolution with selection between two universal clauses*

$$\frac{C_1 \vee A \quad \neg B \vee C_2}{(C_1 \vee C_2)\sigma} \ ,$$

if σ is the most general unifier of A and B, *$A\sigma$ is eligible in $(C_1 \vee A)\sigma$, and $\neg B\sigma$ is eligible in $(\neg B \vee C_2)\sigma$.*

(2) *First-order ordered factoring with selection*

$$\frac{C_1 \vee A \vee B}{(C_1 \vee A)\sigma} \ ,$$

if σ is the most general unifier of A and B, and *$A\sigma$ is eligible in $(C_1 \vee A \vee B)\sigma$.*

(3) *First-order ordered resolution with selection between an initial and a universal clause, between two initial clauses, and ordered factoring with selection on an initial clause*. Defined in analogy to the two deduction rules above with the only difference that the result is an initial clause.

(4) *Ordered fine-grained step resolution with selection.*

$$\frac{C_1 \Rightarrow \bigcirc(D_1 \vee A) \quad C_2 \Rightarrow \bigcirc(D_2 \vee \neg B)}{(C_1 \wedge C_2)\sigma \Rightarrow \bigcirc(D_1 \vee D_2)\sigma} \ ,$$

where $C_1 \Rightarrow \bigcirc(D_1 \vee A)$ and $C_2 \Rightarrow \bigcirc(D_2 \vee \neg B)$ are step clauses, σ is a most general unifier of the literals A and B such that σ does not map variables from C_1 or C_2 into a constant or a functional term (functional terms can be introduced by Skolemisation), *$A\sigma$ is eligible in $(D_1 \vee A)\sigma$, and $\neg B\sigma$ is eligible in $(D_2 \vee \neg B)\sigma$.*

$$\frac{C_1 \Rightarrow \bigcirc(D_1 \vee L) \quad D_2 \vee \neg N}{C_1\sigma \Rightarrow \bigcirc(D_1 \vee D_2)\sigma} \ ,$$

where $C_1 \Rightarrow \bigcirc(D_1 \vee L)$ is a step clause, $D_2 \vee \neg N$ is a universal clause, and σ is a most general unifier of the literals L and N such that σ does not map variables from C_1 into a constant or a functional term, *$N\sigma$ is eligible in $(D_2 \vee \neg N)\sigma$, and $L\sigma$ is eligible in $(D_1 \vee L)\sigma$.*

(5) *Ordered right positive step factoring with selection.*

$$\frac{C \Rightarrow \bigcirc(D \vee A \vee B)}{C\sigma \Rightarrow \bigcirc(D \vee A)\sigma} \ ,$$

where σ is a most general unifier of the atoms A and B such that σ does not map variables from C into a constant or a functional term, and *$A\sigma$ is eligible in $(D \vee A \vee B)\sigma$.*

(6) *Clause conversion.* A step clause of the form $C \Rightarrow \bigcirc\bot$ is rewritten into the universal clause $\neg C$.

(7) *Eventuality resolution rule.*

$$\frac{\forall x(\mathcal{A}_1(x) \Rightarrow \bigcirc(\mathcal{B}_1(x))) \quad \ldots \quad \forall x(\mathcal{A}_n(x) \Rightarrow \bigcirc(\mathcal{B}_n(x))) \qquad \Diamond L(x)}{\forall x \bigwedge_{i=1}^{n} \neg\mathcal{A}_i(x)} \ (\Diamond^{\mathcal{U}}_{res}) \,,$$

where $\forall x(\mathcal{A}_i(x) \Rightarrow \bigcirc\mathcal{B}_i(x))$ are complex combinations of step clauses, called *full merged step clauses* [KDD$^+$05], such that for all $i \in \{1, \ldots, n\}$, the *loop* side conditions $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \neg L(x))$ and $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \bigvee_{j=1}^{n}(\mathcal{A}_j(x)))$, with $\mathcal{U}$ being the current set of all universal clauses, are both valid.

(8) *Ground eventuality resolution rule.*

$$\frac{\mathcal{A}_1 \Rightarrow \bigcirc\mathcal{B}_1 \quad \ldots \quad \mathcal{A}_n \Rightarrow \bigcirc\mathcal{B}_n \qquad \Diamond l}{\bigwedge_{i=1}^{n} \neg\mathcal{A}_i} \ (\Diamond^{\mathcal{U}}_{res}) \,,$$

where $\mathcal{A}_i \Rightarrow \bigcirc\mathcal{B}_i$ are merged derived step clauses such that the *loop* side conditions $\mathcal{U} \wedge \mathcal{B}_i \models \neg l$ and $\mathcal{U} \wedge \mathcal{B}_i \models \bigvee_{j=1}^{n} \mathcal{A}_j$ for all $i \in \{1, \ldots, n\}$ are both valid.

Rules (1) to (6), also called rules of *fine-grained step resolution*, are either identical or closely related to the deduction rules of ordered first-order resolution with selection, a fact that we will exploit in our implementation.

In contrast, rules (7) and (8) are much more complex, as they require not just one or two premises, but an indeterminate (though finite) number of premises which have to satisfy certain conditions. To find premises suitable for an application of the eventuality resolution rule, we

Function FG-BFS

Input: A set $\mathbf{S}$ of universal and step clauses, saturated by fine-grained resolution, and an eventuality clause $\Diamond L(x) \in \mathcal{E}$.

Output: A formula $H(x)$ with at most one free variable.

Method:

1. Let $H_0(x) = \top$; $N_0 = \emptyset$; $i = 0$.
2. Let $\mathbf{S_{i+1}} = \{P(c^l) \Rightarrow \bigcirc M(c^l) \,|\, \text{original } P(x) \Rightarrow \bigcirc M(x) \in \mathbf{S}\} \cup \{\top \Rightarrow \bigcirc(\neg H_i(c^l) \vee L(c^l))\} \cup \mathbf{S}$. Apply the rules of fine-grained step resolution *except the clause conversion rule* to $\mathbf{S_{i+1}}$. If we obtain a contradiction, then return the loop $\top$ (in this case $\forall x \neg L(x)$ is implied by the universal part).
   Otherwise let $N_{i+1} = \{C_j \Rightarrow \bigcirc\bot\}_{j=1}^{k}$ be the set of all *new* final clauses in the saturation of $\mathbf{S_{i+1}}$.
3. If $N_{i+1} = \emptyset$, return $\bot$; else let $H_{i+1}(x) = \bigvee_{j=1}^{k} C_j\{c^l \to x\}$.
4. If $\forall x(H_i(x) \Rightarrow H_{i+1}(x))$ return $H_{i+1}(x)$.
5. $i = i + 1$; goto 2.

Note: The constant c^l is a fresh constant used for loop search only

Figure 1: Breadth-first search using fine-grained step resolution.

use a particular algorithm, called FG-BFS (for fine-grained breadth-first search) shown in Fig. 1. This algorithm internally uses the deduction rules (1) to (5), and, in general, termination of the algorithm is not guaranteed.

Let *ordered fine-grained resolution with selection* be the calculus consisting of the rules (1) to (6) above, together with the ground and non-ground eventuality resolution rules (7) and (8), restricted to loops found by the FG-BFS algorithm. We denote this calculus by $\mathfrak{I}_{FG}^{S,\succ}$. The calculus can be extended by standard first-order redundancy elimination rules as well as analogous rules for step clauses.

Note that for ordered fine-grained step resolution with selection, the ordering and selection function only influence which literals on the right-hand side of an implication are eligible, literals on the left-hand side are not taken into account.

Theorem 2 ([HKS05]) *Ordered fine-grained resolution with selection is sound and complete for monodic temporal problems over expanding domains.*

4 Implementation

The deduction rules of ordered fine-grained step resolution are close enough to classical first-order resolution to allow us to use state of the art first-order resolution provers to provide an implementation of our calculus.

Let $\mathbf{S}$ be a temporal problem in clausal form. For every k-ary predicate, P, occurring in $\mathbf{S}$, we introduce a new $(k+1)$-ary predicate $\widetilde{P}$. We will also use the constant 0 (representing the initial moment in time), and unary function symbols s (representing the successor function on time) and h, which we assume not to occur in $\mathbf{S}$. Let ϕ be a first-order formula in the vocabulary of $\mathbf{S}$. We denote by $[\phi]^T$ the result of replacing all occurrences of predicates in ϕ by their "tilded" counterparts with T as the first argument (e.g. $P(x, y)$ is replaced with $\widetilde{P}(T, x, y)$). The term T will either be the constant 0 or the variable t (intuitively, t represents a moment in time). The variable t is assumed to be universally quantified.

Now, in order to realise fine-grained step resolution by means of classical first-order resolution, we define a set of first-order clauses $\mathbf{FO}(\mathbf{S})$ as follows.

- For every initial clause C from $\mathbf{S}$, the clause $[C]^0$ is in $\mathbf{FO}(\mathbf{S})$.
- For every universal clause D from $\mathbf{S}$, the clause $[D]^t$ is in $\mathbf{FO}(\mathbf{S})$.
- For every step clause $p \Rightarrow \bigcirc q$ from $\mathbf{S}$, the clause $\neg\widetilde{p}(t) \vee \widetilde{q}(s(t))$ is in $\mathbf{FO}(\mathbf{S})$, and for every step clause $P(x) \Rightarrow \bigcirc Q(x)$, the clause $\neg\widetilde{P}(t,x) \vee \widetilde{Q}(s(t),h(x))$ is in $\mathbf{FO}(\mathbf{S})$.

The key insight is that fine-grained step resolution on $\mathbf{S}$, including (implicitly) the clause conversion rule, can be realised using classical ordered first-order resolution with selection (see, e.g. [BG01]) on $\mathbf{FO}(\mathbf{S})$. For universal and initial resolution and factoring rules, rules (1) to (3), this is obvious. For step resolution and (step) factoring, rules (4) and (5) we observe that if a clause contains a *next-state* literal, i.e. a literal whose first argument starts with the function symbol s, a factoring or resolution inference can only be performed on such a literal. This requirement can be enforced by an appropriate atom ordering. Note that all rules performing inferences on step clauses impose the restriction on unifiers (such as σ) that σ does *not* map variables occurring in the left side of a step clause into a constant or a functional term. On first-order clauses, this restriction is enforced by the function symbol h introduced by $\mathbf{FO}$: Each temporal literal $\bigcirc Q(x)$ is mapped by $\mathbf{FO}$ to $\widetilde{Q}(s(t),h(x))$, and the function symbol h "shields" the variable x from being instantiated by a constant or functional term. No explicit clause conversion rule, rule (6), is required for the translated clauses.

Note that standard redundancy deletion mechanisms, such as subsumption and tautology deletion, are also compatible with fine-grained step resolution (for details see [KDD$^+$05]). Note further that the first-order clause $\neg\widetilde{P}(t,x) \vee \widetilde{Q}(s(t),h(x))$ from $\mathbf{FO}(\mathbf{S})$ does not subsume the clause $\neg\widetilde{P}(t,c) \vee \widetilde{Q}(s(t),c)$ stemming from the clausification as introduced in Definition 2. We need clauses of the form $\neg\widetilde{P}(t,c) \vee \widetilde{Q}(s(t),c)$ for the calculus to be complete [KDD$^+$05].

As for the eventuality resolution rule and ground eventuality resolution rule, rules (7) and (8), we find the merged clauses required by means of the FG-BFS algorithm. The only difficulty in implementing the algorithm in Fig. 1 using first-order ordered resolution with selection is that in step (2) of the algorithm, the rules of fine-grained step resolution are applied with the exception of the clause conversion rule, rule (6). As no explicit clause conversion rule is required on $\mathbf{FO}(\mathbf{S})$, this restriction cannot be enforced by disabling one of the deduction rules. Instead we use a variant $\mathbf{FO}_{BFS}$ of $\mathbf{FO}$ which has the desired effect. Let $\mathbf{S_{i+1}}$ be a monodic temporal problem in clausified form as defined in step (2) of the FG-BFS algorithm. Then $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$ is defined as follows:

- For every universal clause D in $\mathbf{S_{i+1}}$, the clause $[D]^t$ is in $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$.
- For every ground step clause $p \Rightarrow \bigcirc l$ in $\mathbf{S_{i+1}}$, the clause $\neg\widetilde{p}(0) \vee \widetilde{l}(s(t))$ is in $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$, and for every non-ground step clause $P(x) \Rightarrow \bigcirc M(x)$ in $\mathbf{S_{i+1}}$, the clause $\neg\widetilde{P}(0,x) \vee \widetilde{M}(s(t),h(x))$ is in $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$.

Recall that initial clauses do not contribute to loop search, so we do not include their translation into $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$. Again, the motivation for $\mathbf{FO}_{BFS}$ is that of saturation of $\mathbf{S_{i+1}}$ under the rules of fine-grained step resolution except that the clause conversion rule corresponds to the saturation of $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$ under ordered first-order resolution as described above. In particular,

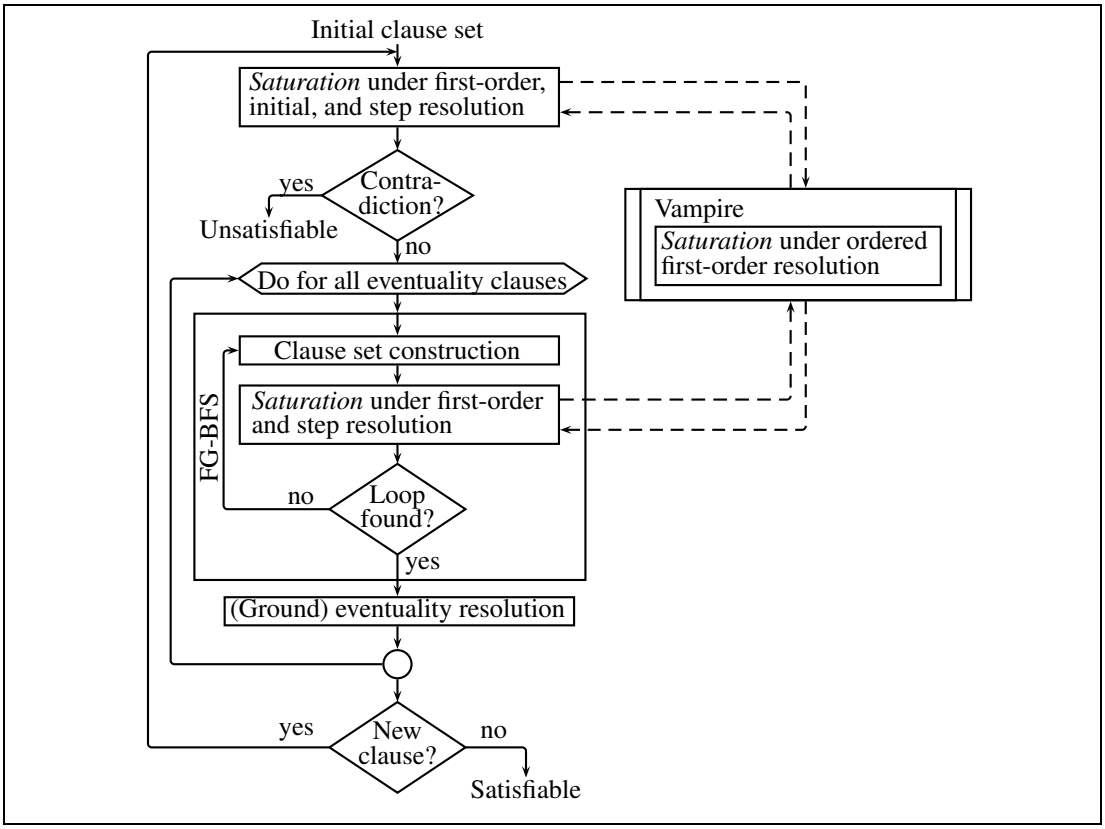


Figure 2: Main loop of **TeMP**

clauses consisting only of literals whose first argument is '0' in the saturation of $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$ correspond to final clauses (up to negation). Using this criterion it is straightforward to extract those clauses from the saturation of $\mathbf{FO}_{BFS}(\mathbf{S_{i+1}})$ to form the set N_{i+1} which is the outcome of step (2) of the FG-BFS algorithm and to proceed with step (3).

The logical consequence check in step (4) of the FG-BFS algorithm is again delegated to a first-order prover: for every $C_i(x) \in H_i(x)$ we form a new clause set $C_i \wedge \neg H_{i+1}(x)$; if all the resulting sets are unsatisfiable, $\forall x(H_i(x) \Rightarrow H_{i+1}(x))$ is valid.

Note that it is straightforward to see whether a clause in $\mathbf{FO}(\mathbf{S})$ is the result of translating an initial, a universal, or a (non-)ground step clause. This makes it possible to compute $\mathbf{FO}_{BFS}(\mathbf{S})$ from $\mathbf{FO}(\mathbf{S})$ instead of from $\mathbf{S}$. Also, the conclusion of an application of one of the eventuality resolution rules can directly be computed as a set of first-order clauses of the appropriate form. Thus, there is no need to ever translate clauses in $\mathbf{FO}(\mathbf{S})$ back to DSNF clauses. Instead, after translating the input given to **TeMP** once using $\mathbf{FO}$, we can continue to operate with first-order clauses. In addition, we use the set-of-support strategy since the set $\mathbf{S}$ is already saturated and only inferences between clauses in $\mathbf{S_{i+1}} - \mathbf{S}$ and $\mathbf{S}$ need to be performed.

In our implementation, we extend the propositional temporal prover, **TRP**++ [HK03], to deal with monodic formulae. The main procedure, depicted in Fig. 2, of our implementation of this calculus consists of a loop where at each iteration (i) the set of temporal clauses is saturated under applications of the rules of fine-grained step resolution, that is, rules (1) to (6), and (ii) then for every eventuality clause in the clause set, an attempt is made to find a set of premises

for an application of the (ground) eventuality resolution rule. If we find such a set, the set of clauses representing the conclusion of the application of the rule is added to the current set of clauses and the resulting set is saturated under application of the step resolution rules (this helps to identify whether the conclusion of the eventuality resolution rule is redundant or not). We have two control strategies concerning how to explore eventualities in loop search: either we go through them one by one regardless of whether a loop is found and application of the (ground) eventuality resolution rule derives new non-redundant clauses (a sort of breadth-first strategy) or we enter the next iteration of the main loop as soon as a loop is found for which an application of the (ground) eventuality resolution rule results in new non-redundant clauses (a sort of depth-first strategy). Fig. 2 illustrates the breadth-first strategy. The main loop terminates if the empty clause is derived, indicating that the initial set of clauses is unsatisfiable, or if no new clauses have been derived during the last iteration of the main loop, which in the absence of the empty clause indicates that the initial set of clauses is satisfiable.

The task of saturating clause sets with classical resolution simulating step resolution is delegated to the **Vampire** kernel [RV02], which is linked to the whole system as a C++ library. **TeMP** communicates with the **Vampire** kernel in a direct way via the kernel API, thus avoiding expensive textual communication. For temporal reasoning (loop search using FG-BFS, ground eventuality resolution and eventuality resolution rules), we use our own data structures, which are efficient enough for our purposes, and there is a special module in **TeMP** which rewrites **TeMP**'s own data structures to, and from, **Vampire** data structures. While there is a little overhead stemming from rewriting, such an architecture opens up the possibility of replacing **Vampire** with any other first-order theorem prover (supporting ordered resolution with selection). Note, however, that minor adjustments have been made in the functionality of **Vampire** to accommodate step resolution: a special mode for literal selection has been introduced such that in a clause containing a next-state literal only next-state literals can be selected. At the moment, the result of a previous saturation step, augmented with the result of an eventuality resolution application, is resubmitted to the **Vampire** kernel, although no inferences are performed between the clauses from the already saturated part. This is only a temporary solution, and in the future we hope **Vampire** will support incremental input in order to reduce communication overhead.

5 Case studies

In this section we demonstrate an application of **TeMP** to the automatic verification of parameterised cache coherence protocols. These protocols play an important role in models of shared-memory multiprocessor systems. Usually, in such systems, every individual processor has its own private cache memory; the processor uses the cache to hold local copies of main memory blocks (for details, see, for example, [Han93]). While reducing the access time, this approach poses the problem of *cache consistency*, whereby one has to ensure that the copies of the *same* memory block in the caches of *different* processors are consistent. Such data consistency can be provided by *cache coherence protocols*, which typically operate as follows: every processor is equipped with a finite state control, which *reacts* to the *read* and *write* requests. Abstracting from the low-level implementation details of read, write and synchronisation primitives, one may model cache coherence protocols as *families of identical finite state machines* together with a primitive form of communication: if one automaton makes a transition (an action) a, then it

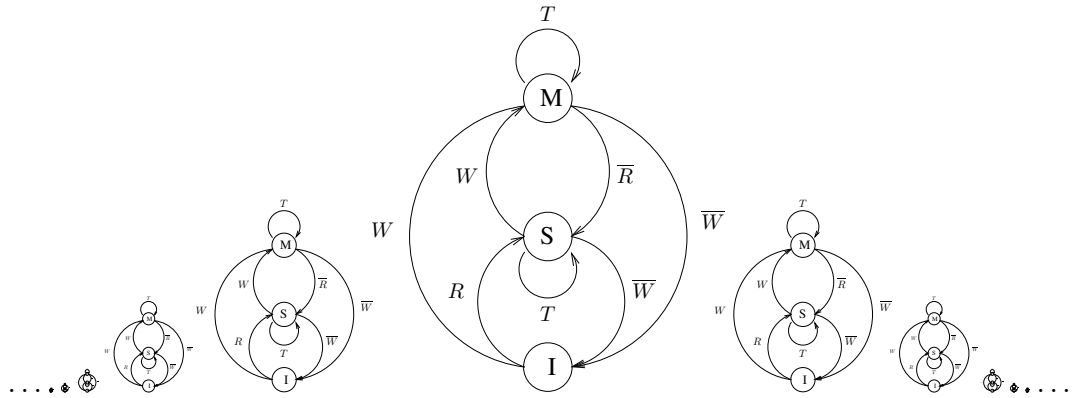


Figure 3: A family of identical automata for the *MSI* Cache Coherence Protocol

is required that *all* other automata simultaneously make a complementary transition (reaction) $\bar{a}$ [Del00, Del03].

We refer to this model as the *communicating broadcast automata model*, and to corresponding protocols as *communicating broadcast automata protocols*.

The following types of correctness conditions (safety for data consistency [Han93]) are most common for communicating broadcast automata protocols.

Non Co-occurrence of states: some local states q_1, q_2 should not appear in the any global state.

At most one: a local state q can appear at most once, that is, be a state of at most one processor, in any global state.

We illustrate this model on a particular example, namely, the *MSI* protocol (the name originates from the names of three states of the protocol, *Modified*, *Shared*, *Invalid*, respectively). Possible actions are *R* (for *read*), *W* (for *write*), and *T* (for *local* τ*-transition*). We present the protocol by the transition relation of the finite state machine below and give its graphical representation in Fig. 3; for the clarity of presentation we omit reaction loops, in particular, $\overline{T}$ loops around every vertex.

$$
\begin{array}{llllllllll}
\tau(I,W) & = & M & \tau(I,R) & = & S & & & & \\
\tau(I,\overline{W}) & = & I & \tau(I,\overline{R}) & = & I & \tau(I,\overline{T}) & = & I \\
\tau(S,W) & = & M & & & & \tau(S,T) & = & S \\
\tau(S,\overline{W}) & = & I & \tau(S,\overline{R}) & = & S & \tau(S,\overline{T}) & = & S \\
 & & & & & & \tau(M,T) & = & M \\
\tau(M,\overline{W}) & = & I & \tau(M,\overline{R}) & = & S & \tau(M,\overline{T}) & = & M
\end{array}
$$

Notice that transition function τ is a partial one and it is not defined, for example, on the pair (s,r). In the beginning of time, all automata are in the state I.

The correctness conditions for this protocol are:

- Non Co-occurrence condition for the states M and S; and

- At most one condition for the state M.

Following [FL03] we now give the temporal specification of the *MSI* protocol and its correctness conditions in the syntax of **TeMP**. The syntax of **TeMP** is an extension of the TPTP[1] syntax with temporal formulae. In what follows, `![X]p(X)` stands for $\forall x P(x)$, `?[X]p(X)` stands for $\exists x P(x)$, operators `~`, `&`, `|`, and `->` stand for $\neg$, $\wedge$, $\vee$, and $\rightarrow$, respectively; `next`, `always`, and `sometime` stand for $\bigcirc$, $\Box$, and $\Diamond$, respectively. The idea of our encoding of the protocol can be explained as follows. For every state of the protocol we introduce an unary predicate symbol: m for M, s for S and i for I. The elements of the (abstract) domain of a temporal model represent automata. An automaton represented by an element s is in a particular state, say M if, and only if, corresponding predicate m holds on s. We introduce one more unary predicate a, where $a(s)$ holds if, and only if, an automaton represented by s is *active* and performs some action. Further unary predicates w, r, and t denote actions that can be executed by an automaton. Only the active automaton can perform an action, all other automata perform a simultaneous reaction.

For example, the formula

$$\forall x(a(x) \wedge i(x) \Rightarrow (w(x) \vee r(x)) \ \wedge \ \forall x((i(x)) \wedge w(x) \Rightarrow \bigcirc m(x))$$

expresses that in the state I both reading and writing actions are possible, and $\tau(I, W) = M$ and

$$\forall x(m(x) \wedge \neg w(x) \wedge \exists y w(y) \Rightarrow \bigcirc i(x))$$

describes the reaction on this writing action of an automaton in the state M. Note that the reaction is expressed as $\neg w(x) \wedge \exists y w(y)$.

In the original representation, [FL03], *equality* is used to express the fact that only one automaton can be active at any time. Equality destroys the completeness of the monodic fragment [WZ02]. So, in [FLK05] we suggested replacing equality with the *congruence* relation giving an incomplete, but correct, verification procedure. It is well-known that resolution on congruence axioms tends to cause the generation of too many (mostly unnecessary) clauses, and resolution is the heart of our temporal prover. Therefore, we apply the prover to even weaker problems where not all of the congruence axioms are included. We express the fact that no more than one instance of the automaton is active at any time saying that, if two instances of the automaton, x and y, are active at the same time, they are in the same state and perform the same action.

Summing up the considerations above, the temporal specification of the *MSI* protocol, SP_{MSI} is characterised by the following formula.

```
![X]i(X) &
%%%%%%%%%% Initially all automata are in the state $I$

always(![X](i(X) | m(X) | s(X))) &
%%%%%%%%%% Every automaton at every moment of time is in the one of
%%%%%%%%%% the possible states

always(~?[X]((m(X) & i(X)) | (m(X) & s(X)) | s(X) & i(X))) &
%%%%%%%%%% Automata states are mutually excluded
```

[1] `http://www.cs.miami.edu/~tptp`

```
always(?[X]a(X) & ![X,Y](a(X) & a(Y) -> ((i(X) & i(Y)) |
                                          (m(X) & m(Y)) |
                                          (s(X) & s(Y) ))))  &
%%%%%%%%%% If two automata are simultaneously active then they both
%%%%%%%%%% are in the same state.


always(~?[X] ((w(X) & r(X)) | (w(X) & t(X)) | (r(X) & t(X))))  &
%%%%%%%%%% Actions are mutually excluded

always(![X,Y](a(X) & a(Y) -> (((w(X) & w(Y)) | (~w(X) & ~w(Y))) &
                              ((r(X) & r(Y)) | (~r(X) & ~r(Y))) &
                              ((t(X) & t(Y)) | (~t(X) & ~t(Y)))
                             ))) &
%%%%%%%%%% An automaton cannot do different actions at the same time

always(![X]((w(X) | r(X) | t(X)) -> a(X))) &
%%%%%%%%%% W, R, T are actions


always(![X] (i(X) & a(X) -> (w(X) | r(X)))) &
%%%%%%%%%% The only actions available for an active automaton in
%%%%%%%%%% the state I are W and R


always(![X] (s(X) & a(X) -> (w(X) | t(X)))) &
%%%%%%%%%% The only actions available for an active automaton in
%%%%%%%%%% the state S are W and T


always(![X] (m(X) & a(X) -> t(X))) &
%%%%%%%%%% The only action available for an active automaton in
%%%%%%%%%% the state M is T

always( ![X] (i(X) & w(X) -> next m(X)) &
        ![X] (i(X) & r(X) -> next s(X)) &
        ![X] (s(X) & w(X) -> next m(X)) &
        ![X] (s(X) & t(X) -> next s(X)) &
        ![X] (m(X) & t(X) -> next m(X)) &
%%%%%%%%%% Results of the action

        ![X] ((i(X) & ~w(X) & ?[Y]w(Y)) -> next i(X)) &

        ![X] ((i(X) & ~r(X) & ?[Y]r(Y)) -> next i(X)) &
        ![X] ((i(X) & ~t(X) & ?[Y]t(Y)) -> next i(X)) &


        ![X] ((s(X) & ~w(X) & ?[Y]w(Y)) -> next i(X)) &
        ![X] ((s(X) & ~r(X) & ?[Y]r(Y)) -> next s(X)) &
        ![X] ((s(X) & ~t(X) & ?[Y]t(Y)) -> next s(X)) &
```

Property	Total time	**Vampire** called	Generated clauses	Subsumed clauses
MSI prop.1	10.272s	15	219892	185870
MSI prop.2	33.054s	48	849143	797765
Synapse N+1 prop. 1	0.472s	12	16490	11938
Synapse N+1 prop. 2	0.663s	14	21204	15874

Figure 4: **TeMP** performance on *MSI* and *Synapse N+1*

```
        ![X] ((m(X) & ~w(X) & ?[Y]w(Y)) -> next i(X)) &
        ![X] ((m(X) & ~r(X) & ?[Y]r(Y)) -> next s(X)) &
        ![X] ((m(X) & ~t(X) & ?[Y]t(Y)) -> next m(X))
%%%%%%%%% Result of the re-action. An automaton performs a re-action
%%%%%%%%% iff it does not do an action, but somebody else does.
       )
```

The negation of the Non co-occurrence condition is given by the following formula

```
sometime (?[X](m(X)) & ?[Y](s(Y)))
```

while the (Skolemised) negation of the *At most one* condition is expressed by

```
q(c) & ~q(d) & sometime (m(c) & m(d))
```

Notice that the "fresh" predicate q is introduced to ensure that constants c and d are interpreted by different elements of the domain.

We apply **TeMP** to the conjunction of the protocol specification and the negation of one of the correctness conditions; if the resulting set is unsatisfiable, the protocol satisfies the condition.

This approach to the verification of parameterised protocols was successfully applied to a number of protocols that are basically described by finite automata. It takes only a few seconds to verify on a desktop computer both the *Non co occurrence of states* and the *At most one* properties for the *MSI* protocol. The same is true for the *Synapse N+1* protocols given in [Del03]. Fig. 4 contains some statistical data concerning the performance of **TeMP** (and calls to **Vampire**) on the *MSI* and *Synapse N+1* protocols.

However, of particular interest is a wider class of parameterised communicating broadcast automata protocols where *global* conditions are allowed. The corresponding parameterised model with global conditions is introduced in [Del03]. The basic model of broadcasting automata is extended to include conditional actions of the form $P \rightarrow \sigma$. A transition S, σ, S' is allowed only if the global state S satisfies P. For example, P may express the fact that in the global state there is an automaton in a specified local state. We have translated the *Illinois* cache coherence protocol [PP84], which uses global conditions, into the the syntax of **TeMP**, but have met practical difficulties in running the prover on the result of the translation: the prover ran out of computing resources.

We can extract the first-order problems given to **Vampire** in TPTP syntax; we tried **SPASS** and **E** on those problems, and none of these systems terminated within reasonable time. (Note however that neither **SPASS** nor **E** support the special selection strategy needed for our implementation as described in Section 4, so even if one of those provers terminated that would not

help **TeMP** immediately, but we could consider replacing **Vampire** with one of those provers, of course.)

There are some interesting points which can be observed from our experiments.

- Our prover relies on **Vampire**'s capability to saturate a set of first-order clauses efficiently. The **Vampire** kernel is called several times, as illustrated by the statistics in Fig. 4, and only one (the last) call might be on an unsatisfiable set of clauses.
- The atom ordering and literal selection function used dramatically influences **Vampire**'s performance on satisfiable clauses. In particular, we only managed to prove both correctness conditions for *MSI* when we used a literal selection strategy different from what the kernel suggests by default.
- Further experiments with **Mace** showed that the satisfiable problems have very simple, two element models, which can be found in very little time. Our prover would benefit from a form of semantic resolution based on the properties of these simple models.

  However, we are also not in a position that we just need to detect satisfiability of a clause set. In particular, in the FG-BFS algorithm we need to deduce *all* final clauses derivable from a given clause set, not just detect whether the given clause set is satisfiable.
- When the prover does not terminate, we see that the memory consumption grows very slowly which suggests that with current strategies, **Vampire** generates and immediately subsumes a very large amount of clauses.

  So, on the one hand we benefit from very efficient subsumption algorithms. On the other, a strategy which could reduce the number of subsumed clauses that are derived would be even more beneficial.

6 Conclusions

Over the past four decades, theory, engineering, and practice of building automated first-order theorem provers reached a mature state, and a number of powerful implementations are available at the moment. The success of those implementations allows one to use a tool originally developed for first-order logic as an integral part of a prover for a different, more complex logic.

Based on verification case studies, we performed experiments with our system. One immediate observation is that while the available first-order provers feature extremely efficient inference engines, their default strategies aim mainly at unsatisfiable clause sets. When the saturation of a set of satisfiable clauses is needed, extra fine-tuning is required. The relatively degraded performance on satisfiable clause sets is somewhat surprising because the 'problematic' clause sets we have encountered have very simple models. Partly, such behaviour of first-order provers can be possibly explained by the format of the CASC competition.

For this first case study we have deliberately chosen a problem with a known solution. To compete with other known approaches capable to solve those problems, such as the one based on integer vector reachability utilised in [Del03], we have to fine-tune our temporal resolution engine. However, we expect that the main advantage of our method is the ability to reason about a wider class of systems, such as protocols with *asynchronous* communication, which seem to be outside of the scope of the integer vector reachability method, but which can still be modelled via the first-order temporal approach. We see verification of asynchronous communication as

the most promising direction for future work. This needs, however, a deeper insight into first-order resolution provers and will undoubtedly require new saturation strategies for satisfiable problems.

We are planning to continue our work on **TeMP** and, in particular, build a pool of possible strategies aimed at the kind of first-order problems stemming from our applications. We believe that this research will be beneficial for both the temporal reasoning and the first-order theorem proving community.

References

[BG01] L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01], chapter 2, pages 19–99.

[CL71] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1971.

[Del00] G. Delzanno. Automatic verification of parametrized cache coherence protocols. In *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 53–68. Springer, 2000.

[Del03] G. Delzanno. Constraint-based verification of parametrized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[DFK] A. Degtyarev, M. Fisher, and B. Konev. Monodic temporal resolution. *ACM Transactions on Computational Logic*. To appear.

[DFK03] A. Degtyarev, M. Fisher, and B. Konev. Monodic temporal resolution. In *Proc. CADE-19*, volume 2741 of *LNAI*, pages 397–411. Springer, 2003.

[FDP01] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, 2001.

[FL03] M. Fisher and A. Lisitsa. Deductive verification of cache coherence protocols. In *Proc. AVOCS 2003*, pages 117–186, 2003.

[FLK05] M. Fisher, A. Lisitsa, and B. Konev. Practical infinite-state verification with temporal reasoning. In *Proceedings of the NATO advance research workshop Verification of infinite-state systems with applications to security VISSAS 2005*, 2005.

[Han93] J. Handy. *The Cache Memory Book*. Academic Press, 1993.

[HK03] U. Hustadt and B. Konev. TRP++ 2.0: A temporal resolution prover. In *Proc. CADE-19*, volume 2741 of *LNAI*, pages 274–278. Springer, 2003.

[HKRV04] U. Hustadt, B. Konev, A. Riazanov, and A. Voronkov. **TeMP**: A temporal monodic prover. In *Proc. IJCAR 2004*, volume 3097 of *LNAI*, pages 326–330. Springer, 2004.

[HKS05] U. Hustadt, B. Konev, and R. Schmidt. Deciding monodic fragments by temporal resolution. In *Proc. CADE-20*. Springer, 2005. To appear.

[HWZ00] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.

[KDD+05] B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt. Mechanising first-order temporal resolution. *Information and Computation*, 199(1–2):55–86, 2005.

[Lei97] Alexander Leitsch. *The Resolution Calculus*. Springer, 1997.

[NW01] A. Nonnengart and Ch. Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [RV01], chapter 6, pages 335–370.

[PP84] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memries. In *Proc. 11th International Symposium on Computer Architecture*, pages 348–354. IEEE, 1984.

[RV01] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier, 2001.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[WZ02] F. Wolter and M. Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic. *Annals of Pure and Applied logic*, 118:133–145, 2002.

MPTP 0.2: Design, Implementation, and First Cross-verification Experiments

Josef Urban
Dept. of Theoretical Computer Science
Charles University
Malostranské nám. 2/25, Praha, Czech Republic
urban@kti.ms.mff.cuni.cz

Abstract

This is a report about development and preliminary testing of the second version of the MPTP (Mizar Problems for Theorem Proving) system. The goal of this project is to make the large formal Mizar Mathematical Library (MML) available to current first-order automated theorem provers (ATPs).

This version of MPTP switches to a generic extended TPTP syntax that adds term-dependent sorts and abstract (Fraenkel) terms to TPTP. We describe these extensions and explain how they are transformed by MPTP to standard TPTP syntax using relativization of sorts and de-anonymization of abstract terms. The system is now based solely on the newly available native XML format of Mizar articles, which is directly transformed using XSLT to the extended TPTP syntax, so no special-purpose MPTP exporter based on the Mizar implementation is necessary. This also makes export of full Mizar proofs much easier, and we discuss the TPTP syntax extensions for their encoding.

One of the goals of MPTP is ATP cross-verification of the Mizar library. The prerequisite for this is the cross-verification of the simplest Mizar inference steps. The feasibility of this has been tested on a small initial part (ca. 40.000 problems) of the library. The methodology and results of this test are described here.

1 Introduction

1.1 Mizar

Mizar [Rud92, RT99, MR04] is a formal Jaskowski-style [Jas34, Pel99] mathematical language and a proof checker for that language. There are two important features that distinguish Mizar from other proof assistants [Wie03]:

- It is essentially a first-order system (based on set theory).
- It focuses on the development of the large Mizar Mathematical Library (MML) which (as of May 2005) contains more than 900 formal articles from various fields of mathematics.[1]

[1] See `http://mizar.uwb.edu.pl/JFM/mmlident.html` or `http://merak.pb.bialystok.pl/` for contents of MML

Following is an example of the Mizar language, It is the Mizar proof of the theorem `COMPLEX1:2` (second theorem in article `COMPLEX1` [Byl90]):

```
theorem Th2:
  for a, b being real number holds
   a^2 + b^2 = 0 iff a = 0 & b = 0
 proof
   let a, b be real number;
   thus a^2 + b^2 = 0 implies a = 0 & b = 0
    proof assume
A1:    a^2 + b^2 = 0;
A2:    0 <= a^2 & 0 <= b^2 by SQUARE_1:72;
     assume a <> 0 or b <> 0;
     then a^2 <> 0 or b^2 <> 0 by SQUARE_1:73;
     then 0 + 0 < a^2 + b^2 by A2,REAL_1:67;
     hence contradiction by A1;
    end;
   assume a = 0 & b = 0;
   hence thesis by SQUARE_1:60;
 end;
```

This article cannot fully explain Mizar (see the above given references), we will just point out several Mizar features on this example.

- The language uses types (like ‘‘`real number`’’ above). Since Mizar is based on set theory, the types however do not play any “foundational” role in the system (as is the case e.g. for HOL and many other higher-order systems). The best way to think of Mizar types is to treat them just as normal first-order predicates, for which some facts are obvious to the Mizar checker (and thus can be used to decrease the verbosity of formalization, and for early error checking very similar to the standard order-sorted type systems [GM92]). These “obvious” facts include e.g. the widening hierarchy, or the non-emptiness of extensions of these predicates. All such “special” facts obviously have to be proved when the types are defined.

- The Jaskowski-style natural-deduction proofs are steered by the implicit *thesis*, which is in the beginning equal to the proposition that is being proved. This *thesis* is reduced by various *Reasoning Items*, like ‘‘`let`’’ (Universal-introduction), ‘‘`assume`’’ (Implication-introduction) and ‘‘`thus`’’, ‘‘`hence`’’ (explicit justification of (a part of) the thesis). The proof is finished when the *thesis* is reduced to verum.

- The keyword ‘‘`by`’’ followed by labels (e.g. `by A2,REAL_1:67;`) introduces *Simple Justifications*. Mizar has a limited fast refutational prover [Wie00, NB04], which is used to discharge the proof obligations which are already sufficiently simple. The labels refer to the propositions from which the current formula should follow. Labels like `REAL_1:67` denote the globally available Mizar theorems (imported from other articles), while other labels (like ‘‘`A2`’’) refer to “local” propositions, previously proved or assumed inside the current proof.

- Similarly, functors, constants and predicates can also be defined either globally (like + and =), or locally in the proof (like the local constants ‘‘`a`’’ `and` ‘‘`b`’’

introduced by the ‘‘let’’ keyword). Obviously, all the local constructs can only be used in their proper scope.

1.2 The goals of MPTP, ILF

There are several goals of the MPTP (Mizar Problems for Theorem Proving) project, described in more detail in [Urb04]. In short, the cooperation of modern ATP systems with large libraries of formalized mathematics is good both for ATPs (large number of testing problems, optimization on various mathematical domains, dealing with large knowledge bases, etc.), and for the formalization efforts (proof assistance, cross-verification, automated theory refactorings, etc.). Such cooperation is also the best candidate for merging the deductive (e.g. ATP) and inductive (e.g. machine learning) methods of Artificial Intelligence, because mathematics is (by definition) the most deductively developed science, and once we have sufficient amount of such data, inductive methods can be applied too.

MPTP has been much inspired by the Mizar-part of the large ILF project [DW97, Dah98], which (unfortunately) stopped in 1998 without finishing the Mizar-to-ATP export. Thanks to ILF, hundreds of ATP problems extracted from several untyped Mizar articles have been for several years already included in the standard TPTP [SS98] library.

1.3 First MPTP version

The first version of MPTP has already been used for initial exploration of the usability of ATP systems on the Mizar Mathematical Library (MML), and of the benefits of assisting deductive tools with trained inductive advisors [Urb04]. The first important number obtained is the 41 % ATP success rate on reproving about 30.000 MML theorems from high-level hints[2] taken from corresponding MML proofs. The second important number is that in about 35 % of these cases (i.e., about every seventh MML theorem), relevant high-level hints sufficient for a successful ATP proof can be selected fully automatically by an independent Bayesian advisor [CCRR99] trained on previous MML proofs.

It was the primary goal of the first MPTP version to make these initial measurements possible in order to get some real feedback about the feasibility of (inductively assisted) ATP over a very large body of formalized mathematics. No such hard evidence had been previously known, which sometimes lead to overly pessimistic views on such a project. Many shortcuts and simplifications were therefore taken in the first MPTP version, naming at least the following:

- Mizar formulas were directly exported to the DFG [HKW96] syntax used by the SPASS [Wei01] system. SPASS seemed to perform best on MPTP problems, probably because of its handling of sort theories. SPASS also has a built-in efficient clausifier [NW01], which the other efficient provers like E [Sch02] and Vampire [RV02] did not have.
- This particularly meant that one concrete method of handling sorts (encoding as predicates) was chosen for the export, yielding standard (untyped) first-order

[2] precisely: other Mizar theorems and definitions used in the proofs; see [Urb04] for details

formulas, from which the original type information could not be recovered and used for different encodings.

- Mizar proofs were not exported. Only the lists of MML references (theorems and definitions) used for proof of each MML theorem were remembered for re-creation of ATP problems. The proof structure and internal lemmas were forgotten.

- Such lists of MML references are in about 80 percent (27,449 out of 33,527) of theorem proofs sufficient as high-level hints for reproving - i.e. the Mizar proofs use only these MML references and some implicit (background) facts like type hierarchy, arithmetical evaluations, etc. In the Mizar proofs of the remaining ca 20 percent (6078) of theorems, Mizar *schemes*[3] and top-level non-theorem lemmas[4] were used. These two kinds of propositions were completely ignored, making these theorems not eligible for ATP reproving.

- The export of Mizar *structure* types was incomplete (some axioms were missing), *abstract terms* were translated incorrectly. Both these shortcuts were justified by the low frequency of such Mizar constructs.

Many of these simplifications however made further experiments with MPTP difficult or impossible. The lack of proof structure prevents measurements of ATP success rate on all internal proof lemmas, and experiments with unfolding lemmas with their own proofs. Additionally, even if only several abstract terms were translated incorrectly, during such proof unfoldings they could spread much wider. All this negatively affects the possibility of full ATP cross-verification of MML. For such cross-verification it will often be necessary to follow the structure of Mizar proofs, and use the internal lemmas as hints for an ATP proof. The working objectives of further versions of MPTP should therefore be:

- the correctness of translation of even the least frequent Mizar constructs

- complete export of Mizar proofs

- a sufficiently generic (i.e. rich) format allowing different handling of e.g. Mizar sorts, but implementing some default transformations for systems that are not specialized in such areas

1.4 Structure of this article

The rest of this article describes the current implementation of MPTP 0.2, and the first experiments done with it. Section 2 shortly introduces the general XML-based solution taken for complete export of MML for other systems. Section 3 explains how the Mizar types and abstract terms are handled in MPTP 0.2, suggests extensions to the TPTP standard for their encoding, and shows how these extensions are transformed to the standard first-order TPTP format by MPTP. Translation of the parts of MML proofs which are needed for the experiments described here is also explained, and we sketch

[3] Mizar schemes are second-order theorems parametrized by functions or predicates.

[4] Vast majority of Mizar propositions proved on the top-level (i.e. not inside proof of other proposition) are in Mizar exported as theorems reusable in other articles. This is however not mandatory.

the overall algorithm used for producing ATP problems from MML problems. In section 4 several such experiments are described, all dealing with the reproving of the Mizar *Simple Justifications*. The influence of a better encoding of the abstract terms on the success rate of ATP systems is measured there. 48 articles where the translation is now believed to be complete are used for an attempt to reprove 100 percent of the ca 18,000 *Simple Justifications* contained in them. Again a machine learning method is used to make the small number of hard problems easier, and all but 67 problems are reproved.

2 XML-ization of Mizar

The most demanding objective among those mentioned above is the complete export of Mizar proofs. By a complete proof export we should mean at least two things:

- complete export of the Mizar proofs into a format that can be easily processed by other systems (e.g. MPTP)
- supplying functions that can generate ATP problems corresponding to parts of these proofs

This task was previously dealt with in the ILF project [DW97, Dah98], and at least the first requirement was completely solved there by having a special-purpose Mizar-to-Prolog exporter (by Czeslaw Bylinski), translating Mizar articles to a Prolog syntax. An obvious solution to the first requirement would therefore be an update of the ILF exporter. Several issues appeared when considering such solution:

- The ILF exporter was not a standard and maintained part of Mizar, and due to the fast development of Mizar in recent years, it became quite outdated and would require a complete rewrite.
- Several other systems working with the exported Mizar articles have appeared in the recent years: MMLQuery [BR03], MoMM [Urb05b], MizarMode [Urb05a, BU04], and also MPTP. Each of these systems used its own special-purpose exporting tool for doing very similar things. Each of these exporters were in different state of up-to-dateness. The natural solution to this situation would be just one well-maintained generic exporter.
- The old internal format used by Mizar itself was designed long ago, when memory and storage were expensive, and it was quite hard to extend for new Mizar constructs and utilities. A new extensible and richer format would be useful for Mizar itself.

All these issues resulted in quite a large reimplementation of Mizar described in [Urb05c]. Mizar started to use XML[5] natively as its internal format produced during parsing. This format was significantly extended, and it now contains a very complete semantically disambiguated form of a Mizar article (even with some ATP-important items, like definitional expansions, which were missing in the ILF exporter). Because of the completeness of this format, and thanks to the wide-spread availability of XML parsers,

[5] See `http://lipa.ms.mff.cuni.cz/~urban/Mizar.html` for specification of the Mizar XML format.

the need for special-purpose Mizar exporters and the problem of their maintenance were thus largely eliminated. The whole Mizar internal library (items reusable in other articles) is now distributed in this format, and complete articles are translated to it just by running the Mizar verifier.

The format still has to be space-economical to keep the Mizar verification times acceptable, so in its native form it avoids redundant information that can be easily recovered by postprocessing. A simple postprocessing XSLT stylesheet is therefore available[6] for creating very rich equivalents. This stylesheet adds absolute MML addresses to the resources used in the article, adds explicit proof levels to proof items, etc. Articles in such rich format are already suitable for a number of data-mining and presentational tasks, some examples are given in [Urb05c]. The new MPTP implementation also starts with this rich Mizar XML format as its input.

3 Export to Prolog and TPTP

The rich XML format of Mizar articles can be directly loaded into many Prolog (and other) systems, and processed as a tree structure. However the first processing step is the transformation to a TPTP-like [SS98] format, which is done by quite a straightforward processing of the XML tree. The XSLT[7] language (declarative functional language with lazy-evaluation) has been designed exactly for such purposes. So the whole MizarXML-to-TPTP transformation[8] is now written in about 700 lines of XSLTXT[9]. It provides all the Mizar-to-ATP translation functionalities (see [Urb04, Urb03]) done earlier by a special purpose exporter based on the Mizar implementation (called `fo_tool` in the first MPTP version). The most important changes and additions are mentioned below.

3.1 Syntax for Mizar types

One of the largest tasks of any Mizar-to-ATP export is dealing with the Mizar term-dependent types (i.e. types parametrized by terms). A more formal description of this type system is given in [Urb05b], where the axioms of *Mizar-like Horn theory* and *Mizar-like Horn theory with attributes* are stated. The following example (written already in the extended TPTP syntax) of matrix multiplication illustrates the usage of term-dependent types:[10]

```
![K:integer,A:matrix(K),B:matrix(K)]:
      sort(matrix_multiply(K,A,B), matrix(K)).
```

This means that such matrix multiplication is well-defined only for two square matrices of dimension K, and its result is also a K matrix. This definition has obvious

[6] `http://kti.ms.mff.cuni.cz/cgi-bin/viewcvs.cgi/xsl4mizar/addabsrefs.xsltxt?view=markup`

[7] `http://www.w3.org/TR/xslt`

[8] `http://kti.ms.mff.cuni.cz/cgi-bin/viewcvs.cgi/xsl4mizar/mizpl.xsltxt?view=markup`

[9] `http://www.zanthan.com/ajm/xsltxt/` - this is a compact syntax for XSLT stylesheets

[10] It is standard in Mizar to speak about *types*, while it is standard in TPTP to speak about *sorts*. We try to stick to the proper word in these two contexts, however these two words are completely equivalent in this article.

generalization for $M \times N$ and $N \times P$ matrices, with the result having the type $M \times P$ matrix. Semantically, the Mizar types are predicates, and the simplest translation method translates N-ary type symbols into $(N + 1)$-ary predicate symbol, and relativizes by such predicates (this means implication for universal quantification and conjunction for existential). For the above given example, the result (written in the standard TPTP notation) would be:

```
![K,A,B]: ( integer(K) & matrix(K,A) & matrix(K,B) )
     => matrix(K, matrix_multiply(K,A,B)) .
```

There are good reasons, why the MPTP formulas should keep the sorted syntax, and not use some direct translation of types as in the first MPTP version:

- There are alternative methods of type translation. E.g. [Dah98] suggests alternative inclusion-operator encoding of types.
- Various MPTP problem-generating stages (e.g. signature filtering) can take advantage of the knowledge that something is a type, and handle it differently from normal predicates.
- If the sorted syntax extensions to TPTP become standardized, some provers may eventually implement their own sort optimizations, and work directly with formulas in the extended syntax.

The current version of the TPTP-like dependent sort syntax used by MPTP is a result of discussions with Geoff Sutcliffe. It extends the quantification part of TPTP formulas, adds the special *sort/2* predicate for explicit expressing of sortedness, and introduces the *sort/0* formula kind for formulas encoding the sort hierarchy. The sorted quantifications now have following syntax (see also the matrix multiplication example above):

```
<quantified formula> ::= <quantifier> <sorted variables> : <literal formula>
<sorted variables> ::= [<variable> : <sort specification>
                        <rest of sorted variables>*]
<sort specification>  ::= <and-not formula>
```

Here `and-not formula` is a formula consisting of a conjunction of literals, that does not contain the *sort/2* predicate[11]. Other logical connectives could be allowed, however Mizar does not use them currently for types. An example of more advanced sorted quantification is e.g.:

```
! [G : (~ finite & graph), W1 : walk(G), W2 : subwalk(W1)]
```

Note that the syntax does not allow mixing sorted and unsorted quantifications. This is because e.g.:

```
! [I,K : integer]
```

has ambiguous interpretation as either

[11] All this is true if no abstract terms (see below) are present. Since abstract terms contain formulas, such definitions would become more complicated.

```
! [I : integer, K : integer]
```

or

```
! [I : $true , K : integer]
```

Therefore all MPTP quantifications are now sorted, and if a sort is not supplied, it has to be expressed using the `$true/0` atom. This atom is handled specially when the sort relativization is done, i.e. instead of `$true(I)` the sort is translated just to `$true`. This is a bit similar to Mizar, where the types have to be always specified, and if no particular type is wanted, the default type `set` must be used. The Mizar type `set` has no semantic content (everything is `set` in Mizar), so it is directly transformed to `$true` by our translation. Original unsorted TPTP quantifications like

```
! [I,J,K]
```

are currently not used, just for simplicity reasons.

The special Prolog *sort/2* predicate is used for expressing sortedness inside formulas, it is a TPTP equivalent of the Mizar (and ILF) *is/2* predicate. Its syntax is following (again, see the matrix example above):

```
<sorted atom> ::= sort(<term>,<sort specification>)
```

Here `<sort specification>` is defined as above. This Prolog predicate again has to be treated specially by MPTP when creating ATP problems (see the example above encoding the result type of matrix multiplication).

3.2 Sort handling in MPTP

The initial problem-generating functions only implement the predicate encoding of sorts. The following fragment of Prolog code does the translation.[12] As mentioned above, sort declarations with arity N are transformed into predicates with arity $N + 1$. Those in universal quantifications imply (relativize) the propositions, while existential must hold simultaneously (in conjunction) with the propositions that they quantify.

```
% return the list of quantified variables and conjunction of predicates
sort_transform_qlist([],[],$true).
sort_transform_qlist([X:S],[X],S1):- sort_transform(sort(X,S),S1).
sort_transform_qlist([(X:S)|T],[X|Qvars1],S1 & Preds1):-
        sort_transform(sort(X,S),S1),
        sort_transform_qlist(T,Qvars1,Preds1).

% end of term traversal
sort_transform(X,X):- atomic(X); var(X).

% relativization
sort_transform(! Svars : Y, ! Qvars : (Preds => Y1)):-
        sort_transform_qlist(Svars,Qvars,Preds),
        sort_transform(Y,Y1).
```

[12] Again, this is for formulas without abstract terms. Abstract terms are removed before sort transformations take place.

```
sort_transform(? Svars : Y, ? Qvars : (Preds & Y1)):-
        sort_transform_qlist(Svars,Qvars,Preds),
        sort_transform(Y,Y1).

% sort/2 predicate translation
sort_transform(sort(X,Y1 & Y2),Z1 & Z2):-
        sort_transform(sort(X,Y1),Z1),
        sort_transform(sort(X,Y2),Z2).
sort_transform(sort(X,~Y),~Z):- sort_transform(sort(X,Y),Z).
sort_transform(sort(_,$true),$true).
sort_transform(sort(_,$false),$false).
sort_transform(sort(X,Y),Z):- Y =.. [F|Args], Z =.. [F,X|Args].

% term traversal
sort_transform(X1,X2):-
        X1 =.. [H1|T1], maplist(sort_transform,T1,T2), X2 =.. [H1|T2].
```

3.3 Abstract terms

Abstract (or Fraenkel) terms are set-theoretical abbreviations for unique objects guaranteed by the Replacement and Comprehension axioms of ZFC. In Mizar they are written using the following syntax:

```
{ N + M where M,N is Integer : N < M }
```

The reasons for extending TPTP to handle abstract terms are very similar to those given for the sorted syntax. After some discussion with Geoff Sutcliffe, the special *all/3* Prolog predicate was chosen for the encoding. Its syntax is following:

```
<abstract term> ::= all(<sorted variables>,<term>,<literal formula>)
```

For instance:

```
all([M:Integer,N:Integer], plus(N,M), less(N,M))
```

As noted above, the existence of abstract terms is guaranteed in set theory by the Comprehension axiom (*"all members of some set satisfying some predicate form a set"*) and the Replacement axiom (*"image of a set under a function is again a set"*). Their uniqueness is guaranteed by the Extensionality axiom (*"two sets are equal if they contain the same elements"*). Note that Comprehension requires the quantified variables to be already members of some set, this is a method of preventing Russel's paradox used by set theory. Mizar checks this requirement by looking at the types of quantified variables, i.e. in the above example it has to know that `Integer` is a *small type*, i.e. a type whose extension is a set. More ingenious methods of checking correctness of abstract terms syntactically have been recently suggested by Arnon Avron [Avr04]. No such syntactic check is now used for the extended TPTP syntax, this is left to the formula providers, and possibly to the ATP systems that will implement this syntactic extension.

Abstract terms are very similar to lambda terms, which are sometimes called anonymous functions. Therefore we now call the process of removing abstract terms *deanonymization*. It is very similar to skolemization, and this is another reason why this syntactic extension could eventually become handled by standard ATP clausifiers, or even dealt

with in calculi which implement delayed transformation to normal forms (e.g. tableaux or [GS03]). It means that we introduce a new functor symbol, corresponding to the abstract term in the following way:

```
![X]:(in(X,all_0_xx) <=> ?[N:Integer,M:Integer]:(X = plus(N,M) & less(N,M))).
```

Here *all_0_xx/0* is the newly introduced "fraenkel" functor for the abstract term given above, the first number in it (0) is its arity and the second number (xx) just a serial numbering of such symbols with the same arity[13]. Obviously fraenkel functors with nonzero arity can arise if their context includes quantified variables, this is similar to skolemization. The predicate *in/2* (set-theoretic membership) would have to be reserved for this purpose too in TPTP.

As with skolemization, a lot of optimizing steps can be done during deanonymization. If one abstract term is used twice, only one fraenkel functor is necessary. This has the additional advantage that the equality of such terms is obvious, while for different fraenkel functors the Extensionality axiom has to be used to find out that they encode the same term. Since the deanonymization algorithm is likely to include more such optimizations in the future, it has been split into two independent parts. The first part just collects the abstract terms from formulas together with their contexts, and replaces them by new Prolog variables. This collecting has to be quite careful, because abstract terms can (and do) appear at any position where normal terms are allowed, i.e. also inside sort specifications or inside other abstract terms. After this part, various optimizations can be done on the collected abstract terms and finally the fraenkel functors are introduced, their definitions created, and the new Prolog variables are instantiated with the corresponding fraenkel functors.

3.4 Export of proofs

The full translation of proofs from XML to TPTP is still work in progress at the moment, however a part which is sufficiently large for initial reproving experiments described below has already been translated. The Mizar proofs consist of various reasoning items implementing various Jaskowski-style natural deduction steps. A proof starts with a thesis equal to the formula that is being proved, and various *Skeleton items* (e.g. assuming the antecedent of a thesis which is an implication) are used to modify the thesis. The proof is successful when the thesis is reduced to verum. The *Skeleton items* operate on the thesis, and they must correspond to the current structure of the thesis when they are used. This is checked by a simple part of the Mizar verifier called Reasoner. The *Auxiliary items* (e.g. proving some useful lemma) do not operate on the thesis, and they can be intermixed with the *Skeleton items* freely in the proofs. Many items (both skeleton and auxiliary) require a justification. Such justification can be either a full subproof, or a *Simple Justification* saying that the current proposition "easily follows" from several other propositions. The phrase "easily follows" refers to the limited Mizar refutational prover. The initial goal of the translation was to make it sufficiently complete for reproving these *Simple Justifications*. For this the following information had to be translated to TPTP:

[13]This format of fraenkel functors was chosen after discussion with Geoff Sutcliffe and Stephan Schulz, who uses similar numbering scheme for skolem symbols.

- All the propositions introduced by the various reasoning items, together with their justifications.
- Information about the constants, functors and predicates that are created locally inside proofs. This includes information about their types, and their definitions.

We do not mention the translation of the globally available Mizar constructs (e.g. functors, predicates, theorems, definitions, etc.) that are also needed for this, because their handling is very similar to the first MPTP version.

3.4.1 Export of propositions

Even though propositions can be introduced by various reasoning items, their separate translation is easy, because all are tagged with the `<Proposition>` tag in the Mizar XML format[14] The TPTP syntax used for encoding Mizar propositions is following:

```
<proposition> ::= fof(<proposition name>,<type>,
                      <fol formula>,<source>,<mptp info>).
<proposition name> ::= e<serial number> {_ <proof level>}
% serial number on the current proof level
<serial number> ::= <unsigned integer>
% encodes path to the current proof block
<proof level> ::= <unsigned integer> {_ <proof level>}
<type> ::= lemma-derived | unknown
<source> ::= file(<name of article>, <proposition name>)
<mptp info> ::= mptp_info(<serial number>,[{<proof level>}],
                  proposition(<line>,<column>,<mizar number>)
                  {, inference(mizar_by,[],[{ <reference names> }]) })
<reference names>::= <reference name> {, <reference names>}
<reference name> ::= <proposition name> | <theorem name> | <definition name>
<theorem name> ::= t<unsigned integer>_<name of article>
<definition name> ::= d<unsigned integer>_<name of article>
<name of article> ::= <lower word>
% original line and column in the mizar article
<line> ::= <unsigned integer>
<column> ::= <unsigned integer>
% original mizar numbering of propositions
<mizar number> ::= <unsigned integer>
```

For instance:

```
fof(e3_17_1,lemma-derived,
   ( k3_xcmplx_0(c2_17,1) =
     k3_xcmplx_0(c3_17,k3_xcmplx_0(c1_17,k5_xcmplx_0(c1_17))) ),
   file(xreal_1,e3_17_1),
   mptp_info(3,[17,1],proposition(591,26,0),
             inference(mizar_by,[],[e2_17_1,e1_17,d7_xcmplx_0]))).
```

encodes the Mizar proposition proved on line 591 of article XREAL_1:

```
then a*1=b*(c*c") by A2,XCMPLX_0:def 7;
```

[14]The Mizar *Iterative equalities* and *Diffuse statements* actually create propositions too, and they are handled very similarly by MPTP, but for simplicity we do not consider them here.

Some more changes will be probably done to this encoding, we may eventually include the kind of reasoning item which introduced the proposition, or add the article background theory (containing e.g. the sort information) explicitly to the inference, so it would like:

```
inference(mizar_by,[],[e2_17_1,e1_17,d7_xcmplx_0,theory(xreal_1)]).
```

3.4.2 Export of local constants

Local constants can be introduced by several reasoning items of the Jaskowski-style proofs. Each has assigned a type, and sometimes they are defined as being equal to some other term. This is typically used when some term is proved to have some non-obvious type, and we want the Mizar checker to remember that typing. Since the checker first does congruence closure of all ground terms, such equalities actually can be used to provide multiple types for terms when necessary. The numbering of local constants is again according to the proof level and their serial number on their proof level. The syntax would be similar to that of propositions, we just give an example here:

```
fof(dt_c6_16,sort,sort(c6_16,m1_subset_1(k1_numbers)),
    file(xreal_1,c6_16),mptp_info(6,[16],constant(reconsider,type))).
fof(de_c6_16,definition,( c6_16 = c3_16 ),
    file(xreal_1,c6_16),mptp_info(6,[16],constant(reconsider,equality))).
```

The first clause expresses the type of the local constant, while the second expresses its definitional equality to another term. Similar descriptions are used for local functors and predicates.

3.5 Problem creation

The creation of reproving problems corresponding to *Simple Justifications* is done in the following steps:

1. Collect the references mentioned in the `inference` slot of the Mizar-proved proposition.
2. Collect all symbols from the proposition and its references.
3. In a fixpoint manner, add the background theory formulas for these symbols (e.g. sort formulas, formulas expressing properties like *reflexivity* or *antisymmetry*, etc.).
4. Create fraenkel functors for all formulas obtained in the previous step, replace by them all the abstract terms appearing there, and add the formulas defining the fraenkel functors.
5. If a fraenkel functor was introduced, add the Extensionality axiom.
6. Do the sort relativization of all formulas.

All this is now done in about 500 lines of Prolog code[15]. The switch from Perl to (SWI) Prolog was necessitated mainly by the need to implement the functions for deanonymization and sort relativization. From the implementational point of view, having all parts

[15]This code is going to be quite unstable for some time, since other problem creating functions are being added. One version is available at `http://kti.mff.cuni.cz/~urban/MPTP/utils.pl`

of MPTP encoded by one Prolog *fof/5* predicate makes problem creation quite slow. SWI Prolog allows simple indexing on four arguments, and if in the fixpoint computation some formula is searched for by the contents of its *mptp_info* slot, a sequential scan is used by SWI. Since tens of thousands of formulas are typically loaded, this is very inefficient. This is a price for trying to be TPTP-compatible, but it can obviously be helped by additional indexings. Such indexes are now available for the formula kinds used most frequently in the fixpoint computation, however generation of 100 problems can still take about 1 minute on 3GHz Pentium 4. Additional profiling and indexing will be needed, since the number of Mizar *Simple Justifications* is almost 600.000.

4 Initial reproving experiments

4.1 Reproving Simple Justifications in 100 initial articles

For the initial experiments with reproving the *Simple Justifications* only the first 100 Mizar articles were selected. About 40.000 reproving problems can be generated from these articles. The E prover version 0.82 was used, and only with 10 second timelimit due to limited resources. On the other hand, the hardware was a cluster of dual Intel Xeons 3.06 GHz with 2GB RAM each, which offsets a bit the low timelimit. The following table shows the results of this experiment: The success rate is 81 %. The

proved	completion found	timeout	total
31286	780	6661	38727

Table 1: Results of the reproving experiment on 100 articles

algorithm for adding the background theory to the reproving problems is now very complete, and we believe that the only remaining source of the 780 completions are the arithmetical evaluations done by the Mizar checker. Another advantage of having the MPTP implemented in Prolog is that we can attempt to mimic these evaluations when the background theory is added, and that will probably be done in near future. Another possibility is to explore the newly available handling of arithmetics in several ATP systems, and map the Mizar arithmetical symbols to their counterparts in those systems. These arithmetical evaluations are now quite frequent in Mizar, it is quite likely that they are also responsible for a lot of the timeouts.

4.2 Evaluation of the encoding of abstract terms

There are 1477 problems containing abstract terms among the ca 40,000 problems extracted from the initial 100 articles. As mentioned above, the simplest encoding creates a new fraenkel functor for each term appearing in the problem. This is likely to be often inefficient, since an abstract term can be used more than once in the problem, and the only way how to find out that the corresponding different fraenkel functors are equal is through the Extensionality axiom. The definitions of such "different" fraenkel functors are however almost the same, and a clever clausifier should be able to discover this similarity. After the first simplest implementation, a more advanced version recognizing the same abstract terms was written, and performance on these two kinds of translations

could be compared. The SPASS 2.1. prover was used in addition to E prover, since its clausifier is probably still the best available. The following table shows the results of the experiments run on the same hardware as above with 30s timelimit:

description	proved	completion found	timeout	total
E, no optimization	1019	0	458	1477
E, with optimization	1143	0	334	1477
SPASS, no optimization	1098	9	370	1477
SPASS, optimization	1203	9	265	1477

Table 2: Results of the experiment with different encoding of abstract terms

For both provers, encoding the same abstract terms by the same fraenkel functor has helped quite significantly. The improvement is 124 problems (8.4 percent of the total 1477 problems) for E, and 105 problems (7.1 percent) for SPASS. SPASS performs significantly better on these problems, which is probably partially caused by its handling of sort theories, and also by its optimizing clausifier. The lower increase in the SPASS performance on the optimized encoding might be caused exactly by the capability of its clausifier to discover similarities in the fraenkel functor definitions that encode the same term in the simple encoding.

4.3 Reproving Simple Justifications in 48 articles without numbers

The Mizar checker used for proving the Simple Justifications is quite simple, and it should not pose serious difficulties to current ATP systems. This seems to be generally true, since the average user time reported by E on the 40,000 problems which did not time out in the first experiment is 0.26 s. In the last experiment so far, a hard attempt to prove all the Simple Justifications from 48 of the initial 100 articles which do not contain any arithmetical evaluations was conducted. The usage of arithmetical evaluations has to be switched on in Mizar by special directives, so articles without these directives should be fully reprovable (and specifically no completions should be found), if the translation (and Mizar) is correctly implemented.

First, the E prover was run with 4s timelimit on all the 18429 problems extracted from the 48 articles. It solved 17022 of them (92 percent) within the timelimit, and found no completion. On the remaining unsolved 1407 problems, the SPASS prover was run with 60s timelimit. SPASS solved 940 of these harder problems, leaving 467 problems unsolved, no completion was found. The combined success rate at the moment was 97.5 percent. However running both SPASS and E with higher timelimits on the remaining 467 problems helped only very little. An overview of the hard problems has indicated, that in many of them the background theory (formulas encoding the type information, various properties of functors and predicates, nonemptiness of types, etc.) has grown quite large, and causes the provers to delay the right inferences. Since we now add only as little background as possible, changes to the algorithm described in 3.5 can already cause incompleteness. So instead of such changes, a general machine learning solution (very similar to that used in [Urb04]) was taken:

The proofs of the 17962 solved problems were analysed and the background formulas used in them remembered for each problem, together with the global symbols

(i.e. excluding the local constants, etc.) appearing in it. Typically, only a few (up to ten) background formulas are needed for these Simple Justification problems, and they are very much correlated to the problem signature. The SNoW system (a multi-class Bayesian classifier [CCRR99]) was trained on these examples to associate the most promising background formulas with the symbol signature. The background formulas of the remaining 467 hard problems were then filtered by this trained advisor. In the first pass, only three most relevant (as judged by the advisor) background formulas were allowed, and in the second pass the number was raised to six. Such specifications are therefore potentially incomplete, however if contradiction can be found in them, it should be much faster than for the complete specifications. This turned out to be very efficient: After running E on the problems created in the first pass, only 69 problems were unsolved, and the solutions was typically found very quickly (below 1s). The second pass left 75 problems unsolved, only two successes were added to the results of the first pass. We are therefore left with 67 out of the 18429 problems (this is ca 0.4 percent) that are not yet proved before submitting the final version of this articles. This number is low enough, to try manual optimizations if necessary, however these problems will more likely be used for suggesting other automated techniques that will make their solution simpler, and obviously also for the analysis of the correctness of the MPTP and Mizar systems.

5 Conclusion and future work

Some of the working objectives set for MPTP in the introduction have been already quite fulfilled. The export of abstract terms is now correct, even though it could be more optimized. To the best of our knowledge, there are no other incorrectly translated Mizar constructs left. The extended TPTP format already allows recognition and different translations of both the abstract terms and the dependent sorts. The Mizar proofs are now completely exported into XML, but the export from XML to TPTP is only partial yet. Such export however turns out to be quite easy, and the estimate is that it will be done on another couple of hundred lines of XSLTXT.

The more demanding part is actually choosing the TPTP encoding in the right way. E.g. the initial TPTP suggestions for sort syntax did not consider the dependent case, and it is still a question whether the syntax introduced by MPTP will be acceptable. Similar problems are with other parts of TPTP. E.g. "sort" will probably be allowed as a TPTP user type of the sort hierarchy formulas, however a sort hierarchy formula can be both a definition or a theorem, which are other TPTP user types. We will have to add to TPTP the derivation rules for the Jaskowski-style natural deduction. Some inspiration for this can be taken from the E-prover's clausifier, which already has names for the clausification steps. The clausification steps also escape from the simple refutational-proving setting.

Another line of work is the cross-verification. The last experiment documents well the interplay between cross-verification and proving new things expressed in the MPTP language. While cross-verification is useful per se, and for all kinds of MPTP (and Mizar) debugging, the information (here the necessary background formulas) obtained from successful proof attempts can be used as a guidance for proving new problems,

or problems which are too hard to prove in the default way. The inductive aspect, i.e. the possibility to learn from tens of thousands of problems expressed consistently in the same language, has turned out to be a decisive help for the set of hardest problems.

Many other experiments made available by MPTP are planned, see [Urb04] for a more detailed enumeration of the possibilities. One short-term goal is the inclusion of more of the MPTP problems into the TPTP library.

6 Acknowledgments

As already mentioned, Geoff Sutcliffe has had quite a big influence on the current encoding of various MPTP constructs in TPTP. Stephan Schulz has also participated in some of these discussions, and helped with setting up the E prover. The ILF project still serves as a very good example for many Mizar-to-ATP issues. The people involved in the Mizar part of ILF were mainly Ingo Dahn, Christoph Wernhard and Czeslaw Bylinski. The XML-ization of Mizar underlying this MPTP version was thoroughly discussed within the Mizar team, and helped by Czeslaw Bylinski. The first version of this article has been largely modified after many suggestions from the ESCAR reviewers. Thanks for all their suggestions and help.

This work was partially supported by the Charles University research grants (205-03/2060985, 205-10/203336). The resources for the reproving experiments were provided by the Czech METACentrum supercomputing project.

References

[ABT04] Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors. *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21, 2004, Proceedings*, volume 3119 of *Lecture Notes in Computer Science*. Springer, 2004.

[Avr04] Arnon Avron. Formalizing set theory as it is actually used. In Asperti et al. [ABT04], pages 32–43.

[BR03] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in MML. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2003.

[BU04] Grzegorz Bancerek and Josef Urban. Integrated semantic browsing of the Mizar Mathematical Library for authoring Mizar articles. In Asperti et al. [ABT04], pages 44–57.

[Byl90] Czeslaw Bylinski. The complex numbers. *Formalized Mathematics*, 2(2), 1990.

[CCRR99] A. J. Carlson, C. M. Cumby, J. L. Rosen, and D. Roth. Snow user's guide. Technical Report UIUC-DCS-R-99-210, UIUC, 1999.

[Dah98] Ingo Dahn. Interpretation of a Mizar-like logic in first-order logic. In *FTP (LNCS Selection)*, pages 137–151, 1998.

[DW97] Ingo Dahn and Christoph Wernhard. First order proof problems extracted from an article in the MIZAR Mathematical Library. In Maria Paola Bonacina and Ulrich Furbach, editors, *Int. Workshop on First-Order Theorem Proving (FTP'97)*, RISC-Linz Report Series No. 97-50, pages 58–62. Johannes Kepler Universität, Linz (Austria), 1997.

[GM92] Joseph A. Goguen and José Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[GS03] Harald Ganzinger and Jürgen Stuber. Superposition with equivalence reasoning and delayed clause normal form transformation. In *CADE*, pages 335–349, 2003.

[HKW96] R. Hähnle, M. Kerber, and C. Weidenbach. Common syntax of the DFGSchwerpunktprogramm deduction. Technical Report TR 10/96, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, 1996.

[Jas34] S. Jaskowski. On the rules of suppositions. *Studia Logica*, 1, 1934.

[MR04] Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. In Grzegorz Bancerek, editor, *MKM Workshop on 30 Years of Mizar*, 2004.

[NB04] Adam Naumowicz and Czeslaw Bylinski. Improving mizar texts with properties and requirements. In Asperti et al. [ABT04], pages 290–301.

[NW01] A. Nonnengart and C. Weidenbach. *Handbook of Automated Reasoning*, volume I, chapter Computing small clause normal forms., pages 335–367. Elsevier and MIT Press, 2001.

[Pel99] F. J. Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20:1 – 31, 1999.

[RT99] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. *J. Autom. Reasoning*, 23(3-4):197–234, 1999.

[Rud92] P. Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*, pages 311–332. Chalmers University of Technology, Bastad, 1992.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *Journal of AI Communications*, 15(2-3):91–110, 2002.

[Sch02] S. Schulz. E – a brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Urb03] Josef Urban. Translating Mizar for first order theorem provers. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 203–215. Springer, 2003.

[Urb04] Josef Urban. MPTP - motivation, implementation, first experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

[Urb05a] Josef Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic*, 2005. forthcoming, available online at http://ktiml.mff.cuni.cz/∼urban/mizmode.ps.

[Urb05b] Josef Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 2005. forthcoming, available online at http://ktiml.mff.cuni.cz/∼urban/MoMM/momm.ps.

[Urb05c] Josef Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. submitted to MKM 2005, available online at http://ktiml.mff.cuni.cz/∼urban/mizxml.ps, 2005.

[Wei01] C. Weidenbach. *Handbook of Automated Reasoning*, volume II, chapter SPASS: Combining Superposition, Sorts and Splitting, pages 1965–2013. Elsevier and MIT Press, 2001.

[Wie00] Freek Wiedijk. CHECKER - notes on the basic inference step in Mizar. available at http://www.cs.kun.nl/∼freek/mizar/by.dvi, 2000.

[Wie03] Freek Wiedijk. Comparing mathematical provers. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003.

The Arrival of Automated Reasoning

Larry Wos,[1]
M. Spinks[2]
[1]Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439
wos@mcs.anl.gov
[2]Department of Philosophy
La Trobe University, Bundoora, Vic 3083 Australia
mspinksau@yahoo.com.au

Abstract

For some, the object of automated reasoning is the design and implementation of a program that offers sufficient power to enable one to contribute new and significant results to mathematics and to logic, as well as elsewhere. One measure of success rests with the number and quality of the results obtained with the assistance of the program in focus. A less obvious measure (heavily in focus here) rests with the ability of a novice, in the domain under investigation, to make significant contributions to one or more fields of science by relying heavily on a given reasoning program. For example, if one who is totally unfamiliar with the area of study but skilled in automated reasoning can discover with an automated reasoning program impressive proofs, previously unknown axiom dependencies, and far more, then the field of automated reasoning has indeed arrived. This article details such—how one novice, with much experience with W. McCune's program OTTER but no knowledge of the domains under investigation, obtained startling results in the study of areas of logic that include the *BCSK* logic and various extensions of that logic. Among those results was the discovery of a *variety* weaker than has been studied from what we know, a variety that appears to merit serious study, as, for example, does the study of semigroups when compared with that of the study of groups. A quite different result concerns the discovery of a most unexpected dependency in two extensions of the *BCSK* logic.

1 Setting the Stage

When a researcher, who is a master of some field, uses an automated reasoning program and finds a proof of a significant theorem in said field, applause is more than appropriate. That success contributes to the mystique of automated reasoning, providing yet one more bit of evidence that substantial progress has occurred. Evidence of this type exists that includes studies of K. Kunen [Kun92], D. Phillips [Phi], and J. Belinfante [Bel01]. An expert, if the program in use provides the appropriate means, can give advice, make enlightened conjectures, and otherwise restrict and direct the program's attack in a manner that sharply increases the likelihood of success. For example, through the use of R. Veroff's hints strategy [Ver96] or with the resonance strategy [Wos95],

one can guide the program toward or away from paths of reasoning. In addition, the researcher can restrict the program's attack by instructing it to avoid certain lemmas and certain types of term (through the use of demodulation) and, most effective, block the program (with the set of support strategy [WRC65]) from applying inference rules to sets of hypotheses whose consideration could bury the program in irrelevant conclusions. Sometimes a paper results, stating clearly that automated reasoning played an important role, perhaps a vital role. Even better is the case when the paper is published in a journal devoted to mathematics or to logic rather than to automated reasoning.

In contrast, one might consider the case in which a novice, an amateur, in the field of study in focus makes important discoveries by relying on a reasoning program. (By a novice is meant one who knows nothing of the area under investigation, but one who may know much about automated reasoning.) If the discoveries include impressive proofs, previously unknown axiom dependencies, and far more, a landmark has been reached, one that predicts greatness for the future of automated reasoning. This article offers a story of such discoveries, a story of a novice studying the *BCSK* logic, as well as extensions of that logic, with absolutely no knowledge of the fields under study.

At this point, we briefly provide some of the underlying formalism. Recall from [BP94] that the *fixedpoint discriminator* on a set A is the function $f : A^3 \to A$ defined for all a, b, c in A by

```
f(a, b, c) = c if a = b
             1 otherwise
```

for some element $1 \in A$. The element 1 is called the *discriminating element*. The fixedpoint discriminator arises naturally in algebraic logic as a generalization of the ternary discriminator; see, for instance, [BP94].

The *generic fixedpoint discriminator variety*, in symbols FPD_1, is the variety generated by the class of all algebras $\langle A; f, 1\rangle$ of type $\langle 3, 0\rangle$, where f is the fixedpoint discriminator on A and 1 is a nullary operation, the range of which is the discriminating element of f. The 1-assertional logic of FPD_1, in symbols $S(FPD_1, 1)$, is the consequence relation from sets of terms to terms determined by the equivalence $\Gamma|-_{S(FPD_1,1)}\phi$ if and only if $\psi = 1 : \psi \in \Gamma \models_{FPD_1} \phi$. Since FPD_1 is a variety $|-_{S(FPD_1,1)}$ and is both finitary and substitution invariant, and hence is a deductive system in the sense of Blok and Pigozzi [BP99], our interest in *BCSK* logic stems from the observation, made in [BSV], that it is formula equivalent to $S(FPD_1, 1)$.

The following nine axioms, for the *BCSK* logic, initiated the study, where the functions i and j denote *strong* and *weak* implication, respectively.

```
P(i(x,i(y,x))).                          %  A1
P(i(i(x,i(y,z)),i(i(x,y),i(x,z)))).      %  A2
P(i(i(i(x,y),x),x)).                     %  A3
P(i(x,j(y,x))).                          %  A4
P(i(j(x,j(y,z)),j(j(x,y),j(x,z)))).      %  A5
P(i(j(x,j(y,z)),j(y,j(x,z)))).           %  A6
P(i(j(j(x,y),x),x)).                     %  A7
P(i(j(i(x,y),y),j(i(y,x),x))).           %  A8
P(j(i(x,y),j(x,y))).                     %  A9
```

The nature of the contributions in focus here strongly suggests that automated reasoning has indeed arrived.

Detailed is the adventure that began with a set of axioms and target theorems in logic, a collaboration of one expert in that logic with another expert in automated reasoning—but truly a novice in the logic—and some hard-to-find proofs mostly supplied by Veroff using his powerful sketches [Ver01] approach. (More generally, an individual knowing essentially nothing about a field to be studied—but having much knowledge and experience with a reasoning program in hand—can make significant contributions to that field. Further, if the corresponding logical formulation is available, one who is a novice in logic or mathematics can fearlessly seek one valuable proof after another with the expectation of some or much success by relying on a program that offers a variety of strategy. At the other end of the spectrum, one who possesses substantial knowledge of the field to be studied but knows little of automated reasoning can also succeed.) A novice in the field under study has the advantage of not being trapped by knowledge of how one typically proceeds (perhaps, implicitly, must proceed) and can, therefore, follow paths not previously explored. The automated reasoning program has this advantage, for it knows nothing of any field and lacks bias or orientation—such a program is indeed a novice. For example, one can instruct the program to totally avoid some type of term or some lemma that the literature suggests must be relied upon. Such instruction can lead to most satisfying and wondrous discoveries. Well demonstrated here is the current state of automated reasoning in that, eventually, (as one learns here) startling results were obtained.

Our objective was to find "short" proofs, and seeking that objective led to marvelous and unexpected discoveries, which is the basis of the story to unfold. Not surprising, especially to the person familiar with the new book *Automated Reasoning and the Discovery of Missing and Elegant Proofs* [WP03] and the strategies and methodologies it offers, a number of short proofs were in fact completed. More pertinent to this article, unknown axiom dependencies were found, a new concept was formulated—*proof dependence*—and a variety was unearthed that may merit serious study. In particular, by way of a foretaste, of the nine axioms that prompted the original study, two were proved to be dependent, $A3$ and $A6$. These two axioms as well as $A7$ were shown to be totally avoidable (even as deduced formulas) for completion of the proofs being sought, which revealed a promising weaker variety to consider. This variety is axiomatized with axioms 1, 2, 4, 5, 8, and 9.

When one is introduced to some field of mathematics or logic, one is typically presented with a set of axioms from which the theorems are deducible. And that is how this story begins, with nine axioms of the *BCSK* logic. A glance at the set of axioms (of the area in focus) often does not readily reveal which, if any, are dependent on the remaining. For example, if one is introduced to group theory with the axiom set consisting of associativity of product, the existence of a two-sided identity element e, and, with respect to the identity, a two-sided inverse, one might not immediately see that dependencies exist among the given five axioms. But, they are present. Indeed, the axioms of right inverse and right identity are each dependent on the remaining three; equally, those of left inverse and left identity are dependent.

The proofs of the cited dependencies are well within reach of various automated reasoning programs or well within reach of the unaided researcher. With W. McCune's

program OTTER, one can simply negate the axiom to be proved dependent, place the negation in the passive list, place the other axioms in the initial set of support list, and seek (when the notation is equational) with the inference rule paramodulation a proof by contradiction, which, at least for group theory, will be in hand almost at once.

Rather than the deducibility from the remaining axioms, the key focus for this article about dependence is that, at least axiomatically, the dependent item is not needed (in the input). However, a dependent axiom might still be required to complete one or more proofs of interest, needed at the deduced level. Here, as one learns, we are concerned with items that are not needed even at the deduced level, a topic that is featured as we introduce the notion of *proof dependent*. For a foretaste of what is to come, we note that the total avoidance of some thought-to-be indispensable lemma when seeking to complete a proof of a theorem of substantial interest can be challenging. Because of the nature of dependence and the importance of axioms, to totally avoid the use of an independent axiom may be far more challenging and, if successful, may mark the beginning of a study of a weaker variety (field), as in the case of the study of groups versus semigroups. Further, when compared with avoiding the use of some lemma, more interest may rest with the total avoidance of some axiom; after all, axioms are not typically thought of as lemmas.

The axiom in focus need not be a dependent axiom. For example, we might begin with a three-axiom system consisting of independent axioms and seek proofs in which one of the three is selected to be avoided. If we find such proofs, for each of the corresponding theorems, we say we have established proof dependence, because we have shown the selected item to be unnecessary. (The term *proof dependent* is intended to suggest to one that its establishment for a particular formula or equation depends on finding an appropriate proof.) When the selected item is an independent axiom and we are, nevertheless, able to prove one significant theorem after another without its participation (at the deduced level)—proof dependence is present—then we might be in the presence of a weaker theory that merits study. For a well-known situation, one need only consider group theory and its weakening to that of the theory of semigroups, where certain group-theory axioms are dropped. In this article, by offering proofs that totally avoid the use of a key independent axiom ($A7$), we offer a theory (that might merit study) weaker than the *BCSK* logic. For extensions of that logic, we found and offer proofs in which $A7$ is totally avoided, as well as a proof of its dependence that was indeed unexpected. These proofs provide powerful evidence that automated reasoning has arrived and that one with little or no knowledge can find treasure.

A second example, relevant to proof dependency, nicely illustrates one of the limiting points. Let us consider a logic in which condensed detachment is the only rule of inference such that the logic is studied in terms of a single axiom A. Let F denote the formula obtained by applying condensed detachment to two copies of A. Every proof of length greater than or equal to 1 in this study must have as its first step F. In other words for proofs of nonzero length, one cannot dispense with F. Therefore, F is never proof dependent (because it is always needed), regardless of the theorem under consideration when its proof requires at least one deduced step for its completion.

Hilbert himself might have been interested in proof dependence. Indeed, many of us learned as students of the famous 1900 lecture by Hilbert in Paris, a talk in which he offered twenty-three problems for study. As it turns out, a twenty-fourth problem exists,

one he said in his notes that he did not have time to adequately formulate for the Paris lecture. (For that find, thanks goes to R. Thiele [TW02] and his thorough examination of Hilbert's notebooks.) That problem focuses on finding simpler proofs. A proof can be simplified in many ways, including shortening, removing a messy formula or equation, or avoiding some type of term. Also, pertinent to this article, a proof can be simplified by avoiding in all senses some axiom; if said axiom is independent, so much the better and more intriguing. We have a 30-step proof for the dependence of the fifth of the five Łukasiewicz axioms for his infinite-valued sentential calculus that is simpler than the original Meredith proof in various ways. For example, it is shorter, and, surprising to many, it avoids the use of any *double-negation terms*, terms of the form $n(n(t))$ for some term t. (The book citeWos2003b, offers the 30-step proof and many others of its type and features in detail various refinement methodologies. The book also offers open questions and challenges, in Chapter 7, that readers may find interesting.)

In the spirit of Hilbert's twenty-fourth problem is finding a proof that relies on fewer axioms than that in hand. If, for example, one has a proof P of a theorem T that relies on a set of axioms that include dependent axioms, and if one removes the dependent axioms to produce a set S of independent axioms, then there must exist a proof Q from S of the theorem T. The proof Q is simpler than the proof P in an axiomatic sense.

Of course, the absence of a dependent axiom in a set of hypotheses says nothing about its absence in a proof obtained from the so-called smaller (independent) axiom set. In this article, offered is methodology for finding proofs in which the dependent axioms not only are avoided as hypotheses but also are absent among the deduced steps of the proof, are, therefore, proof dependent. The inference rule or rules being employed are taken into account. This article details how such proof-dependent items were found with substantial aid from OTTER. The method that is given is extended to finding proof-dependent items even when the axiom in focus is in fact independent. When one finds that an independent axiom is proof dependent for a number of interesting theorems, then (as noted) the variety obtained by its omission may merit serious study. Proofs that avoid reliance on independent axioms might indeed have been of interest to Hilbert for they are simpler in an important way.

Finally, good fortune, occurring during the search (in the spirit of the new Hilbert problem) for short proofs, takes center stage as the experiments are discussed that led to the discovery of certain dependencies among the original nine axioms that in turn provided the wellspring for the study reported here. Here the article offers proofs that establish proof dependence for various axioms of the *BCSK* logic, as well as for some of its extensions. These proofs support the position that a weaker logic, obtained by the omission of the axiom $A7$, might offer unexpected power and interest.

2 A Wellspring for Ideas

The entire article came into being because of an attempt to find pleasing proofs for the following three theses (theorems), each given in its negated form.

```
-P(i(i(A,B),j(A,B))) | $ANS(THESIS_1).
-P(j(i(A,B),i(j(B,C),j(A,C)))) | $ANS(THESIS_2).
-P(j(i(B,C),i(j(A,B),j(A,C)))) | $ANS(THESIS_3).
```

The study was based on the nine axioms (for the *BCSK* logic that were given in Section 1) as hypotheses. (One might find interesting the fact that the first three axioms serve well for a complete system for the implicational fragment of intuitionistic logic.) Rather than seeking first proofs—three were in hand from Veroff—the object was to find shorter proofs, perhaps far shorter. Because of the nature of the first thesis, namely, it plays the role of a key lemma in a paper under consideration [BSV], the goal was to prove it by itself. The other main goal was to find a short proof of the join of the other two theses, 2 and 3.

Of the aspects of the approach to proof refinement with respect to length, two were prominent. First, we used ancestor subsumption, which causes the program to compare pairs of paths to a conclusion, preferring the shorter derivation. Second, we used demodulation to block steps of a proof, one at a time, to prevent their participation. (Demodulation is typically used for simplification and canonicalization.) By blocking the use of some given step, the program is forced to seek other paths to a proof—and it often is a most effective move to make when seeking shorter proofs.

3 Consequences of the Refinement Phase

As we made progress in finding ever shorter proofs, one of them was of particular note. Specifically, it failed to rely on $A3$ as a hypotheses. In other words, we had a proof from a smaller set of axioms, a set in which $A3$ was omitted.

The next move was to remove $A3$ from the axioms (by commenting it out in the initial set of support list). Somewhat later, we had in hand an even shorter proof, one with a most unexpected property. This proof relied on $A3$—as a deduced step. OTTER thus had established $A3$ to be dependent on the remaining eight of the nine original axioms. Now one sees why, near the close of Section 1, a reference was made to the discovery by good fortune of axiom dependencies.

Because we had found a satisfying proof in which $A3$ was not relied upon as an axiom, but was relied upon as a deduced step, we decided to seek a proof that totally avoided its presence—and the concept of *proof dependent* was born. The approach chosen, which succeeded, was to block, by demodulating unwanted new conclusions to "junk", the retention of $A3$ when and if it was deduced. OTTER later found a 14-step proof (which we give) of the dependency of $A3$, a proof relying on but six of the nine original axioms, omitting $A3$ (of course) but also omitting $A6$ and $A7$.

Stimulated by the discovery of a dependency within the original set of nine axioms, we sought to find other dependencies, focusing on $A6$ perhaps because of its position within the axiom set. In particular, $A3$ is the third of the given axioms concerned exclusively with the function i, and $A6$ is the third of those concerned almost exclusively with the function j. Again, our approach was to comment out $A6$ in the input, and we found appropriate proofs. We thus knew that $A6$ was not needed, at the axiomatic level, to find proofs of the three given theses. Eventually, we had a nice proof of the first of the three theses in which neither $A3$ nor $A6$ was used as an axiom. In that proof $A3$ was not present as a deduced step, but $A6$ was.

We paused before resuming the main journey to seek a nice proof of the dependence of $A6$ on seven of the nine original axioms, with $A3$ not participating. OTTER found

one, a proof of length 27 (not given here) relying (as was the case for $A3$) on but six of the nine original axioms, omitting totally the use of $A3$, $A6$, and $A7$.

We therefore resumed the main journey, seeking a proof in which $A6$ was totally avoided, again relying on demodulating unwanted formulas to junk. The various attempts failed, which (in effect) brings us to the methodology that was promised for this article.

To put all in perspective, a review is in order. OTTER had succeeded in completing satisfying short proofs of the dependency of both $A3$ and $A6$ on the remaining seven axioms of the nine that prompted the study. We had a proof in which $A3$ participated in no way, $A6$ was not relied upon as an axiom, but $A6$ was present as a deduced step. Further, all attempts at completing a proof in which $A6$ was totally absent and all of the other given conditions were met had failed—with the numerous standard approaches we take.

We were thus forced to depart from our usual practice, that of paying little or no attention (in the vast majority of studies) to the actual proofs themselves. More precisely, our typical approach does not call for a close examination of a completed proof, in detail or as a whole. Instead, we rely on years of experimentation for a feel for which options and which values, if assigned to parameters, are likely to enable the program to complete a given assignment. In other words, we have found that the reading of a proof usually sheds little or no light on how one might proceed to refine it. Instead, such a reading can play a role in the formulation of new strategies and new methodologies that apply to many areas.

The so-to-speak forced inspection of the proof in hand that was the focus of attention showed that $A6$ was used as a parent for only one formula that followed its derivation. In that none of the standard approaches had enabled OTTER to find the sought-after proof, the obvious conjecture asserts that a number of steps greater than 1 might be needed to obtain the child of $A6$, where the formula $A6$ was not allowed to participate. Indeed, intuitively, removing one of the two parents of a deduced conclusion, especially when the removed parent is itself a deduced conclusion late in a proof—in the case under discussion, the 46th step in a 53-step proof—can cause havoc. Our choice was to invoke the use of the command set(sos_queue) to cause OTTER to conduct a breadth-first search for a proof of the child of $A6$. We placed in the initial set of support (in addition to the axioms $A1$, $A2$ $A4$, $A5$, $A7$, $A8$, and $A9$) the first 45 deduced steps of the proof in hand up to but not including $A6$. The target, negated and placed in list(passive), was the child of $A6$. (The approach we took is indeed reminiscent of the cramming strategy [Wos03], a strategy that enables the program to force or cram formulas in the initial set of support into the desired proof.) Just for total clarity, with almost certainty, additional deduced steps would be needed. After a thorough level-saturation search through level 1, at level 2 the desired proof of the child of $A6$ was completed, a proof of length 2.

We pause briefly to note that the approach just given would have merited use even if $A6$ had been the parent of more than one formula that followed its derivation. Iteration would be the way to proceed. One would proceed as we did but now with the negation of the first child of $A6$ placed in list(passive) with the goal of obtaining the needed proof that culminates with the derivation of the first child and without allowing $A6$ to participate in any manner. Then one would amend further the list(sos) with the new proof steps (that led to the derivation of the first child of $A6$ without $A6$ participating),

as well as proof steps of the original proof preceding the second child and not dependent on $A6$, and used as target the second child, placing its negation in list(passive), now with the goal of deriving the second child and with the given constraints. One would proceed in this manner, gathering proof steps along the way, until the last child of $A6$ was proved. Of course, the method we are presenting is useful when the goal is to avoid any unwanted formula or equation and replace its role by other formulas or equations, whether establishing proof dependence is the intention or not.

We had the components that almost guaranteed we could complete a proof that avoided the use of $A3$ and $A6$ as axioms and, more important in the context of proof dependence, avoided the use of those two formulas even as deduced steps. To enable OTTER to return the proof of interest, we placed in the initial set of support the original nine axioms but with $A3$ and $A6$ commented out. In list(passive), we still placed the negation of thesis_1 and, for monitoring purposes, the negations, respectively, of thesis_2 and thesis_3. In list(usable), we placed the two rules for condensed detachment, one for the function i and one for the function j, and the negation of the join of theses 2 and 3. To ensure that both $A3$ and $A6$ would not participate in any proof, we included the following.

```
list(demodulators).
(P(i(i(i(x,y),x),x)) = junk).             %  A3
(P(i(j(x,j(y,z)),j(y,j(x,z)))) = junk).   %  A6
(i(x,junk) = junk).
(i(junk,x) = junk).
(j(x,junk) = junk).
(j(junk,x) = junk).
(P(junk) = $T).
end_of_list.
```

The crucial move directed OTTER to the proof we expected it to find, a proof quite like that which relied on both $A6$ as a deduced clause and exactly one of its children. Throughout the experiments, we had relied upon the use of *resonators* to direct the program's search for one or more proofs. A resonator [Wos95] is a formula or an equation that does not itself take on the value **true** or **false**. Instead, its functional pattern is the key, where all variables within a resonator are treated as indistinguishable from each other, just denoting that a variable occurs in the corresponding position, and where the value assigned to a resonator reflects its conjecture importance (the smaller the value, the higher the priority given to similar deduced items). For the resonators intended to guide the program to the expected goal, we used the set that had led to the proof relying on $A6$, and, to enable the program to find the newer proof (not depending on $A6$) of the child of $A6$, we included the two resonators that corresponded to the proof found with level saturation.

As expected, OTTER was successful, and we had established both $A3$ and $A6$ proof dependent and, of course, not relying on either at the axiomatic level. We immediately attempted to further prune the original nine with regard to axiom dependencies and, more relevant to this article, seek proofs establishing additional proof dependence than that in hand. We did not expect that $A1$, $A2$, $A4$, or $A5$ would extend what we had in hand so far, in part because they appeared to be vital. However, $A7$ did look promising.

Therefore, we began a study with $A7$ commented out, as well as $A3$ and $A6$ not accessible as axioms or as deduced formulas.

The capture was quickly made: we had proofs in which neither $A3$ nor $A6$ nor $A7$ was present, as an axiom or as a deduced formula. We therefore turned to an attempt to prove $A7$ dependent on but six of the nine axioms. The experiment failed. Z. Ernst came to the rescue—or perhaps rescue is the wrong word in that we would have preferred $A7$ to be dependent—finding the following three-element model (with Mace4; see the Web www.mcs.anl.gov/AR/mace4) showing that $A7$ is in fact independent of the six.

```
-------- Model 1 at 0.01 seconds --------

 a : 1

 b : 2

 i :
      | 0 1 2
   ---+------
   0 | 0 1 2
   1 | 0 0 2
   2 | 0 1 0

 j :
      | 0 1 2
   ---+------
   0 | 0 1 2
   1 | 0 0 2
   2 | 0 0 0

 P :
        0 1 2
   ---------
        1 0 0

-------- end of model --------
```

Nevertheless, we now had in hand an example of an extension of the original concept of proof dependence in that we had considered an independent axiom. Specifically, although $A7$ is in fact independent, we had in hand proofs establishing each of $A3$, $A6$, and $A7$ to be proof dependent, with $A3$, $A6$, and $A7$ absent from the axiom system. We were thus ready for a serious effort at proof refinement in the context of length, within the given constraints, seeking "short" proofs of thesis_1, the join of theses 2 and 3, the dependence of $A3$, and the dependence of $A6$.

4 Pleasing Proofs

Our main effort in the context of proof refinement was aimed at proof shortening; shorter proofs are usually more pleasing than longer. As noted earlier, of the various aspects that can be brought to bear, two played the key role: ancestor subsumption and demodulation (to block the retention of conclusions one classes as unwanted). In particular, we took each proof in hand and instructed OTTER to block its steps one at a time, forcing it to seek a somewhat different proof, occasionally a sharply different proof. One might immediately conjecture that a direct attack on finding shorter proofs is in order, some type of exhaustive search, for example. The task cannot be so subtle, or can it?

Of course, one would prefer applying an algorithm that simply seeks and finds the shortest proof that exists for any given theorem and given axiom set. Studies of more than a decade prove (to me) that such an algorithm in many, many cases does not exist. Further, an unexpected obstacle (illustrating the cited subtlety) exists when seeking a proof shorter than that in hand. The following aphorism (found in some books and papers) nicely captures the obstacle. "Shorter subproofs do not necessarily a shorter total proof make." For the curious, how can this aphorism hold? An example is in order.

Let us consider a proof of, say, 20 steps in which the tenth step is proved by using steps 6 through 9. In other words, the length of the subproof concluding with step 10 is five. Now let us assume that OTTER or some person finds a proof of 10 that relies on 6a and 8a, a proof of length three. With ancestor subsumption in use, the program will prefer this second derivation because its length is three rather than five. A program or a person might then attempt to complete a proof relying on the three-step shorter subproof, with the expectation that the total proof (of step 20) will clearly be shorter. Such may not occur, for example, in the event that steps 6 through 9 play a vital role in the twenty-step proof. If a proof is completed that uses the cited three steps (of the shorter subproof), the resulting proof may have length at least 22—and far worse may occur.

Our efforts were indeed rewarded, as seen with the following proofs.

The given proofs are the shortest, for their respective conclusions, we have been able to complete. (For the curious, we note that the inference rule regarding the function i can be dispensed with; it is a derived inference rule. Its inclusion enables the program to find shorter proofs. In the presence of *A1,* *A*2, *A*4, *A*5, *A*8, and *A*9, OTTER finds a two-step proof showing that the corresponding clause is dependent. With the cited axiom system and the derived inference rule present, OTTER finds a 14-step proof showing *A*3 to be dependent; when the derived inference rule is removed, the best proof we have found has length 20.)

A 14-Step Proof of the Dependency of A3

```
----- Otter 3.3d, April 2004 -----
The process was started by wos on jaguar.mcs.anl.gov,
Thu May 27 10:43:03 2004
The command was "otter".  The process ID is 31886.
----> UNIT CONFLICT at   0.02 sec ----> 150 [binary,149.1,17.1] $ANS(a3).
```

```
Length of proof is 14.  Level of proof is 10.

---------------- PROOF ----------------

6 [] -P(i(x,y)) | -P(x) | P(y).
7 [] -P(j(x,y)) | -P(x) | P(y).
9 [] P(i(x,i(y,x))).
10 [] P(i(i(x,i(y,z)),i(i(x,y),i(x,z)))).
11 [] P(i(x,j(y,x))).
12 [] P(i(j(x,j(y,z)),j(j(x,y),j(x,z)))).
13 [] P(i(j(i(x,y),y),j(i(y,x),x))).
14 [] P(j(i(x,y),j(x,y))).
17 [] -P(i(i(i(a1,a2),a1),a1)) | $ANS(a3).
24 [hyper,6,9,9] P(i(x,i(y,i(z,y)))).
38 [hyper,6,10,10] P(i(i(i(x,i(y,z)),i(x,y)),i(i(x,i(y,z)),i(x,z)))).
40 [hyper,6,10,24] P(i(i(x,y),i(x,i(z,y)))).
41 [hyper,6,10,9] P(i(i(x,y),i(x,x))).
46 [hyper,7,14,40] P(j(i(x,y),i(x,i(z,y)))).
58 [hyper,6,38,41] P(i(i(x,i(x,y)),i(x,y))).
97 [hyper,7,14,58] P(j(i(x,i(x,y)),i(x,y))).
115 [hyper,6,13,97] P(j(i(i(x,y),x),x)).
120 [hyper,6,11,115] P(j(x,j(i(i(y,z),y),y))).
124 [hyper,6,12,120] P(j(j(x,i(i(y,z),y)),j(x,y))).
129 [hyper,7,124,46] P(j(i(i(i(x,y),z),y),i(x,y))).
138 [hyper,7,124,129] P(j(i(i(i(i(x,y),x),z),x),x)).
144 [hyper,6,13,138] P(j(i(x,i(i(i(x,y),x),z)),i(i(i(x,y),x),z))).
149 [hyper,7,144,9] P(i(i(i(x,y),x),x)).
```

5 Extending the Logic

At this point, one might ask about the power of the abbreviated axiom set consisting of $A1$, $A2$, $A4$, $A5$, $A8$, and $A9$. For example, with $A3$ and $A6$ and $A7$ omitted, can we prove significant theorems when the logic is extended by adjoining yet another axiom of interest or by adjoining a set of axioms focusing on different functions? Obviously, the omission of both $A3$ and $A6$ presents no problem at the axiomatic level in that they have been proved dependent on the set consisting of $A1$, $A2$, $A4$, $A5$, $A8$, and $A9$. However, as noted, $A7$ is independent of that set. Further, perhaps $A3$ or $A6$ or both will be needed at the deduced level, and $A7$ will be needed at the axiomatic level or at the deduced level.

With the following formula, $A10$, we have such an extended logic, $BCSK+$.

```
P(i(j(j(x,y),y),j(j(y,x),x))).  % A10
```

An interesting theorem to prove is captured, in its negated form, with the following clause; the formula to be proved is equivalent to $A10$, and the proof found by OTTER avoids totally $A3$, $A6$, and $A7$.

```
-P(i(j(A,B),i(A,B))) | $ANS(thm).
```

To complete proof of the equivalence, OTTER found an 18-step proof that deduces $A10$ from the following formula in clause notation, and, again, a proof completely free of reliance on $A3$, $A6$, and $A7$.

```
P(i(j(x,y),i(x,y))).
```

An appropriate move to test the power of the abbreviated axiom system, now consisting of seven axioms with the cited addition of $A10$, is to give OTTER an input file whose initial set of support consists of the seven axioms. The demodulator list contains equalities that, respectively, block the retention of $A3$, $A6$, and $A7$ if and when each is deduced. After all, for example, $A7$ might now be dependent on the seven-axiom system. From Veroff, we had in hand a 42-step proof of the theorem under consideration to initiate the study, a proof that does depend on the three axioms we intended to avoid, (at both the axiomatic and deduced levels). The original goal was to shorten that proof. More pertinent from the viewpoint of this article, we sought to find a proof establishing each of $A3$, $A6$, and $A7$ to be proof dependent simultaneously.

All went smoothly, with the discovery of a 23-step proof. In examining the proof, we observed that the added axiom, $A10$, is used but once. This observation caused us to ask about the independence of $A7$ in this extended logic. After all, perhaps the use of the added axiom ($A10$) leads to a proof of the dependence of $A7$. Therefore, we turned after a short time to studying this possible dependence. Is $A7$ independent or dependent in the extended logic? The effort paid off: OTTER found a proof of dependence, the following in which both $A3$ and $A6$ are totally absent.

A 24-Step Proof of the Dependence of A7

```
----- Otter 3.3g-work, Jan 2005 -----
The process was started by wos on jaguar.mcs.anl.gov,
Tue Mar  8 16:10:03 2005
The command was "otter".  The process ID is 4337.
----> UNIT CONFLICT at   0.13 sec ----> 679 [binary,678.1,17.1] $ANS(a7).

Length of proof is 24.  Level of proof is 11.

---------------- PROOF ----------------

1 [] -P(i(x,y)) | -P(x) | P(y).
2 [] -P(j(x,y)) | -P(x) | P(y).
5 [] P(i(x,i(y,x))).
6 [] P(i(i(x,i(y,z)),i(i(x,y),i(x,z)))).
7 [] P(i(x,j(y,x))).
8 [] P(i(j(x,j(y,z)),j(j(x,y),j(x,z)))).
9 [] P(i(j(i(x,y),y),j(i(y,x),x))).
10 [] P(j(i(x,y),j(x,y))).
11 [] P(i(j(j(x,y),y),j(j(y,x),x))).
17 [] -P(i(j(j(a1,a2),a1),a1)) | $ANS(a7).
```

```
57 [hyper,1,6,6] P(i(i(i(x,i(y,z)),i(x,y)),i(i(x,i(y,z)),i(x,z)))).
58 [hyper,1,5,6] P(i(x,i(i(y,i(z,u)),i(i(y,z),i(y,u))))).
59 [hyper,1,6,5] P(i(i(x,y),i(x,x))).
61 [hyper,1,5,7] P(i(x,i(y,j(z,y)))).
69 [hyper,2,10,5] P(j(x,i(y,x))).
80 [hyper,1,6,58] P(i(i(x,i(y,i(z,u))),i(x,i(i(y,z),i(y,u))))).
83 [hyper,1,57,59] P(i(i(x,i(x,y)),i(x,y))).
93 [hyper,1,7,69] P(j(x,j(y,i(z,y)))).
124 [hyper,1,80,5] P(i(i(x,y),i(i(z,x),i(z,y)))).
127 [hyper,2,10,83] P(j(i(x,i(x,y)),i(x,y))).
136 [hyper,1,8,93] P(j(j(x,y),j(x,i(z,y)))).
188 [hyper,1,6,124] P(i(i(i(x,y),i(z,x)),i(i(x,y),i(z,y)))).
198 [hyper,1,9,127] P(j(i(i(x,y),x),x)).
288 [hyper,1,188,61] P(i(i(j(x,y),z),i(y,z))).
344 [hyper,1,288,11] P(i(x,j(j(x,y),y))).
395 [hyper,2,10,344] P(j(x,j(j(x,y),y))).
398 [hyper,1,124,344] P(i(i(x,y),i(x,j(j(y,z),z)))).
550 [hyper,1,8,395] P(j(j(x,j(x,y)),j(x,y))).
564 [hyper,1,398,288] P(i(i(j(x,y),z),j(j(i(y,z),u),u))).
615 [hyper,1,11,550] P(j(j(j(x,y),x),x)).
618 [hyper,2,198,564] P(j(j(i(x,y),x),x)).
632 [hyper,2,136,615] P(j(j(j(x,y),x),i(z,x))).
650 [hyper,1,11,618] P(j(j(x,i(x,y)),i(x,y))).
678 [hyper,2,650,632] P(i(j(j(x,y),x),x)).
```

A second and more intriguing extension of the original logic, *SBPC*, was studied with the goal of determining the need, at the deduced level, of $A3$, $A6$, and $A7$. For the study, we began again with the now so-to-speak famous six axiom system, that consisting of $A1$, $A2$, $A4$, $A5$, $A8$, and $A9$, and adjoined the following six axioms (expressed in clause notation), where the function a denotes logical **and** and the function o denotes logical **or**.

```
P(j(x,o(x,y))).                          %  A11
P(i(y,o(x,y))).                          %  A12
P(j(j(x,z),j(j(y,z),j(o(x,y),z)))).      %  A13
P(i(a(x,y),x)).                          %  A14
P(j(a(x,y),y)).                          %  A15
P(i(i(x,y),i(i(x,z),i(x,a(y,z))))).      %  A16
```

In this extended logic, we attempted to find proofs, preferably short ones, of the following four theorems, each given in its negated form, and, as one might predict, we sought proofs in which $A3$, $A6$, and $A7$ are totally absent.

```
-P(j(i(A,B),i(o(A,C),o(B,C)))) | $ANS(1).
-P(j(i(A,B),i(o(C,A),o(C,B)))) | $ANS(2).
-P(j(i(A,B),j(i(B,A),i(a(A,C),a(B,C))))) | $ANS(3).
-P(j(i(A,B),i(a(C,A),a(C,B)))) | $ANS(4).
```

We began the study of the four theorems with proofs supplied by Veroff, obtained by him using his powerful technique called *sketches*. Perhaps because of the goal of finding appropriate proofs, four of them, establishing proof dependence for the three unwanted axioms, we were unable to complete the studies until we relied on a 92-step proof that deduced (without using $A3$ in any way) a (former) child of $A3$. In other words, the earlier studies of proof dependence came into play, enabling us (and OTTER) to overcome an obstacle. Success eventually was the result. OTTER returned a 53-step proof of the first of the four theorems, a 64-step proof of the second, a 104-step proof of the third and a 99-step proof of the fourth. Many experiments were required, as well as much use of refinement methodology detailed in the book [WP03]. The last significant reductions in proof length (of the proofs of the third and fourth theorems) were obtained by heavy reliance on cramming. Briefly, OTTER was given proofs of steps near the end of the proofs in hand and asked to (in effect) force their proof steps into (we hoped) shorter proofs of the targets.

The discovery that $A7$ is dependent in the $BCSK+$ logic (obtained by adding $A10$ to the original nine axioms, then removing any use of $A3$ and $A6$) led us to consider the possibility that that formula is dependent in this second extension of the $BCSK$ logic. Indeed, would it not be more than piquant to find that $A7$ is independent in the original study and then find it dependent in two extensions of the logic? And, as the following proof shows—the shortest so far discovered—that is exactly what was found.

A 35-Step Proof of the Dependence of A7 in a Second Extension

```
----- Otter 3.3g-work, Jan 2005 -----
The process was started by wos on theorem.mcs.anl.gov,
Sun Mar 20 12:31:56 2005
The command was "otter".  The process ID is 20352.
----> UNIT CONFLICT at   0.09 sec ----> 904 [binary,903.1,24.1] $ANS(a7).

Length of proof is 35.  Level of proof is 20.

---------------- PROOF ----------------

10 [] -P(i(x,y)) | -P(x) | P(y).
11 [] -P(j(x,y)) | -P(x) | P(y).
12 [] P(i(x,i(y,x))).
13 [] P(i(i(x,i(y,z)),i(i(x,y),i(x,z)))).
14 [] P(i(x,j(y,x))).
15 [] P(i(j(x,j(y,z)),j(j(x,y),j(x,z)))).
16 [] P(i(j(i(x,y),y),j(i(y,x),x))).
17 [] P(j(i(x,y),j(x,y))).
18 [] P(j(x,o(x,y))).
19 [] P(i(y,o(x,y))).
20 [] P(j(j(x,z),j(j(y,z),j(o(x,y),z)))).
24 [] -P(i(j(j(a1,a2),a1),a1)) | $ANS(a7).
130 [hyper,10,13,13] P(i(i(i(x,i(y,z)),i(x,y)),i(i(x,i(y,z)),i(x,z)))).
```

```
133 [hyper,10,12,14] P(i(x,i(y,j(z,y)))).
135 [hyper,10,12,15] P(i(x,i(j(y,j(z,u)),j(j(y,z),j(y,u))))).
138 [hyper,11,17,16] P(j(j(i(x,y),y),j(i(y,x),x))).
139 [hyper,11,17,15] P(j(j(x,j(y,z)),j(j(x,y),j(x,z)))).
140 [hyper,11,17,14] P(j(x,j(y,x))).
142 [hyper,11,17,12] P(j(x,i(y,x))).
180 [hyper,10,130,133] P(i(i(x,i(j(y,x),z)),i(x,z))).
197 [hyper,11,140,140] P(j(x,j(y,j(z,y)))).
244 [hyper,10,180,135] P(i(j(x,y),j(j(z,x),j(z,y)))).
285 [hyper,11,17,244] P(j(j(x,y),j(j(z,x),j(z,y)))).
290 [hyper,10,244,142] P(j(j(x,y),j(x,i(z,y)))).
298 [hyper,10,15,285] P(j(j(j(x,y),j(z,x)),j(j(x,y),j(z,y)))).
347 [hyper,11,298,197] P(j(j(j(x,y),z),j(y,z))).
362 [hyper,11,285,347] P(j(j(x,j(j(y,z),u)),j(x,j(z,u)))).
371 [hyper,11,347,138] P(j(x,j(i(x,y),y))).
416 [hyper,11,362,139] P(j(j(x,j(y,z)),j(y,j(x,z)))).
449 [hyper,11,285,371] P(j(j(x,y),j(x,j(i(y,z),z)))).
488 [hyper,11,416,416] P(j(x,j(j(y,j(x,z)),j(y,z)))).
506 [hyper,11,416,138] P(j(i(x,y),j(j(i(y,x),x),y))).
559 [hyper,11,449,18] P(j(x,j(i(o(x,y),z),z))).
584 [hyper,11,139,488] P(j(j(x,j(y,j(x,z))),j(x,j(y,z)))).
603 [hyper,11,506,19] P(j(j(i(o(x,y),y),y),o(x,y))).
604 [hyper,11,506,14] P(j(j(i(j(x,y),y),y),j(x,y))).
657 [hyper,11,584,559] P(j(x,j(i(o(x,y),j(x,z)),z))).
681 [hyper,11,416,657] P(j(i(o(x,y),j(x,z)),j(x,z))).
698 [hyper,11,603,681] P(o(x,j(x,y))).
709 [hyper,11,488,698] P(j(j(x,j(o(y,j(y,z)),u)),j(x,u))).
727 [hyper,11,709,140] P(j(x,x)).
738 [hyper,11,20,727] P(j(j(x,y),j(o(y,x),y))).
767 [hyper,11,709,738] P(j(j(j(x,y),x),x)).
814 [hyper,11,285,767] P(j(j(x,j(j(y,z),y)),j(x,y))).
845 [hyper,11,814,604] P(j(j(i(j(j(x,y),x),x),x),x)).
851 [hyper,11,814,290] P(j(j(j(i(x,y),z),y),i(x,y))).
903 [hyper,11,851,845] P(i(j(j(x,y),x),x)).
```

For the curious, the first study yielded a 39-step proof, a proof that the usual methods were unable to improve upon. However, with a most unsophisticated form of cramming, the given 35-step proof was found. In particular, rather than relying on a subproof of one of the late steps, OTTER was merely given the first 34 steps of the 39-step proof and told to apply level saturation. In other words, no attention was paid to the possible presence of steps among the thirty-four that were not used in the proof of the thirty-fourth step.

6 Summary

In this article, we have extended the notion of axiom dependence to one of *proof dependence.* Briefly, a formula or equation is proof dependent if it can be dispensed with, even as a deduced item; in other words there exists at least one proof that shows the item to be totally unnecessary. The new term, proof dependent, was chosen because of the nature of a dependent axiom, namely, one that is unnecessary at the so-called input level (from the viewpoint of automated reasoning). We have given methodology for finding an appropriate proof, one that completely avoids the use of some selected item, even when the item is in fact an independent axiom.

We have included various proofs discovered with indispensable aid from McCune's OTTER, the shortest proofs that we could discover, given the conditions to be satisfied. Such conditions included total avoidance of one or more items. Among our successes was the discovery of various axiom dependencies. In two of the three logics we studied, both extensions of the *BCSK* logic, we found (most unexpectedly) that a key axiom, $A7$, is dependent, although it is independent among the axioms for *BCSK.*

Acknowledgments

*This work was supported in part 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.

References

[Bel01] J. Belinfante. Computer assisted proofs in set theory, 2001. Web site http://www.math.gatech.edu/∼belinfan/research/autoreas/index.html.

[BP94] W. J. Blok and D. Pigozzi. On the structure of varieties with equationally definable principal congruences III. *Algebra Universalis*, 32:545–608, 1994.

[BP99] W. J. Blok and D. Pigozzi. Algebraisable logics. *Mem. Amer. Math. Soc.*, 77, 1999.

[BSV] R. J. Bignall, M. Spinks, and R. Veroff. On the assertional logics of the generic pointed discriminator and generic pointed fixedpoint discriminator varieties. Preprint, 2003.

[Kun92] K. Kunen. Single axioms for groups. *J. Automated Reasoning*, 9:291–308, 1992.

[Phi] J. D. Phillips. Private Communication, Argonne Workshop on Automated Reasoning and Deduction (AWARD), 2003.

[TW02] R. Thiele and L. Wos. Hilbert's twenty-fourth problem. *J. Automated Reasoning*, 29(1):67–89, 2002.

[Ver96] R. Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *J. Automated Reasoning*, 16(3):223–239, 1996.

[Ver01] R. Veroff. Solving open questions and other challenge problems using proof sketches. *J. Automated Reasoning*, 27(2):175–199, 2001.

[Wos95] L. Wos. The resonance strategy. *Computers and Mathematics with Applications*, 29(2):133–178, 1995.

[Wos03] L. Wos. The strategy of cramming. *J. Automated Reasoning*, 30(2):179–204, 2003.

[WP03] L. Wos and G. W. Pieper. *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press, Paramus, N.J., 2003.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12:536–541, 1965.

MiniSAT and some Applications of SAT

Niklas Sörensson
Chalmers University of Technology
nik@cs.chalmers.se

Abstract

SAT solving is important not only because of its theoretical interest, but also because it has gained success as a practical tool for problem solving. In this talk I will give an overview of my (and my colleagues') contributions to the research field.

The SAT solver MiniSAT was designed to be simple and extensible, yet similar in efficiency and architecture compared to other state-of-the-art SAT solvers. It supports incremental SAT solving, where a sequence of similar SAT problems can be expressed relative to their predecessors. The problems are solved more efficiently because the solver can reuse work from earlier runs. Another feature is the possibility to extend the solver programmatically with other types of boolean constraints, for instance 0-1 linear inequalities. In recent work, we improved MiniSAT by conflict clause simplification, preprocessing based on variable elimination and subsumption, and some changes to the variable order heuristics (i.e., black magic). At the time of writing, the current MiniSAT is doing very well in the 2005 edition of the SAT competition.

On the user end of SAT solving, there is often much to gain by choosing the right type of encoding for a particular problem. As a simple example, propositional formulas may be encoded to CNF by (or-)distribution, or by introducing new propositions for each formula, or a combination of the two. The resulting CNFs may give radically different runtime s when fed to a SAT solver. We have studied several applications of SAT, that can be seen as more involved examples of encoding techniques, including Model Checking, First Order Model Finding, and 0-1 Optimization.

Tau: A Web-Deployed Hybrid Prover for First-Order Logic with Identity, with Optional Inductive Proof

Jay Halcomb, Randall R. Schulz
H&S Information Systems
http://hsinfosystems.com
mailto:hsis@hsinfosystems.com

April 29, 2005

Abstract

We outline Tau, a practical and extensible hybrid theorem prover for first-order predicate calculus with identity. Tau is flexible and user-configurable, accepts the KIF Language, is implemented in Java, and has multiple user interfaces. Tau combines rule-based problem rewriting with Model Elimination, uses Brand's Modification Method to implement identity, and accepts user-configurable heuristic search to speed the search for proofs. Tau optionally implements mathematical induction. Formulas are input and output in KIF or infix FOPC, and other external forms can be added. Tau can be operated from a Web interface or from a command-line interface. Tau is implemented entirely in Java and can run on any system for which a current Java Virtual Machine is available.

Keywords: automated theorem proving (ATP), first-order logic (FOL), hybrid prover, Knowledge Interchange Format (KIF), web-based, interactive, model elimination (ME), resolution, rewriting, Java.

1 Introduction: How Tau Works

Consider the remarks of [Bachmair and Ganzinger, 1998] in the Handbook of Automated Reasoning, "Resolution Theorem Proving":

"It has been pointed out that **a weakness of resolution is its lack of goal orientation**. Simplification and clause elimination based on redundancy helps ameliorate the problem, but one might also consider possible combinations of resolution with such goal-oriented methods as the sequent calculus or semantic tableaux. Semantic tableaux and variants thereof, including the Davis-Putnam method, model elimination and SL-resolution can be viewed as tree-like theorem proving process in which the limits of the individual branches are saturated under (ordered) resolution with selection. **This view may serve as a basis for further investigations of the combination problem.**" P. 94, Vol. 1, emphasis added.

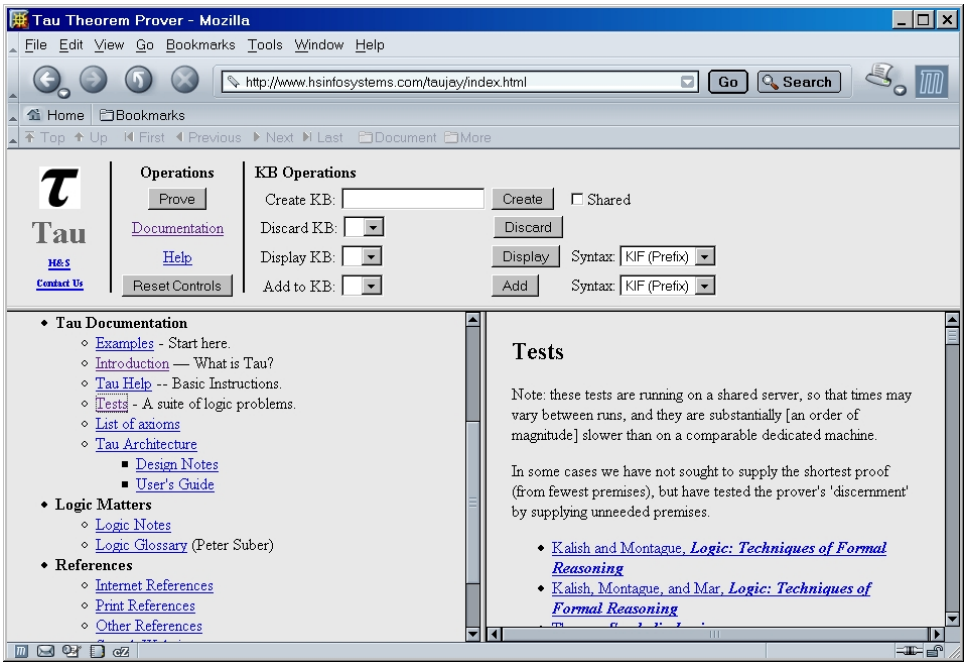


Figure 1: Initial Tau Screen

That was the spirit with which we approached the Tau project. It is perhaps as difficult to induce a computer to 'reason logically' as it is to induce a human to do so, but with the Tau theorem prover and knowledge base (eventually we hope Tau to be a formal theory repository), we are aiming to produce the first interactive, easy-to-use, and comprehensive prover of its kind on the Internet. Tau is sound and theoretically complete for the First Order Predicate Calculus with Identity – a phrase which can cover a multitude of sins, due to the general undecidability of FOPC. Tau's syntax is full FOPC with sentential constants, relation symbols, function symbols, complex terms, and identity.

What you can presently do with Tau: test the FOL validity of symbolic formulas; test the FOL validity of formal arguments (derive a conclusion from premises); normalize formulas – command line interface only at this time;construct a formal FOPC theory and make deductions from it. Examples already constructed include: theorems in Presburger and Peano arithmetic, both with and without mathematical induction; theorems in the theory of commutative ordered fields; theorems in graph theory.

Tau's initial screen is shown in Figure 1. Use of Tau is quite simple: well-formed formulas of KIF can be typed or pasted into the browser window. Then (after perhaps selecting an option check box) press 'Prove'.

Tau is written in the Java programming language ([Sun Microsystems]). Its Web interface uses the Tomcat servlet container ([Apache Jakarta Project]). The user interface of the browser version of Tau is implemented in HTML and CSS using a forms-based submission. The primary proof procedure employed by Tau is Loveland's well-known Model Elimination algorithm ([Loveland, 1968], [Loveland, 1969], and [Loveland, 1978])

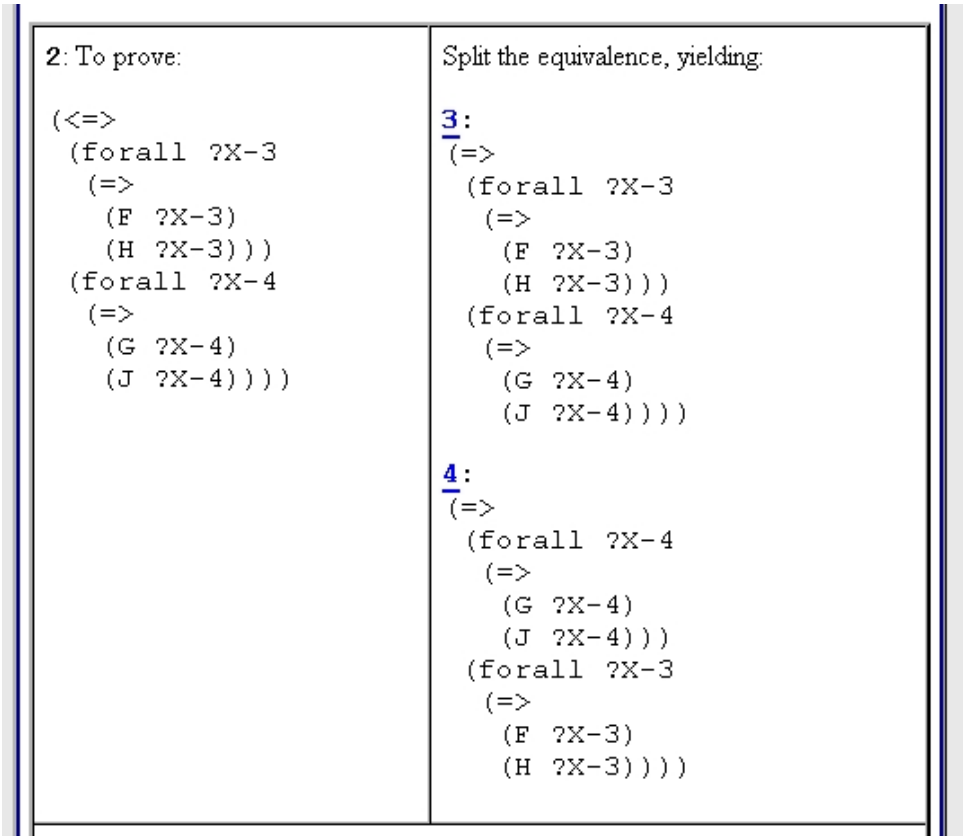


Figure 2: An Example of Rewriting

augmented with a selectable variety of search algorithms, including heuristic search guided by a user-supplied heuristic ranking function.

Prior to submission to the Model Elimination algorithm, problems are optionally subjected to a process of rewriting, in which the original conclusion is rewritten into logically equivalent formulas that are more tractable. The rewriting process is recursive in the sense that the result of a rewriting may itself be rewritten further. When the problem submitted includes use of the identity predicate, the Model Elimination prover stage applies the necessary transformations, using a variant of Brand's Modification Method [Brand, 1975].

Tau is intended for experimentation and educational use. Tau can be used as a proof assistant and as a teaching aid.

Note: The URLs for test cases mentioned in this paper refer to a shared, commercial Internet hosting server with limited computational resources. As a result, problems run slowly there. Full performance of Tau can be witnessed on a dedicated host. Interested parties should contact the authors at their email address for access to this host. Also, note that in some test cases we have not sought to supply the shortest proof (from fewest premises), but we have tested the prover's 'discernment' by supplying unneeded premises.

Tau proofs are based upon proof-by-contradiction using a linear restriction of resolution invented by Loveland called Model Elimination. Tau's implementation of Model Elimination also incorporates (by default) the so-called Set-of-Support restriction, in which the contradiction sought in the indirect proof must exist between the negated conclusion and the premises or within the negated conclusion itself. Tau proof displays, being based upon proof-by-contradiction using a resolution (Model Elimination) strategy, are not informative in the way a natural deduction style of presentation is. We intend later to expose in a more natural way some N.D. proof structure, and to provide further aid to using Tau as a semi-automated proof assistant. We do presently display some of the initial rewriting techniques used (see Figure 2).

Tau can prove either valid theorems or arguments. Other examples with which to try Tau are at the URLs:

T260 `http://www.hsinfosystems.com/taujay/doc/samples/tests/T260.jsp`

T265 `http://www.hsinfosystems.com/taujay/doc/samples/tests/T265.jsp`

T327 `http://www.hsinfosystems.com/taujay/doc/samples/tests/T327.jsp`

HOH `http://www.hsinfosystems.com/taujay/doc/samples/HeadOfAHorse.jsp`

AssocAdd `http:`
`//www.hsinfosystems.com/taujay/doc/samples/PrA.web/AssocAdd.html`

We've been using theorems from [Kalish and Montague, 1964], (through the chapter on identity) for many of our basic tests. The entire test directory (containing over 250 tests) is at:
`http://www.hsinfosystems.com/taujay/doc/samples/testsJSP.html.`

2 Tau and the KIF Language

Typical KIF looks like:

```
(<=>
 (exists ?Y
  (and
   (forall ?X (<=> (f ?X) (= ?X ?Y)))
   (g ?Y)))
 (and
  (exists ?Y
   (forall ?X (<=> (f ?X) (= ?X ?Y))))
  (forall ?X (=> (f ?X) (g ?X)))))
```

You can run a Tau proof of this theorem by clicking 'Prove' at:

`http://www.hsinfosystems.com/taujay/doc/samples/tests/T324.jsp`

After running that test you will see the trace of Tau's proof of the theorem. That page will show a verbose trace of Tau's actions in proving this theorem, displaying how the theorem was broken down into sub-proofs and how the formulas were rewritten and normalized to facilitate the proof. More concise proof display options are also available from running Tau in a command line mode.

KIF (Knowledge Interchange Format) is essentially a parenthesized prefix version of common first-order logical notation, which largely emanates from environs of Stanford University; KIF is also a part of the ISO (International Organization for Standardization) [Common Logic Standard] effort. Being a prefix form, it is efficient for many computer applications and for that reason we adopted KIF as Tau's first internal language. Tau also has an infix syntax which is consistent with typical conventions used in ASCII computer settings. Both of these concrete syntaxes are intended for situations where no special logic symbols are available. Our architecture admits unlimited additional concrete syntaxes, including those which include proper mathematical and logical symbology such as TeX or MathML; we intend to incorporate graphical notations and I/O into Tau.

Tau accepts both a prefix version of FOL, called KIF (Knowledge Interchange Format), and a related infix form of FOL. Tau KIF is a Lisp-like, S-Expression prefix syntax based on the KIF 3 standard. Details of KIF 3 are available at the Knowledge Interchange Format home page, `http://ksl-web.stanford.edu/knowledge-sharing/kif/`. An HTML conversion of the TeX original from the preceding page is here: `http://logic.stanford.edu/kif/Hypertext/kif-manual.html`.

Variables in KIF are preceded by a '?'; individual constants, predicates, relations, and functions may be a single alphabetic character or a string of such. Computer generated individual constants appearing in our proofs are preceded by a '$'.

For further details, please see:

Knowledge Interchange Format (KIF) An HTML conversion of the TeX original from the preceding page `http://www-ksl.stanford.edu/knowledge-sharing/kif/`

Knowledge Interchange Format, dpAns `http://logic.stanford.edu/kif/dpans.html`

Knowledge Interchange Format `http://logic.stanford.edu/kif/specification.html`

3 The Logical Theory of Tau

The Tau prover is essentially an indirect prover that proves formulas by establishing the mutual unsatisfiability of the set of clauses that result from the Skolemized form of the original input problem's formulas with the conclusion to prove first negated.

Before Skolemization, clausalization and the application of Model Elimination, the conclusion is subject to a process of rule-driven rewriting that replaces the original conclusion with other more tractable but (collectively) equivalent conclusions, each of

which is proved independently. The result of any given rewriting is itself subject to rewriting.

This recursive decomposition process produces a tree of sub-proofs. Both conjunctive (all sub-proofs produced by a given rewriting must succeed) and disjunctive (only one of a rewriting's sub-proof need succeed) sub-proof combination rules are allowed. The system can optionally compute estimates of the proof complexity of each resulting sub-proof and then order the attempts to prove them so as to conclude the overall proof successfully (or fail) in the shortest time.

For some types of problems we have also implemented a direct instantiation method and an optional incremental satisfiability checker (based upon Davis-Putnam-Loveland); see [Hooker, 1993] and [Hooker, 1993a].

Tau is based on: reductio ad absurdum, or contradiction testing; normalization (see, e.g., [Baaz et al, 2002], and also see [Nonnengart and Weidenbach, 2002]); our version of Brand transformations (see, e.g.,[Brand, 1975], and [Degtyarev and Voronkov, 1999], "Equality reasoning in sequent-based calculi"), to implement identity rewriting strategies.

Tau's use of a Model Elimination technique in conjunction with selection heuristics and proof strategizing helps overcome some of the difficulties resulting from a lack of goal-directedness.

Our primary emphasis is on the logical soundness of the proof method and the integrity of the software design. Principally via the command line interface we have a good deal of control and flexibility in choosing proof strategies, and over the presentations and annotations, and we are adding these options judiciously to the browser interface.

4 Algorithms

4.1 Resolution

Resolution proof was introduced in [Robinson, 1965] and [Robinson, 1971]; the well-known [Chang and Lee, 1973] gave resolution further impetus. However, the resolution method requires considerable augmentation by efficient search techniques to be of practical use.

4.2 Model Elimination and Proof Search

The Model Elimination technique was introduced in [Loveland, 1968], [Loveland, 1969], and [Loveland, 1978], and is theoretically sound and complete. Interest in it was more lately revived with Stickel's work on the theorem prover PTTP, e.g. [Stickel, 1984]. Tau uses a version of Model Elimination with refinements to handle certain completeness issues which may arise from an uncareful application of search techniques; for example, the Inoue problem (see below). In this regard, Tau also offers multiple search strategies, with selection heuristics (clausal weighting).

As with all automated theorem proving, search plays a central role. Tau's implementation of the Model Elimination procedure implements these kinds of search: breadth-first search, depth-first search, heuristic search, and modified search. In all cases, a user-specified depth-cutoff is applied.

Breadth-first search is guaranteed to find the shortest proof possible for the problem, but will typically examine far too many clauses in the process of finding that shortest proof. Breadth-first search also tends to consume excessive amounts of primary storage holding clauses at the frontier of the proof search tree.

Depth-first search requires the least amount of storage and depends strongly on the depth cutoff to prevent its becoming trapped in unbounded sub-trees of the proof tree.

Heuristic search is the default and almost always produces the best overall results. Each clause in the set of clauses produced by the conclusion and each clause (or *chain*, in Loveland's terminology) that arises by successful applications of the Model Elimination inference operations is evaluated by a user-specified *heuristic function* whose purpose is to estimate the distance from the specified clause to a successful proof (i.e., an empty clause). Pending clauses, those that occupy the current frontier of the Model Elimination proof search tree, are held in a priority queue that is ordered by the aforementioned heuristic function. At each cycle of the Model Elimination proof search, the clause with the lowest heuristic value (i.e., the one deemed closest to yielding a successful proof) is chosen for processing.

Modified search, as described in [Chang and Lee, 1973], is an option for any of the three basic proof search procedures mentioned above. Model Elimination includes three kinds of inference operation: Factorization, Reduction and Extension. Factorization and Reduction operate on single clauses, while Extension operates on pairs of clauses. Modified search differs from basic search only with respect to the pairs of clauses that participate in the Extension operation. Instead of computing all of the Extension operations possible for a given clause as a single operation, all potential Extension side (or *auxiliary*) clauses are determined and each of the resulting center-clause / side-clause Extension pairs are scheduled independently. This allows for a more refined heuristic to be computed than is possible if only the center clause is examined, because the heuristic function has access to both the center and the side clauses. Factorization is itself optional at the user's discretion. In most cases, modified search produces better performance than basic search.

4.3 Heuristic Search

Under heuristic proof search, the choice of which clause or clause pair to expand next is governed by the value produced by the *heuristic function*. At any time during the ongoing search for a Model Elimination proof, the frontier of the proof tree is held in a priority queue which is ordered by the value produced by the heuristic function. Each cycle of the proof search consists of removing from the priority queue and then expanding the lowest-valued clause or clause pair (when modified search is in effect).

The primitive heuristic functions, one for single-clause tree nodes and one for extension clause pair nodes, are defined by the user and take the form of a simple linear function combining any of a variety of built-in parameters describing the clause or clause pair. The function is specified in a Lisp-like S-Expression that includes decimal numeric constants and the names of the clause / clause-pair parameter functions following one of the target keywords *node*, *pair*, *center* or *side* combined using any of the four arithmetic operators.

Single Clause Parameter Functions

nLiterals The number of literals, framed or unframed, in the clause

nFLiterals The number of framed literals in the clause

nUFLiterals The number of unframed literals in the clause

nIdentity The number of identity literals in the clause

nIdentityIF The number of unframed identity literals

termDepth The maximum depth of nesting of complex terms

termDepthIF The maximum depth of nesting of complex terms in unframed literals

termVolume The total number of symbols in the clause

nVars The number of distinct variables in the clause

nVarsIF The number of distinct variables in unframed literals

depth The depth of the clause in the proof search tree

Extension Pair Parameter Functions

nLiterals The sum of the number of literals, framed or unframed, in each clause

nFLiterals The sum of the number of framed literals in each clause

nUFLiterals The sum of the number of unframed literals in each clause

nIdentity The sum of the number of identity literals in the each clause

nIdentityIF The sum of the number of unframed identity literals in each clause

termDepth The maximum depth of nesting of complex terms in either clause

termDepthIF The maximum depth of nesting of complex terms in the unframed literals of each clause

termVolume The sum of the total number of symbols in each clause

termVolumeIF The sum of the total number of symbols in unframed literals in each clause

nVars The sum of the number of distinct variables in each clause

nVarsIF The sum of the number of distinct variables in unframed literals of each clause

depth The depth of the center clause in the proof search tree

As described above, the clause pair functions are applied to both clauses in the extension clause pair. When desired, these functions can instead be applied to the center or side clause alone. Examples: fewest literals first, (node nLiterals); literal count + maximum term nesting depth without regard for the kind of proof tree node (single clause or extension clause pair), (+ (node nLiterals) (node termDepth)). Note that framed literals are also known as *A literals* and unframed literals as *B literals*.

4.4 Brand Transformations

Brand transformations are rewrites of standard clausal forms which contain identities. There is a transformation corresponding to the transitivity of identity, and one to the symmetry of identity. These transformations were introduced in [Brand, 1975]; they are further discussed in [Degtyarev and Voronkov, 1999]. Apart from Brand's *flattening* transform, which supplies the substitutivity of identity and is applied unconditionally to problems that include application of the identity predicate, the transitivity and symmetry properties of identity may be supplied either by introducing the pertinent axioms as additional premises or by the application of the corresponding Brand transformation.

4.5 Martelli and Montanari

Resolution theorem proving and all its derivatives and variants rely heavily on the use of unification between first-order expressions. The efficiency of the unifier bears heavily on the overall speed of the prover. In addition to the classic recursive "mesh" unification algorithm presented in many texts, papers and books, Tau implements the efficient unification algorithm discussed in [Martelli and Montanari, 1977] and in [Martelli and Montanari, 1982]. This unification algorithm treats the expressions to be unified, any number of them, as a system of simultaneous equations and solves that system. It is folklore that the Martelli and Montanari algorithm, although providing the best theoretical complexity result, is not always the best algorithm in practice due to the overloading of handling complex data structure. With Tau, we have found that with some problems M&M has substantially improved typical and worst-case complexity by comparison with the classical mesh unification algorithm; i.e., there are problems that generate terms whose structure tips the balance of net run-time cost in favor of M&M. In fact, we use the mesh unifier by default (because measurement confirmed this folklore), but the advantage is small and while we have not confirmed the claim generally, we believe there are problems that produce term structures for which it is true. There are also optimization techniques (low-level programming, not algorithmic) that could yet close the gap for the majority of problems and make M&M the overall winner. The biggest problem is the large number of very short-lived set data structures produced when executing the M&M algorithm. If we can cut the overhead of their generation and reclamation, we hope to see M&M to surpass the mesh unifier.

4.6 Stillman's Subsumption Algorithm

Another time-consuming operation for resolution-based theorem provers is computing clause subsumption. In addition to the classic subsumption algorithm described in [Chang and Lee, 1973], Tau implements the better-performing subsumption algorithm invented by Stillman and described in [Gottlob and Leitsch, 1985].

5 Computational Results

The notion of an empirically successful theorem prover is difficult to define, and has a problematic history. As with human provers, it is not clear or uncontroversial exactly what to count as virtue in a prover. Is it: speed, some idea of completeness or comprehensiveness, ease of use, subtlety and originality, or some other factor, or some combination of these? In a practical sense, the idea is one of instrumental virtue, and thus relative to the various conceptions of good use of logic. The TPTP (Thousands of Problems for Theorem Provers) Problem Library, however, is now providing a more uniform basis for assessments; `http://www.cs.miami.edu/~tptp/`.

We have begun testing Tau on the TPTP library, which provides a large and challenging repository of benchmarks for provers. TPTP provides tools for translation of TPTP problems into KIF, but due to Tau's preference for FOF over CNF forms, its treatment of identities, and its rewrite strategies, in some cases further aligning of the TPTP tests is necessary before a reasonably full and fair comparison can be made. To date, using the automatic translation tools, we have translated the Geo (geometry) set of problems, with a solution rate of about one-third. For illustration, a proof trace, showing the ME proof steps, is given below of TPTP (Number Theory) NUM016-1, the intended interpretation of which is that there exist infinitely many prime numbers. This is followed by sample statistics of a run of this problem. [Note: some of the run time includes initial invocation of the prover when run in the shell mode.]

```
Root:
{  (not (prime ?X));
   (not (less a ?X));
        (less (factorial_plus_one a) ?X) }
Extend:
{  (not (less ?X-2 ?Y-3));
   (not (less ?Y-3 ?X-2)) }
[ ?Y-3 -> (factorial_plus_one a), ?X -> ?X-2 ]

Clause:
{  (not (prime ?X-2));
   (not (less a ?X-2));
       [(less (factorial_plus_one a) ?X-2)];
   (not (less ?X-2 (factorial_plus_one a))) }
Reduce: [ ?X-2 -> (factorial_plus_one a) ]

Clause:
{  (not (prime (factorial_plus_one a)));
   (not (less a (factorial_plus_one a))) }
Extend:
{  (not (divides ?X-4y (factorial_plus_one ?Y-4z)));
        (less ?Y-4z ?X-4y) }
[ ?Y-4z -> a, ?X-4y -> (factorial_plus_one a) ]
```

```
Clause:
{  (not (prime (factorial_plus_one a)));
  [(not (less a (factorial_plus_one a)))];
   (not (divides (factorial_plus_one a)
                 (factorial_plus_one a))) }
Extend:
{       (divides ?X-6f ?X-6f) }
[ ?X-6f -> (factorial_plus_one a) ]

Clause:
{  (not (prime (factorial_plus_one a))) }
Extend:
{       (prime ?X-71);
        (divides (prime_divisor ?X-71) ?X-71) }
[ ?X-71 -> (factorial_plus_one a) ]

Clause:
{ [(not (prime (factorial_plus_one a)))];
        (divides (prime_divisor (factorial_plus_one a))
                                (factorial_plus_one a)) }
Extend:
{  (not (divides ?X-77 ?Y-78));
   (not (less ?Y-78 ?X-77)) }
[ ?X-77 -> (prime_divisor (factorial_plus_one a)),
                 ?Y-78 -> (factorial_plus_one a) ]

Clause:
{ [(not (prime (factorial_plus_one a)))];
       [(divides (prime_divisor (factorial_plus_one a))
                                (factorial_plus_one a))];
   (not (less (factorial_plus_one a)
              (prime_divisor (factorial_plus_one a)))) }
Extend:
{  (not (prime ?X-8c));
   (not (less a ?X-8c));
        (less (factorial_plus_one a) ?X-8c) }
[ ?X-8c -> (prime_divisor (factorial_plus_one a)) ]

Clause:
{ [(not (prime (factorial_plus_one a)))];
       [(divides (prime_divisor (factorial_plus_one a))
                                (factorial_plus_one a))];
  [(not (less (factorial_plus_one a)
                (prime_divisor (factorial_plus_one a))))];
   (not (prime (prime_divisor (factorial_plus_one a))));
   (not (less a (prime_divisor (factorial_plus_one a)))) }
```

```
Extend:
{  (not (divides ?X-r5 (factorial_plus_one ?Y-r6)));
        (less ?Y-r6 ?X-r5) }
[ ?Y-r6 -> a, ?X-r5 -> (prime_divisor (factorial_plus_one a)) ]

Clause:
{ [(not (prime (factorial_plus_one a)))];
       [(divides (prime_divisor (factorial_plus_one a))
                                 (factorial_plus_one a))];
  [(not (less (factorial_plus_one a)
                (prime_divisor (factorial_plus_one a))))];
   (not (prime (prime_divisor (factorial_plus_one a)))) }
Extend:
{       (prime ?X-1cx);
        (prime (prime_divisor ?X-1cx)) }
[ ?X-1cx -> (factorial_plus_one a) ]

Clause:
{  }
```

Step Stats: elapsedTime=0.364; cpuTime=0.0; steps=9; roots=1; inputs=12; hornInputs=9; definiteInputs=6; generalInputs=3; factors=0; premiseLiterals=19; rootLiterals=3; generated=349; predicates=3; functions=2; constants=1; skFunctions=0; skConstants=0; expanded=348; derivations=348; factorizations=0; reductions=4; extensions=345; symIDUnif=0; outOrder=1.0; maxQueue=472; nResidual=468; nTooDeep=0; nUnacceptable=9; nXUnacceptable=0; nSubsumed=0; nVacuous=0; proofLength=9

Proof Stats: proved=1; elapsedTime=0.364; cpuTime=0.0; subproofs=1; successes=1; steps=9; roots=1; inputs=12; hornInputs=9; definiteInputs=6; generalInputs=3; factors=0; premiseLiterals=19; rootLiterals=3; generated=349; predicates=3; functions=2; constants=1; skFunctions=0; skConstants=0; expanded=348; derivations=348; factorizations=0; reductions=4; extensions=345; symIDUnif=0; outOrder=1.0; maxQueue=472; nResidual=468; nTooDeep=0; nUnacceptable=9; nXUnacceptable=0; nSubsumed=0; nVacuous=0; proofLength=9

Elapsed: 0m2s; User: 0m1.7s; System: 0m0.1s

Our work with TPTP has just begun. We will not be so rash as to claim that Tau surpasses any of the well-known provers, such as Otter, Setheo, Meteor, Protein, or Snark, but initial tests indicate that Tau behaves respectably on a variety of TPTP problems. However, that may be, we shall give below more sample Tau statistics, after a discussion of some various types of problems.

5.1 Logic Theorems

There are at present 78 logical theorems available for testing in the Tau browser, derived from [Kalish and Montague, 1964] and [Montague, Kalish, and Mar, 1980].

A simple theorem which caused an incompleteness problem for some older resolution style provers was posed in [Inoue, 1992].

```
(=> (and
      (forall ?X (or (not (Q ?X)) (P ?X) (P a))
      (not (P B))
      (Q B) )
(P a))
```

Tau handles such problems easily; a test run can be made at `http://hsinfosystems.com/taujay/doc/samples/tests/In001.jsp`.

The 'Los theorem' was considered a surprise in the early days of theorem proving, as no one seems to have thought it intuitive, and it was discovered first by a theorem prover. The theorem is:

```
(=> (and
(forall (?X ?Y ?Z) (=> and (P ?X ?Y) (P ?Y ?Z)) (P ?X ?Z))
(forall (?X ?Y ?Z)(=> (and (Q ?X ?Y) (Q ?Y ?Z)) (Q ?X ?Z)))
(forall (?X ?Y)(=> (Q ?X ?Y) (Q ?Y ?X)))
(forall (?X ?Y) (or (P ?X ?Y)(Q ?X ?Y))) )
(or (forall (?X ?Y) (P ?X ?Y)) (forall (?X ?Y)(Q ?X ?Y))))
```

Tau also handles this problem easily; a test run can be seen at `http://hsinfosystems.com/taujay/doc/samples/tests/Los001.jsp`.

5.2 Identity Problems

As you have seen, Tau solves a variety of identity tests. However, there are two theorems involving identity from [Montague, Kalish, and Mar, 1980] which Tau has not yet been able to prove (except in simplified form) are T328 and T329:

```
T328
(forall (?A ?B ?C)
 (=>
  (and (=> (exists ?Z (forall ?X (<=> (f ?X) (= ?X ?Z))))(f ?A))
       (=> (not (exists ?Z (forall ?X (<=> (f ?X) (= ?X ?Z))))) (= ?A ?C))
       (=> (exists ?Z (forall ?Y (<=> (f ?Y) (= ?Y ?Z)))) (f ?B))
       (=> (not (exists ?Z (forall ?Y (<=> (f ?Y) (= ?Y ?Z))))) (= ?B ?C)) )
  (= ?A ?B)))
```

```
T329
(forall (?A ?B ?C)
 (=>
  (and (=> (exists ?Y (forall ?X (<=> (f ?X) (= ?X ?Y)))) (f ?A))
       (=> (not (exists ?Y (forall ?X (<=> (f ?X) (= ?X ?Y))))) (= ?A ?C))
       (=> (exists ?Y (forall ?X (<=> (g ?X) (= ?X ?Y)))) (g ?B))
       (=> (not (exists ?Y (forall ?X (<=> (g ?X) (= ?X ?Y))))) (= ?B ?C))
       (forall ?X (<=> (f ?X) (g ?X))) )
  (= ?A ?B)))
```

There are particularly interesting problems for intelligent automation, because a human proof would naturally employ (after instantiation) the lemmas which the premises intuitively reveal, as in .e.g. T328, (and (=¿ phi (f a)) (=¿ (not phi) a=c) (=¿ phi (f b)) (=¿ (not phi) (= b c))), together with the knowledge that :- (or phi (not phi)), in order to reach the conclusion expeditiously. Tau's ME strategy, however, becomes swamped with these problems when run under its normal limits. The adoption of a named subformula / rewriting strategy, together with an intelligent lemmaization scheme is clearly called for in such cases, and we are now working on adding lemmaization to Tau.

See [Bachmair and Ganzinger, 1998] for more discussion of identity handling in theorem provers.

5.3 Theory of a Successor, Presburger and Peano Arithmetic

We denote the theory of a successor, Succ. It is a subtheory of Peano Arithmetic, expressed in KIF by:

```
(forall ?X (not (= 0 (succ ?X))))
(forall ?X (forall ?Y (=> (= (succ ?X) (succ ?Y)) (= ?X ?Y))))
```

Tau tests in the theory Succ are at:
`http://www.hsinfosystems.com/taujay/doc/samples/testsJSP.html#Succ`

Note that to prove even the simple (forall ?X (not (= ?X (succ ?X)))) requires the use of mathematical induction, discussed in the next section.

Presburger Arithmetic axioms (the theory PrA), is also a subtheory of Peano arithmetic, lacking multiplication; it is decidable, but already has difficult computational complexity. It is expressed in KIF by the axioms:

```
(forall ?X (not (= 0 (succ ?X))))
(forall ?X (forall ?Y (=> (= (succ ?X) (succ ?Y)) (= ?X ?Y))))
(forall ?X (= (+ ?X 0) ?X))
(forall ?X (forall ?Y (= (+ ?X (succ ?Y)) (succ (+ ?X ?Y)))))
```

Tau tests in the theory PrA are at:
`http://www.hsinfosystems.com/taujay/doc/samples/testsJSP.html#Presburger`

Peano Arithmetic adds the multiplication axioms:

```
(forall ?X (forall ?Y (= (* 0 ?X) 0)))
(forall ?X (forall ?Y (= (* ?X (succ ?Y)) (+ (* ?X ?Y) ?X))))
```

Tau tests in simple Peano Arithmetic are at:
`http://www.hsinfosystems.com/taujay/doc/samples/testsJSP.html#Enderton`
and tests using induction at:
`http://www.hsinfosystems.com/taujay/doc/samples/testsJSP.html#PAI`.

Combinations and reductions of these sets of axioms (PA, PrA, and Succ), together with the introduction of definitions give us other theories, which we will denote below, while presenting some sample axioms of each theory. Each of these theories may also be extended by the use of induction. Some of the theories involve only 'succ', some involve addition and multiplication also. There are various courses possible with extension

by definition and with axiomatization by primitives. For example, '<' may be defined axiomatically in an extension of the theory Succ; alternatively, it may be defined in PrA. Tau also has sample problems which are extensions of PrA and PA: these involve various definitions of the predicates 'even', 'odd', '=<', and others.

See [Enderton, 2001] for further discussion.

5.4 Mathematical Induction

Mathematical induction may be used (see the checkbox on the Tau website) in Tau: (all instances of)

$$(=>\ (and\ (F\ 0)\ (forall\ ?X\ (=>\ (F\ ?X)\ (F\ (succ\ ?X))))\ (forall\ ?X\ (F\ ?X)))),$$

where F represents any formula with one free variable.

Note that some proofs in Peano arithmetic and Presburger arithmetic require that Tau apply mathematical induction or use results which haven been previously proved by induction, while others do not. It is a good exercise for the user or student to determine what the dependencies are.

A sample proof using induction is may be run at `http://hsinfosystems.com/taujay/doc/samples/PALT.web/PALT02Ind.html`. You will note that in this proof a logical axiom for an identity substitution is also supplied as a premise. We have found that, in some instances, the Brand transformations are speeded up thereby, when heuristic search is also used.

Below is the ME proof, followed by statistics.

```
Proof Chain -- 5 steps:

Root:
{  (not (< (+ (succ $MVI-1) $UI-1) (* (succ $MVI-1) $UI-1))) }
Extend:
{  (not (< ?Fl1-17p ?Y-17q));
        (< ?X-17r ?Y-17q);
   (not (= ?Fl1-17p (succ ?X-17r))) }
[ ?X-17r -> (+ (succ $MVI-1) $UI-1), ?Y-17q ->
                        (* (succ $MVI-1) $UI-1) ]

Clause:
{ [(not (< (+ (succ $MVI-1) $UI-1) (* (succ $MVI-1) $UI-1)))];
   (not (< ?Fl1-17p (* (succ $MVI-1) $UI-1)));
   (not (= ?Fl1-17p (succ (+ (succ $MVI-1) $UI-1)))) }
Extend:
{       (= ?Eq-2r0 ?Eq-2r0) }
[ ?Fl1-17p -> (succ (+ (succ $MVI-1) $UI-1)), ?Eq-2r0 ->
                             (succ (+ (succ $MVI-1) $UI-1)) ]

Clause:
{ [(not (< (+ (succ $MVI-1) $UI-1)
```

```
          (* (succ $MVI-1) $UI-1)))];
   (not (< (succ (+ (succ $MVI-1) $UI-1))
               (* (succ $MVI-1) $UI-1))) }
Extend:
{  (not (< (succ 0) ?X-9tt));
   (not (< (succ 0) ?Y-9tu));
        (< (succ (+ ?X-9tt ?Y-9tu)) (* ?X-9tt ?Y-9tu)) }
[ ?X-9tt -> (succ $MVI-1), ?Y-9tu -> $UI-1 ]

Clause:
{ [(not (< (+ (succ $MVI-1) $UI-1) (* (succ $MVI-1) $UI-1)))];
  [(not (< (succ (+ (succ $MVI-1) $UI-1))
                  (* (succ $MVI-1) $UI-1)))];
   (not (< (succ 0) (succ $MVI-1)));
   (not (< (succ 0) $UI-1)) }
Extend:
{       (< (succ 0) $UI-1) }

[ <empty> ]

Base step:

Clause:
{ [(not (< (+ (succ $MVI-1) $UI-1) (* (succ $MVI-1) $UI-1)))];
  [(not (< (succ (+ (succ $MVI-1) $UI-1))
                   (* (succ $MVI-1) $UI-1)))];
   (not (< (succ 0) (succ $MVI-1))) }

Extend:
{       (< (succ 0) (succ $MVI-1)) }
1
[ <empty> ]

Clause:
{  }
```

For the logical basis of this application, see, e.g.,: [Bundy, 2001]

5.5 Graph Theory

We have axiomatized in KIF several problems in finite graph theory. A sample can be seen at `http://hsinfosystems.com/taujay/doc/samples/GMGT.web/GM08d.html` . These problems are over very small domains, so the universal quantifiers are equivalent to finite conjunctions of atomic sentences, and the existential quantifiers are equivalent to finite disjunction of atomic sentences. Accordingly, we have taken advantage of these equivalences to introduce corresponding proof rewrites into Tau, for such problems.

5.6 Sample Statistics

Through the command line interface, Proof statistics can be displayed for each submitted problem, as previously shown, or for each sub-proof in a submitted problem, or collectively for batch runs of multiple problems. The sample statistics shown next are for a batch test suite of 233 problems, many taken from . Note that the inferences counted are complex Model Elimination inferences, including factorizations, subsumptions, reductions and extensions, not simply resolutions. Tau's speed of inference upon these problems runs from a few hundred per second up to tens of thousands per second, depending upon the logical complexity of the clausal forms, and upon the specific search mechanism invoked. The overall average for this test suite was about a thousand inferences per second.

Proved: The number of problems successfully proved out of the total number of problems submitted followed by the total number of inference steps in the resulting proofs and lastly the average length of the proofs obtained over the entire test run.

Roots: The number of root clauses used to prime the Model Elimination search tree. This number is greater than or equal to the number of problems attempted because each FOF in the conclusion in general produces multiple clauses, each of which must be used as a root for a Model Elimination proof search.

Inputs: The total number of clauses submitted, whether they derive from conclusions to prove or from the premises.

Horn / Definite / Other: Histogram of input clauses according to type. (Horn clauses have at most one positive literal, Definite clauses have exactly one literal.)

Root Literals: The number of literals in the clauses used as Model Elimination proof search tree roots.

Side Literals: The number of literals in clauses used as Model Elimination side or auxiliary clauses.

Generated: The number of clauses produced before the proof attempt concluded, whether successfully or not.

Expanded: The number of clauses processed by application of the Model Elimination inference operations.

Derivations: The number of successful (new clause-producing) applications of Model Elimination inference operations.

Factorizations: The number of successful applications of the Model Elimination factorization operation.

Reductions: The number of successful applications of the Model Elimination reduction operation.

Extensions: The number of successful applications of the Model Elimination extension operation.

Totals: Proved: 216 of 233; 1755 inferences; 8.12 Inferences/Proof Elapsed Time: 708.54 sec; Roots: 977; Inputs: 8858; Horn / Definite / Other: 8101 / 6329 / 483; Root Literals: 4001; Side Literals: 30841; Generated: 320434; Expanded: 276289; Derivations: 319457; Factorizations: 1971; Reductions: 22414; Extensions: 295511; Out-order: 1.16; Residual: 1651309; Too deep: 0; Unacceptable: 30051; XUnacceptable: 1815; Subsumed: 106; Vacuous: 7542;

6 Disproofs

Tau can in certain cases disprove theorems. A few of our tests are deliberate disproofs, as a check on soundness. These tests are positively proved invalid and are noted as such when they are run. The soundness of such disproof depends upon the prover's noticing in simple cases that it has exhausted all the possibilities for obtaining a proof by contradiction. When the Model Elimination proof tree is finite and we can build it fully without deriving the empty clause / chain, then we have disproved the conjecture. Simple and not always possible, but when it is, that's all there is to it when these conditions are recognized. A refinement which Tau uses in certain cases is to notice that a theorem has only a very small number of finite models, up to isomorphism, via recognition of identity constraints. A typical disproof (from [Thomas, 1977]) may be run at `http://hsinfosystems.com/taujay/doc/samples/tests/Th267-gA.jsp`.

7 Next Extensions of Tau

"De l'audace, encore de l'audace, et toujours de l'audace!" - Danton

Tau is an ongoing project and its authors plan to follow several paths, including: implementation of MathML notation; implementation of notation for the treatment of variable-binding operators in [Montague, Kalish, and Mar, 1980] (see Chapters X and XI), and of theorem schemata; persistent KB storage across browser sessions; formal language translation facilities; simplified English translation facilities; translation between systems of logic (i.e., intuitionistic and FOL); formal definitions (work partially implemented); implementing the use of metalogical expressions in deduction, as axiom schemata and, particularly, in forming inductive axioms (work now underway); implementation of HOL and the use of theorem schemata; implementation of sequent style proofs (work now underway) [For an introduction to, and discussion of sequent calculi, see, e.g., [Robinson and Voronkov (Eds.), 2002] or [Buss, 1998]]; further TPTP testing (work now underway); graphical notations.

We hope that users will find Tau stimulating.

References

[Apache Jakarta Project] The Tomcat servlet container, a product of the Apache Jakarta Project; `http://jakarta.apache.org/tomcat/index.html`.

[Bachmair and Ganzinger, 1998] Bachmair, Leo and Harald Ganzinger. "Equational reasoning in saturation based theorem proving", in Wolfgang Bibel and Peter H. Schmidt, editors, Automated Deduction: A Basis for Applications. Volume I, Foundations: Calculi and Methods, pages 353–398. Kluwer Academic Publishers, Dordrecht, 1998.

[Baaz et al, 2002] Baaz, Matthias, Uwe and Leitsch, "Normal Form Transformations", in Robinson and Voronkov (Eds), Handbook of Automated Reasoning (2 vols), MIT Press, Cambridge, 2002.

[Brand, 1975] Brand, Daniel. "Proving theorems with the modification method", SIAM Journal on Computing, 4(4):412430, 1975.

[Bundy, 2001] Bundy, Alan. "The Automation of Proof by Mathematical Induction", Handbook of Automated Reasoning 2001: 845-911

[Buss, 1998] Buss, Samuel R. (Ed.), Handbook of Proof Theory, Elsevier, New York, NY, 1998

[Chang and Lee, 1973] Chang, C.L., and R. C. T. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[Common Logic Standard] Common Logic Standard, an ISO effort towards an international standard for Common Logic `http://philebus.tamu.edu/cl/`

[Degtyarev and Voronkov, 1999] Degtyarev, Anatoli and Andrei Voronkov. "Equality reasoning in sequent-based calculi". In Alan Robinson and Andrei Voronkov, editors, Handbook of Automated Reasoning, Elsevier Science Publishers, 1999.

[Enderton, 2001] Enderton, Herbert B., A Mathematical Introduction to Logic, 2nd Ed., Harcourt Academic Press, New York, 2001

[Gottlob and Leitsch, 1985] Gottlob, G. and L. Leitsch. "On the efficiency of subsumption algorithms", Journal of the ACM, Volume 32, Issue 2, April 1985, pp. 280 - 295; `http://doi.acm.org/10.1145/3149.214118`

[Hooker, 1993] Hooker, J.N. "Solving the incremental satisfiability problem", Journal of Logic Programming 15 (1993) 177-186.

[Hooker, 1993a] Hooker, J.N. "New methods for computing inferences in first order logic", Annals of Operations Research (1993) 479-492.

[Inoue, 1992] Inoue, K. "Linear resolution for consequence finding", Artificial Intelligence, 56:301–353, 1992.

[Loveland, 1968] Loveland, D.W. "Mechanical theorem-proving by model elimination," Journal of the ACM, Volume 15, Issue 2, April 1968, pp. 236-251; http://doi.acm.org/10.1145/321450.321456,ACM DOI bookmark.

[Loveland, 1969] Loveland, D.W. "A simplified format for the model elimination procedure", Journal of the ACM, Vol. 15, Issue 2 1969, pp. 349-363; http://doi.acm.org/10.1145/321526.321527,ACM DOI bookmark.

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

[Martelli and Montanari, 1982] Martelli, Alberto and Ugo Montanari, "An Efficient Unification Algorithm", ACM Trans. Program. Lang. Syst. 4(2): 258-282 (1982).

[Martelli and Montanari, 1977] Martelli, A., and Montanari, U. "Theorem proving with structure sharing and efficient unification", Internal Rep. S-77-7, Ist. di Scienze della Informazione, University of Pisa, Pisa, Italy; also in Proceedings of the 5th International Joint Conference on Artificial Intelligence, Boston, 1977, p. 543.

[Kalish and Montague, 1964] Montague, Richard and Donald Kalish, Logic, Techniques of Formal Reasoning, New York, Harcourt, Brace and World, Inc., 1964.

[Montague, Kalish, and Mar, 1980] Montague, Richard , Kalish, Donald, and Mar, Gary (Ed. Robert Fogelin), Logic: Techniques of Formal Reasoning, Harcourt Brace, New York, 1980.

[Nonnengart and Weidenbach, 2002] Andreas Nonnengart, Christoph Weidenbach, "Computing Small Clause Normal Forms", In Alan Robinson and Andrei Voronkov, editors, Handbook of Automated Reasoning. Elsevier Science Publishers, 1999.

[Robinson, 1965] Robinson, J.A. "A machine-oriented logic based on the resolution principle", Jour. Assoc. for Comput. Mach., 1965, 23-41.

[Robinson, 1971] Robinson, J.A., "Computational logic: The unification computation", In Machine Intelligence, vol. 6, B. Meltzer and D. Michie (Eds.). Edinburgh Univ. Press, Edinburgh, Scotland, 1971, pp. 63-72.

[Robinson and Voronkov (Eds.), 2002] Robinson and Voronkov (Eds), Handbook of Automated Reasoning (2 vols), MIT Press, Cambridge, 2002.

[Sun Microsystems] http://java.sun.com/.

[Stickel, 1984] Stickel, M.E., "A Prolog technology theorem prover", New Generation Computing, 1984, 371-383.

[Sutcliffe, 2001] Sutcliffe, Geoff and Suttner, Christian, "The TPTP (Thousands of Problems for Theorem Provers) Problem Library", http://www.cs.miami.edu/~tptp/.

[Thomas, 1977] Thomas, James A., "Symbolic Logic", Merrill, Columbus, Ohio, 1977.

The implementation of Logiweb

Klaus Grue
Dept.Comp.Sci. University of Copenhagen
grue@diku.dk

Abstract

This paper describes the implementation of the 'Logiweb' system with emphasis on measures taken to support classical reasoning about programs.

Logiweb is a system for authoring, storing, distributing, indexing, checking, and rendering of 'Logiweb pages'. Logiweb pages may contain mathematical definitions, conjectures, lemmas, proofs, disproofs, theories, journal papers, computer programs, and proof checkers.

Reading Logiweb pages merely requires access to the World Wide Web. Two example pages are available on `http://yoa.dk/`. Writing, checking, and publishing Logiweb pages requires Logiweb to be downloaded and installed.

Logiweb comes with a hierarchy of features: Lemmas and proofs are stated in a theory named 'Map Theory', Map Theory is implemented on top of a calculus named 'Logiweb sequent calculus', and Logiweb sequent calculus is implemented on top of the 'Logiweb reduction system' (a version of λ-calculus). The Logiweb reduction system is implemented in the Logiweb core software which is currently implemented in Common Lisp.

The levels above the Logiweb core software are defined on Logiweb pages, allowing users to use the features as they are or to define and publish new ones on new Logiweb pages. As an example, a user may want to use ZFC in place of Map Theory, in which case the easiest approach is to publish a Logiweb page that defines ZFC in Logiweb sequent calculus and proceed from there.

The 'base' page on `http://yoa.dk/`, which is 180 pages long when printed out, was checked in 40 seconds. This is non-trivial to achieve for a proof checker implemented in lamdba calculus.

The Logiweb sequent calculus is defined on the base page mentioned above. A user who wants to define e.g. ZFC set theory on top of that may publish a new page, call it 'zfc', and let the 'zfc' page reference the 'base' page. That makes all definitions on the 'base' page available to the 'zfc' page. After that, another user may state and prove lemmas about e.g. real numbers on a third page, call it 'real', which references the 'zfc' page. When the proofs on the 'real' page are checked, Logiweb will arrange that the 'zfc' and 'base' pages are available in a predigested form suitable for proof checking.

Seen from the point of view of proof checking and publication, the World Wide Web has the drawback that once submitted pages can be modified after submission. In the example above, modification of the 'base' page could ruin the correctness of the 'real' page.

To avoid problems with pages being modified, Logiweb implements its own referencing system which forces immutability upon once submitted pages. Once a Logiweb page is submitted, it cannot be changed, just like papers cannot change after publication.

When a Logiweb page is submitted, a unique Logiweb 'reference' is computed from its contents. The Logiweb system allows to look up a Logiweb page given its reference.

Once a Logiweb page is submitted, it may be moved and duplicated such that its http url may change and such that a page may be available many places in the world under different urls, but the Logiweb reference remains constant. One of the tasks of Logiweb is to keep track of the relation between the fixed references and the associated, fluctuating set of http urls.

1 Introduction

Logiweb is a web-like system that allows mathematicians and computer scientists to web-publish pages with high typographic quality and high human readability which are also machine verifiable. Among other, Logiweb allows pages to contain definitions of formal theories, definitions of new constructs, programs, lemmas, conjectures, and proofs. Furthermore, Logiweb allows pages to refer to each other across the Internet, and allows proof checking of proofs that span several pages that reside different places in the world. As an example, a lemma on one page may refer to a construct which is defined on another page situated elsewhere, in which case the proof checker must access both pages to establish the correctness of the proof.

Logiweb is accumulative and provides a medium for archived mathematics. In contrast, the World Wide Web, which supports mathematics through MathML and OMDoc [Koh03, MS01], is a medium suited for information in flux.

Like the Internet and the WWW, Logiweb is a robust, 'anarchistic' system that runs without any central authority; it has been designed in the hope that such a system is the missing piece of software for widespread usage of automated reasoning.

Currently, Logiweb is used as it is, but it also has the potential to run silently and transparently underneath other systems like Mizar [Muz93, TB85]. Support for other systems requires substantial effort, but the hooks for doing so in many different ways are available in Logiweb.

Logiweb gives complete notational freedom to its users as well as complete freedom to choose any axiomatic theory (e.g. ZFC) as basis for their work. Logiweb also allows different notational systems and theories to co-exist and interact smoothly.

Logiweb was originally designed to support Map Theory [BG97, Gru92, Ska02, Val03] which has the same power as ZFC but relies on very different foundations in that, e.g., it relies on λ-calculus *instead* of first order predicate calculus. However, Logiweb has been designed such that it supports all axiomatic theories equally well so the ability to support Map Theory should be seen as a widening rather than a narrowing of the scope.

Logiweb puts no restrictions on what logic is used in the sense that it can support any theory for which one can program a mechanical proof checker. The ease with which Logiweb supports highly distinct theories like ZFC and Map Theory indicates that use of arbitrary logic is not only possible but also feasible. Logiweb supports classical as well as intuitionistic logic, it supports theories built on first order predicate calculus as well as other brands of theories, and it supports theories (such as Map Theory) which admits general recursive definitions.

The absence of restrictions on the choice of logic of course makes it impossible to

supply a code-from-theorems extraction facility like `term_of` of Nuprl [CAB+86], but functions for manipulation of theorems and proofs of individual theories are expressible in the programming language of Logiweb.

One goal of Logiweb was to design a simple proof system which allows to cope with the complexity of mathematical textbooks. To ensure that the system can cope with the complexity of a full, mathematical textbook in a human readable style, two books [Gru01, Gru02] have been developed 1992-2002 to test the system.

Reference [Gru01] is a discrete math book for first year university students and is of interest here because it has been possible to test the human readability of the book in practice. The associated course has been given ten times with a total of more than a thousand students. The course has been a success and runs as the first course on the computer science curriculum at DIKU in parallel with a course on ML.

Reference [Gru02] is a treatise on Map Theory and is of interest here because it contains a substantial proof (a proof of the consistency of ZFC expressed in Map Theory) that can stress test Logiweb. To allow comparison with other proof systems and to ensure correctness, [Gru02] has been ported by hand to Isabelle [Pau98a, Pau98b, Ska02].

At the time of writing, Logiweb is used on a graduate course in logic (c.f. `http://www.diku.dk/~grue/logiweb/20050502/home/index.html`) and Logiweb is being adapted according to user requests. After that, it is the intension to run first [Gru02] and then [Gru01] through the system. Running those two books through Logiweb requires adaption of the books to the current syntax of the Logiweb compiler plus programming of a number of proof tactics that are described but not formally defined in the books. Running [Gru02] through Logiweb will also allow a comparison with Isabelle.

Map Theory essentially is the Logiweb programming language extended with a quantifier. As a long term goal, this makes it interesting to use Map Theory to reason *about* Logiweb, possibly leading to a situation where one can solve the academic exercise to let a proof checker prove its own correctness. A more immediate application is to use Map Theory to reason about code fragments expressed in the Logiweb programming language as is done in [Gru01].

1.1 Overview of the paper

Logiweb is a simple system with a simple programming language, a simple macro expansion facility, a simple proof checking facility, a simple protocol for exchange of documents, a simple format for storing Logiweb pages and so on. While each feature is simple in itself, the sum of features may make Logiweb look complex at first sight. For a comprehensive introduction to Logiweb, consult Logiweb itself at `http://yoa.dk/` and read the 'base' page.

The present paper gives an overview of the system from an implementation perspective in Section 1.2 and from a user perspective in Section 2. Then Section 3 describes how Logiweb pursues its goal to allow classical reasoning about programs without sacrificing generality and efficiency of computation. Section 4 describes the data structures used for representing terms, lemmas, proofs, pages, and so on. Section 5 describes the proof checking algorithm and Section 6 summarizes.

1.2 System overview

A user may use the World Wide Web as shown in Figure 1. In the figure, the user may use the text editor to construct an html page and store it in the file system within reach of the http server. Then the user (or another user) may use the html browser to request the html page from the http server which in turn retrieves the html page from the file system.

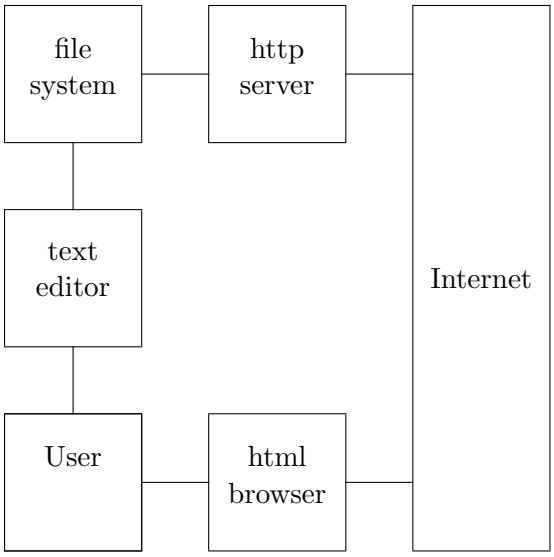


Figure 1: World Wide Web

Figure 2 shows how a user may use Logiweb. To write a Logiweb page, the user prepares a source text and invokes the Logiweb compiler on it. This is similar to running TEX on a TEX source [Knu83]. Actually, much of a Logiweb source consists of TEX source code.

When and if the compiler succeeds in interpreting the source, it translates it to a compressed format, checks its mathematical correctness, and stores it back in the file system in the format of a Logiweb page within reach of the http server. The compiler also renders the page in PDF so that users without a genuine Logiweb browser can view it. After that, any user that knows the url of the page can retrieve it using an html browser.

When the compiler succeeds in translating a Logiweb page, it also computes the Logiweb reference of the page and notifies the Logiweb server (c.f. Figure 2). The Logiweb server keeps track of the relationship between http urls and Logiweb references and makes the relationship available via the Internet using the Logiweb protocol. The Logiweb protocol allows Logiweb servers to cooperate on indexing pages such that each server merely has to keep track of local pages plus some information about which other Logiweb servers to refer non-local requests to.

A Logiweb reference contains a RIPEMD-160 [DBP96] hash key and a time stamp. The RIPEMD-160 hash key is computed on basis of the bytes of the associated page. As long as RIPEMD-160 stands up against collision attacks, not even a malicious user can get away modifying as much as a single byte of a Logiweb page without getting

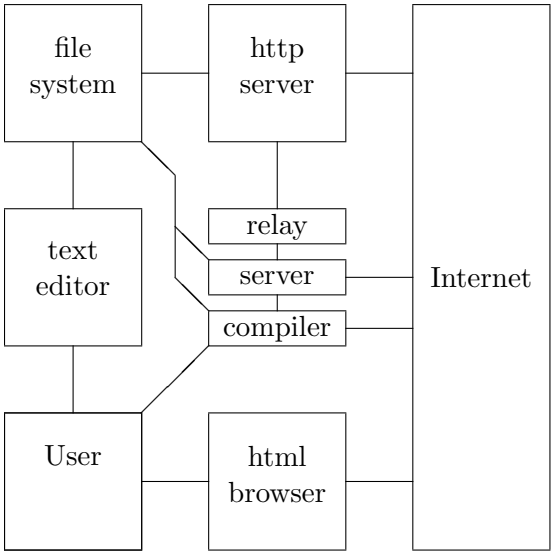


Figure 2: Logiweb

caught by a RIPEMD-160 check.

When the compiler translates a Logiweb page that references other Logiweb pages (which is the normal case), it uses the Logiweb server to locate the references and then transitively loads the referenced pages so that all definitions on transitively referenced pages are available.

When referencing a Logiweb page from the World Wide Web, one may construct an http url from the Logiweb reference by expressing the reference in hexadecimal and prepending it with the url of a Logiweb relay (c.f. Figure 2). A Logiweb relay is a CGI-program which, given a reference, contacts the nearest Logiweb server, translates the reference to an ordinary url, and returns an html indirection to that url. This instructs the html browser of the user to fetch the associated page. The net experience for the user is that clicking a Logiweb reference in an html document makes the html browser navigate to the referenced Logiweb page.

Referencing from Logiweb pages to html pages is trivial but not necessarily advisable since the immutability of Logiweb pages makes it impossible to repair broken links.

In addition to the Logiweb server, compiler, and relay mentioned above, the current implementation of Logiweb includes an 'lgwping' program which allows to ping a Logiweb server to see if it is responding.

For more details on Logiweb see `http://yoa.dk/` or [Gru04].

2 A Logiweb tutorial

2.1 Hello world

To give an overview of the system from a user perspective, we now follow the first steps of a new user. The steps are close to the steps actually followed by the current users (c.f. `http://www.diku.dk/~grue/logiweb/20050502/home/index.html`).

Previous versions of Logiweb offered a WYSIWYG authoring tool, but that has been abandoned until further and replaced by a lean and mean compiler that offers high speed and reasonably intelligible error messages, but no help beyond that. In other words, our new user is in a situation that resembles the situation of the first time user of a new programming language.

So a reasonable way to get started is to copy the source text of a "hello world" Logiweb page and try to compile that. The source of a "hello world" page is available at `http://www.diku.dk/~grue/logiweb/20050502/home/grue/hello-world/fixed/source/source.pyk`. The essential lines read

```
\begin{document}
"[ math pyk define hello world as "hello world" end define end math ]"
\end{document}
```

which requires quite a lot of explanation to make sense. Instead of looking for an explanation, our new user stores the source text in the file "page.pyk" and runs the compiler by issuing a command like the following:

```
> pyk pyk=page url=http://my.domain/my/directory level=all
```

After that, our user starts an html browser and looks up `http://my.domain/my/directory/hello-world/fixed`, then clicks "body", and then clicks "PDF" to see the following:

$$[\text{hello world} \stackrel{\text{pyk}}{=} \text{“hello world”}]$$

2.2 A minor update

Our new user, encouraged by seeing output from the system, modifies the source:

```
\begin{document}
The definition "[ math pyk define hello world as "hello world" end
define end math ]" defines the name of {\em my} page.
\end{document}
```

Then the user reruns the compiler and asks the html browser to reload the page to get

$$\text{The definition } [\text{hello world} \stackrel{\text{pyk}}{=} \text{“hello world”}] \text{ defines the name of } \textit{my} \text{ page.}$$

A key feature of Logiweb is that pages are immutable, so it may seem peculiar how easily the user changed `http://my.domain/my/directory/hello-world/fixed` above. To Logiweb, however, the two pages have different Logiweb references and the first "hello world" page was immutable as long as it existed. Immutability means that, given a Logiweb reference, one can locate the associate page (if it exists anymore) and, furthermore, one can check whether or not anybody has tampered with the page.

Our user invoked the compiler with a "level=all" argument. That indicates that the backend of the compiler should render not only the page itself but also a lot of additional material. A "level=submit" is equivalent to "level=all", but in addition requests the

compiler to notify the nearest Logiweb server about the submission and to store the page as `http://my.domain/my/directory/hello-world/TIME` where TIME is the date and time of submission. This is useful for versions of a page that are expected to exist for more than a debug round trip.

2.3 Guarding against haphazardness

Now our hopeful user is ready for doing some proof checking. However, suppose the source text of the "hello world" page contains something like

```
BIBLIOGRAPHY
base: "http://yoa.dk/logiweb/page/base/fixed/vector/page.lgw"
```

These lines tell the compiler that the "hello world" page references whatever Logiweb page happens to be at that particular URL at the moment the "hello world" page is translated. The `.lgw` file is the real Logiweb page in a standardized, binary format. If the referenced page is overwritten, and no copies of the page exists anymore, then the "hello world" page will be ruined. So to guard against this, the user issues the following command:

```
> pyk lgw=http://yoa.dk/logiweb/page/base/fixed/vector/page.lgw \
> url=http://my.domain/my/directory level=submit
```

That makes the compiler make a local copy of the given Logiweb page. The local copy will have exactly the same reference as the original, and the local copy ensures that the Logiweb page will remain in existence even if the original instance of the page ceases to exist. Then the user may look up the raw Logiweb reference at `http://yoa.dk/logiweb/page/base/fixed/reference/kana.html` and insert that in the bibliography:

```
BIBLIOGRAPHY
base:nani
nuse siti sete tiku kata  sana susu siku kitu naku
kake sisu suni nusa tini  tesa kika sutu siku neke
saku kesa keke seke kine  suki sise nasa natu
```

This ensures that the "hello world" page will reference the same Logiweb page each time the user re-translates the "hello world" page.

2.4 Defining a theory

Having guarded against the haphazardness of the external world, our user may write

```
Propositional calculus "[ intro prop pyk "prop" tex "L_p" end
intro ]" as defined in \cite{mendelson} is defined thus:
"[ math theory prop end theory end math ]", "[ math in theory
prop rule a one says all meta a indeed all meta b indeed meta
a imply meta b imply meta a end rule end math ]", ...
```

to get

> Propositional calculus [$\mathrm{L_p}$] as defined in [Men87] is defined thus: [**Theory** $\mathrm{L_p}$], [$\mathrm{L_p}$ **rule** A1: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $\mathcal{A} \Rightarrow \mathcal{B} \Rightarrow \mathcal{A}$], [$\mathrm{L_p}$ **rule** A2: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $\forall\mathcal{C}$: $(\mathcal{A} \Rightarrow \mathcal{B} \Rightarrow \mathcal{C}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow \mathcal{A} \Rightarrow \mathcal{C}$], [$\mathrm{L_p}$ **rule** A3: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow (\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}$], and [$\mathrm{L_p}$ **rule** MP: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $\mathcal{A} \vdash \mathcal{A} \Rightarrow \mathcal{B} \vdash \mathcal{B}$].

For a less cramped and more complete example see Section 1.6 of the body of `http://yoa.dk/logiweb/page/check/fixed/`.

The `"[ intro prop pyk "prop" tex "L_p" end intro ]"` in the source above says that the construct named "prop" should be rendered as "`L_p`" in TEX and can be referred to as "prop" on pages referencing the page. Normally, a construct should have the same name on the page and on pages referencing the page, so the latter piece of information is a bit redundant.

2.5 Proving something

Our user may now state a lemma and a proof:

> [$\mathrm{L_p}$ **lemma** I: $\forall\mathcal{A}$: $\mathcal{A} \Rightarrow \mathcal{A}$]

$\mathrm{L_p}$ **proof of** I:

L01:	Arbitrary $\gg$	$\mathcal{A}$	;
L02:	A1 $\gg$	$\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A}$	;
L03:	A1 $\gg$	$\mathcal{A} \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}$	;
L04:	A2 $\gg$	$(\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A}) \Rightarrow$ $(\mathcal{A} \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}$	;
L05:	MP $\triangleright$ L02 $\triangleright$ L04 $\gg$	$(\mathcal{A} \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}$	;
L06:	MP $\triangleright$ L03 $\triangleright$ L05 $\gg$	$\mathcal{A} \Rightarrow \mathcal{A}$	$\square$

2.6 How the page is verified

Logiweb pages are verified by the Logiweb core software. That software would fit naturally into a Logiweb browser. But, at present, there is no such Logiweb browser since Logiweb piggybacks the World Wide Web, and the core software actually resides in the compiler, sandwiched between a frontend and a backend. When the "hello world" page is translated, the compiler does as follows:

The frontend reads the source file and loads all Logiweb pages transitively referenced by the page. During this process, all transitively referenced pages are processed and verified by the core software unless they already reside in the cache of the compiler (which may be saved to disk).

Then the compiler reads declarations of associativity and priority of all constructs used and parses the source. The output from this process is a list of bytes called a "Logiweb vector" in the format used for storing and transmitting Logiweb pages.

At this point, the compiler could store the vector and halt. Instead, the compiler invokes the core software on the vector. The core software unpacks the vector into a "body", a "bibliography", and a "dictionary". The body essentially is one, big Lisp

S-expression, the bibliography is a list of pages directly referenced by the page, and the dictionary contains the arities of all constructs introduced on the page.

Then the compiler invokes a macro engine. That engine is defined on the first, direct reference of the page, i.e. on the "base" page in the case of "hello world". The macro engine on the base page happens to implement an outside-in macro expansion strategy which supports but does not enforce recursive macro expansion. Among other, the base page contains a macro defined by [(x)≝x] which says that parentheses disappear during macro expansion.

After macro expansion, the compiler "harvests" the page, i.e. collects all definitions present on the page (including macro definitions). After harvesting, new macro definitions may affect how the page should have been expanded, so the compiler macro expands the page once more from scratch. The compiler then alternates between expansion and harvesting until a fixed point is reached (if ever) or until the user kills the compiler. The output from this process (if any) is a "codex" which contains all the definitions and an "expansion" which is the macro expanded version of the page.

Then the compiler invokes a verifier. That verifier is also defined on the first direct reference of the page. The verifier on the base page happens to be the conjunction of two verifiers, one that verifies test cases like $[2 + 3 = 5]$ and one that verifies proofs.

Since definitions is what Logiweb collects, anything interesting should be expressed as definitions. As an example, [L_p **lemma** I:$\forall\mathcal{A}$:$\mathcal{A} \Rightarrow \mathcal{A}$] macro expands into a definition that says that the "statement aspect" of [I] equals [$L_p \vdash \forall\mathcal{A}$:$\mathcal{A} \Rightarrow \mathcal{A}$]. [**Theory** L_p] is particularly complicated; it macro expands into a definition that says the the statement aspect of [L_p] equals the intuitionistic conjunction of the four rules of [L_p]. The [**Theory**] macro finds the rules of [L_p] by scanning the codex of the page.

The proof of [I] above macro expands into a definition of the "proof aspect" of [I]. The right hand side of that definition comprises a "proof engine" applied to a quoted version of the macro expanded proof. The proof verifier evaluates the right hand side so that control is passed to the proof engine which in turn evaluates all constructs for which a "tactic aspect" is defined. When all proof tactics are evaluated, the proof verifier returns a term expressed in Logiweb sequent calculus to the proof verifier which then evaluates that according to the rules of that calculus to see if the proof is correct and proves what it is supposed to prove. The proof above uses a unification tactic [$\mathcal{A} \gg \mathcal{B}$] which instantiates $\mathcal{A}$ to fit $\mathcal{B}$.

After verification, regardless of whether the page is correct or not, the compiler invokes the backend to render the body, bibliography, dictionary, codex, expansion, diagnose (in case of errors), reference and vector of the page in a number of different formats.

3 Classical reasoning about programs

Logiweb has been designed with the goal to support classical reasoning about programs. At the same time, however, Logiweb has been designed to be as neutral as possible with respect to choice of logic and notation. In other words, the requirement to support classical reasoning about programs should be seen as a widening of the scope compared to systems which focus on constructive reasoning or which focus on classical mathematics.

Constructive reasoning often leads to unnecessary complications. On the other hand, classical reasoning about programs is non-trivial because general recursion and infinite looping is cumbersome to deal with in classical theories like ZFC set theory. As an example, if $[n! \doteq \mathbf{if}(n = 0, 1, n \cdot (n-1)!)]$ then it is trivial that $[(-1)!]$ loops indefinitely, but that is non-trivial to express and prove in ZFC.

In Logiweb, the chosen solution to that problem is to base all computable definitions on a version of λ-calculus that allows classical reasoning, and to ensure that Logiweb is able to support such classical reasoning.

This is non-trivial for two reasons. Firstly, λ-calculus programs are inefficient compared to programs expressed in other languages unless special measures are taken. Secondly, it is much easier to develop first order predicate calculus in lamdba calculus than the other way round, so Logiweb must support classical logic that is not based on first order predicate calculus (we consider a theory "classical" if it allows proof by cases such as use of the law of excluded middle; the theory we shall arrive at also allows to use the axiom of choice).

In the following we first introduce a version of λ-calculus which is suited to classical reasoning and then deal with the efficiency problem.

3.1 λ-calculus for classical reasoning

Pure λ-calculus [Chu41] is inherently syntactic of nature and cumbersome to handle in classical theories like ZFC. In contrast, impure λ-calculi do support classical reasoning and have models that are classical of nature [BG97].

As an example, λ-calculus to which one adds integers is suited for classical reasoning. λ-calculus enriched with truth values T and F also supports classical reasoning.

Hence, λ-calculus enriched with two or countably many new values is suited to classical reasoning. Actually, λ-calculus enriched with any number of new values from one and up are equally suited to classical reasoning.

To make matters as simple as possible, Logiweb is based on λ-calculus enriched with just one new value plus an operation which makes the new value useful. We shall refer to the new value as T. In terms of the C programming language, a lambda construct corresponds to a pointer to a function and T corresponds to the NULL pointer. Among many things, we shall use T to represent truth, which explains the choice of name.

The λ-calculus used by Logiweb is defined by the Logiweb reduction system $\lambda\mathsf{T}$:

$$\begin{array}{lcl} (\lambda x.y)z & \rightarrow & \langle y|x{:=}z\rangle \\ \mathsf{T}z & \rightarrow & \mathsf{T} \\ \mathbf{if}(\mathsf{T}, y, z) & \rightarrow & y \\ \mathbf{if}(\lambda u.v, y, z) & \rightarrow & z \end{array}$$

3.2 Equality of lambda terms

In pure λ-calculus, two terms are considered "equal" or "beta-equivalent" if they reduce to the same term; this is what makes λ-calculus syntactic of nature and cumbersome to deal with classically. In $\lambda\mathsf{T}$, equality is defined semantically as follows:

Define $\lambda\mathsf{T}\|$ as the system above extended with a new, binary operator $x \parallel y$ and the following reduction rules:

$$\begin{array}{lcl} \mathsf{T} \parallel x & \to & \mathsf{T} \\ x \parallel \mathsf{T} & \to & \mathsf{T} \\ \lambda u.v \parallel \lambda x.y & \to & \lambda z.\mathsf{T} \end{array}$$

We say that a term t of $\lambda\mathsf{T}\|$ is a "true" term if it reduces to T, a "function" term if it reduces to a term of form $\lambda x.y$, and a "bottom" term if it is neither a true nor a function term. We say that two terms s and t are "root equivalent", written $s \sim t$, if they are both true, both function, or both bottom terms. We say that s and t are "equal", written $s = t$, if $fs \sim ft$ for all terms f of $\lambda\mathsf{T}\|$. Terms of $\lambda\mathsf{T}$ are considered equal if they are equal in $\lambda\mathsf{T}\|$.

Classical equality $s = t$ is undecidable (since otherwise $s = \bot$ could decide the halting problem where $\bot \doteq (\lambda x.xx)(\lambda x.xx)$). Furthermore, $s = t$ differs from $\beta\eta$-equivalence $s =_\eta t$. Actually, neither of the two relations imply the other. As an example, $(\lambda x.\lambda y.xy)\mathsf{T} \to \lambda y.\mathsf{T}y$ whereas $(\lambda x.x)\mathsf{T} \to \mathsf{T}$ so $\lambda x.\lambda y.xy \neq \lambda x.x$ even though $\lambda x.\lambda y.xy =_\eta \lambda x.x$. As another example, $\mathsf{Y}\lambda f.\lambda x.\lambda x.f = \mathsf{Y}\lambda f.\lambda x.\lambda x.\lambda x.f$ (they both equal $\lambda x.\lambda x.\lambda x.\cdots$) but the two terms are not $\beta\eta$-equivalent. β-equivalence does imply classical equality.

3.3 Classical reasoning about $\lambda\mathsf{T}$

In $\lambda\mathsf{T}$, any term f satisfies $f = \mathsf{T}$ or $f = \lambda x.fx$ or $f = \bot$, there is no fourth possibility. This 'quartum non datur' (QND) rule makes $\lambda\mathsf{T}$ classical because it allows proof by cases. A two-valued logic like ordinary propositional calculus satisfies a 'tertium non datur' rule whereas a three valued logic like $\lambda\mathsf{T}$ satisfies a 'quartum non datur' rule like the one above. The proof-theoretic strength is the same.

As an example, if we define $\mathsf{F} \doteq \lambda x.\mathsf{T}$ and $x \wedge y \doteq \mathbf{if}(x, \mathbf{if}(y, \mathsf{T}, \mathsf{F}), \mathbf{if}(y, \mathsf{F}, \mathsf{F}))$ then QND allows to prove $x \wedge y = y \wedge x$. If $x \vee y \doteq \mathbf{if}(x, \mathbf{if}(y, \mathsf{T}, \mathsf{T}), \mathbf{if}(y, \mathsf{T}, \mathsf{F}))$ and $\neg x \doteq \mathbf{if}(x, \mathsf{F}, \mathsf{T})$ then $x \vee \neg x = \mathsf{T}$ fails (counterexample: $x = \bot$) but QND allows to prove $x \vee \neg x = !x$ where $!x \doteq \mathbf{if}(x, \mathsf{T}, \mathsf{T})$. In general, QND allows to translate lemmas and proofs of classical propositional calculus to $\lambda\mathsf{T}$.

The QND-inference belongs to Map Theory. In Logiweb, Map Theory has four connections to $\lambda\mathsf{T}$. Firstly, Map Theory allows to reason about $\lambda\mathsf{T}$. Secondly, the reduction rules of $\lambda\mathsf{T}$ are axioms of Map Theory. Thirdly, any $\lambda\mathsf{T}$ program like $x! \doteq \mathbf{if}(x = 0, 1, n \cdot (n-1)!)$ (where integers and $=$ and $\cdot$ on integers is defined suitably) automatically becomes a definition that can be used in Map Theory proofs. Fourthly, the proof checker for Map Theory is implemented in $\lambda\mathsf{T}$.

3.4 Stress test of Map Theory

The ultimate test for a theory is to prove the consistency of ZFC set theory within it. The result itself is not important since ZFC set theory is generally accepted to be consistent, but proving the consistency of ZFC in a theory proves that that theory is as powerful as ZFC theory which in turn is known to be sufficiently powerful to express all of classical and most of modern mathematics.

For Map Theory, [Gru02] contains a formal proof of the consistency of ZFC within Map Theory. Among other, [Gru02] was written to develop the language of Logiweb, so

[Gru02] was expressed in the language of Logiweb before Logiweb was implemented. The correctness of [Gru02] has been established in Isabelle as reported in [Ska02]. Adaption of [Gru02] to the final version of the Logiweb language and proof-checking in that framework is the next, major task in the Logiweb project and will allow comparison with the Isabelle implementation.

Note: If SI denotes the assumption that there exists a strongly inaccessible ordinal then ZFC+SI can prove the consistency of Map Theory [BG97, Gru92] which in turn can prove the consistency of ZFC [Gru92, Gru02]. This supports a claim that Map Theory has strength between ZFC and ZFC+SI. But "strength" is defined on basis of Gödels 1931 paper [Göd31] which only considers theories that build on first order predicate calculus so, for technical reasons, the "strength" of Map Theory is not defined. The claim in Section 1 that Map theory has the "same" power as ZFC is imprecise but close to the truth.

3.5 Efficiency of $\lambda\mathsf{T}$

The core software of Logiweb supports $\lambda\mathsf{T}$ and no other programming language. Since classical reasoning about $\lambda\mathsf{T}$ is possible and rather straightforward ([Gru02], Chapter 6), this ensures the possibility to reason classically about any program expressed in Logiweb.

This leaves two problems: How to handle programs expressed in other languages, and how to ensure efficiency?

The first problem is trivial in principle. To handle e.g. C programs, define a compiler from C to $\lambda\mathsf{T}$ in $\lambda\mathsf{T}$. Such a compiler is typically referred to as a 'semantics' of C.

The second problem is non-trivial, and typical implementations of λ-calculus are inefficient to a degree where a compiler from C to $\lambda\mathsf{T}$ would be of little practical use.

Logiweb has a rather simple solution to the efficiency problem which is described in the following.

3.6 Definitions

Logiweb allows Logiweb pages to contain definitions. As an example, consider the following definitions:

$$\begin{array}{lcl} x::y & \doteq & \lambda z.\mathbf{if}(z,x,y) \\ x^h & \doteq & x\mathsf{T} \\ x^t & \doteq & x\mathsf{F} \\ \mathsf{F} & \doteq & \lambda x.\mathsf{T} \end{array}$$

It is straightforward to prove $(x::y)^h = x$ and $(x::y)^t = y$ so $x::y$ is a pairing construct and x^h and x^t are the associated 'head' and 'tail' operations.[1]

If the definitions above are stated on a Logiweb page P, then they will be available in P as well as all pages referencing P. The definitions effectively extend the Logiweb reduction system with new reduction rules like $x::y \rightarrow \lambda z.\mathbf{if}(z,x,y)$.

[1] In Map Theory, which has inference rules of transitivity and substitution of equals and which has all $\lambda\mathsf{T}$ reduction rules and all valid $\lambda\mathsf{T}$ definitions as axioms, a proof of $(x::y)^h = x$ essentially reads $(x::y)^h = (x::y)\mathsf{T} = (\lambda z.\mathbf{if}(z,x,y))\mathsf{T} = \mathbf{if}(\mathsf{T},x,y) = x$.

Computation of e.g. $(x::y)^h = x$ will not be particularly efficient, however. It is possible to implement the pairing operation much more efficiently than using $\lambda z.\mathbf{if}(z, x, y)$.

To allow efficient implementation without affecting the ability of classical reasoning, Logiweb has two definition constructs, which we shall refer to as $\doteq$ and $\doteqdot$. Formally,

$$x::y \doteq \lambda z.\mathbf{if}(z, x, y)$$

and

$$x::y \doteqdot \lambda z.\mathbf{if}(z, x, y)$$

mean exactly the same. Backstage, however, Logiweb has a finite list of 'optimized functions' which Logiweb can compute efficiently. For each optimized function, Logiweb stores both the efficient version of the function and the 'semantics' of the function. The 'semantics' of an optimized function is a definition of the function expressed in $\lambda\mathsf{T}$. When Logiweb sees a definition like

$$x::y \doteqdot \lambda z.\mathbf{if}(z, x, y)$$

it searches its list of optimized functions for one whose 'semantics' is identical to

$$\lambda z.\mathbf{if}(z, x, y)$$

(except for naming of bound variables). If Logiweb finds a match, it translates $x::y$ to the optimized function found. Otherwise, Logiweb treats $\doteqdot$ like $\doteq$. A "match" must be exact (modulo naming). As an example, $\lambda z.\mathbf{if}(z, (\lambda x.x)x, y)$ does not match $\lambda z.\mathbf{if}(z, x, y)$.

The $\doteq$ and $\doteqdot$ constructs are identical from the point of view of reasoning as long as optimized functions behave exactly as specified by their semantics. It is the responsibility of the implementer of the core software to ensure this correspondence.

Distinct implementations of Logiweb may have different lists of optimized functions; that may affect the speed of a computation but not the result of a computation.

Actually, the current implementation of Logiweb has no optimized function for the untagged pair construct $x::y$ above. Instead, that implementation has an optimized function for a particular tagged pair construct, and also has optimized functions for handling cardinals (i.e. non-negative integers). Finally, the current implementation of Logiweb does some type inference and strictness analysis to make programs run fast (c.f. Section 3.6 of the base page). All that is invisible from the point of view of reasoning about programs.

3.7 Feasibility

The measures above and those mentioned in Section 4 have proven sufficient to make it feasible to implement macro expansion and proof checking on top of $\lambda\mathsf{T}$. As mentioned in the abstract, the 'base' page on `http://yoa.dk/`, which is 180 pages long when printed out, was macro expanded and checked in 40 seconds.

Each time an efficiency enhancement was implemented, the efficiency gain was measured in a rather crude way (with a manual stop watch on an otherwise unloaded

machine). The product of efficiency gains indicate a total speed-up around 10^9 with an uncertainty of several orders of magnitude. If the 10^9 figure is correct, the base page would take around 1200 years to macro expand and check without optimizations.

The 40 second measure is just a feasibility demonstration, not one suited for comparison with other systems. The measure shows that Logiweb can survive a 180 page document with around 1500 definitions, 180 test cases, and 10 proof lines. The 40 seconds are mainly spent on macro expanding the rather complex base page seven times (for macro expansion iteration see Section 2.6). Applying Logiweb to longer proofs is currently done by about ten students on a graduate course.

4 Data structures

In this section we describe the data structures Logiweb uses when verifying pages. The choice of such data structures has proven to impact the efficiency of verification considerably.

4.1 Terms

The current implementation of Logiweb has a pairing function and support for cardinals (non-negative integers) among its optimized functions. We shall refer to the pairing function as $x::y$ even though it differs from the particular pairing function defined in Section 3.

The term is one of the most fundamental data structures of a system for formal logic. Logiweb terms are data structures implemented using cardinals, $x::y$, and T.

Logiweb terms are used for representing ordinary terms like $2+3$. But they are also used for representing formulas like $\forall x\colon x+1=1+x$. Furthermore, Logiweb terms are used for representing lemmas and proofs. Actually, an entire Logiweb page is one big term as seen from the point of view of Logiweb.

Logiweb terms are trees whose nodes are labeled by 'Logiweb symbols'. A Logiweb term with root r and subterms $t_1,\ldots,t_n$ is represented by

$$r::t_1::\cdots::t_n::\mathsf{T}$$

This representation of terms is effectively the same as a Lisp S-expression [McC60]. The differences are (1) Logiweb symbols differ from Lisp symbols/Lisp atoms, (2) Logiweb terms are terminated by T where Lisp S-expressions are terminated by NIL, and (3) each Logiweb symbol has an arity which must match the n above.

As mentioned previously, each Logiweb page has a reference r. That reference is a sequence of bytes when stored on disk or transmitted via a network, but when handled inside Logiweb software, it is a cardinal.

Each Logiweb page declares a finite number of Logiweb symbols. Each Logiweb page has a unique reference r and each symbol declared by a page has an identifier i which is unique within the page, so a Logiweb symbol is uniquely determined by r and i together.

A Logiweb symbol with reference r and identifier i is represented by a structure of form $r::i::d$. The last item d in a Logiweb symbol comprises debugging information

which is irrelevant to formal reasoning. The debugging information notes where the term was located before macro expansion and thus allows to produce meaningful error messages.

The term that makes up a Logiweb page can only contain symbols from the page itself and pages directly referenced by the page. After macro expansion, the term can contain symbols from the page itself and pages transitively referenced by the page.

Large parts of a Logiweb page typically consists of ordinary text. Ordinary text is encoded as terms inside the Logiweb software. When stored on disk or transmitted over a network, care has been taken to encode text efficiently. In particular, Unicode characters below 128 (i.e. ASCII characters) take up one byte each. Ordinary text of Logiweb pages is expressed as TEX source text.

4.2 Arrays

Concerning efficiency, the main drawback of pure functional programming is the lack of constant time array access. Logiweb is based on $\lambda\mathsf{T}$ which certainly is a pure functional programming language, and constant time array access is not tenable. Fortunately, it is not important whether or not array access is constant in time. It is more important that array access is fast.

Apart from terms, the main data structures of Logiweb are based on 'Logiweb arrays'. A Logiweb array a represents a function f from cardinals to arbitrary data which has the property that $f(n) \neq \mathsf{T}$ holds for at most finitely many cardinals n.

We shall refer to the value associated to the cardinal n by the array a as $a[n]$ and to the set $\{n|a[n] \neq \mathsf{T}\}$ as the 'domain' of a.

A Logiweb array a is represented as a binary tree whose leafs have form $n::x$ where $x \neq \mathsf{T}$. A leaf of form $n::x$ represents the information that $a[n] = x$.

A leaf of form $n::x$ is placed at a location in a which depends on the index n and on what other indices are stored in the array. As an example, consider a leaf of form $6::\mathsf{F}$ which indicates that $a[6] = \mathsf{F}$. The binary expansion of 6, written with the least significant bit first, reads $01100000\cdots$. The address at which the leaf $6::\mathsf{F}$ is placed in a is the shortest prefix of $01100000\cdots$ which distinguishes 6 from all other indices of the array a.

As a result, the access time of an array a with a contiguous domain depends on the logarithm of the size of the domain. The access time of a sparse array with randomly distributed indices also depends on the logarithm of the size of the domain.

The arrays used in Logiweb are typically accessed either by small cardinals (e.g. identifiers of Logiweb symbols) or by randomly distributed cardinals (e.g. references of Logiweb symbols).

4.3 Logiweb codices

A definition like $\mathsf{F} \doteq \lambda x.\mathsf{T}$ defines the value of F, and a system for mathematical reasoning certainly must keep track of such definitions. Logiweb collects all definitions present on a Logiweb page in a data structure which we shall refer to as a Logiweb codex, c.f. Section 2.6.

A value definition like $\mathsf{F} \doteq \lambda x.\mathsf{T}$ is what one normally thinks of as a definition. But a computational system must handle many other kinds of definitions: definitions of how

constructs macro expand, how they should be rendered, how a user should input them via a keyboard, and many other things.

Logiweb handles different kinds of definitions by the introduction of Logiweb 'aspects'. Each definition in Logiweb consists of a left hand side, a right hand side, and an aspect. As an example, definition of three aspects of F could read

$$\begin{aligned}\mathsf{F} &\overset{\text{val}}{=} \lambda x.\mathsf{T}\\ \mathsf{F} &\overset{\text{tex}}{=} \text{“\mathsf\{F\}”}\\ \mathsf{F} &\overset{\text{pyk}}{=} \text{“false”}\end{aligned}$$

The first definition defines the value aspect of F. Or, stated in a more straightforward way, it defines the value of F. The second definition defines how F should be rendered and the third definition states what a user should type on a keyboard or say in a microphone to enter an F to an authoring tool. On traditional, site based proof checkers one typically stores "pyk"- and "tex"-like information separately, but when sending pages around on the internet, each page must be a capsule containing all information needed to e.g. render the page. The latter two definitions above form a convenient way to include information that is normally hidden away. The "intro" construct in Section 2.4 macro expands into pyk and tex definitions (see the base page for a precise definition).

The macro facility allows to keep the de Bruijn factor[2] low. The macro facility allows authors to write pages in a style that is appealing to the human reader but still macro-reduces into a more machine understandable form. The author of a page can define a construct to be a macro by defining its macro aspect.

In Logiweb, aspects are represented by symbols. Definitions contain a left hand side (which may contain parameters), an aspect (which may also contain parameters), and a right hand side. When Logiweb "codifies" a Logiweb page, it macro expands it and collects definitions from it, leading to a "codex" and a macro expanded version of the page. The macro expanded version is a term whereas the codex is an array c with the property that

$$c[r_s][i_s][r_a][i_a]$$

is the definition of the aspect with reference r_a and identifier i_a for the symbol with reference r_s and identifier i_s.

The codex allows fast access to any aspect of any symbol during verification.

4.4 Logiweb racks

We shall refer to an array that contains heterogeneous data as a 'rack' and to each index of a rack as a 'hook'.

Logiweb assigns a rack r to each Logiweb page. The hooks of the rack are various cardinals that represent various concepts. As an example, Logiweb hangs the codex c of a page on the 'codex' hook, meaning that a particular cardinal c' represents the concept of a codex and meaning that $r[c'] = c$. The hooks of a Logiweb rack include the following:

[2] the ratio by which formalization increases the length of a text [dB80]

vector The list of bytes that makes up the Logiweb page when it is stored on disk or transmitted over a network.

bibliography The list of pages directly referenced by the present page. Reference number zero (the first reference) is the reference of the page itself.

cache Explained later.

dictionary Symbol declarations of the page, represented as an array d. A symbol with identifier i 'exists' if $d[i] \neq \mathsf{T}$ in which case $d[i]$ is the arity of the symbol.

body The term that makes up the page.

codex The codex of the page as explained previously.

expansion The macro expanded version of the body.

diagnose Logiweb hangs T on this hook if the page passes verification. Otherwise, the diagnose will be a term which, when typeset, is supposed be be a meaningful error message.

code A compiled version of the codex (for an example of use see the base page, Section 4.4.1)

4.5 Logiweb caches

As explained previously, there is a one-to-one correspondence between Logiweb references and immutable Logiweb pages. The correctness of a Logiweb page only depends on the contents of the page, which is immutable, and the contents of transitively referenced pages, which are also immutable. For that reason, each Logiweb page only needs to be verified once. The current implementation of the compiler verifies each page the first time it is referenced within a session.

Independently of any caches maintained by Logiweb software for efficiency reasons, Logiweb also defines a "Logiweb cache" for each Logiweb page. The cache of a page collects information about a page and all its transitively referenced pages.

The cache of a Logiweb page is an array c for which $c[r]$ is the rack of the page with Logiweb reference r. The domain of the cache c comprises the references of the page itself and pages transitively referenced by the page. As a dirty trick, $c[0]$ contains the reference of the page itself so that e.g. $c[c[0]]$ is the rack of the page.

Racks and caches are defined mutually recursively. A cache is an array that maps references to racks. A rack maps the previously mentioned 'cache' hook to an array which maps references of transitively referenced pages to their caches. The resulting structure is a non-cyclic one with considerable sharing which gives efficient access to all data needed during verification and during all other activities undertaken by Logiweb.

5 Verification

5.1 Page symbols

Each Logiweb page implicitly declares a symbol whose identifier is zero, and the arity of that symbol is forced to be zero. We shall refer to that symbol as the 'page symbol' of the page. The reference of a page symbol equals the reference of the page so there is a one-to-one correspondence between pages and page symbols, c.f. Section 4.1.

Each Logiweb definition assigns an aspect to a symbol. Aspects assigned to a page symbol, however, should be interpreted as aspects of the page. As an example, the name of a page symbol effectively becomes the name of the page.

5.2 Verification

From the point of view of the Logiweb core software, verification of a Logiweb page is trivial. The core software just codifies the page, looks up the 'claim' aspect of the page symbol (which, if defined, is a term), applies that term to the cache of the page, and considers the page correct if the result is T. Otherwise, Logiweb hangs the result of the computation on the 'diagnose' hook of the page. The diagnose is supposed to be a term which, when typeset, is supposed to explain what went wrong. Supplying meaningful diagnoses is the responsibility of programmers of claims, proof tactics, etc.

If a page makes no claim (i.e. if no claim aspect is defined for its page symbol) then Logiweb uses the claim of reference number one of the page, and if that reference makes no claim either, then the page is considered trivially correct. In the "hello world" example in Section 2.6, the "hello world" page makes no claim but reference number one (the base page) does make a claim.

The claim made by the base page is a substantial one. It scans the codex of the page for all proof definitions, invokes proof compilers which in turn invoke proof tactics, verifies the proofs, checks for cyclic references between proofs, and checks that the proofs prove the statement aspects they are supposed to prove.

It is an important feature of Logiweb that a complex beast like a proof checker is not included in the core software. Firstly, it reduces the complexity of the core software. Secondly, at gives the users of the system the flexibility to use the proof checker that comes with logiweb (simply by referencing the 'base' page as reference number one) or to define another one.

To establish confidence in the formal correctness of a Logiweb page, a human reader can check that it has been verified by a proof checker that the reader trusts. That can be done by inspecting the claim aspect of the relevant page symbol.

Proof checkers are faced with the same problem as the human reader. The proof checker that comes with Logiweb is 'arrogant' in the sense that it only trusts lemmas that it has checked itself. When a proof being checked references a lemma on another page, the proof checker looks up the claim of the other page, which is supposed to be a conjunction, and then checks that the proof checker itself is a member of that conjunction. The proof checker also checks that the diagnose of referenced pages equals T. Hence, the proof checker that comes with Logiweb only trusts itself but is willing to coexist with other checkers. On the base page, the proof checker coexists with a test case verifier, c.f. Section 2.6.

One can easily adapt any TEX source to the format of Logiweb and get it accepted as a trivially correct page, simply because TEX sources make no claims that Logiweb can understand. That is useful for communication to readers but of course such trivially correct pages are of no formal use.

Macro expansion is just as simple as verification from the point of view of the Logiweb core software. Logiweb pages are macro expanded by applying the macro aspect of the page symbol of a page to the body of the page and hanging the result on the 'expansion'

hook of the page, c.f. Section 2.6..

6 Status

Logiweb allows users to publish, verify, retrieve, and read pages that contain formal mathematics.

The Logiweb core software comprises about 900 kilobyte of source code (including comments). It implements the features described in the present paper. It also implements many other features like features for rendering. The core software is kept simple by moving essential features like the definition of the proof checker from the core software to Logiweb pages.

In particular, the measures taken to allow λ-calculus programs to be efficient have been implemented with success. Also, the data structures of codices, racks, and caches have proven to support proof checking well.

At the time of writing, Logiweb allows referencing within a single site covered by a single Logiweb server. The Logiweb protocol allows cooperation among Logiweb servers. When that is implemented, the web-part of the system will allow a single formal development to consist of papers that reside different places in the world. Until then, users are forced to copy referenced papers to their own site. Section 2.3 describes copying as a convenient possibility, but until further copying is a necessity.

At the time of writing, 600 kilobyte of Logiweb source text has been verified by Logiweb. Those 600 kilobyte define the computing machinery, the macro expansion facility, and the proof checker, and verify the feasibility of the system (c.f. Section 3.7). 800 kilobyte of formal proofs [Gru02] await verification.

References

[BG97] C. Berline and K. Grue. A κ-denotational semantics for Map Theory in ZFC+SI. *Theoretical Computer Science*, 179(1–2):137–202, June 1997.

[CAB+86] Robert L. Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[dB80] Nicolaas Govert de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996. `http://citeseer.nj.nec.com/dobbertin96ripemd.html`.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 12(XXXVIII):173–198, 1931.

[Gru92] K. Grue. Map theory. *Theoretical Computer Science*, 102(1):1–133, July 1992.

[Gru01] K. Grue. *Mathematics and Computation*, volume 1–3. DIKU (lecture notes), 7. edition, 2001. `http://www.diku.dk/~grue/papers/mac0102/`.

[Gru02] K. Grue. Map theory with classical maps. Technical Report 02/21, DIKU, 2002. `http://www.diku.dk/publikationer/tekniske.rapporter/2002/`.

[Gru04] Klaus Grue. Logiweb. In Fairouz Kamareddine, editor, *Mathematical Knowledge Management Symposium 2003*, volume 93 of *Electronic Notes in Theoretical Computer Science*, pages 70–101. Elsevier, Feb 2004.

[Knu83] D. Knuth. *The TeXbook*. Addison Wesley, 1983.

[Koh03] Michael Kohlhase. OMDoc: An open markup format for mathematical documents (version 1.1), 2003. `http://www.mathweb.org/omdoc.ps`.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[Men87] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks, 3. edition, 1987.

[MS01] Robert Miner and Jeff Schaeffer. A gentle introduction to MathML, 2001. `http://www.dessci.com/en/support/tutorials/mathml/default.htm`.

[Muz93] Michał Muzalewski. *An Outline of PC Mizar*. Foundation of Logic, Mathematics and Informatics, Mizar User Group, Brussels, 1993.

[Pau98a] Lawrence C. Paulson. Introduction to Isabelle. Technical report, University of Cambridge, Computer Laboratory, 1998.

[Pau98b] Lawrence C. Paulson. The Isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory, 1998.

[Ska02] Sebastian C. Skalberg. *An Interactive Proof System for Map Theory*. PhD thesis, University of Copenhagen, October 2002. `http://www.mangust.dk/skalberg/phd/`.

[TB85] Andrzej Trybulec and Howard Blair. Computer assisted reasoning with MIZAR. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, Los Angeles, CA, August 1985. Morgan Kaufmann. `http://www.mizar.org/`.

[Val03] Thierry Vallée. *"Map Theory" et Antifondation*, volume 79 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

Things to know when implementing KBO

Bernd Löchner
Technische Universität Kaiserslautern
loechner@informatik.uni-kl.de

Abstract

The Knuth-Bendix Ordering (KBO) is one of the term orderings in widespread use. We present a new algorithm to compute KBO, which is (to our knowledge) the first asymptotically optimal one. Starting with an "obviously correct" version we use the method of program transformation to stepwise develop an efficient version, making clear the essential ideas, while retaining correctness. By theoretical analysis we show that the worst-case behavior is thereby changed from quadratic to linear. Measurements show the practical improvements of the different variants.

1 Introduction

For the development of a practically successful prover it is important to have efficient implementations of the most time-consuming subtasks. Recently the efficient implementation of term orderings has received attention [Löc04, RV04]. Having decent implementations of indexing techniques available, the time that is spent on determining ordering relations can amount to a significant part of the overall running time. Schulz gives an estimation of up to 35 % for the prover E [Sch02] (personal communication). In [RV04], Riazanov and Voronkov give a figure of about 40 % on the average for the prover Vampire [RV02] and their straightforward implementation, with up to 80 % for the hardest problems. For Waldmeister [LH02] we observe a more modest figure of typically 5–10 % as rewriting with unorientable equations is very restricted in the default settings [BH96]. Even so, for some proof tasks nearly 50 % are needed by the orderings.

The aim of this work is the development of an efficient version of the *Knuth-Bendix Ordering* (KBO) [KB70], one of the orderings in widespread use. It is widely believed that an implementation of KBO is asymptotically optimal if it shows quadratic worst-case behavior. In the following, however, we will show the derivation of a variant that needs only linear time. Similar to previous work, where we investigated the efficient implementation of the Lexicographic Path Ordering [Löc04], our approach is based on *program transformations* [BD77, PP93]: Using a language that is close to functional programming or algebraic specification, we start with some "obviously correct" implementation, which is as close as possible to the original definition of the ordering. Then we refine the implementation in several small steps that preserve correctness. To keep the presentation concise (and interesting to read) we omit intermediate steps that occurred in our paper-and-pencil use of a program transformation calculus. Instead, for each function we present its initial specification and its final version. The advantage of

this detailed approach is that the central ideas and their influence on the running time become very clear.

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. All experiments were performed on machines equipped with 1 GHz Pentium III processors and 1 GByte RAM.

2 The definition of the Knuth-Bendix Ordering

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

The KBO is named after its first use by Knuth and Bendix in [KB70]. Since then its definition has been slightly generalized by relaxing the variable condition. The KBO is parameterized by a *precedence* $>_\mathcal{F}$, that is a partial ordering on $\mathcal{F}$, and a *weight function* $\varphi : \mathcal{F} \cup \mathcal{V} \to \mathbb{N}$. A weight function φ is *admissible* to a precedence $>_\mathcal{F}$ if there is some $\mu > 0$ such that $\varphi(x) = \mu$ for all $x \in \mathcal{V}$, $\varphi(c) \geq \mu$ for all constants $c \in \mathcal{F}$, and if $f \in \mathcal{F}$ is a unary function symbol with $\varphi(f) = 0$ then $f >_\mathcal{F} g$ for all $g \in \mathcal{F} - \{f\}$. The weight function φ is extended to terms by $\varphi(f(t_1, \ldots, t_n)) = \varphi(f) + \varphi(t_1) + \cdots + \varphi(t_n)$.

DEFINITION 1 *Let $>_\mathcal{F}$ be a precedence on $\mathcal{F}$, φ a weight function admissible to $>_\mathcal{F}$, and $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. Then $s \succ_{\mathrm{kbo}} t$ iff*

- $s \equiv f(s_1, \ldots, s_n)$, $t \equiv g(t_1, \ldots, t_m)$, *and*

  (1) $|s|_x \geq |t|_x$ *for all $x \in \mathcal{V}$ and*

  (2a) $\varphi(s) > \varphi(t)$ *or*

  (2b) $\varphi(s) = \varphi(t)$, $f >_\mathcal{F} g$ *or*

  (2c) $\varphi(s) = \varphi(t)$, $f = g$, *and there is some k with*
  $s_1 \equiv t_1$, $\ldots$, $s_{k-1} \equiv t_{k-1}$, $s_k \succ_{\mathrm{kbo}} t_k$,

- *or* $s \equiv f(s_1, \ldots, s_n)$, $t \equiv x \in \mathcal{V}$, *and* $x \in \mathrm{Var}(s)$.

The main idea of the KBO is that larger terms are greater than smaller terms; weight function φ allows to differentiate between the different symbols (case (2a)). This weight-based ordering is then refined by the precedence (case (2b)) and a recursive comparison (case (2c)). The purpose of the variable condition (1) is to make $\succ_{\mathrm{kbo}}$ stable against substitutions (i. e., $s \succ_{\mathrm{kbo}} t$ implies $\sigma(s) \succ_{\mathrm{kbo}} \sigma(t)$ for any substitution σ).

THEOREM 1 *The Knuth-Bendix ordering $\succ_{\mathrm{kbo}}$ to precedence $>_\mathcal{F}$ and weight function φ admissible to $>_\mathcal{F}$ is a reduction ordering. If $>_\mathcal{F}$ is total on $\mathcal{F}$ then $\succ_{\mathrm{kbo}}$ is total on ground terms.* □

3 Implementing the Knuth-Bendix Ordering

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 : Bool. For natural numbers it has a sort Nat. Sorts Vid and Fid represent variables $\mathcal{V}$ and function symbols $\mathcal{F}$. We define the data type Term via two constructors using an additional sort Termlist for lists of terms.

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{V} & : & \mathsf{Vid} \to \mathsf{Term} & [\,] & : & \mathsf{Termlist} \\ \mathsf{F} & : & \mathsf{Fid\ Termlist} \to \mathsf{Term} & \centerdot & : & \mathsf{Term\ Termlist} \to \mathsf{Termlist} \end{array}$$

Note that for a term $\mathsf{F}(f, ts)$ there is no relationship encoded in the data type between the arity of f and the length of ts. It is therefore possible to construct Terms that are not well-formed, that is, they do not represent elements of $\mathrm{Term}(\mathcal{F}, \mathcal{V})$.

3.1 The reference implementation

By Definition 1, there are three sub-problems to solve for implementing KBO: the variable condition (1), the computation of the weight $\varphi(s)$, and the lexicographic comparison. Of these, the variable condition is the most complex, whereas implementing $\varphi(s)$ is straightforward:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{phi} & : & \mathsf{Term} \to \mathsf{Nat} & \mathsf{phi_{tl}} & : & \mathsf{Termlist} \to \mathsf{Nat} \\ \mathsf{phi}(\mathsf{V}(x)) & = & \mu & \mathsf{phi_{tl}}([\,]) & = & 0 \\ \mathsf{phi}(\mathsf{F}(f, ss)) & = & \varphi(f) + \mathsf{phi_{tl}}(ss) & \mathsf{phi_{tl}}(s \centerdot ss) & = & \mathsf{phi}(s) + \mathsf{phi_{tl}}(ss) \end{array}$$

The variable condition (i. e., condition (1) of Def. 1) is not effective in that the set $\mathcal{V}$ is infinite by definition. Therefore, several authors restrict the test to $\mathrm{Var}(s, t)$. As this complicates later optimizations, we use a different approach. Whereas $\mathcal{V}$ is infinite, for any invocation of the ordering there exists only a finite subset $\mathcal{V}_{\mathrm{fin}}$ in the state of the prover. Let $K = |\mathcal{V}_{\mathrm{fin}}|$. We assume that the module implementing the KBO can access K and that there is a function $\mathsf{index} : \mathsf{Vid} \to \mathsf{Nat}$ that transforms efficiently (i. e., in constant time) a variable identifier into a natural number between 1 and K. Then the following function xlen computes the x-length of a term: $\mathsf{xlen}(s, i) = |s|_x$ iff $\mathsf{index}(x) = i$.

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{xlen} & : & \mathsf{Term\ Nat} \to \mathsf{Nat} & \mathsf{xlen_{tl}} & : & \mathsf{Termlist\ Nat} \to \mathsf{Nat} \\ \mathsf{xlen}(\mathsf{V}(x), i) & = & \textbf{if } \mathsf{index}(x) = i \textbf{ then } 1 \textbf{ else } 0 & \mathsf{xlen_{tl}}([\,], i) & = & 0 \\ \mathsf{xlen}(\mathsf{F}(f, ss), i) & = & \mathsf{xlen_{tl}}(ss, i) & \mathsf{xlen_{tl}}(s \centerdot ss, i) & = & \mathsf{xlen}(s, i) + \mathsf{xlen_{tl}}(ss, i) \end{array}$$

We use xlen in function $\mathsf{varCheck}_1$ which checks the variable condition:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{varCheck}_1 & : & \mathsf{Term\ Term} \to \mathsf{Bool} & \mathsf{varCheck}' & : & \mathsf{Term\ Term\ Nat} \to \mathsf{Bool} \\ \mathsf{varCheck}_1(s, t) & = & \mathsf{varCheck}'(s, t, 1) & \mathsf{varCheck}'(s, t, i) & = & \textbf{if } \ i > K \qquad\qquad\qquad \textbf{then } \mathsf{true} \\ & & & & & \textbf{elif } \ \mathsf{xlen}(s, i) \not\geqslant \mathsf{xlen}(t, i) \textbf{ then } \mathsf{false} \\ & & & & & \textbf{else } \mathsf{varCheck}'(s, t, i + 1) \end{array}$$

As $\mathsf{xlen}(s, i)$ needs $O(|s|)$ steps, $\mathsf{varCheck}_1(s, t)$ needs $O(K \cdot (|s| + |t|))$ steps.

With the auxiliary functions available the implementation of KBO is straightforward. We call it the *reference implementation.*[1] Functions $>_{\mathsf{F}}$, $=_{\mathsf{t}}$, and $=_{\mathsf{tl}}$ compute the

[1] The subscript 1 denotes that this is the first version we consider. It will increase subsequently.

precedence, syntactic equality on Term, and syntactic equality on Termlist. Function contains tests whether a variable symbol occurs in a term.

$$
\begin{array}{rcl}
\mathsf{kbo}_1 & : & \mathsf{Term}\,\mathsf{Term} \to \mathsf{Bool} \\
\mathsf{kbo}_1(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) & = & \mathsf{varCheck}_1(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) \wedge \\
 & & (\mathsf{phi}(\mathsf{F}(f,ss)) > \mathsf{phi}(\mathsf{F}(g,ts)) \ \vee \\
 & & \ \mathsf{phi}(\mathsf{F}(f,ss)) = \mathsf{phi}(\mathsf{F}(g,ts)) \wedge f >_{\mathsf{F}} g \ \vee \\
 & & \ \mathsf{phi}(\mathsf{F}(f,ss)) = \mathsf{phi}(\mathsf{F}(g,ts)) \wedge f = g \wedge \mathsf{kbolex}_1(ss,ts)) \\
\mathsf{kbo}_1(\mathsf{F}(f,ss),\mathsf{V}(y)) & = & \mathsf{contains}(\mathsf{F}(f,ss),y) \\
\mathsf{kbo}_1(\mathsf{V}(x),t) & = & \mathsf{false} \\
 & & \\
\mathsf{kbolex}_1 & : & \mathsf{Termlist}\,\mathsf{Termlist} \to \mathsf{Bool} \\
\mathsf{kbolex}_1([\,],[\,]) & = & \mathsf{false} \\
\mathsf{kbolex}_1(s\,.\,ss,t\,.\,ts) & = & \textbf{if } s =_{\mathsf{t}} t \textbf{ then } \mathsf{kbolex}_1(ss,ts) \textbf{ else } \mathsf{kbo}_1(s,t)
\end{array}
$$

Note that kbolex_1 is a partial function, it is only defined if the arguments have the same length. This reflects that $\mathcal{F}$ contains function symbols of fixed arity only. In a theorem prover it is useful to have also a bidirectional version $\mathsf{ckbo}(s,t)$ available. Its result sort Res has four constructors $\mathsf{E},\mathsf{G},\mathsf{L},\mathsf{N}$ which encode whether $s \equiv t$, $s \succ_{\mathrm{kbo}} t$, $t \succ_{\mathrm{kbo}} s$, or whether s and t are not comparable.

$$
\begin{array}{rcl}
\mathsf{ckbo}_1 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Res} \\
\mathsf{ckbo}_1(s,t) & = & \textbf{if} \quad s =_{\mathsf{t}} t \quad \textbf{then } \mathsf{E} \\
 & & \textbf{elif } \mathsf{kbo}_1(s,t) \textbf{ then } \mathsf{G} \\
 & & \textbf{elif } \mathsf{kbo}_1(t,s) \textbf{ then } \mathsf{L} \\
 & & \textbf{else } \mathsf{N}
\end{array}
$$

When we analyze the worst-case running time of kbo_1, we see that it is quadratic in the size of the arguments:

THEOREM 2 *Let s and t be well-formed* Terms*. Then $\mathsf{kbo}_1(s,t) = \mathsf{true}$ iff $s \succ_{\mathrm{kbo}} t$. The worst-case running time of $\mathsf{kbo}_1(s,t)$ is $O(KN^2)$ where $K = |\mathcal{V}_{\mathrm{fin}}|$ and $N = |s| + |t|$.*

PROOF The correctness of the implementation is easy to see as it closely follows Definition 1. It remains to consider the worst-case running time. If s is a variable then kbo_1 performs one step. If t is a variable then the function performs $O(|s|)$ steps. In the remaining case, performing the variable check needs $O(KN)$ steps and determining the weights of the terms needs $O(N)$ steps. Furthermore, after testing some subterms for syntactic equality, which is bound linearly in N, the evaluation of kbolex_1 may perform one recursive call to kbo_1. In the worst case, the top-symbol of both terms is a unary function symbol. Hence, for the recursive call the size of each argument is only reduced by one. Therefore, we can estimate the costs of $\mathsf{kbo}_1(s,t)$ by the recurrence $C(N) = c_1KN + c_2N + c_3 + C(N-2)$, where c_1, c_2 and c_3 are some constants. Solving this recurrence leads to $C(N) \leq c'_1KN^2 + c'_2N^2 + c'_3N + c'_4$ for some suitable constants $c'_1, \ldots, c'_4$. This implies that the worst-case running time of $\mathsf{kbo}_1(s,t)$ is $O(KN^2)$. □

By this analysis, it is easy to come up with instances that show the quadratic behavior of kbo_1. As the number of recursive calls is bound by the depth of the terms, the worst case occurs for terms consisting mostly of unary function symbols. We use KBO_1 to refer to the C-implementation of kbo_1.

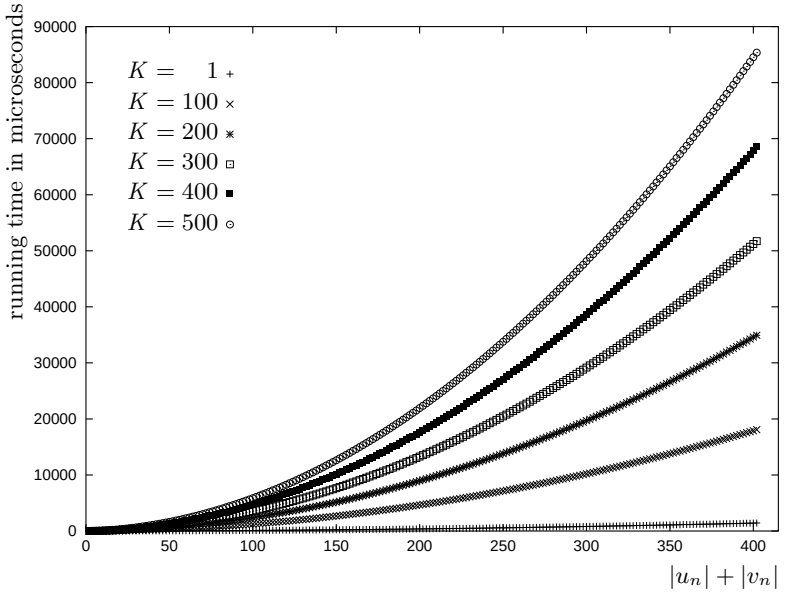


Figure 1: Time needed to evaluate $\text{KBO}_1(u_n, v_n)$ with $n = 1, \ldots, 200$ and six different values of K (cf. Example 1)

EXAMPLE 1 *Let $f >_\mathcal{F} a >_\mathcal{F} b$ and φ return 1 for each symbol. Let $u_n \equiv f^n(a)$ and $v_n \equiv f^n(b)$. Then $\mathsf{kbo}_1(u_n, v_n)$ needs time that grows linearly in K and quadratically in n to determine $u_n \succ_{\text{kbo}} v_n$. Figure 1 depicts the running times of $\text{KBO}_1(u_n, v_n)$ for $n = 1, \ldots, 200$ and six different values of K.*

Of course, the definition of $\mathsf{kbo}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts))$ can easily be optimized by extracting common subexpressions into let-variables and re-bracketing the main expression:

$$
\begin{aligned}
&\mathsf{kbo}(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \;=\; \\
&\quad \textbf{let}\;\; vc = \mathsf{varCheck}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \\
&\qquad\;\; w_s = \mathsf{phi}(\mathsf{F}(f, ss)) \\
&\qquad\;\; w_t = \mathsf{phi}(\mathsf{F}(g, ts)) \\
&\quad \textbf{in}\;\; vc \wedge (w_s > w_t \vee (w_s = w_t \wedge (f >_\mathsf{F} g \vee (f = g \wedge \mathsf{kbolex}(ss, ts)))))
\end{aligned}
$$

However, this modification only improves the constants, the asymptotic running time is still $O(KN^2)$ in the worst case. The main problems are the expensive check of the variable condition and the repeated work induced by recursive calls.

Standard techniques to avoid repeated computations such as dynamic programming or memoization use an additional data structure to store intermediate results. This is unproblematic for the weights of the subterms. However, to implement the variable condition, we have to store the multisets of variables occurring in a subterm. The following example shows that this may lead to space problems.

EXAMPLE 2 *Let $g >_\mathcal{F} a >_\mathcal{F} b$ and φ return 1 for each symbol. The terms u_n are given by $u_0 \equiv a$ and $u_{n+1} \equiv g(u_n, x_n, y_n)$. The terms v_n are given by $v_0 \equiv b$ and $v_{n+1} \equiv g(v_n, y_n, x_n)$. Then $u_n \succ_{\text{kbo}} v_n$. However, the multisets of variables for u_n and v_n contain $2n$ different variables. This means that simply storing the multisets of variables for each term leads to a quadratic memory requirement.*

It is easy to develop representations of the multisets that circumvent the quadratic memory requirement for this example. To find general solutions, however, is quite challenging. As it turns out, we can derive an efficient version of KBO without the use of a memo-table. First, we want to improve the check of the variable condition.

3.2 Implementing the variable condition with arrays

The main problem with function $\mathsf{varCheck}_1$ is that it needs K traversals of each term. An obvious way to avoid this is to compute the multisets of variables contained in s and t and to test whether the multiset belonging to t is a sub-multiset of the multiset belonging to s. This needs only one traversal per term. However, the use of multisets complicates later transformations. Hence, we use a different approach which is based on arrays and which is also closer to an imperative implementation. The arrays contain one entry for each variable which we can efficiently access by using function index.

In the imperative implementation arrays are modified destructively. To model this, we thread arrays as additional parameters through the function calls and require that a variable bound to an array must be used exactly once in each branch of the function definition.[2] This facilitates a linearity analysis that some compilers of functional languages use as an optimization. Instead of being copied, linearly used data structures can be modified destructively.

It is therefore convenient to reduce the number of arrays. Instead of two arrays of naturals for the number of variables in s and the number of variables in t we can use one single array of integers which stores for each variable x the *variable balance* $|s|_x - |t|_x$. The test of condition (1) of Definition 1 is then replaced by $|s|_x - |t|_x \geq 0$ for all $x \in \mathcal{V}_{\mathrm{fin}}$.

To enhance modularity and ease later optimizations we use a dedicated data type VarBal for variable balances with corresponding methods. We use $\mathbf{1}$ to denote the type of the empty tuple with the single value $\langle\rangle$, which plays the same role as `void` in the C language. The following two functions allocate and deallocate an instance of VarBal. We assume that $\mathsf{newArray}(\mathsf{Int}, 1, K)$ returns a new array of integers with indices ranging from 1 to K such that all entries are initialized to 0. We write $vb = \vec{0}$ to denote that all entries of vb have value 0. Function $\mathsf{freeArray}$ frees in some unspecified way the memory occupied by the array passed as parameter.

$$\begin{array}{rcl} \mathsf{newVB} & : & \mathbf{1} \to \mathsf{VarBal} \\ \mathsf{newVB}(\langle\rangle) & = & \mathsf{newArray}(\mathsf{Int}, 1, K) \end{array} \qquad \begin{array}{rcl} \mathsf{freeVB} & : & \mathsf{VarBal} \to \mathbf{1} \\ \mathsf{freeVB}(vb) & = & \mathsf{freeArray}(vb) \end{array}$$

A conservative estimation is that both functions run in $O(K)$ time. We will discuss later about how to improve on this (see Section 3.5).

To increment and decrement the entry for a given variable in a VarBal we use the following two functions:

$$\begin{array}{rcl} \mathsf{inc} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\ \mathsf{inc}(vb, x) & = & \mathbf{let}\ \ i = \mathsf{index}(x) \\ & & \quad\ \langle vb', n\rangle = \mathsf{read}(vb, i) \\ & & \mathbf{in}\ \ \mathsf{update}(vb', i, n+1) \end{array} \qquad \begin{array}{rcl} \mathsf{dec} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\ \mathsf{dec}(vb, x) & = & \mathbf{let}\ \ i = \mathsf{index}(x) \\ & & \quad\ \langle vb', n\rangle = \mathsf{read}(vb, i) \\ & & \mathbf{in}\ \ \mathsf{update}(vb', i, n-1) \end{array}$$

Functions read and update are standard functions to read and modify entries of an array. We assume that they need constant time. Therefore, functions inc and dec need

[2] The cognoscenti will recognize a state monad [Wad92].

constant time as well. Note the explicit threading of the array, even read returns it. This facilitates the aforementioned linearity analysis.

The following function noNeg tests whether for all variables the balance is not negative (i. e., $|s|_x - |t|_x \geq 0$ for all $x \in \mathcal{V}_{\text{fin}}$). Function noPos tests whether for all variables the balance is not positive. This is useful for the bidirectional version of KBO.

$$
\begin{array}{rcl}
\mathsf{noNeg} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\
\mathsf{noNeg}(vb) & = & \mathsf{noNeg}'(vb, 1)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{noPos} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\
\mathsf{noPos}(vb) & = & \mathsf{noPos}'(vb, 1)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{noNeg}' & : & \mathsf{VarBal}\,\mathsf{Nat} \to \mathsf{VarBal} \times \mathsf{Bool} \\
\mathsf{noNeg}'(vb, i) & = & \\
\quad \textbf{if } i > K \textbf{ then } \langle vb, \mathsf{true} \rangle & & \\
\quad \textbf{else let } \langle vb', n \rangle = \mathsf{read}(vb, i) & & \\
\quad\quad \textbf{in if } n < 0 \textbf{ then } \langle vb', \mathsf{false} \rangle & & \\
\quad\quad\quad \textbf{else } \mathsf{noNeg}'(vb', i+1) & &
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{noPos}' & : & \mathsf{VarBal}\,\mathsf{Nat} \to \mathsf{VarBal} \times \mathsf{Bool} \\
\mathsf{noPos}'(vb, i) & = & \\
\quad \textbf{if } i > K \textbf{ then } \langle vb, \mathsf{true} \rangle & & \\
\quad \textbf{else let } \langle vb', n \rangle = \mathsf{read}(vb, i) & & \\
\quad\quad \textbf{in if } n > 0 \textbf{ then } \langle vb', \mathsf{false} \rangle & & \\
\quad\quad\quad \textbf{else } \mathsf{noPos}'(vb', i+1) & &
\end{array}
$$

It is easy to see that the worst-case running time of both functions is $O(K)$. Note that only functions newVB, freeVB, inc, dec, noNeg and noPos rely on the knowledge how the data type VarBal is represented.

Function mdfyVB modifies the variable balances in one traversal. The argument *pos* indicates whether mdfyVB has to increase or to decrease the entries in vb.

$$
\begin{array}{rcl}
\mathsf{mdfyVB} & : & \mathsf{VarBal}\,\mathsf{Term}\,\mathsf{Bool} \to \mathsf{VarBal} \\
\mathsf{mdfyVB}(vb, \mathsf{V}(x), pos) & = & \textbf{if } pos \textbf{ then } \mathsf{inc}(vb, x) \\
 & & \textbf{else } \mathsf{dec}(vb, x) \\
\mathsf{mdfyVB}(vb, \mathsf{F}(f, ss), pos) & = & \mathsf{mdfyVB}_{\mathsf{tl}}(vb, ss, pos)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{mdfyVB}_{\mathsf{tl}} & : & \mathsf{VarBal}\,\mathsf{Termlist}\,\mathsf{Bool} \to \mathsf{VarBal} \\
\mathsf{mdfyVB}_{\mathsf{tl}}(vb, [\,], pos) & = & vb \\
\mathsf{mdfyVB}_{\mathsf{tl}}(vb, s \,.\, ss, pos) & = & \textbf{let } vb' = \mathsf{mdfyVB}(vb, s, pos) \\
 & & \textbf{in } \mathsf{mdfyVB}_{\mathsf{tl}}(vb', ss, pos)
\end{array}
$$

The running time of both functions is linear in the size of the second argument.

With these functions it is possible to implement function $\mathsf{varCheck}_2$:

$$
\begin{array}{rcll}
\mathsf{varCheck}_2 & : & \mathsf{Term}\,\mathsf{Term} \to \mathsf{Bool} & \\
\mathsf{varCheck}_2(s, t) & = & \textbf{let} & vb = \mathsf{newVB}(\langle\rangle) \\
 & & & vb' = \mathsf{mdfyVB}(vb, s, \mathsf{true}) \\
 & & & vb'' = \mathsf{mdfyVB}(vb', t, \mathsf{false}) \\
 & & & \langle vb''', res \rangle = \mathsf{noNeg}(vb'') \\
 & & & \langle\rangle = \mathsf{freeVB}(vb''') \\
 & & \textbf{in} & res
\end{array}
$$

Taking into account the running times of the functions concerning VarBal it is clear that the worst-case running time of $\mathsf{varCheck}_2(s, t)$ is in $O(K + |s| + |t|)$.

Function kbo_2 (and hence ckbo_2) differs from function kbo_1 in using $\mathsf{varCheck}_2$ instead of $\mathsf{varCheck}_1$ and having the improved definition for $\mathsf{kbo}(\mathsf{F}(f, ss), \mathsf{F}(g, ts))$, thereby avoiding the multiple computations of identical weights.

THEOREM 3 *Let s and t be well-formed* Terms*. Then $\mathsf{kbo}_2(s, t) = \mathsf{kbo}_1(s, t)$. The worst-case running time of $\mathsf{kbo}_2(s, t)$ is $O(KN + N^2)$ where $K = |\mathcal{V}_{\text{fin}}|$ and $N = |s| + |t|$.*

PROOF It is easy to see that $\mathsf{varCheck}_2(s, t) = \mathsf{varCheck}_1(s, t)$ and that the modified definition for $\mathsf{kbo}_2(\mathsf{F}(f, ss), \mathsf{F}(g, ts))$ is equivalent to the original one. It remains to consider the worst-case running time. If at least one of the arguments is a variable then $\mathsf{kbo}_2(s, t)$ behaves identical to $\mathsf{kbo}_1(s, t)$: Its running time is either $O(1)$ or $O(|s|)$.

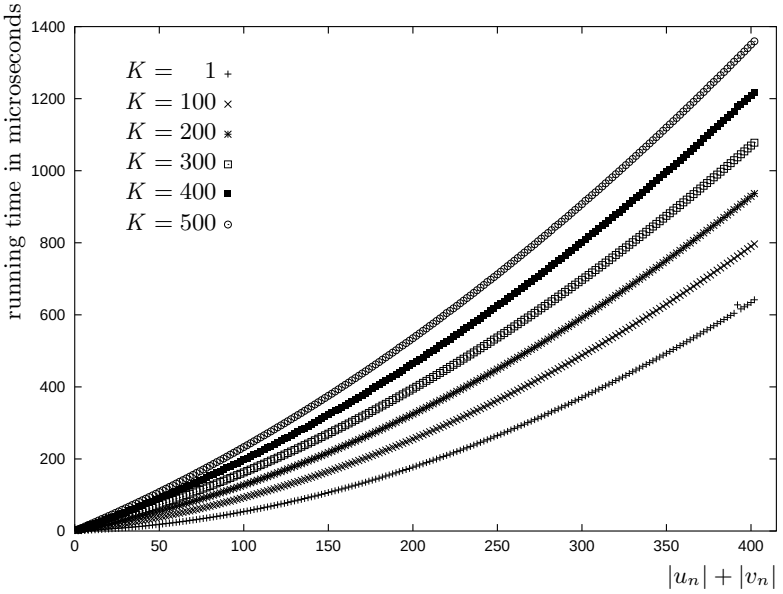


Figure 2: Time needed to evaluate $\text{KBO}_2(u_n, v_n)$ with $n = 1, \ldots, 200$ and six different values of K (cf. Example 1)

In the remaining case, performing the variable check with $\mathsf{varCheck}_2$ needs $O(K + N)$ steps. Determining the weights of the terms still is $O(N)$. (However, the constants are smaller than for kbo_1 as repeated work is avoided). For the evaluation of kbolex nothing changes. Therefore, we can estimate the costs of $\mathsf{kbo}_2(s, t)$ by the recurrence $C(N) = c_1K + c_2N + c_3 + C(N - 2)$, where c_1, c_2 and c_3 are some constants. Solving this recurrence leads to $C(N) \leq c'_1KN + c'_2N^2 + c'_3N + c'_4$ for some suitable constants c'_1, $\ldots$, c'_4. This implies that the worst-case running time of $\mathsf{kbo}_2(s, t)$ is $O(KN + N^2)$. □

Revisiting Example 1, we see in Figure 2 that KBO_2, the C-implementation of kbo_2, is significantly faster than KBO_1 for these test cases. (Figure 2 has a significantly smaller scale than Figure 1). Note the different slope of the graphs depicted in Figures 1 and 2 which indicates the different dependency on K of the two implementations.

3.3 Deriving a linear version

The *tupling strategy* is a standard approach in program transformations [PP93]. The key insight in optimizing kbo_2 is that the tupling strategy allows us not only to combine the calculation of the variable balances and the weights, which is straightforward, but also to include the lexicographic comparison. Hence, all three sub-computations are combined into one traversal of the terms. As this avoids the iterated computations of variable balances and weights it leads to an algorithm that is linear in $|s| + |t|$.

To achieve the desired result we need a variant of kbo that incorporates the tests for syntactic equality. As it turns out, developing a version for the bidirectional ckbo is only slightly more involved than developing a version for a three-valued unidirectional variant. For the presentation, we therefore prefer the bidirectional variant for its greater symmetry. The unidirectional variant can easily be derived by simplifying the bidirectional one.

Because of the variable balances, we have to thread some state through the computation. By the tupling strategy the state is extended to incorporate the weights of the terms and the result of the lexicographic comparison. To reduce the number of components of the state, we therefore combine both weights into one *weight balance* wb which is defined as $wb = \varphi(s) - \varphi(t)$. Analogously to the variable balances, the test $\varphi(s) \geq \varphi(t)$ is then replaced by $wb \geq 0$. To get used to the tupling approach we first define some auxiliary functions. Function mdfyVWB combines the modification of the variable balances with the modification of the weight balance. Its initial definition is the following:

$$\begin{array}{rcl} \mathsf{mdfyVWB}(vb, wb, s, pos) & = & \textbf{let } vb' = \mathsf{mdfyVB}(vb, s, pos) \\ & & \quad\; w = \mathsf{phi}(s) \\ & & \textbf{in } \textbf{if } pos \textbf{ then } \langle vb', wb + w\rangle \textbf{ else } \langle vb', wb - w\rangle \end{array}$$

With the tupling strategy we can avoid the multiple traversals of the term:

$$\begin{array}{rcl} \mathsf{mdfyVWB} & : & \mathsf{VarBal\ Int\ Term\ Bool} \to \mathsf{VarBal} \times \mathsf{Int} \\ \mathsf{mdfyVWB}(vb, wb, \mathsf{V}(x), pos) & = & \textbf{if } pos \textbf{ then } \langle \mathsf{inc}(vb, x), wb + \mu\rangle \\ & & \textbf{else } \langle \mathsf{dec}(vb, x), wb - \mu\rangle \\ \mathsf{mdfyVWB}(vb, wb, \mathsf{F}(f, ss), pos) & = & \textbf{let } \langle vb', wb'\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb, wb, ss, pos) \\ & & \textbf{in } \textbf{if } pos \textbf{ then } \langle vb', wb' + \varphi(f)\rangle \\ & & \quad \textbf{else } \langle vb', wb' - \varphi(f)\rangle \\ \mathsf{mdfyVWB}_{\mathsf{tl}} & : & \mathsf{VarBal\ Int\ Termlist\ Bool} \to \mathsf{VarBal} \times \mathsf{Int} \\ \mathsf{mdfyVWB}_{\mathsf{tl}}(vb, wb, [\,], pos) & = & \langle vb, wb\rangle \\ \mathsf{mdfyVWB}_{\mathsf{tl}}(vb, wb, s \,.\, ss, pos) & = & \textbf{let } \langle vb', wb'\rangle = \mathsf{mdfyVWB}(vb, wb, s, pos) \\ & & \textbf{in } \mathsf{mdfyVWB}_{\mathsf{tl}}(vb', wb', ss, pos) \end{array}$$

Obviously, evaluating $\mathsf{mdfyVWB}(vb, wb, s, pos)$ needs $O(|s|)$ steps.

Function mdfyVWBc integrates in addition the test whether some variable symbol y is contained in the term s:

$$\begin{array}{rcl} \mathsf{mdfyVWBc}(vb, wb, s, y, pos) & = & \textbf{let } \langle vb', wb'\rangle = \mathsf{mdfyVWB}(vb, wb, s, pos) \\ & & \quad\; res = \mathsf{contains}(s, y) \\ & & \textbf{in } \langle vb', wb', res\rangle \end{array}$$

With tupling we get the following variant:

$$\begin{array}{rcl} \mathsf{mdfyVWBc} & : & \mathsf{VarBal\ Int\ Term\ Vid\ Bool} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Bool} \\ \mathsf{mdfyVWBc}(vb, wb, \mathsf{V}(x), y, pos) & = & \textbf{if } pos \textbf{ then } \langle \mathsf{inc}(vb, x), wb + \mu, x = y\rangle \\ & & \textbf{else } \langle \mathsf{dec}(vb, x), wb - \mu, x = y\rangle \\ \mathsf{mdfyVWBc}(vb, wb, \mathsf{F}(f, ss), y, pos) & = & \textbf{let } \langle vb', wb', res\rangle = \mathsf{mdfyVWBc}_{\mathsf{tl}}(vb, wb, ss, y, pos) \\ & & \textbf{in } \textbf{if } pos \textbf{ then } \langle vb', wb' + \varphi(f), res\rangle \\ & & \quad \textbf{else } \langle vb', wb' - \varphi(f), res\rangle \\ \mathsf{mdfyVWBc}_{\mathsf{tl}} & : & \mathsf{VarBal\ Int\ Termlist\ Vid\ Bool} \to \mathsf{VarBal} \times \mathsf{Int} \\ \mathsf{mdfyVWBc}_{\mathsf{tl}}(vb, wb, [\,], y, pos) & = & \langle vb, wb, \mathsf{false}\rangle \\ \mathsf{mdfyVWBc}_{\mathsf{tl}}(vb, wb, s \,.\, ss, y, pos) & = & \textbf{let } \langle vb', wb', res\rangle = \mathsf{mdfyVWBc}(vb, wb, s, y, pos) \\ & & \textbf{in } \textbf{if } \neg res \textbf{ then } \mathsf{mdfyVWBc}_{\mathsf{tl}}(vb', wb', ss, y, pos) \\ & & \quad \textbf{else let } \langle vb'', wb''\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb', wb', ss, pos) \\ & & \qquad \textbf{in } \langle vb'', wb'', \mathsf{true}\rangle \end{array}$$

As each symbol in the term is considered once at constant costs, the evaluation of $\mathsf{mdfyVWBc}(vb, wb, s, y, pos)$ needs $O(|s|)$ steps.

The tupled variant of ckbo modifies the given variable and weight balances and computes ckbo of the terms:

$$\begin{array}{rcl}\mathsf{tckbo}(vb, wb, s, t) & = & \textbf{let} \;\; \langle vb', wb'\rangle = \mathsf{mdfyVWB}(vb, wb, s, \mathsf{true}) \\ & & \quad\;\; \langle vb'', wb''\rangle = \mathsf{mdfyVWB}(vb', wb', t, \mathsf{false}) \\ & & \quad\;\; res = \mathsf{ckbo}(s, t) \\ & & \textbf{in} \;\; \langle vb'', wb'', res\rangle\end{array}$$

Note that if $s =_{\mathsf{t}} t$ then $vb'' = vb$ and $wb'' = wb$ (i. e., the possible modifications of each variable balance and the weight balance cancel each other).

Similar to tckbo, we need a tupled variant of the lexicographic comparison. We first specify a Res-valued variant of the lexicographic comparison. Like function kbolex it is only defined for Termlists of the same length.

$$\begin{array}{rcl}\mathsf{ckbolex}(ss, ts) & = & \textbf{if} \;\; ss =_{\mathsf{tl}} ts \;\; \textbf{then} \; \mathsf{E} \\ & & \textbf{elif} \;\; \mathsf{kbolex}(ss, ts) \;\; \textbf{then} \; \mathsf{G} \\ & & \textbf{elif} \;\; \mathsf{kbolex}(ts, ss) \;\; \textbf{then} \; \mathsf{L} \\ & & \textbf{else} \; \mathsf{N}\end{array}$$

It is easy to see that the following variant is equivalent:

$$\begin{array}{rcl}\mathsf{ckbolex}([\,], [\,]) & = & \mathsf{E} \\ \mathsf{ckbolex}(s \,.\, ss, t \,.\, ts) & = & \textbf{let} \; res = \mathsf{ckbo}(s, t) \\ & & \textbf{in} \;\; \textbf{if} \; res = \mathsf{E} \; \textbf{then} \; \mathsf{ckbolex}(ss, ts) \\ & & \quad\;\; \textbf{else} \; res\end{array}$$

The tupled variant tckbolex takes into account the variable and weight balances. It must be called only with Termlists of the same length as otherwise it is undefined.

$$\begin{array}{rcl}\mathsf{tckbolex}(vb, wb, ss, ts) & = & \textbf{let} \;\; \langle vb', wb'\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb, wb, ss, \mathsf{true}) \\ & & \quad\;\; \langle vb'', wb''\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb', wb', ts, \mathsf{false}) \\ & & \quad\;\; res = \mathsf{ckbolex}(ss, ts) \\ & & \textbf{in} \;\; \langle vb'', wb'', res\rangle\end{array}$$

Note that $\mathsf{tckbolex}(vb, wb, s \,.\, ss, t \,.\, ts)$ returns the same result as $\mathsf{tckbolex}(vb, wb, ss, ts)$ if $s =_{\mathsf{t}} t$.

In the specification of tckbo and tckbolex the calculation of the ordering relation and the modification of the variable and weight balances occur independently. However, the whole point of the optimizations is to perform them simultaneously and to use the variable and weight balances for computing the ordering relations. To get correct results we therefore have to establish the following invariant I: At each invocation of tckbo and tckbolex both functions are called with $vb = \vec{0}$ and $wb = 0$. Thus, it is sufficient that the optimized versions of tckbo and tckbolex give the same results as the unoptimized versions only if called with $vb = \vec{0}$ and $wb = 0$.

The optimized version of tckbolex is called $\mathsf{tckbolex}_3$. Note the close relationship to the second variant of ckbolex.

$$\begin{array}{rcl}\mathsf{tckbolex}_3 & : & \mathsf{VarBal\ Int\ Termlist\ Termlist} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Res} \\ \mathsf{tckbolex}_3(vb, wb, [\,], [\,]) & = & \langle vb, wb, \mathsf{E}\rangle \\ \mathsf{tckbolex}_3(vb, wb, s \,.\, ss, t \,.\, ts) & = & \textbf{let} \; \langle vb', wb', res\rangle = \mathsf{tckbo}_3(vb, wb, s, t) \\ & & \textbf{in} \;\; \textbf{if} \; res = \mathsf{E} \; \textbf{then} \; \mathsf{tckbolex}_3(vb', wb', ss, ts) \\ & & \quad\;\; \textbf{else let} \;\; \langle vb'', wb''\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb', wb', ss, \mathsf{true}) \\ & & \qquad\qquad\;\; \langle vb''', wb'''\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb'', wb'', ts, \mathsf{false}) \\ & & \qquad\;\; \textbf{in} \;\; \langle vb''', wb''', res\rangle\end{array}$$

In the following, we describe the optimized version of tckbo, which admittedly looks rather complex. However, most of the complexity comes from the threading of the state. The complicated-looking **if-elif**-expression simply checks the different possibilities in a systematic way.

$$
\begin{array}{rcl}
\mathsf{tckbo}_3 & : & \mathsf{VarBal\ Int\ Term\ Term} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Res} \\
\mathsf{tckbo}_3(vb, wb, \mathsf{V}(x), \mathsf{V}(y)) & = & \textbf{let}\ \ vb' = \mathsf{inc}(vb, x) \\
 & & \qquad vb'' = \mathsf{dec}(vb', y) \\
 & & \qquad res = \textbf{if}\ x = y\ \textbf{then}\ \mathsf{E}\ \textbf{else}\ \mathsf{N} \\
 & & \textbf{in}\ \ \langle vb'', wb, res\rangle \\
\mathsf{tckbo}_3(vb, wb, \mathsf{V}(x), \mathsf{F}(g, ts)) & = & \textbf{let}\ \langle vb', wb', ctn\rangle = \mathsf{mdfyVWBc}(vb, wb, \mathsf{F}(g, ts), x, \mathsf{false}) \\
 & & \qquad res = \textbf{if}\ ctn\ \textbf{then}\ \mathsf{L}\ \textbf{else}\ \mathsf{N} \\
 & & \qquad vb'' = \mathsf{inc}(vb', x) \\
 & & \textbf{in}\ \ \langle vb'', wb' + \mu, res\rangle \\
\mathsf{tckbo}_3(vb, wb, \mathsf{F}(f, ss), \mathsf{V}(y)) & = & \textbf{let}\ \langle vb', wb', ctn\rangle = \mathsf{mdfyVWBc}(vb, wb, \mathsf{F}(f, ss), y, \mathsf{true}) \\
 & & \qquad res = \textbf{if}\ ctn\ \textbf{then}\ \mathsf{G}\ \textbf{else}\ \mathsf{N} \\
 & & \qquad vb'' = \mathsf{dec}(vb', y) \\
 & & \textbf{in}\ \ \langle vb'', wb' - \mu, res\rangle \\
\mathsf{tckbo}_3(vb, wb, \mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \textbf{let}\ \langle vb', wb', lex\rangle = \mathsf{tckbo}'_3(vb, wb, f, g, ss, ts) \\
 & & \qquad wb'' = wb' + \varphi(f) - \varphi(g) \\
 & & \qquad \langle vb'', nNeg\rangle = \mathsf{noNeg}(vb') \\
 & & \qquad \langle vb''', nPos\rangle = \mathsf{noPos}(vb'') \\
 & & \qquad G\text{-}or\text{-}N = \textbf{if}\ nNeg\ \textbf{then}\ \mathsf{G}\ \textbf{else}\ \mathsf{N} \\
 & & \qquad L\text{-}or\text{-}N = \textbf{if}\ nPos\ \textbf{then}\ \mathsf{L}\ \textbf{else}\ \mathsf{N} \\
 & & \textbf{in}\ \ \textbf{if}\ \ wb'' > 0\ \textbf{then}\ \langle vb''', wb'', G\text{-}or\text{-}N\rangle \\
 & & \quad \textbf{elif}\ \ wb'' < 0\ \textbf{then}\ \langle vb''', wb'', L\text{-}or\text{-}N\rangle \\
 & & \quad \textbf{elif}\ \ f >_{\mathsf{F}} g\ \ \textbf{then}\ \langle vb''', wb'', G\text{-}or\text{-}N\rangle \\
 & & \quad \textbf{elif}\ \ g >_{\mathsf{F}} f\ \ \textbf{then}\ \langle vb''', wb'', L\text{-}or\text{-}N\rangle \\
 & & \quad \textbf{elif}\ \ f \neq g\ \ \ \textbf{then}\ \langle vb''', wb'', \mathsf{N}\rangle \\
 & & \quad \textbf{elif}\ \ lex = \mathsf{E}\ \ \textbf{then}\ \langle vb''', wb'', \mathsf{E}\rangle \\
 & & \quad \textbf{elif}\ \ lex = \mathsf{G}\ \ \textbf{then}\ \langle vb''', wb'', G\text{-}or\text{-}N\rangle \\
 & & \quad \textbf{elif}\ \ lex = \mathsf{L}\ \ \textbf{then}\ \langle vb''', wb'', L\text{-}or\text{-}N\rangle \\
 & & \quad \textbf{else}\ \langle vb''', wb'', \mathsf{N}\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{tckbo}'_3 & : & \mathsf{VarBal\ Int\ Fid\ Fid\ Termlist\ Termlist} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Res} \\
\mathsf{tckbo}'_3(vb, wb, f, g, ss, ts) & = & \textbf{if}\ f = g\ \textbf{then}\ \mathsf{tckbolex}_3(vb, wb, ss, ts) \\
 & & \textbf{else let}\ \ \langle vb', wb'\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb, wb, ss, \mathsf{true}) \\
 & & \qquad\quad \langle vb'', wb''\rangle = \mathsf{mdfyVWB}_{\mathsf{tl}}(vb', wb', ts, \mathsf{false}) \\
 & & \qquad \textbf{in}\ \ \langle vb'', wb'', \mathsf{N}\rangle
\end{array}
$$

Note that $\mathsf{tckbolex}_3$ is only called if $f = g$ which implies that ss and ts have the same length. Analogously, the value of lex is only considered if $f = g$ (i. e., results from invoking $\mathsf{tckbolex}_3$).

LEMMA 1 *Let* $vb = \vec{0}$ *and* $wb = 0$. *If* $s =_{\mathsf{t}} t$ *then* $\mathsf{tckbo}_3(vb, wb, s, t) = \langle\vec{0}, 0, \mathsf{E}\rangle$. *If* $ss =_{\mathsf{tl}} ts$ *then* $\mathsf{tckbolex}_3(vb, wb, ss, ts) = \langle\vec{0}, 0, \mathsf{E}\rangle$.

PROOF Simultaneous induction on $|s| + |t|$ and $|ss| + |ts|$. If $s =_{\mathsf{t}} t$ then either $s = \mathsf{V}(x)$, $t = \mathsf{V}(y)$, and $x = y$, or $s = \mathsf{F}(f, ss)$, $t = \mathsf{F}(g, ts)$, $f = g$, and $ss =_{\mathsf{tl}} ts$. In the first case, tckbo_3 first increments, then decrements the same entry of vb, then sets res to E and

uses wb unmodified. Hence, it returns $\langle\vec{0}, 0, \mathsf{E}\rangle$. In the second case, tckbo'_3 calls $\mathsf{tckbolex}_3$ which returns $\langle\vec{0}, 0, \mathsf{E}\rangle$ by induction hypothesis. By adding and subtracting the same value the weight balance is not changed. Hence, tckbo_3 returns $\langle\vec{0}, 0, \mathsf{E}\rangle$.

If $ss =_{\mathsf{tl}} ts$ then either both lists are $[\,]$ or $ss = s \centerdot ss'$, $ts = t \centerdot ts'$, $s =_{\mathsf{t}} t$, and $ss' =_{\mathsf{tl}} ts'$. In the first case, $\mathsf{tckbolex}_3$ lets vb and wb unmodified. Hence, it returns $\langle\vec{0}, 0, \mathsf{E}\rangle$. In the second case, it calls tckbo_3 which by induction hypothesis returns $\langle\vec{0}, 0, \mathsf{E}\rangle$. Then $\mathsf{tckbolex}_3$ calls itself with $vb' = \vec{0}$ and $wb' = 0$. Hence, induction hypothesis applies, the result is $\langle\vec{0}, 0, \mathsf{E}\rangle$. □

Because recursive calls of tckbo_3 and $\mathsf{tckbolex}_3$ occur only as long as the symbols of s and t are identical, we immediately get the following corollary.

COROLLARY 1 *The invariant I is established by* tckbo_3 *and* $\mathsf{tckbolex}_3$. □

LEMMA 2 *Let* $vb = \vec{0}$ *and* $wb = 0$. *Then* $\mathsf{tckbo}_3(vb, wb, s, t) = \mathsf{tckbo}(vb, wb, s, t)$. *If* $\mathsf{length}(ss) = \mathsf{length}(ts)$ *then* $\mathsf{tckbolex}_3(vb, wb, ss, ts) = \mathsf{tckbolex}(vb, wb, ss, ts)$.

PROOF Simultaneous induction on $|s| + |t|$ and $|ss| + |ts|$. Lemma 1 covers the cases where $s =_{\mathsf{t}} t$ and $ss =_{\mathsf{tl}} ts$. Hence, it suffices to consider the remaining cases.

If s and t are different variables then the variable balances are updated accordingly and N is determined for the ordering relation. As φ maps all variables to μ, wb remains unchanged. If one term is a variable and the other is not then $\mathsf{mdfyVWBc}$ modifies the balances for the nonvariable terms. Furthermore, it tests whether the variable occurs in the term which determines the value of *res*. Finally, the balances are updated to take the variable into account. It remains the case where s and t are both nonvariable terms. If both top-symbols are identical then $\mathsf{tckbolex}_3$ is called which by induction hypothesis returns the correct result for the comparison of the arguments. Otherwise, tckbo'_3 updates the balances for the arguments by calling $\mathsf{mdfyVWB}_{\mathsf{tl}}$. Handling the weights of the top-symbols is the duty of tckbo_3. Variables *G-or-N* and *L-or-N* record whether or not the variable balances approve a possible G or L result. The **if-elif**-expression then simply follows the definition of KBO and considers first the weights, then the precedence, and finally the result of the lexicographic comparison.

If ss and ts are not identical then, after considering some possibly empty prefix, $\mathsf{tckbolex}_3$ calls tckbo_3 with $vb = \vec{0}$, $wb = 0$, and two different terms. Hence, by induction hypothesis, $res \neq \mathsf{E}$ and the balances are updated accordingly. □

With the help of tckbo_3 it is easy to compute ckbo_3 which is mainly a wrapper that handles the allocation and deallocation of the array for the variable balances.

$$\begin{array}{lcll} \mathsf{ckbo}_3(s,t) & = & \textbf{let} & vb = \mathsf{newVB}(\langle\rangle) \\ & & & \langle vb', wb, res\rangle = \mathsf{tckbo}_3(vb, 0, s, t) \\ & & & \langle\rangle = \mathsf{freeVB}(vb') \\ & & \textbf{in} & res \end{array}$$

THEOREM 4 *Let s and t be well-formed* Terms. *Then* $\mathsf{ckbo}_3(s,t) = \mathsf{ckbo}_2(s,t)$. *The worst-case running time of* $\mathsf{ckbo}_3(s,t)$ *is* $O(KN)$ *where* $K = |\mathcal{V}_{\mathrm{fin}}|$ *and* $N = |s| + |t|$.

PROOF Function ckbo_3 calls newVB, tckbo_3, and freeVB. Lemma 2 ensures the correctness of ckbo_3. The costs for newVB and freeVB are $O(K)$. During the evaluation of

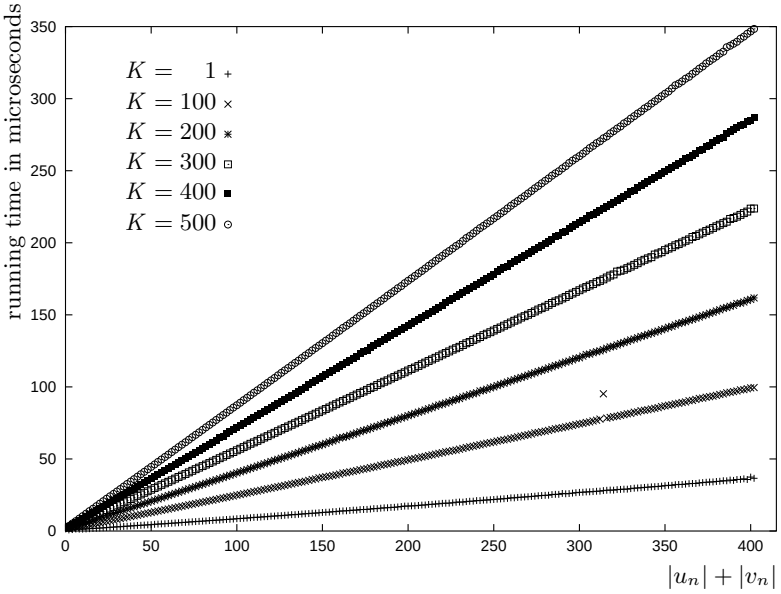


Figure 3: Time needed to evaluate $\text{KBO}_3(u_n, v_n)$ with $n = 1, \ldots, 200$ and six different values of K (cf. Example 1)

$\mathsf{tckbo}_3(vb, wb, s, t)$ each symbol in s and t is considered once. Except functions noNeg and noPos all operations need constant time. Functions noNeg and noPos need $O(K)$ time for each invocation. In the worst case, both functions are called $N/2$ times. Hence, $\mathsf{tckbo}_3(vb, wb, s, t)$ needs at most $O(KN)$ steps. Therefore, the worst-case running time of $\mathsf{ckbo}_3(s, t)$ is $O(KN)$. □

Revisiting Example 1 again, we can see in Figure 3 that the running time of KBO_3, the C-implementation of kbo_3, depends linearly on the size of its arguments. It is also considerably faster than KBO_2 for these test cases. (Note the different scale of Figures 2 and 3). The slope of the lines is linear with K.

3.4 Optimizing the variable test

By using the tupling strategy we have successfully avoided the quadratic behavior. However, as can be seen in Figure 3, the running time of KBO_3 still depends noticeably on K, the number of different variables in the prover's state. Each invocation of noNeg and noPos tests for each of the K variables its balance, which apparently is in $O(K)$. The main idea to transform both operations into tests with constant costs is to keep counters for the number of positive variable balances and for the number of negative variable balances. Let getPC and getNC return the values of these counters. Then we can implement noNeg and noPos in the following way:

$$\begin{array}{rcl} \mathsf{noNeg} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noNeg}(vb) & = & \textbf{let } \langle vb', n\rangle = \mathsf{getNC}(vb) \\ & & \textbf{in } \langle vb', n = 0\rangle \end{array} \qquad \begin{array}{rcl} \mathsf{noPos} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noPos}(vb) & = & \textbf{let } \langle vb', n\rangle = \mathsf{getPC}(vb) \\ & & \textbf{in } \langle vb', n = 0\rangle \end{array}$$

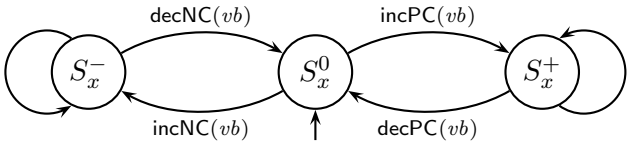
A nice effect of this change is that functions noNeg and noPos are now independent from the representation of VarBal. This knowledge is hidden in getNC and getPC.

We keep the two counters in two extra entries of the array that represents the variable balances. We use entry 0 to count the negative variable balances and entry $K+1$ to count the positive variable balances. We therefore have to change function newVB:

$$\begin{array}{rcl} \mathsf{newVB} & : & \mathbf{1} \to \mathsf{VarBal} \\ \mathsf{newVB}(\langle\rangle) & = & \mathsf{newArray}(\mathsf{Int}, 0, K+1) \end{array}$$

To increment or decrement the counters we use functions incNC, incPC, decNC, and decPC, which all have straightforward constant-time definitions.

We modify the counters each time the variable balance for a variable x leaves or reaches 0. Thus, we treat the variable balance for x as the following finite automaton:



Functions inc and dec are then defined accordingly:

$$\begin{array}{rcl} \mathsf{inc} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\ \mathsf{inc}(vb,x) & = & \mathbf{let} \quad i = \mathsf{index}(x) \\ & & \quad \langle vb', n\rangle = \mathsf{read}(vb, i) \\ & & \quad\quad vb'' = \mathsf{update}(vb', i, n+1) \\ & & \mathbf{in\ if} \quad n = 0 \quad \mathbf{then}\ \mathsf{incPC}(vb'') \\ & & \quad \mathbf{elif}\ \ n = -1\ \mathbf{then}\ \mathsf{decNC}(vb'') \\ & & \quad \mathbf{else}\ vb'' \end{array}$$

$$\begin{array}{rcl} \mathsf{dec} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\ \mathsf{dec}(vb,x) & = & \mathbf{let} \quad i = \mathsf{index}(x) \\ & & \quad \langle vb', n\rangle = \mathsf{read}(vb, i) \\ & & \quad\quad vb'' = \mathsf{update}(vb', i, n-1) \\ & & \mathbf{in\ if} \quad n = 0\ \mathbf{then}\ \mathsf{incNC}(vb'') \\ & & \quad \mathbf{elif}\ \ n = 1\ \mathbf{then}\ \mathsf{decPC}(vb'') \\ & & \quad \mathbf{else}\ vb'' \end{array}$$

Keeping the two counters has a profound influence on the running time. As getNC and getPC both need constant time, functions noNeg and noPos are both in $O(1)$. Functions inc and dec need more time than before, but are still in $O(1)$. Hence, we get the following result for function ckbo_4 which differs from ckbo_3 by using the counter-based variable tests:

THEOREM 5 *Let s and t be well-formed* Terms*. Then* $\mathsf{ckbo}_4(s,t) = \mathsf{ckbo}_3(s,t)$*. The worst-case running time of* $\mathsf{ckbo}_4(s,t)$ *is* $O(K+N)$ *where* $K = |\mathcal{V}_{\mathrm{fin}}|$ *and* $N = |s| + |t|$*.*

PROOF It is easy to see that the counter-based variable test is equivalent to the previous one. For determining the costs, recall that function ckbo_4 invokes newVB, tckbo_4, and freeVB. The costs for newVB and freeVB are $O(K)$. During the evaluation of $\mathsf{tckbo}_4(vb, wb, s, t)$ each symbol in s and t is considered once. Including functions noNeg and noPos the operations performed for each symbol need constant time. Hence, $\mathsf{tckbo}_4(vb, wb, s, t)$ needs $O(N)$ time. Therefore, the worst-case running time of $\mathsf{ckbo}_4(s,t)$ is $O(K+N)$. □

We denote the C-implementation of kbo_4 with KBO_4. Revisiting Example 1 again, its dependency on K shows as a small constant offset in the two graphs depicted in Figure 4. (For clarity reasons, we have omitted the data for the other values of K). For $K = 500$, KBO_4 is also considerably faster than KBO_3, whereas for $K = 1$ the differences are tiny, as can be seen by comparison with Figure 3.

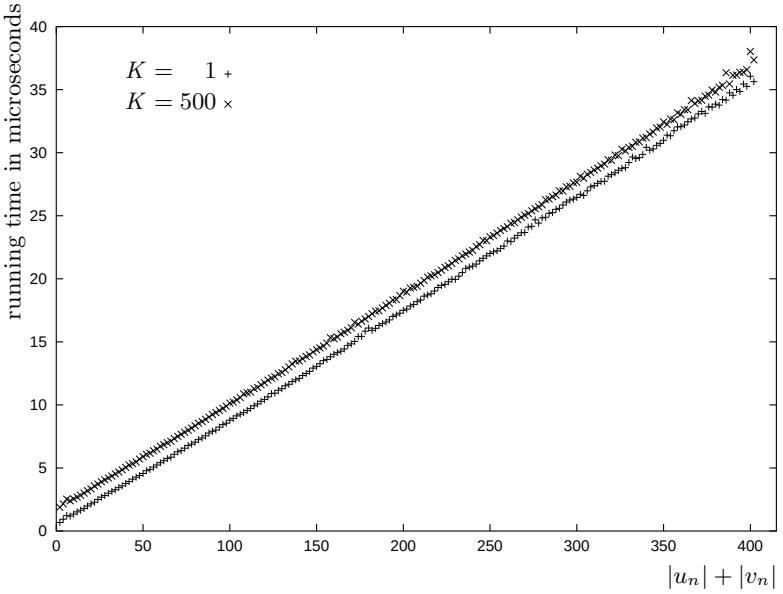


Figure 4: Time needed to evaluate $\text{KBO}_4(u_n, v_n)$ with $n = 1, \ldots, 200$ and two different values of K (cf. Example 1)

3.5 One final modification

The optimized handling of the variable test has successfully avoided the dependency on K of functions noNeg and noPos. However, functions newVB and freeVB may still depend on K. The aim of our final modification is to change that. As a result, the evaluation of $\mathsf{ckbo}_5(s,t)$ is independent of the number of variables in the prover's state. The practical improvements we can expect are rather small, as Figure 4 indicates. The real importance of this result is that it is asymptotically optimal: the running time of $\mathsf{ckbo}_5(s,t)$ solely depends on $|s|$ and $|t|$, the size of its actual arguments.

A general observation with the theorem prover WALDMEISTER is that calls to the ordering occur much more frequently than changes of K. We therefore make the following modification: the memory for the variable balances is not allocated and deallocated for each invocation of ckbo_5. Instead, this memory is handled outside the ordering in some global variable *globalVB*. If $\mathcal{V}_{\text{fin}}$ changes then the size of *globalVB* is adjusted accordingly.[3] This is not the duty of the module implementing the ordering. Therefore, we have to establish some invariant about the contents of *globalVB*. We assume that all entries of *globalVB* are zero at the invocation of ckbo_5. Hence, we have to ensure that this holds true when ckbo finishes.

The task of function freeVB is therefore to zero out all entries that are modified during the evaluation of $\mathsf{tckbo}_5(vb, wb, s, t)$. To do this independently of K, we have to record the modified entries during the computation. Therefore, we change the representation of variable balances. Data type VarBal is now an array of pairs of integers with indices

[3]It is not strictly necessary to keep a global array for that purpose. Many theorem provers keep for each variable a structure for recording its name, sort, etc. Given a certain variable, the corresponding structure is usually quickly accessible. We could therefore avoid *globalVB* by storing its contents in these structures. It is easy to adapt the simpler array-based formulation we have chosen to such a setting.

ranging from 0 to $K+1$. Of each entry, the first component keeps the variable balance as before. With the second component we establish a singly-linked list of the modified entries.[4] We use functions read_1 and update_1 to read and modify the first component of an entry, and similarly read_2 and update_2 for the second component.

The new versions of functions getNC, getPC, incNC, decNC, incPC, and decPC are identical to the old ones, except that read is replaced by read_1 and update is replaced by update_1. Hence, their running times remain constant. Functions inc and dec are changed analogously. In addition, they call function record to indicate that the entry is changed:

$$
\begin{array}{rcl}
\mathsf{inc} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\
\mathsf{inc}(vb,x) & = & \textbf{let} \quad i = \mathsf{index}(x) \\
 & & \quad \langle vb', n\rangle = \mathsf{read}_1(vb,i) \\
 & & \quad\quad vb'' = \mathsf{update}_1(vb', i, n+1) \\
 & & \quad\quad vb''' = \mathsf{record}(vb'', i) \\
 & & \textbf{in}\ \textbf{if}\ \ n = 0\ \ \textbf{then}\ \mathsf{incPC}(vb''') \\
 & & \quad \textbf{elif}\ n = -1\ \textbf{then}\ \mathsf{decNC}(vb''') \\
 & & \quad \textbf{else}\ vb'''
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{dec} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\
\mathsf{dec}(vb,x) & = & \textbf{let} \quad i = \mathsf{index}(x) \\
 & & \quad \langle vb', n\rangle = \mathsf{read}_1(vb,i) \\
 & & \quad\quad vb'' = \mathsf{update}_1(vb', i, n-1) \\
 & & \quad\quad vb''' = \mathsf{record}(vb'', i) \\
 & & \textbf{in}\ \textbf{if}\ \ n = 0\ \textbf{then}\ \mathsf{incNC}(vb''') \\
 & & \quad \textbf{elif}\ n = 1\ \textbf{then}\ \mathsf{decPC}(vb''') \\
 & & \quad \textbf{else}\ vb'''
\end{array}
$$

The singly-linked list is handled by the following functions. We use the value $K+1$ as sentinel for the end of the list. The anchor of the list is the second component of the zeroth array entry. We assume that variable *globalVB* contains the result of $\mathsf{newArray}(\mathsf{Int} \times \mathsf{Int}, 0, K+1)$.

$$
\begin{array}{rcl}
\mathsf{newVB} & : & \mathbf{1} \to \mathsf{VarBal} \\
\mathsf{newVB}(\langle\rangle) & = & \textbf{let}\ \ vb = \mathit{globalVB} \\
 & & \quad vb' = \mathsf{update}_2(vb, 0, K+1) \\
 & & \textbf{in}\ \ vb'
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{clear} & : & \mathsf{VarBal}\,\mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{clear}(vb,i) & = & \textbf{let}\ \langle vb', i'\rangle = \mathsf{read}_2(vb,i) \\
 & & \quad\quad vb'' = \mathsf{update}(vb', i, \langle 0,0\rangle) \\
 & & \textbf{in}\ \ \textbf{if}\ i = K+1\ \textbf{then}\ vb'' \\
 & & \quad \textbf{else}\ \mathsf{clear}(vb'', i')
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{freeVB} & : & \mathsf{VarBal} \to \mathbf{1} \\
\mathsf{freeVB}(vb) & = & \textbf{let}\ vb' = \mathsf{clear}(vb, 0) \\
 & & \textbf{in}\ \ \langle\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{record} & : & \mathsf{VarBal}\,\mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{record}(vb,i) & = & \textbf{let}\ \langle vb', i'\rangle = \mathsf{read}_2(vb,i) \\
 & & \textbf{in}\ \ \textbf{if}\ i' = 0\ \textbf{then}\ \mathsf{record}'(vb', i) \\
 & & \quad \textbf{else}\ vb'
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{record}' & : & \mathsf{VarBal}\,\mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{record}(vb,i) & = & \textbf{let}\ \langle vb', i'\rangle = \mathsf{read}_2(vb,0) \\
 & & \quad\quad vb'' = \mathsf{update}_2(vb', i, i') \\
 & & \quad\quad vb''' = \mathsf{update}_2(vb'', 0, i) \\
 & & \textbf{in}\ \ vb'''
\end{array}
$$

Functions newVB and record need constant time. Hence, functions inc and dec need constant time, too. The running time of freeVB is determined by the length of the list. If there are M different variables in s and t then function clear overwrites $M+2$ entries of the array: In addition to the entries representing variable balances the two entries at position 0 and $K+1$ which represent the counters. Let $N = |s| + |t|$. Then $M \le N$. This leads to the desired result for function ckbo_5 which uses the new versions of the functions handling the variable balances.

THEOREM 6 *Let s and t be well-formed* Terms*. Then* $\mathsf{ckbo}_5(s,t) = \mathsf{ckbo}_4(s,t)$*. The worst-case running time of* $\mathsf{ckbo}_5(s,t)$ *is* $O(|s| + |t|)$*.*

[4] This is a simple way to achieve the desired result. There are other sensible approaches.

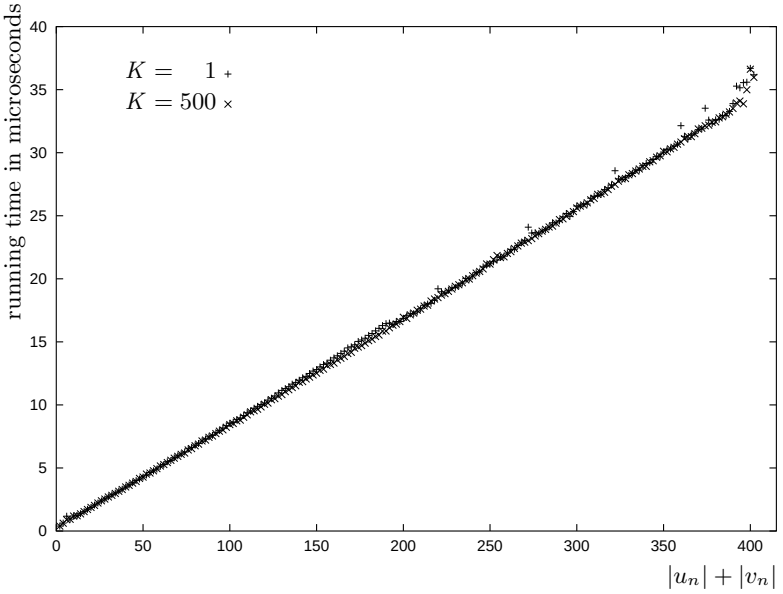


Figure 5: Time needed to evaluate $\text{KBO}_5(u_n, v_n)$ with $n = 1, \ldots, 200$ and two different values of K (cf. Example 1)

PROOF The different memory management does not affect correctness as all modified entries are zeroed out by clear. Concerning the costs, function ckbo_5 calls besides tckbo_5 the new versions of newVB and freeVB. The costs for newVB are $O(1)$. The costs for freeVB are $O(M)$ where $M = |\text{Var}(s,t)| \leq |s|+|t|$. Except calling the new versions of the auxiliary functions, function tckbo_5 is identical to function tckbo_4. As these auxiliary functions all need constant time, evaluating $\mathsf{tckbo}_5(vb, wb, s, t)$ needs $O(|s| + |t|)$ time. Therefore, the worst-case running time of $\mathsf{ckbo}_5(s,t)$ is $O(|s| + |t|)$. □

KBO_5 is our C-implementation of kbo_5. Figure 5 depicts the running times of KBO_5 for the test cases of Example 1 and two different values of K. The two graphs are practically indistinguishable, which indicates that the running time of KBO_5 does not depend on K. The differences to KBO_4 are, however, rather small, cf. Figure 4.

4 Experimental evaluation

During the development of the different versions of kbo we presented experimental data for a well chosen worst-case example to illustrate the different asymptotic running time behavior. Unfortunately, such experiments tell us nothing about the improvements we can expect in practice: the terms tested for their ordering relations in real proof attempts do not follow such simple patterns.

To capture ordering tests originating from real proof attempts, we performed two kinds of experiments: For five individual examples we measured the time for the calls to the ordering until WALDMEISTER finished successfully. The examples belong to different domains of TPTP [SS98]. Furthermore, we made measurements for all 722 UEQ-problems of TPTP-2.7.0. For that, we first ran WALDMEISTER using KBO_1 with a time limit of 10 seconds and recorded how often the ordering routines were called. We then

problem	number of calls to ordering	time needed by ordering				
		KBO$_1$	KBO$_2$	KBO$_3$	KBO$_4$	KBO$_5$
BOO031-1	455 754	1.500	0.613	0.477	0.531	0.401
GRP179-1	436 624	0.881	0.527	0.433	0.463	0.324
LCL109-2	106 160	0.337	0.166	0.107	0.121	0.081
LAT085-1	195 780	1.370	0.356	0.258	0.287	0.185
ROB006-1	511 148	2.544	0.894	0.757	0.767	0.590
TPTP$_{10}$	64 151 129	307.800	114.400	85.400	97.350	77.660

Table 1: Time (in seconds) needed for ordering comparisons

performed the actual measurements by running the prover with the different versions KBO$_i$ and by aborting the run after the corresponding number of calls is reached. We use TPTP$_{10}$ to refer to the summarized results.

As we can learn from Table 1, the optimizations usually improve the time needed for the ordering comparisons. However, the improvements are rather moderate compared to our running example. An interesting observation is that KBO$_4$ usually needs more time than KBO$_3$. This means that in these experiments the counter-based variable test is inferior to simply checking the variable balances individually. The reason is that in WALDMEISTER the number of different variables is kept as small as possible. Therefore, we can observe rather small values of K. For the runs of TPTP$_{10}$ the average value of K is 9.5, and for only 13 of the 722 problems it is greater than twenty. Therefore, only few comparisons can be saved, whereas for each invocation of inc and dec additional comparisons are necessary. The potential of the optimization lies in cases where K is greater than the number of variables occurring in the terms. In the test runs of TPTP$_{10}$ these cases are too rare to lead to speed-ups. Considering the small average value of K, we have to attribute the improvements of KBO$_5$ to the significantly reduced number of allocations and deallocations. The linear dependency on K of KBO$_4$ does not seem relevant for such small values of K.

5 Related work

We are unaware of any work discussing the efficient implementation of the basic KBO test. The thesis of Steinbach contains a short overview about the time complexities of orderings [Ste94, Chap. 6.2]. He shows that KBO can be evaluated in $O(|s| \cdot |t|)$ time. Unfortunately, his discussion does not cover the variable test. Weidenbach suggests to implement the KBO straight after its definition [Wei01].

We analyzed the KBO-implementations of several systems that participated in CASC, the system competition of automated theorem provers affiliated with the Conference on Automated Deduction (see `http://www.tptp.org/CASC`). Typically, a recursive variant similar to kbo$_2$ is employed performing the variable test with the help of some data structure. We find the use of arrays, hash tables, and association lists. An important variation is to delay the variable test, that is, it is only performed after the tests belong-

ing to part (2) of Definition 1 have been checked successfully. This helps to reduce the costs for the variable test. Some systems do not implement the full KBO, but use simple weight-based orderings instead thus avoiding the recursive calls in the lexicographic part and therefore the quadratic worst-case behavior.

A different approach is used in the Vampire system where the variable test and the weight-based test are combined by using (linear) polynomials. Recurring tests with two terms under different substitutions are optimized by using a specialized partial evaluation technique (see [RV04]). This is beneficial, because most invocations of the ordering test do not only concern two terms s and t but in addition some substitution σ. The simplest way to determine whether $\sigma(s) \succ \sigma(t)$ holds is to apply the substitution and to call the basic ordering tests on the instances. However, explicitly constructing the instantiated terms is rather costly. By passing σ as an additional parameter in the recursive calls and looking up bindings of variables if necessary, we can improve on that considerably. The modification to our implementations is straightforward, but has the disadvantage that for some variable x the term $\sigma(x)$ may be traversed several times in case x occurs more than once. It will be interesting to see how our linear version of KBO can be combined with the technique of [RV04] which avoids this multiple traversal.

6 Conclusions

Starting from an "obviously correct", but inefficient variant, we developed step-by-step an efficient implementation of KBO. From an initially quadratic algorithm, which depends furthermore on the number of variables in the system K, we derive a linear one which is independent of K. The key is to use techniques known from the program-transformation community and appropriate data structures. To our knowledge, this is the first linear implementation of KBO. At least, several developers of CASC-systems were surprised to hear that KBO can be implemented in linear time.

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 – even by doing it by hand without a dedicated system. The manual proofs revealed two bugs in the derivation of kbo_3. In our C translations we found about a dozen bugs; most of them occurred in the development of KBO$_3$. This indicates that the step between kbo_2 and kbo_3 is too wide to do without machine support. With a proper program transformation system it should be easy. Nevertheless, this low number of errors is far better than what is usually achieved by traditional coding practice, especially when we take into account the intricacy of the algorithms. We are convinced that the chosen step-by-step approach is superior to presenting the final algorithm and proving its correctness at once. Our experience suggests to use this two-level development approach for other subtasks in a prover, especially when they need a significant amount of the running time and an efficient implementation is not obvious.

Acknowledgments: Thomas Hillenbrand was the first who made me aware of the difficulties with the variable check. Some e-mail discussion with Alexandre Riazanov and Andrei Voronkov stimulated me to properly write down my rough ideas about efficiently implementing KBO.

References

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BH96] A. Buch and T. Hillenbrand. WALDMEISTER: Development of a High Performance Completion-Based Theorem Prover. SEKI-Report 96-01, Universität Kaiserslautern, 1996.

[DP01] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. I, ch. 9. Elsevier, 2001.

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, ed., *Computational Algebra*, pp. 263–297. Pergamon Press, 1970.

[LH02] B. Löchner and T. Hillenbrand. A phytography of WALDMEISTER. *AI Communications*, 15(2–3):127–133, 2002. See `http://www.waldmeister.org`.

[Löc04] B. Löchner. Things to know when implementing LPO. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proc. of the 1st Workshop on Empirically Successful First Order Reasoning (ESFOR '04)*, 2004. Extended version to appear in *International Journal on Artificial Intelligence Tools*.

[PP93] A. Pettorossi and M. Proietti. Rules and Strategies for Program Transformation. In B. Möller, H. Partsch, and S. Schuman, eds., *Formal Program Development*, volume 755 of *LNCS*, pp. 263–304. Springer, 1993.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15:91–110, 2002. See `http://www.vampire.fm`.

[RV04] A. Riazanov and A. Voronkov. Efficient checking of term ordering constraints. In D. Basin and M. Rusinowitch, eds., *Proc. 2nd International Joint Conference on Automated Reasoning*, LNCS, pp. 60–74. Springer, 2004.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *AI Communications*, 15:111–126, 2002. See `http://www.eprover.org`.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. 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`.

[Wad92] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wei01] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, vol. II, ch. 27. Elsevier Science, 2001.