Proceedings of the Second International Workshop on the Implementation of Logics

Hans de Nivelle and Stephan Schulz

MPI–I–2001–2–006 November 2001

Author's Address

Hans de Nivelle
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany
Phone: +49 681 9325-223
Fax: +49 681 9325-299
Email: `nivelle@mpi-sb.mpg.de`

Stephan Schulz
Technische Universität München
Institut für Informatik
80290 München
Phone: +49 89 289 27914
Fax: +49 89 289 27902
Email: `schulz@informatik.tu-muenchen.de`

Abstract

This volume contains the papers that were accepted for presentation at the Second International Workshop on the Implementation of Logics. The workshop took place in Havana, Cuba on December 8th 2001 in conjunction with the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. (LPAR 2001) All contributions were reviewed by an international programm committee, whose names can be found elsewhere in this volume.

Keywords

Implementation of Logics, First Order Theorem Proving, Linear Logic, Explicit Substitutions.

Preface

This volume contains the papers that were presented at the *Second International Workshop on the Implementation of Logics.* The workshop was held in Havana (La Habana), Cuba, on December 8th, 2001. The workshop was held in conjunction with the *8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, (LPAR 2001).

Aim of the workshop was to bring together people working on implementation of logic in the broadest sense, in order to exchange ideas and techniques. There were contributions from different branches of logic. The majority of contributions is about first order theorem proving, but there are papers about higher order logic and linear logic as well.

The level of concreteness of the papers differs. Some of the papers describe concrete systems. Others discuss design dilemmas that reccur in many systems, or general techniques that can be reused in other systems.

Implementation of logic is not only an academic exercise, but also a something that matters to the outside world. Interactive verification systems are gradually finding applications. They are being used for verification of hardware, for verification of micro-code, and for verification of safety-critical parts of software.

Automated deduction systems have significantly improved during the last few years, and are now increasingly being used for various industrial and scientific applications. This progress was caused not only by improvement of hardware, but also by the appearance of more refined calculi, and better implementation techniques.

Until recently, there has been no dedicated forum to discuss implementation techniques. As a conseqeunce, implementation papers were scattered through larger conferences on AI, functional and declarative programming, or automated deduction. At these conferences, implementation papers were not always processed in the way they deserved. As a result, much knowledge about efficient implementation techniques is hard to find, and is often not written down at all.

In order to change this situation, Andrei Voronkov decided to organize the *Reunion Workshop on Implementations of Logic.* This workshop took place November 2000 on Reunion Island, in conjunction with LPAR 2000.

Because the first workshop was considered successful, we have organized the second workshop, this time as an open workshop. We received submissions from 5 countries spread over 3 continents. We express our gratitude to the Programme Committee, who were able to produce high quality reviews in short time. Eight papers were selected for presentation. We hope you enjoy them.

November 2001 Hans de Nivelle and Stephan Schulz

Programme Committee

- Thom Frühwirth, Ludwig-Maximilians-Universität München, Germany
- Joseph D. Horton, University of New Brunswick, Canada
- Ullrich Hustadt, University of Liverpool, UK
- Reinhold Letz, Technische Universität München, Germany
- Bernd Löchner, Universität Kaiserslautern, Germany
- William McCune, Argonne National Laboratory, USA
- Roberto Nieuwenhuis, Technical University of Catalonia, Spain
- Hans de Nivelle (Co-Chair), Max Planck Institut für Informatik, Saarbrücken, Germany
- Stephan Schulz (Co-Chair), Technische Universität München, Germany
- Mark E. Stickel, SRI International, USA
- Tanel Tammet, University of Göteborg *and* Chalmers University of Technology, Sweden

Contents

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kähler, R. Schneider, M. Botsch, C. Rössl, J. Vorsatz	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames A Technique for Making Lines Look Like Segments

MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification

Experiments with Marmoset, a Deduction System Based on Clausetrees

Graham Fyffe, Joseph D. Horton, and Yanping He
University of New Brunswick
email: jdh@unb.ca

1 Introduction

Marmoset (Marmoset Automated Reasoner Mostly Only Solves Easy Theorems) is a automated theorem prover being developed at the University of New Brunswick. It is a binary resolution [6] system which uses some of the ideas developed using clause trees [1] and binary resolution trees [8]. The major ideas are:

1. the rank/activity restriction, which is only developed using clause trees, and clause trees do not need to be implemented to use this restriction;
2. minimality, an extension of the regularity restriction;
3. surgery, in which pieces of a clause tree can be removed to improve the result of the clause tree.

The main purpose of Marmoset is to be a flexible theorem prover with which to test these ideas, and compare them with other resolution methods such as a-ordering. If one wants to compare two different procedures, it is better to use the same underlying system than to have two independent systems. One of the two systems will likely be much better at some tasks than the other, and the comparisons would be unfair.

There is a secondary hope that eventually Marmoset will be a competitive system, or at least part of a competitive system. It is still closer to a "proof of concept" level than an effective theorem prover. This note reports results of the first experiments run using Marmoset, specifically tests of the rank/activity restriction[2] versus a-ordering.

The next section discusses the theoretical background: clause trees, surgery, binary resolution trees, and the rank/activity restriction. The next section discusses Marmoset, and the remainder of the paper discusses the experiments and the results.

In a logical setting, lower case letters are used for predicate symbols, function symbols and constants. Upper case letters are used for variables. A bar over a predicate symbol represents negation. A clause is represented as a string of literals conjoined together, and a set of clauses is represented by a sequence of clauses separated by commas. In a mathematical setting, lower case letters are used for nodes and edges of graphs, upper case letters are used for sets, trees, and L is used for a labeling function.

2 Background

2.1 Clause trees

Binary resolution proofs can be represented by a clause tree, introduced in [1]. Conceptually, a clause tree represents a clause together with a proof from a set of input clauses. An input clause is represented by a complete bipartite graph $K_{1,n}$ or claw, in which the leaves correspond to the atoms of the literals of the clause, modified by the sign on the edge connecting the leaf to the central vertex. Such a clause tree is said to be *elementary.* A new clause tree can be built by resolving two complementary literals from different elementary clause trees. Identify the two leaves, so the resolved literal becomes an internal node of the tree, thereby building a clause tree with leaves still corresponding to the other literals of the clauses. Thus leaves of the clause tree correspond to the literals of the clause. If there are two leaves with unifiable or identical literals, then two unifiable or identical literals occur in the clause. Merging two such literals is represented in the clause tree by applying a substitution if necessary, and choosing a *merge path* from the leaf corresponding to the removed literal to the other leaf corresponding to the now identical literal.

The above discussion suggests a procedural definition, by giving the operations to construct clause clause trees, as in [1]. Here the definition is structural.

Definition 1 (Clause Tree) *$\mathcal{T} = (N, E, L, M)$ is a clause tree on a set S of input clauses if:*

1. *(N, E) as a graph is an unrooted tree.*
2. *L is a labeling of the nodes and edges of the tree. $L : N \cup E \to S \cup A \cup \{+, -\}$, where A is the set of instances of atoms in S. Each node is labeled either by a clause in S and called a clause node, or by an atom in A and called an atom node. Each edge is labeled $+$ or $-$.*
3. *No atom node is incident with two edges labeled the same.*
4. *Each edge $e = \{a, c\}$ joins an atom node a and a clause node c; it is associated with the literal $L(e)L(a)$.*
5. *For each clause node c, $\{L(a, c)L(a) | \{a, c\} \in E\}$ is an instance of $L(c)$. A path $(v_0, e_1, v_1, \ldots, e_n, v_n)$ where $0 \leq i \leq n$, $v_i \in N$ and $e_j \in E$ where $1 \leq j \leq n$ is a* merge path *if $L(e_1)L(v_0) = L(e_n)L(v_n)$. Path $(v_0, \ldots, v_n)$* precedes *$(\prec)$ path $(w_0, \ldots, w_m)$ if $v_n = w_i$ for some $i = 1, \ldots, m - 1$.*
6. *M is the set of chosen merge paths such that:*

(a) *the tail of each is a leaf (called a* closed *leaf),*

(b) *the tails are all distinct and different from the heads, and*

(c) *the relation $\prec$ on M can be extended to a partial order, that is, does not contain a cycle.*

An *open leaf* is an atom node leaf that is not the tail of any chosen merge path. The set the literals at the open leaves of a clause tree $\mathcal{T}$ is called the *clause* of $\mathcal{T}$, $cl(\mathcal{T})$.

When a merge path is chosen between two open leaves, there is no reason to choose one direction over the other, unless one specifies some arbitrary heuristic. The corresponding proofs remain exactly the same. One can define a *path reversal* operation which changes the clause tree except that one merge path runs in the opposite direction, which may cause some other merge paths to be modified somewhat. Then two clause trees are said to be *reversal equivalent* if there is a sequence of path reversals which transform one tree to the other. Perhaps a better alternative, developed in [7] in a slightly different context (tableaux) and put into general clause trees in [1], is the foothold restriction, which can be used to make an arbitrary choice that is consistent regardless of the order of the resolutions.

2.2 Clause tree surgery

Surgery is a process in which a clause tree has pieces removed, and the remaining pieces can be reassembled to form a new clause tree. The resulting tree has as its open leaves, a subset of the open leaves of the original. Hence the result of surgery subsumes the initial clause tree. Moreover the removal of some of the resolutions (and factors) means that possibly some of the substitutions caused by unifications are no longer necessary, so that even if no open leaves are removed, the resulting clause may strictly subsume the original.

The exact conditions under which surgery can be applied are rather technical, and are specified in detail in [1]. The main idea is that there can be two distinct atom nodes which can be identified or merged, with the result that parts of the original proof are no longer needed. Another way to look at surgery is that there is a proof corresponding to the clause tree which does not satisfy the regularity condition of [11], which implies that the proof can be shortened.

Figures 1 and 2 give examples of surgery. The first clause tree in 1 proves $q(a)$ from the clauses $q(X)p(X), \bar{p}(X)r(X), \bar{r}(X)\bar{p}(X), p(a)$. After surgery, the result is $q(X)$. In this example, a new merge path is chosen, and as a result a unification is removed. The merge path in the "before" clause tree in Figure 1 must be searched for, but in the "after" clause tree of Figure 1, the merge path is a chosen merge path and is part of the clause tree data structure.

In the second example Figure 2, a *tautology path* is found, which allows a piece from the middle of the clause tree to be removed. Note that the tautology

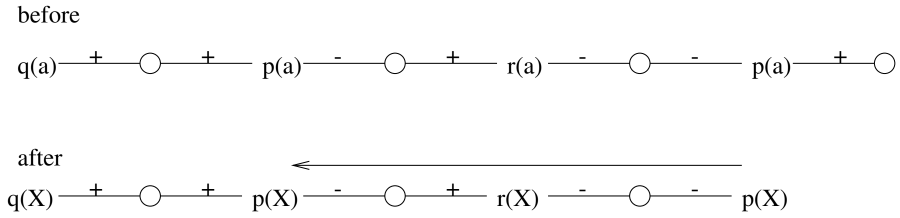


Figure 1: Surgery on a clause tree on clauses $q(X)p(X)$, $\bar{p}(X)r(X)$, $\bar{r}(X)\bar{p}(X)$, $p(a)$

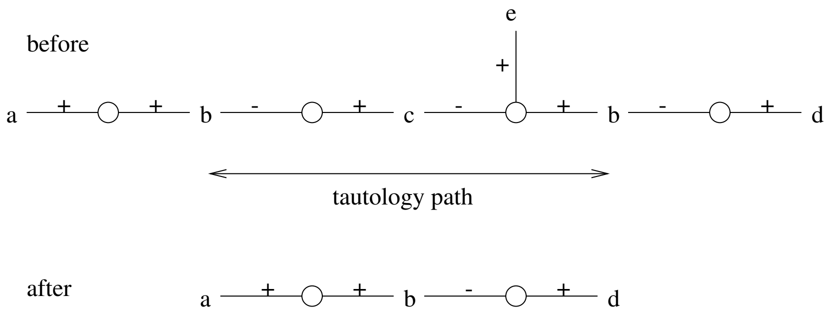


Figure 2: Surgery on a clause tree on clauses $a\bar{b}$, $\bar{b}c$, $\bar{b}\bar{c}e$, bd

path is not recorded as part of the clause tree structure, but must be searched for. It disappears from the "after" clause tree.

2.3 Binary resolution trees

Clause trees are not easy to implement directly efficiently. Instead Marmoset uses binary resolution trees (brt) [8]. A brt is a type of proof tree, which can be defined recursively as follows. An input clause by itself is a brt. Otherwise a brt is a clause, together with the brts of the two clauses which are resolved to form the clause, and the literals which were resolved. A detailed structural definition is given in [8].

To be able construct a brt that is used to prove a clause, it is only necessary that each clause know its own parents, and what literals were resolved. This is also what is needed to be recorded in order to write out a proof, so this is not very difficult to store. What one must be able to do to use brts to implement clause trees, is to determine when surgery can be applied, and how to implement surgery. Efficient algorithms for these operations are given in [8]. A good implementation of clause trees using brts should be no less efficient than an ordinary clause-based resolution system which maintains its proofs, if the clause tree ideas are turned off.

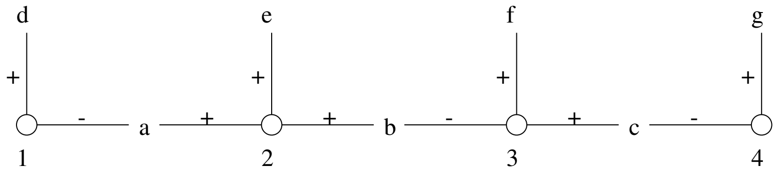


Figure 3: A clause tree rank/activity example

2.4 The rank/activity restriction

The goal of the rank/activity restriction is to be able to construct each clause tree exactly once. In the rank/activity restriction, every literal in a clause is either active or inactive. Only pairs of active literals may be resolved. In the child clause generated by a resolution, inactivity is inherited from the parent literal. However if two literals are merged or factored, then the resulting literal becomes active, regardless of whether the parent literals are active or inactive.

All active literals in a clause are also given a rank. When a binary resolution is performed, in the child clause any literal which has a higher rank in the parent clause than the resolved literal becomes inactive, while lower ranked literals remain active. Thus flexibility is retained if highly ranked literals are resolved.

The rank/activity restriction is complete [2], and completeness is retained in conjunction with many other restrictions of resolution [3]. Moreover, every clause tree is built exactly once using the rank/activity restriction. A clause tree with n resolutions on distinct atoms and with m merges is equivalent to at least $2^{2n-\Theta(m\log(n/m))}$ different binary resolution trees [4]. Moreover the number of equivalent binary resolution trees can be as large as $n!/(m+1)$.

As an example consider the clause tree in Figure 3 based on the clauses $\bar{a}d, abe, \bar{b}cf, \bar{c}g$. Consider the following proofs of the clause $defg$.

1	$\bar{a}d$	input clause								
2	abe	input clause								
3	$\bar{b}cf$	input clause								
4	$\bar{c}g$	input clause								
5	bde	$1 *_a 2$	bde	$1 *_a 2$	$acef$	$2 *_b 3$	$acef$	$2 *_b 3$	$\bar{b}fg$	$3 *_c 4$
6	$cdef$	$3 *_b 5$	$\bar{b}fg$	$3 *_c 4$	$cdef$	$1 *_a 5$	$aefg$	$4 *_c 5$	$aefg$	$2 *_b 5$
7	$defg$	$4 *_c 6$	$defg$	$5 *_b 6$	$defg$	$4 *_c 6$	$defg$	$1 *_a 6$	$defg$	$1 *_a 6$

In most purely binary resolution procedures, all five proofs would be generated, unless the procedure stops. Not all fifteen steps would be performed, because three of them are duplicated: clauses $bde, \bar{b}fg, acef$ each occur in two of the proofs. Two other intermediate clauses are repeated in two proofs: $cdef, aefg$. One copy of each would presumably be subsumed away, and the

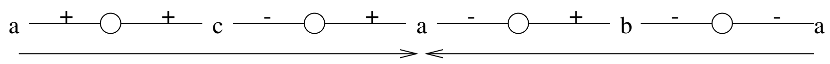


Figure 4: A clause tree rank/activity example with merges

final steps of these two proofs would be duplicates. This leaves 10 resolutions and four subsumptions to be performed. A rank/activity restricted procedure only does six resolutions and no subsumptions are needed, assuming that the d, e, f, g literals are of lower rank than the other literals in the clauses. For example, if the literals are ranked always in alphabetical order, a with the highest rank, then only the following resolutions are done:

5	bde	$1 *_a 2$
6	a^*cef	$2 *_b 3$
7	b^*fg	$3 *_c 4$
8	$cdef$	$3 *_b 5$
9	a^*efg	$4 *_c 6$
10	$efgh$	$4 *_c 8$

The starred literals are inactive. Different rank/activity procedures could do them in different orders but only these resolutions could be performed with these clauses alone, with this rank function.

It is very important that inactive literals which are merged be reactivated. Consider the clause tree in Figure 4. It is based on the clauses $ac, a\bar{c}, \bar{a}b, \bar{a}\bar{b}$. Suppose that the rank priority is alphabetical ordering. When the b and $\bar{b}$ are resolved, the $\bar{a}$'s are deactivated; similarly the a's are de-activated when the c and $\bar{c}$ are resolved. If they were not re-activated when they were merged, this clause tree could not be built. But every clause tree is supposed to be built by the rank/activity procedure. The point is that a resolution of a literal which is to be eventually merged must be delayed until after other resolutions are done. Hence such a literal can be allowed to be de-activated before it is merged, and re-activated when it is merged.

3 The overall procedure used by Marmoset

Marmoset maintains a set of retained clauses, and a queue of clauses to be processed, which starts with the input clauses. At each step, the clauses in the queue are processed. A clause is removed from the queue and is either retained, or rejected if the appropriate option is set. A clause can be rejected if it:

1. is subsumed by a retained clause (forward subsumption);
2. can have clause tree surgery applied (non-minimality);

3. has no active literal (inactivity); or

4. exceeds some size criterion, like depth of term.

When a clause is retained, its most general factors are generated and added to the queue, and its literals are inserted into a discrimination tree for indexing [5]. The clause is then used for back subsumption. The active literals in the retained clauses are inserted into a priority queue, the priority being assigned by a user defined function.

This repeats until the queue of clauses is empty or the empty clause is found. If the queue is empty, then a literal with highest priority is chosen and resolved with all other retained complementary literals. The new resolvents are added to the queue of clauses, and the process repeats.

All literals of retained clauses are stored in the discrimination tree, but each node has two different lists. The active literals are stored in one list, and can be found when looking for a literal to be resolved. The discrimination tree is also used for subsumption; in this case inactive literals must also be considered.

Marmoset is a literal-based reasoning system, as opposed to clause-based. The search control is based on the evaluation of literals rather than the evaluation of clauses, so that some literals of a clause may have been resolved while others have not. This structure allows several different types of procedures to be implemented. A clause-based reasoning system is produced by making the score of every literal in a clause the same. An a-ordered system can be produced by inserting only literals of the highest priority in a clause into the priority queue. The rank/activity restriction can perhaps be improved by giving literals of higher priority lower scores, and hence decreasing the number of inactive literals produced, but this heuristic is not tested yet.

4 The experiments and results

As Marmoset does not yet have any special method of dealing with equality, and has no advantage in dealing with Horn clauses, the problem set on which it was tested consisted of the set of non-Horn, non-equality problems in CNF form in the TPTP problem library, version 2.3.0 [10]. This includes 765 problems. In each run, each problem was given 5 minutes on an Alphaserver DS10 6/600 [1].

The weight of a literal was set equal to the number of predicate and function symbols in the literal, excluding non-duplicate variables, plus the maximal term depth. Thus $p(X)$ would have weight 1, with one predicate symbol, no duplicate literals, and 0 term depth, while $q(X, f(X))$ has score 4, with 1 each for q, f and the duplicate X, and one for term depth. The score of a clause was the sum of the weights of all literals in the clause, plus the number of resolutions required to produce the clause. No restrictions on the size of a clause or depth

[1] A 616Mhz 21264 processor with 256MB of memory, running Linux. The processors were in a Beowulf cluster in the Laboratory for the Investigation of Discrete Structures at UNB, which is supported by the Canadian Foundation for Innovation.

RUN	Unsatisfiable	Satisfiable	Total
CLAUSE	212	9	221
LITERAL	205	9	214
CLAUSE_AO	217	13	230
LITERAL_AO	207	14	221
CLAUSE_RA	220	9	229
LITERAL_RA	227	9	236
LITERAL_RAO	228	9	237
LITERAL_RAO_REV	225	9	234
CLAUSE_AO_REV	205	16i	221

Table 1: Number of solved problems by run

of terms were introduced. Only subsumption by unit clauses was included. The set of support restriction was used. Minimality was not used.

The first question considered was whether the literal-based version would be different from a clause-based algorithm. In the first run [CLAUSE], all literals in a clause were given the same score, the score for each literal was just the score of the clause as defined above. In the second run [LITERAL], the weight of the literal resolved was subtracted from the score of the clause, to force heavier literals in a clause to be resolved earlier.

The next question investigated was the rank/activity restriction [RA], and how it compares to a-ordered resolution [AO]. In a-ordered resolution, all atoms are ordered by a liftable ordering, and only the literal(s) with the highest priority in a clause is(are) allowed to resolve. Literals were ordered alphabetically up to the first variable. Four more runs were done, with both AO and RA using both CLAUSE and LITERAL. The ranking function for RA was the order of the literals in the clause. The number of solved problems are included in Table 1.

We investigated the impact of changing the order used in the rank/activity and a-ordering runs. The original r/a run did not use the same order as a-ordering, but used the order in which the clauses were stored. To make a better comparison we re-ran r/a with the same order as the a-ordering. Then we ran both r/a and a-ordering with the reverse ordering.

Table 2 gives the geometric average of the ratio of the times between each problem which is solved by both of the runs. The smaller the number, the better the procedure on the line, and the worse is the procedure at the head of the column. If either run failed to solve the problem, or if both solved it in less than 1 second, the problem is thrown out. The last number is the geometric mean of the times taken to solve 139 problems. These problems were solved by at least one of the procedures, yet took more than one second for one of them.

The number of problems solved within a given time is displayed all of the runs in Figure 5. The second diagram. Figure 6, compares rank/activity with a-ordering.

Marmoset averaged about 2000 inferences per second on solved problems,

Run	C	L	CAO	LAO	CRA	LRA	LRO	LROR	CLAR	Mean
CLAUSE	1.00	0.95	2.03	1.14	1.90	1.72	1.37	1.33	1.80	15.9
LITERAL	1.05	1.00	2.03	1.02	2.00	2.15	1.56	1.51	1.70	15.7
C_AO	0.49	0.49	1.00	0.44	0.84	0.89	0.73	0.69	0.84	8.8
L_AO	0.88	0.98	2.25	1.00	1.48	1.83	1.49	1.40	1.53	14.2
C_RA	0.53	0.50	1.19	0.68	1.00	0.92	0.80	0.75	0.95	10.4
L_RA	0.58	0.47	1.13	0.55	1.08	1.00	0.77	0.71	1.00	10.3
L_RAO	0.73	0.64	1.37	0.67	1.26	1.31	1.00	0.93	1.22	12.1
L_RAO_R	0.75	0.66	1.45	0.71	1.34	1.41	1.07	1.00	1.24	12.7
C_AO_R	0.55	0.59	1.18	0.65	1.05	1.00	0.82	0.81	1.00	9.8

Table 2: Average Ratio of Times between Pairs of Runs

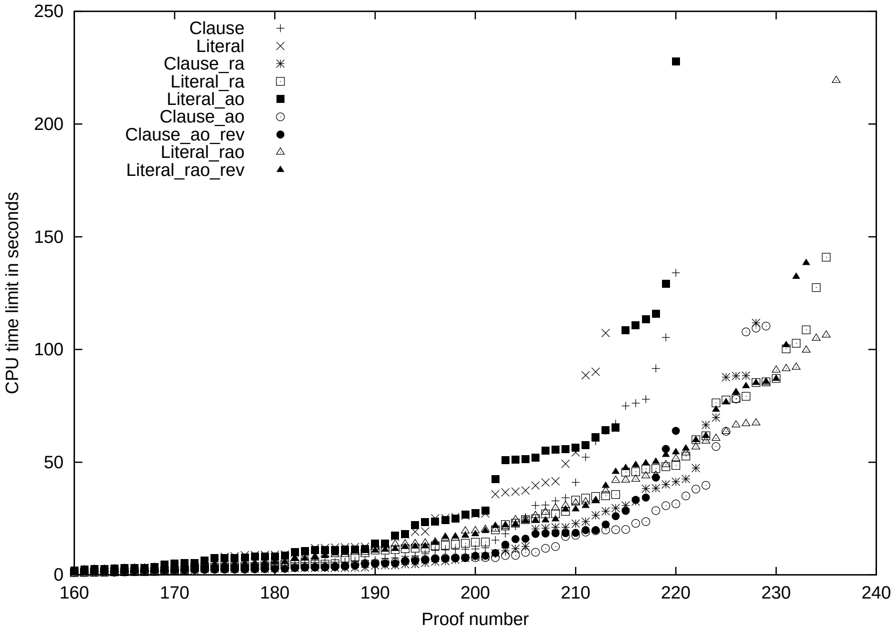


Figure 5: Problems solved under a given time — all runs

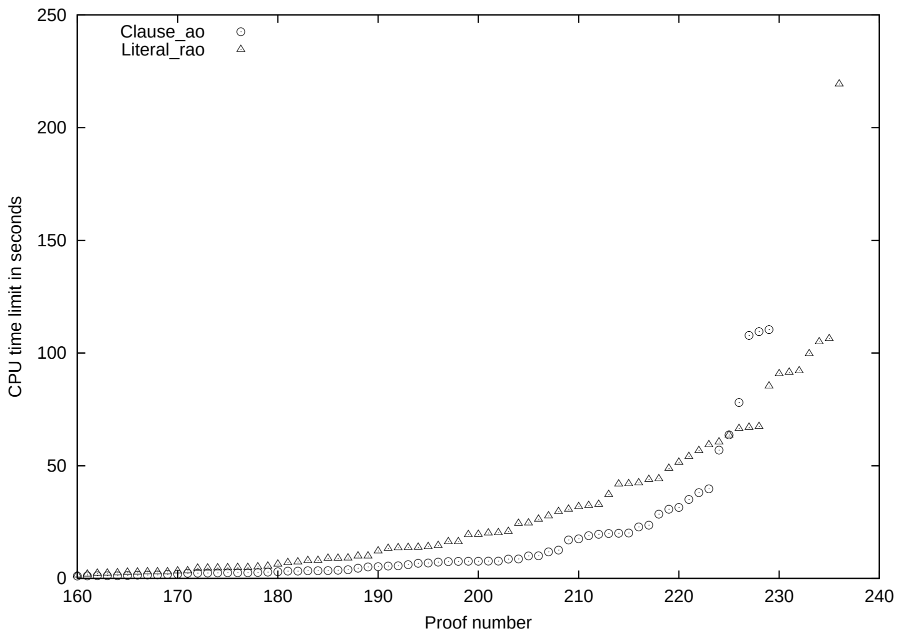


Figure 6: Problems solved under a given time — RA vs AO (same order)

but by five minutes the inference rate fell below 1000. The inference rates were similar for each of the procedures.

5 Conclusions

We were surprised that the literal-based runs did not solve as many problems as the clause-based runs, for both the basic and a-ordered runs. Our intuition that larger, and hence more restrictive, literals should be resolved earlier is apparently wrong in general. However for the rank/activity runs, the literal-based run did better than the clause-based run, as expected. We believe that with some heuristics, the literal-based procedures will do better than clause-based procedures.

The rank/activity restriction did better than the a-ordering in terms of the number of problems solved. But a-ordering did better in terms of the average time to solve a problem, and did better than no ordering at all. As time increases, we expect that rank/activity should improve relatively, which shows in both Figure 5 and Figure 6. CLAUSE_AO solved more problems up to about a minute, but by a minute and a half, LITERAL_RA was solving more problems.

The difference between rank/activity and a-ordering was minimal when rank/activity used the same order as the a-ordering originally used. But when the order was reversed, the order was worse for both, especially a-ordering. This shows that both procedures are sensitive to the ordering, as one would expect. It is known that a-ordering can be exponentially slower depending on the order of the variables chosen [9]. Rank/activity should not not be as sensitive as a-ordering to changes in the ordering of the literals, because it does allow resolutions to be done in other orders.

Marmoset is not a very fast theorem prover, but it is a useful tool to investigate some ideas for new calculi developed using clause trees. In addition to rank/activity, we will soon be testing clause tree surgery, and surgical minimality.

References

[1] J. D. Horton and B. Spencer. Clause trees: a tool for understanding and implementing resolution in automated reasoning. *Artificial Intelligence*, 92:25–89, 1997.

[2] J. D. Horton and B. Spencer. Rank/activity: a canonical form for binary resolution. In C. Kirchner and H. Kirchner, editors, *Automated Deduction — CADE-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 412–426, Lindau, Germany, July 1998. Springer.

[3] J. D. Horton and Bruce Spencer. Combining rank/activity with set of support, hyperresolution and subsumption. Technical Report TR99-125,

Faculty of Computer Science, University of New Brunswick, Fredericton, 1999.

[4] Joseph D. Horton. Counting the number of equivalent binary resolution proofs. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR 2001*, number 2250 in LNAI, Havana, Cuba, December 2001. Springer-Verlag.

[5] W. McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9:147–167, 1992.

[6] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[7] Bruce Spencer. Avoiding duplicate proofs with the foothold refinement. *Annals of Mathematics and Artificial Intelligence*, 12:117–140, 1994.

[8] Bruce Spencer and J. D. Horton. Efficient algorithms to detect and restore minimality, an extension of the regular restriction of resolution. *Journal of Automated Reasoning*, 25:1–34, 2000.

[9] Bruce Spencer and J.D. Horton. Support ordered resolution. In D. McAllester, editor, *Automated Deduction — CADE-15*, number 1831 in Lecture Notes in Artificial Intelligence, pages 385–400, Pittsburgh, PA, USA, June 2000. Springer.

[10] G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In D. Kapur, editor, *Automated Deduction CADE-12*, number 814 in Lecture Notes in Artificial Intelligence, pages 252–266. Springer-Verlag, Berlin, 1994.

[11] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics*, Seminars in Mathematics: Matematicheskii Institute, pages 115–125. Consultants Bureau, 1969.

The Next WALDMEISTER Loop (Extended Abstract)

Thomas Hillenbrand[1] and Bernd Löchner[2]

[1]Max-Planck-Institut für Informatik, Saarbrücken, Germany,
hillen@mpi-sb.mpg.de

[2]FB Informatik, Universität Kaiserslautern, Kaiserslautern, Germany,
loechner@informatik.uni-kl.de

1 Introduction

Very recently the question of how to organize the proof search within saturation-based theorem provers has attracted renewed attention [Vor01]: Two variants of the given-clause algorithm are under discussion, now being termed OTTER loop resp. DISCOUNT loop. For each of them, competitive systems have been built, demonstrating that both approaches meet the state of the art. But it is unclear which path to follow in the future in order to "increase performance of the modern provers by several orders of magnitude" [loc. cit.].

With this paper, we want to further the discussion of this question within the frame of the implementation workshop. We present our understanding of the DISCOUNT loop as evolved during the development of the WALDMEISTER prover for unit equational deduction [HJL99]. In our view, the benefits of this loop variant are threefold: (i) only active facts have to be indexed, (ii) passive facts may be stored in a compressed, space-saving representation, and therefore (iii) completeness can be sustained much longer. This contrasts with the need for weight limits or eligibility estimations in the case of the OTTER loop (cf. [RV99]).

In this contribution we focus on the second issue. We refine our previous conceptions towards a clarified system design that allows even more compression and thereby ends the need to sacrifice completeness after several hours of running time. Moreover, any overhead for the proof construction is removed from the proof search. It should even become possible to depart from the given-clause algorithm, to simulate a given-pair algorithm [McC97], and especially to develop hybrid versions, which to our knowledge has not been studied so far. The implementation which is currently under construction will give rise to further insights.

Algorithm 1 The proof procedure of WALDMEISTER

FUNCTION WALDMEISTER($\mathcal{E}, \mathcal{H}, >, \varphi$) : BOOLEAN
1: $(\mathcal{A}, \mathcal{P}) := (\varnothing, \mathcal{E})$
2: **WHILE** $\neg$trivial($\mathcal{H}$) $\wedge\ \mathcal{P} \neq \varnothing$ **DO**
3: $\quad e := \arg\min \varphi(\mathcal{P});\ \mathcal{P} := \mathcal{P} \backslash \{e\}$
4: $\quad$ **IF** $\neg$orphan(e) **THEN**
5: $\qquad e := \text{Normalize}^{>}_{\mathcal{A}}(e)$
6: $\qquad$ **IF** $\neg$redundant(e) **THEN**
7: $\qquad\quad (\mathcal{A}, P_1) := \text{Interred}^{>}(\mathcal{A}, e)$
8: $\qquad\quad \mathcal{A} := \mathcal{A} \cup \{\text{Orient}^{>}(e)\}$
9: $\qquad\quad P_2 := \text{CP}^{>}(e, \mathcal{A})$
10: $\qquad\quad \mathcal{P} := \mathcal{P} \cup \text{Normalize}^{>}_{\mathcal{A}}(P_1 \cup P_2)$
11: $\qquad\quad \mathcal{H} := \text{Normalize}^{>}_{\mathcal{A}}(\mathcal{H})$
12: $\qquad$ **END**
13: $\quad$ **END**
14: **END**
15: **RETURN** trivial($\mathcal{H}$)

2 An inspection of Waldmeister's proof procedure

2.1 The proof procedure itself

The input to the procedure (cf. Alg. 1) consists of an axiom set $\mathcal{E}$, a set of hypotheses $\mathcal{H}$, a reduction ordering $>$, and a weighting function φ for heuristical evaluation. The saturation is performed in a cycle working on a set $\mathcal{A}$ of *active facts* which participate in inferences and a set $\mathcal{P}$ of *passive facts* waiting to become members of $\mathcal{A}$. We denote, slightly extending the imperative notation, by $\mathcal{A}_i$ the value of $\mathcal{A}$ after the i-th iteration of the loop core in lines 7–11; this iteration count gives a notion of abstract time. The sets $\mathcal{A}_i$ induce the rewrite relations $\rightarrow_i\ :=\ \rightarrow_{\mathcal{A}_i}\ \subset\ >$. The weighting function φ, e.g. $\varphi(s{=}t) = |s| + |t|$, has to ensure that every passive fact becomes minimal at some point in time. This guarantees the fairness of the proof procedure.

(L. 1) Initially, $\mathcal{A}$ is empty and $\mathcal{P}$ contains the input axiomatization. (L. 2) The saturation proceeds as long as the hypotheses have not become trivial and there are still passive facts to be selected. Inside the completion loop, the following steps are performed: (L. 3) Select an equation e with minimal weight from the set $\mathcal{P}$. (L. $4, 5$) Unless a parent equation has been reduced, simplify e to a normal form. (L. $6, 7$) Interreduce the set $\mathcal{A}$ with e, i. e. take active facts that are reducible by e out of $\mathcal{A}$. (L. 8) Orient e if possible, and add it to $\mathcal{A}$. (L. 9) Generate all new critical pairs. (L. 10) Normalize them and the reduced active facts, and insert everything into $\mathcal{P}$. (L. 11) Normalize the hypotheses.

As can be seen in the proof procedure, the passive facts are subject to normalization only right after their generation and in case they get selected. In contrast, within an OTTER loop the whole set of passive facts is normalized in every iteration of the cycle. Line 10 then would read "$\mathcal{P} :=$ Normalize$^{>}_{\mathcal{A}}(\mathcal{P} \cup P_1 \cup P_2)$". Some implementations moreover use the rewrite relation generated by the union of active and passive facts, i.e. they always employ Normalize$^{>}_{\mathcal{A}\cup\mathcal{P}}$ instead of Normalize$^{>}_{\mathcal{A}}$. In that case the orphan predicate (L. 4) must always be evaluated to FALSE, cf. Sect. 2.2.

2.2 Aspects of space consumption

Typically $\mathcal{P}$ contains far more equations than $\mathcal{A}$, as a rule of thumb $|\mathcal{P}| = O(|\mathcal{A}|^2)$ on the average. Therefore the space requirements of WALDMEISTER are dominated by the representation of the passive facts. Hence, methods to save space mainly have to tackle $\mathcal{P}$. One approach is to minimize the number of elements, another, independent one, is to minimize the space needed for one element.

As to the first, redundancy criteria are under demand. But it is clear that such redundancy checks must not be too expensive, since otherwise the DISCOUNT loop would fall back into an OTTER loop again.

A cheaper and more efficient check is a special feature of the DISCOUNT loop, named *"orphan" criterion*: A critical pair can be deleted if one of its "parent" active facts has been interreduced, for the critical pair has not participated in any inference. This would not be possible if passive facts were used for simplification purposes.

The second approach, namely reducing the space of a single passive fact, means to employ simple *compression schemes*. Within WALDMEISTER, this can be done in three ways: (i) Not at all, i.e. flatterms are employed at the cost of 12 bytes per symbol. (ii) On the level of terms: The passiveness of facts in the DISCOUNT loop entails that they, after generation, are not touched again until they get selected, which in theorem proving is the case for a small fraction only. It is therefore possible to store terms simply as strings of symbols, at the cost of one byte per symbol. (iii) Overlap representation: After heuristical evaluation, the terms themselves are thrown away, and just minimal reconstruction information is kept, at constant cost.

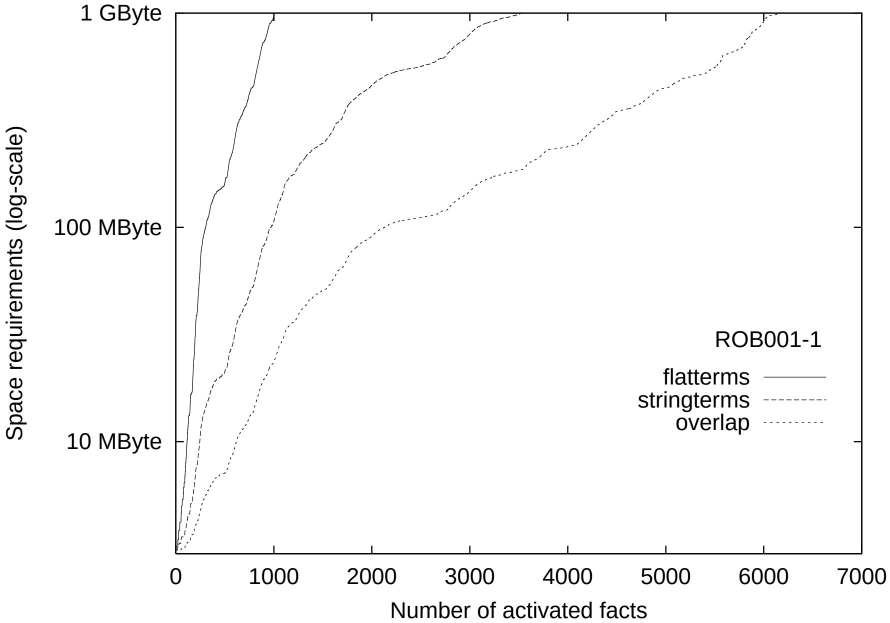
The influence of these representations on the process size over abstract time can be studied in Fig. 1. For each variant, the WALDMEISTER system was run on two large proof tasks from the TPTP problem library on a UltraSPARC-IIi workstation. Stringterms allow compression rates of about 10, which is comparable to what can be achieved with a shared term representation [LS01]. The overlap representation improves on this by another factor of nearly 10. This is quite typical for long-lasting runs.

Nevertheless, the space consumption becomes prohibitive after the activation of several thousand facts (cf. Fig. 1). After some hours we are currently faced with the decision to give up or to throw away critical pairs and to sacrifice completeness. In the sequel we will show how to escape this dilemma.

2.3 A drawback and its remedy

The above-mentioned overlap representation is apparently the best one in terms of space. We have employed it when memory was tight, but it has up to now not become our method of choice. The reason for this is that the proof search is influenced: When a critical pair e is generated at time i, it is normalized to some $e{\downarrow_i}$ with respect to the then current rewrite relation $\rightarrow_i$. Let e be selected at time $j \gg i$. The reconstruction process then yields $e{\downarrow_j}$, whereas with the two other representations the critical pair would enter the completion loop in the shape $(e{\downarrow_i}){\downarrow_j}$. Since the rewrite relation changes during completion, these shapes may differ. According to our observations, the effect on the proof search is frequently negative.

To overcome this obstacle, that is, to make the reconstruction perfect, one has to remember history: All versions of the rewrite relation $\rightarrow_i$ and the time the critical pair at hand was constructed and normalized are recorded. This is by far not as costly as it sounds: It is sufficient to annotate the elements of $\mathcal{A}$ with their *activation* time, and their *deactivation* time if necessary, i. e. when they get reduced by a new equation. Along with each set $\mathcal{A}_i$, the order in which its elements are traversed when searching for rewrite patterns must be reproducable as well so that still the same matching rewrite rules are delivered. Then the normalization steps can be perfectly reproduced; the overlap representation becomes neutral w.r.t. the search



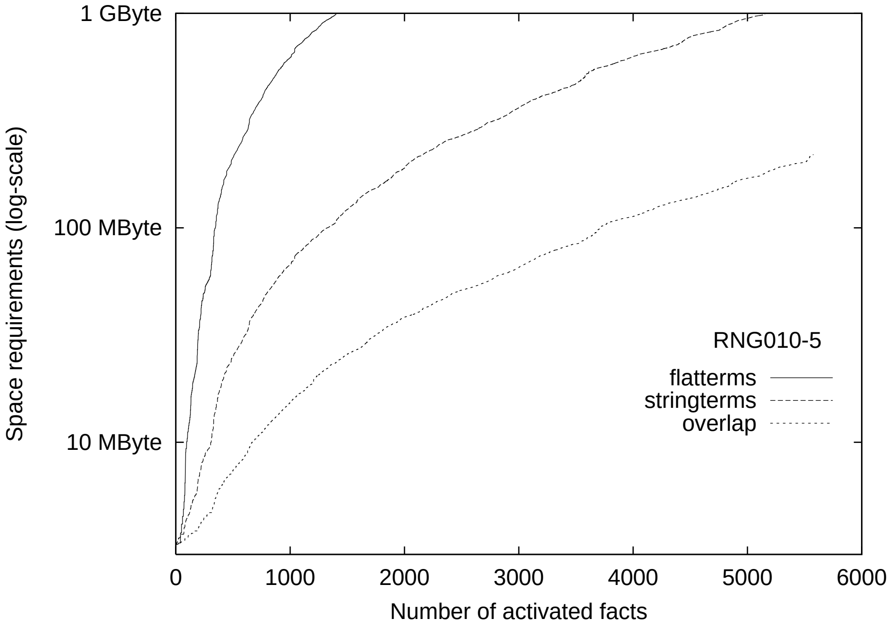


Figure 1: Size of the process over abstract time

behavior.

As a consequence, this changes our view of the rôle of $\mathcal{P}$ in the proof procedure: Abstractly said, $\mathcal{P}$ has to enumerate in a fair way all critical pairs that are not known to be redundant. This sheds a new light on the normalization of freshly generated critical pairs: Its purpose is, besides testing redundancy, only to make a well-behaved heuristical evaluation possible; it is decoupled from the "real" inferences which occur merely at the point of selection; and only "good" critical pairs ever enter the completion loop. Going further, the one-to-one relationship between the elements of $\mathcal{P}$ and the equations they represent is no longer necessary: An item in $\mathcal{P}$ could e. g. denote the set of all critical pairs between two equations e_i and e_j, or between an equation e_i and all members of $\mathcal{A}$ (cf. Sect. 3.1).

2.4 On generating proof objects

A further issue is the provision of a proof object. The first approach realized in WALDMEISTER was to protocol each inference step in an external file. This led to a considerable slowdown of the proof search because a proof task of say 5 seconds could produce a proof log of nearly 100 MBytes.

To overcome this unacceptable situation, we next changed to recording internally for every inference step the minimal information necessary to reproduce it. When a proof is found, a dependency analysis reveals which steps actually contribute to the proof. These inferences are then re-performed step by step to generate the proof object. All in all, one gets proof objects with a run-time penalty of about 5 %.

But this approach has another, still severe drawback: Even with the minimized protocol information the space requirements of the internal proof log can become a limiting factor, for all the reductions on newly generated critical pairs have to be protocolled. Remembering the history of the rewrite relation, however, can drastically change the picture: During proof search, we just keep for every active fact the overlap information it descends from. Within the DISCOUNT loop, only members of $\mathcal{A}$ contribute to a proof. Hence, after the proof is found, the subset $\mathcal{A}^p$ of $\mathcal{A}$ actually needed for the proof is determined in a backward-oriented dependency analysis. (Experience shows that $|\mathcal{A}^p| \ll |\mathcal{A}|$.) In a second, forward-oriented phase all inferences needed to construct the elements of $\mathcal{A}^p$ are performed while logging of the inference steps is enabled. As we can see, the complete inference protocol of the proof search is no longer needed.

3 Representational issues

3.1 The set of passive facts

A simple, but seldomly exploited fact is that a position within a term or a clause can be encoded in a single integer, e.g. as number of the top symbol of $s|_p$ when the symbols in s are counted from left to right. This is straightforward especially when flatterms are employed.

Given two parent equations $s[l']_p{=}t$ and $l{=}r$, the critical pair at the overlap position p is uniquely determined as $\sigma(s[r]_p{=}t)$ where $\sigma = \mathrm{mgu}(l, l')$. The parents e_i and e_j are represented by their numbers i and j, and the critical pair is encoded as $\langle i, j, xp\rangle$ where xp denotes an extended position to clarify on which side to overlap. Instead of distinguishing "into" equation and "from" equation by the argument position in this tuple, we encode this information into the extended position as well and then may require such an arrangement that $i > j$, for the following reasons: Assuming a 32-bit address space, the tuple can with some bit-twiddling be represented in one 64-bit integer. If then such tuples are compared as integers, the result coincides with that of a comparison of the critical pairs by age. Note that many systems employ explicit age information in the facts for that purpose.

Introducing wildcards, tuples can represent sets of critical pairs, e.g. $\langle i, j, *\rangle$ for all the overlaps between the equations e_i and e_j, or $\langle i, *, *\rangle$ for everything generated at the activation of e_i. Equations stemming from interreduction can be encoded in a similar way.

Since $\mathcal{P}$ is ordered according to φ these items are augmented by a weight w. Whenever the number of elements in $\mathcal{P}$ reaches a certain limit, the following compression scheme is applied: Some or all tuples $\langle w_1,\ i, j, xp_1\rangle$ $\ldots$ $\langle w_n,\ i, j, xp_n\rangle$ are combined into $\langle w',\ i, j, *\rangle$ where $w' = \min_{1\leq k\leq n} w_k$. If this is not sufficient, the tuples $\langle w_k,\ i, j, *\rangle$, $1 \leq k \leq n$, can be packed into $\langle \min_{1\leq k\leq n} w_k,\ i, *, *\rangle$.

This should allow to represent most sets $\mathcal{P}$ that occur in practice within a reasonable amount of memory: In the extreme, $\mathcal{P}$ just consists of an initial segment of individual critical pairs and then is linearly bound by the size of $\mathcal{A}$, i.e., $|\mathcal{P}| = O(|\mathcal{A}|)$ even in the worst case. Whenever such a wildcard tuple is selected by the main loop, it gets expanded, i.e., the represented critical pairs are recomputed. The whole approach therefore trades time for space; but note that in general only the light-weighted tuples ever need to be expanded again.

Finally, we also get some new flexibility: The wildcards in the tuples allow to depart from the pure given-clause algorithm. Instead of generating and normalizing all new critical pairs, the corresponding tuples $\langle i, j, * \rangle$ can directly be inserted into $\mathcal{P}$. With some appropriate setting of the initial weights for these tuples it is possible to simulate a given-pair algorithm [McC97] or to develop hybrid versions.

3.2 The set of active facts and its accumulated history

To record all the rewrite relations $\rightarrow_j$, $j \leq i$, it is sufficient to store the elements e_j as they are. The problem is that for a given relation $\rightarrow_j$ and a redex t several equations may match. Which of them is chosen for reduction is determined by the traversal through the indexing structure. It is of course not feasible to save for every $j \leq i$ the index for $\mathcal{A}_j$. We would like to use only a single index for the accumulated history; how can this be achieved?

At this point the perfect discrimination trees employed in WALDMEISTER reveal a beautiful property: The ordering relation of two rewrite patterns being traversed is invariant under insertion and deletion of other elements. Hence, traversing an index over $\{e_j \mid j \leq i\}$ with some filtering to determine the desired $\rightarrow_k$ leads to the same traversal order of matching equations as in the index for $\mathcal{A}_k$. The reason for this property is that the trees are ordered ones, i. e. the order of the successors of a node is fixed. It is unclear to us how the same effect can be achieved with other indexing schemes.

For performance reasons we can keep two indexes: one of $\mathcal{A}_i$ for the current rewrite relation and another of $\{A_j \mid j \leq i\}$ for the accumulated history.

4 Conclusion

This work answers a question which has been on our agenda for several years: How to retain completeness in situations when even our best compression techniques hit the limitations of memory or address space? Compared with the methods mentioned in [Wei01, p. 1980f] our new approach is – in our opinion – superior: Neither do we lose completeness, nor do we have to restart the whole search. If the size of $\mathcal{P}$ is set to modest values we can even expect some improvements in the run time of the system: Since the process size is then considerably smaller, the working set becomes smaller and the cache utilization increases.

As an unexpected, but most welcome, effect we gain more conceptional clarity in the architecture of our prover. First, the initial generation and normalization of critical pairs can be interpreted to be for heuristical evaluation only. These steps are no "real" inferences. Second, the provision of a proof object can be completely decoupled from the proof search. There is no need to record each inference performed in case that inference will contribute to the proof. Third, the given-clause algorithm is no longer hard-wired. We can simulate the given-pair algorithm or explore new hybrid versions.

Finally, let us say that our approach, being fit for the unit equality prover WALDMEISTER, can straightforwardly be adapted to any resolution-based prover that implements the DISCOUNT loop variant. Since in the full clausal case the space requirements are usually more pronounced, the effects should be even more beneficial. Thus, the conclusion is the same for both cases: Even for long-running proof tasks there is no need to sacrifice completeness.

References

[HJL99] Th. Hillenbrand, A. Jaeger, and B. Löchner. System description: WALDMEISTER – Improvements in Performance and Ease of Use. In *Proc. CADE-16*, LNAI 1632, pages 232–236, 1999.

[LS01] B. Löchner and S. Schulz. An Evaluation of Shared Rewriting. 2nd International Workshop on Implementation of Logics (these proceedings), 2001.

[McC97] W. McCune. 33 basic test problems: A practical evaluation of some paramodulation strategies. In R. Veroff, editor, Automated Reasoning and its Applications: Essays in Honor of Larry Wos, chapter 5, pages 71–114. MIT Press, 1997.

[RV99] A. Riazanov and A. Voronkov. Vampire. In *Proc. CADE-16*, LNAI 1632, pages 292–296, 1999.

[Vor01] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction (invited talk). In *Proc. IJCAR*, LNAI 2083, pages 13–28, 2001.

[Wei01] Chr. Weidenbach. Combining Superposition, Sorts and Splitting. In A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, Elsevier, 2001.

Integration of Equality Reasoning into the Disconnection Calculus

Reinhold Letz & Gernot Stenz
Institut für Informatik
Technische Universität München
D-80290 Munich, Germany
{letz,stenzg}@in.tum.de

November 22, 2001

Abstract

Equality handling has always been a traditional weakness of tableau calculi. In this paper we present a tableau calculus suited for the integration of a large class of different methods for equality handling. This is demonstrated with several of those methods, both saturation based and goal-oriented ones.

1 Introduction

In the past decade automated theorem proving using tableau style calculi was developed into a number of very successful systems. However, one of the great weaknesses of tableau style theorem proving is that few methods were found for enabling tableau provers to efficiently solve problems containing equality. Approaches like theory unification [Bec98] turned out to be unfeasible in practice; and they additionally presented new completeness and even undecidability problems. A number of variants of Brand's modification method [Bra75] were developed into usable systems [MIL$^+$97] and must be considered as the most successful means of integrating equality reasoning into a tableau framework hitherto. Methods based on ordered equality handling are not compatible with the most successful refinements of tableaux like the connection conditions. In contrast to such refinements the disconnection calculus offers a more robust framework for the integration of various approaches of equality handling. We present some methods based on orderings and some unordered approaches. These approaches extend the calculus which is the basis for the DCTP prover described in [LS01], where equality was handled in the generic axiomatic way only. This paper is organized as follows. In the following section, a brief introduction into the disconnection calculus will be presented. In Section 3 we show how the paramodulation rule can be adapted to our calculus. Then, in

Section 4 we will consider a different, more goal-oriented approach to equality handling based on disagreement sets. Section 5 contains some comments on our implementations of the introduced approaches, followed by Section 6 containing example tableau proofs for the methods introduced in this paper and, finally, we present some conclusions in Section 7.

2 The Disconnection Tableau Calculus

The disconnection tableau calculus was first developed in [Bil96]. Essentially, this proof system can be viewed as an integration of Plaisted's *clause linking* method [PL92] into a tableau control structure.

The original clause linking method works by iteratively producing instances of the input clauses, which are occasionally tested for unsatisfiability by a separate propositional decision procedure. The use of a tableau as a control structure has two advantages. On the one hand, the tableau format restricts the number of clause linking steps that may be performed. On the other hand, the tableau method provides a propositional decision procedure for the produced clause instances, thus making a separate propositional decision procedure superfluous. For the description of the proof method, we use the standard terminology for clausal tableaux. The disconnection tableau calculus consists of a single complex inference rule, the so-called *linking rule.*

> **Linking rule**. Given a tableau branch B containing two literals K and L in tableau clauses c and d, respectively, if there exists a unifier σ for the complement of K and a variable-renamed variant L, then successively expand the branch with renamings of the two clauses $c\sigma$ and $d\sigma$ as illustrated in Figure 1.

In other terms, we perform a clause linking step and attach the coupled instantiated clauses at the end of the current tableau branch. Afterwards, the respective connection cannot be used any more on the branches expanding B, which explains the naming "disconnection" tableau calculus for the proof method. Additionally, in order to be able to start the tableau construction, one must choose an arbitrary *initial active path* through all the input clauses, from which the initial connections can be selected. This initial active path has to be used as a common initial segment of all proper tableau branches considered later on.

As branch closure condition we use the same notion as employed in the clause linking method. That is, a branch of a tableau is $\forall$-*closed* if it contains two literals K and L such that $K\sigma$ is the complement of $L\sigma$ where σ is a substitution mapping all variables in the tableau to a new constant.

As usual in the development of theorem provers, implementing a simple calculus in its pure form will not result in a competitive system [LS01]. In order to improve the performance of the system, we have integrated a number of refinements, which preserve completeness and increase the performance of the disconnection tableau calculus tremendously. These refinements include

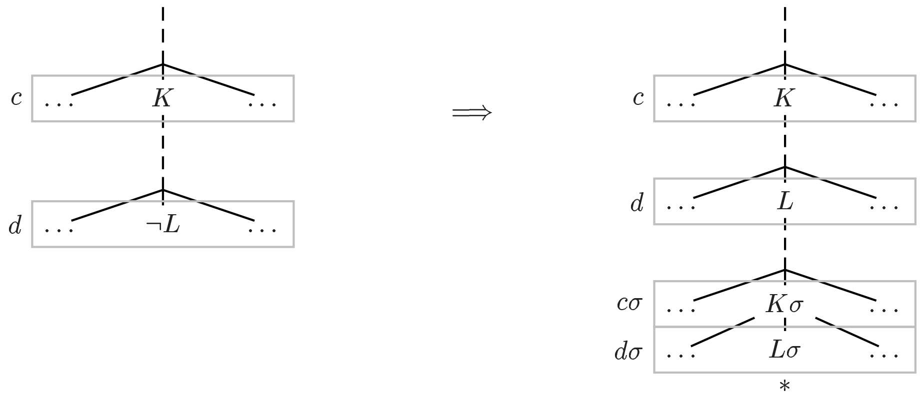


Figure 1: Illustration of a linking step.

different variations of subsumption, a strong unit theory element and several deletion strategies for redundancy elimination.

3 Paramodulation

Apart from using the equality axioms added to the proof problem, paramodulation is perhaps the most traditional and conservative means of handling equality. It also is the basic inference rule underlying the successful superposition calculus [BG98]. We refer the reader to [Lov78, RW69] for a description of paramodulation in the resolution environment.

The simplest form of paramodulation is unordered paramodulation, where overlapping is allowed in an unrestricted manner with all sides of all equations into all terms. Adaptation of this inference rule to the disconnection calculus leads to the *eq-linking rule* as it was introduced in [Bil96].

> **Eq-linking rule**. Given a tableau branch B containing an equation $s \approx t$ and a literal L in tableau clauses c and d, respectively, if there exists a unifier σ for s and a subterm position $L_{|p}$ in L, then successively expand the branch with renamings of the two clauses $c\sigma$ and $d'\sigma$ where $d' = \{s \not\approx t\} \cup \{L_{|p}(t)\} \cup (d \setminus \{L\})$ as illustrated in Figure 2.

Unlike the paramodulation rule of the resolution calculus, eq-linking introduces two independent clauses, the instantiated overlapping equation and the overlapped clause. As these two clauses can be used independently of each other for further inference steps, the equality condition $s \approx t$ has to be added in negated form to the overlapped clause. This way the soundness of inferences involving the overlapped clause can be guaranteed.

Additionally the following simplification rule can be applied to the tableau clauses.

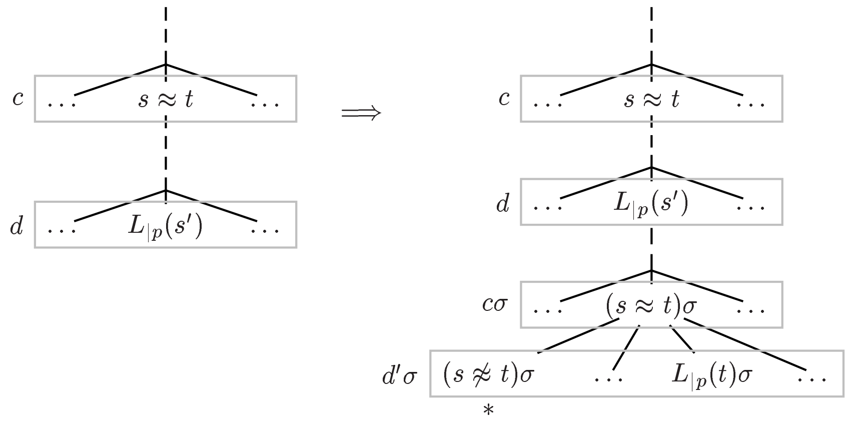


Figure 2: Tableau paramodulation by the eq-linking rule.

Forced instantiation. A clause of the form $x \not\approx t \vee R$ can be seen as $x \approx t \Rightarrow R$ and destructively rewritten to $R\{x/t\}$.

The reflexivity property of equality must be accounted for by either introducing a special closure rule for branches on which a disequation of the form $t \not\approx t$ is placed or by adding the axiom $x \approx x$ to the set of input clauses. The symmetry property of equality as far as linking steps are concerned is taken care of by the eq-linking-rule. The $\forall$-closure of branches as well as all tests for subsumption and variant deletion must explicitly take the symmetry issue into account.

Ordered Paramodulation. The unrestricted unordered form of paramodulation can lead to the generation of a multitude of redundant clauses. The introduction of a term ordering therefore is vital for the success of the method by controlling the symmetry property of equality. As usual, we do this by imposing a reduction ordering on the Herbrand universe of each problem. Overlapping is allowed with the maximal sides of equations only and, in case the overlapped literal is an equality literal, into maximal sides only.[1]

Refinements of Unordered Paramodulation. As an alternative to imposing term ordering restrictions, we also investigate the following restrictions of paramodulation. First, overlapping into general literals can be restricted to literals with an arbitrary but fixed (for each predicate symbol) sign. Also, overlapping into equality literals can be restricted to disequations, and there to an arbitrary but fixed (for each instance) side of the disequation. This feature we call *side selection*.

Rewriting. It is obvious that the huge advances in automated equational reasoning are not due to either paramodulation or orderings by themselves, the most powerful tool developed in this field is the destructive rewriting of terms. However, the application of rewriting to clauses interconnected in a tableau

[1] In case an equality literal cannot be ordered, both sides are considered maximal.

control structure is not trivial. Of course, some rewriting of terms can be easily performed in a separate preprocessing phase, before the tableau construction is started. The rewriting of tableau literals, however, poses problems in theory as well as in implementation. Currently we use a restricted form of rewriting, where each subgoal marking a tableau leaf is rewritten using all unit equations available before it is solved. This, of course, is far from optimal, as new equational lemmas that are generated over the course of the proof attempt cannot be used to rewrite literals already placed on the path. Only when those literals are chosen for a linking step, their instances can be rewritten. The enormous number of links connected to a literal and thus the corresponding search space cannot be directly reduced. This in particular concerns path equations. Therefore, a more powerful future version of tableau rewriting should perform destructive rewriting of arbitrary path literals.

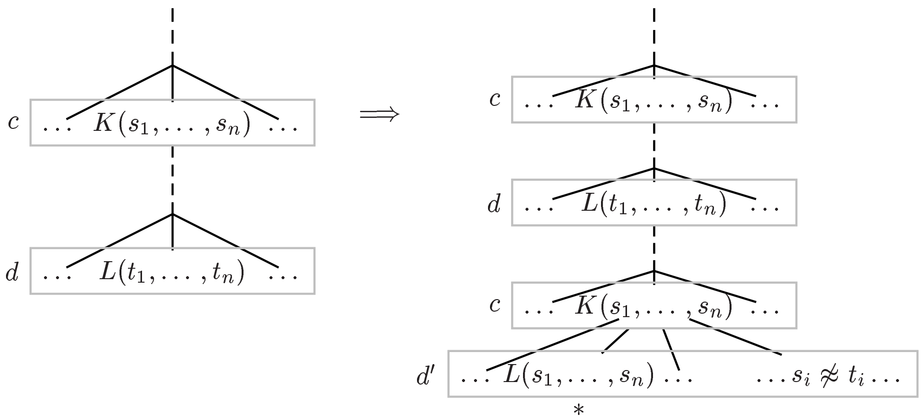


Figure 3: The disagreement linking rule.

4 Disagreement linking

One of the fundamental drawbacks of paramodulation is its inherent lack of goal-orientedness. While at least for the ordered variant the overlapping terms ideally converge to some sort of normal form, the entire procedure is based on saturation and the completion of the given equational system. In the 1980's, Digricoli developed a more goal-oriented method of dealing with equality that he called RUE (Resolution by Unification and Equality) [DH86]. The RUE approach shows some resemblance to Brand's modification method, only that Digricoli's method does not pre-saturate the clause set but performs the flattening of terms on the fly and on demand only. The method is centered around the concept of the disagreement set[2].

[2]Note that our use of this term differs from the one used in [DH86], since we uniquely associate exactly one disagreement set with each pair of terms or literals

Definition 1 (Disagreement set) *If $L(s_1,\dots,s_n)$ and $L(t_1,\dots,t_n)$, $n \geq 0$, are two terms or literals, then their* disagreement set *is the clause $\{s_1 \not\approx t_1,\dots,s_n \not\approx t_n\}$. For pairs of terms or literals with different top functors or predicate symbols or identical signs the disagreement set is not defined.*

Using the disagreement set concept, we have additional links between literals of identical signs and complementary predicate symbols, both with unifiable and non-unifiable terms. This leads us to the *disagreement linking rule*, which differs for equational and non-equational literals.

> **Disagreement linking for equality literals**. Given a tableau branch B containing a disequation K with top terms s and s' and an equation L with top terms t and t' in tableau clauses c and d, respectively, such that s and t have a disagreement set $\mathcal{D}$, let $\mathcal{D}\sigma$ be the result of applying forced instantiation to $\mathcal{D}$. Then successively expand the branch with renamings of the two clauses $c\sigma$ and $d'\sigma$, where $d' = \mathcal{D} \cup (d \setminus \{L\}) \cup \{s' \not\approx t', s \approx s'\}$.
>
> **Disagreement linking for other literals**. Given a tableau branch B containing two literals K and L in tableau clauses c and d, respectively, such that the atoms of K and L have a disagreement set $\mathcal{D}$, let $\mathcal{D}\sigma$ be the result of applying forced instantiation to $\mathcal{D}$. Then successively expand the branch with renamings of the two clauses $c\sigma$ and $d'\sigma$, where $d' = \mathcal{D} \cup (d \setminus \{L\}) \cup \{\neg K\}$ as illustrated in Figure 3.

The difference for the disagreement linking rule for equality literals is that the outermost function symbol can be stripped away when forming the disagreement sets. To account for the symmetry requirements, equations must be disagreement-linked in both directions and the same necessities apply for $\forall$-closure as for eq-linking. Also the reflexive axiom $x \approx x$ must be added to the input formula.

$$\frac{f(s_1,\dots,s_n) \not\approx f(t_1,\dots,t_n)}{s_1 \not\approx t_1 \vee \dots \vee s_n \not\approx t_n}$$

Table 1: The decomposition rule.

Finally, in order to guarantee completeness, we need the *decomposition rule* shown in Table 1, which is defined for disequations only, for other predicate symbols the decomposition is done in the course of the disagreement linking by discarding the predicate symbol itself.

Further refinements like the viability restriction [DH86] can be applied. Finally, and most importantly, the side selection restriction mentioned above can also be employed without affecting completeness. This refinement may lead to an exponential reduction of the search space wrt. the original RUE.

One fundamental disadvantage of disagreement linking is its incompatibility with any sort of orderings. On the other hand it is very restrictive in applying equalities. A practical evaluation will have to show how both approaches compare.

5 Implementation

All of the approaches presented here have been implemented as part of the DCTP prover system [LS01]. It has turned out that ordered paramodulation, in particular in combination with rewriting, is far more successful than unordered paramodulation. As this became clear already in an early stage of development, all further evaluations were restricted to ordered paramodulation. The implementation of disagreement linking turned out more complicated than expected, and no evaluation of the performance of disagreement linking could be done yet. The results of the evaluation of ordered paramodulation, however, clearly indicate the dramatic increase in performance of the prover system, both compared to the version of DCTP without special equality handling and to other state of the art systems. In the following tables, we will present the performance of DCTP and selected other theorem provers for the problems of the CASC-JC system competition held in summer 2001.

Prover	DCTP/Param.	DCTP	E-SETHEO	Otter	Bliksem
Attempted	120	120	120	120	120
Solved	35	14	93	31	29

The above table shows the results for the MIX class of the competition, which was won by E-SETHEO and VampireJC. Allotted time for each problem was 300 seconds. While still being far behind the really successful systems in this category, DCTP with paramodulation performs far better than the competition version and also better than some other well known systems. It also should be noted that DCTP was run on a single strategy only, as opposed to most other systems.

Prover	DCTP/Param.	DCTP	GandalfSat	SCOTT	MACE
Attempted	90	90	90	90	90
Solved	50	20	48	41	25

This table gives the results for the problems of the SAT category of CASC-JC. The competition version of DCTP could not yet verify the claim made in [Bil96] that disconnection as a proof method is particularly suitable for solving satisfiable problems. The new version of DCTP with ordered paramodulation in its current state outperforms the winner of this class.

It must be noted, however, that DCTP still can only solve 2 out of the 90 problems of the unit equality class of CASC-JC, as opposed to the 69 problems solved by the winning Waldmeister 601 system. The main reason for this may be that in our current system rewriting is done in a restricted way.

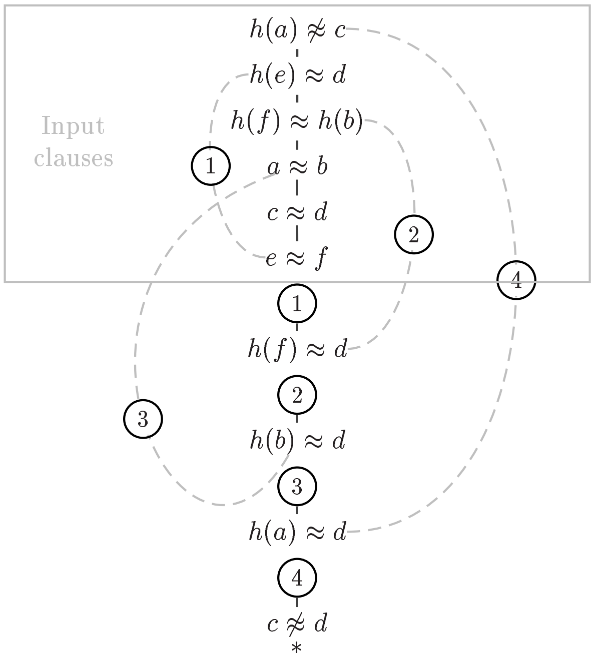
6 Examples

To demonstrate how the two approaches to equality reasoning introduced here work, let us give an example proof for the clause set $\{\{h(a) \not\approx c\}, \{h(e) \approx$

$d\}, \{h(f) \approx h(b)\}, \{a \approx b\}, \{c \approx d\}, \{e \approx f\}\}$ for each method. In each proof, the connecting lines with the encircled numbers indicate the respective links for each method, the encircled numbers in the tableau trees indicate where each link has been used for an inference step. For the sake of clarity all redundant clauses in the tableaux have been left out.

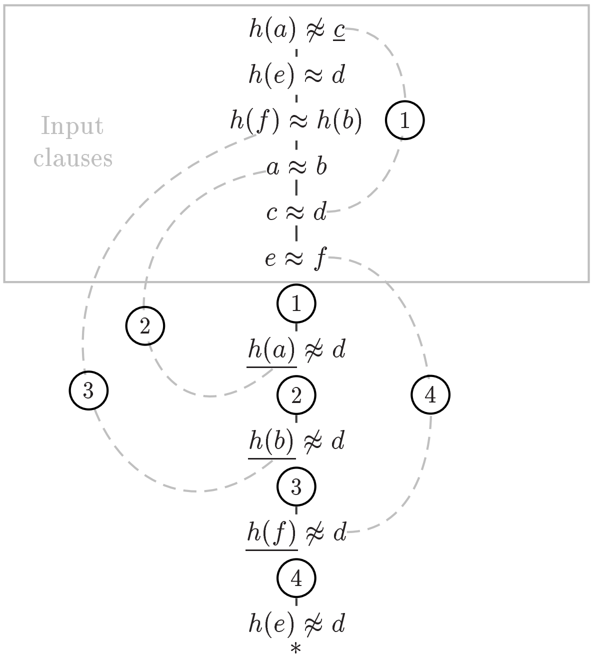
6.1 Ordered Eq-linking

In this section we present an eq-linking proof for the given clause set. The inferences are all ordered eq-linking steps. Since the negated overlapping equations would unnecessarily inflate the proof, all unit-simplifiable subgoals have been omitted for clarity.



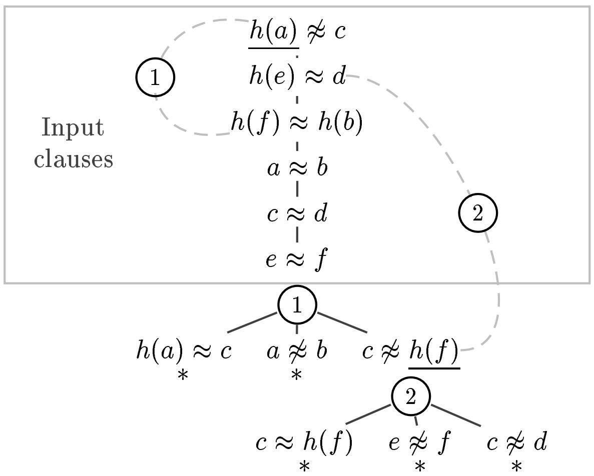
6.2 Unordered Eq-linking

Below we depict an eq-linking proof for the given clause set, this time using unordered eq-linkings only, but with the other restrictions mentioned above. The selected sides of disequations are underlined. As for the ordered case, unit-simplifiable subgoals have been omitted for clarity.



6.3 Disagreement linking

Here we show a proof for our input clause set using disagreement linking. Again, the selected sides of the disequations are underlined.



7 Conclusion

In this paper we have described different methods of integrating theory-based equality handling into a tableau framework. Due to its proof confluence, the framework of disconnection tableaux turned out to be well-suited to this task.

We conclude with some remarks concerning the soundness and completeness of the presented methods, which were not directly addressed in this paper. For all of the approaches described above, the soundness of each of the presented rules is obvious from the fact that, in every case, the new clauses are E-implied by the involved input clauses. Proving the completeness of the presented methods is more difficult and a detailed presentation of the proofs would be beyond the scope of this paper. As an example for these difficulties we could identify certain incompatibilities of the equality handling with standard pruning mechanisms such as regularity. These topics will be elaborated in detail in a future paper.

References

[Bec98] Bernhard Beckert. Rigid E-unification. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, pages 265–289. Kluwer, Dordrecht, 1998.

[BG98] Leo Bachmair 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.

[Bil96] Jean-Paul Billon. The disconnection method: a confluent integration of unification in the analytic framework. In P. Migliolo, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proceedings of the 5th International Workshop on Theorem Proving with analytic Tableaux and Related Methods (TABLEAUX)*, volume 1071 of *LNAI*, pages 110–126, Berlin, May15–17 1996. Springer.

[Bra75] Daniel Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[DH86] Vincent J. Digricoli and Malcolm C. Harrison. Equality-based binary resolution. *Journal of the ACM*, 33(2):253–289, April 1986.

[Lov78] Don W. Loveland. *Automated theorem proving: A logical basis*. North Holland, New York, 1978.

[LS01] Reinhold Letz and Gernot Stenz. DCTP: A Disconnection Calculus Theorem Prover. In Rajeev Goré, Alexander Leitsch, and Tobias

Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR-2001), Siena, Italy*, volume 2083 of *LNAI*, pages 381–385. Springer, Berlin, June 2001.

[MIL+97] Max Moser, Ortrun Ibens, Reinhold Letz, Joachim Steinbach, Christoph Goller, Johann Schumann, and Klaus Mayr. SETHEO and E-SETHEO—The CADE-13 Systems. *Journal of Automated Reasoning*, 18(2):237–246, April 1997.

[PL92] David A. Plaisted and Shie-Jue Lee. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[RW69] G. A. Robinson and L. Wos. Paramodulation and theorem proving in first-order theories with equality. *Machine Intelligence*, 4:135–150, 1969.

An Evaluation of Shared Rewriting

Bernd Löchner[1] and Stephan Schulz[2]

[1]FB Informatik, Universität Kaiserslautern, D-67663 Kaiserslautern, Germany, loechner@informatik.uni-kl.de
[2]Fakultät für Informatik, Technische Universität München, D-80290 München, Germany, schulz@informatik.tu-muenchen.de

Abstract

We present an experimental study on the use of shared rewriting in equational theorem proving. We identify the main effects that lead to term sharing in the proof state and experimentally show their influence. Besides the analysis of sharing and the *sharing factor* we gain insights on the benefits for rewriting and matching operations by a comparison of the theorem provers E and Waldmeister. This also allows us to conclude that the influence of the side effects of E's use of shared rewriting on the search process is quite minor.

1 Introduction

The thorough understanding of techniques found in current theorem provers is a prerequisite for the creation of more powerful new systems. Especially the potential of unusual approaches is often not well understood and, due to the complexity of the issue, eludes theoretical analysis.

Shared term data structures represent repeated instances of the same subterm in the proof state only once. Such representations have been in use for a number of years. However, we are not aware of any more than cursory analysis of the behavior of shared representations. In addition to the obvious benefit of reducing the memory consumption (the original motivation for shared representations), they also allow completely new techniques. Shared, non-local rewriting on shared terms, as realized in the theorem prover E [Sch01], is such a technique.

Most high-performance theorem provers use indexing techniques to replace many operations working with an element at at time with one operation working with a whole set at at time. This is typically used for matching and (less often) unification. Instead of sequentially trying one clause after the other, an index on the clause set is used to quickly determine a set of candidate clauses for the desired operation. Shared rewriting applies the *set-at-a-time* concept to a new class of operations, namely to term rewriting and normal form computation, by rewriting all instances of a given term at the same time.

In this paper we analyze the behavior of E's implementation of shared rewriting and compare it with the more conventional techniques found in Waldmeister [HJL99]. This system was chosen because both provers have been developed using experiences gained with the DISCOUNT system [DKS97] and are therefore similar in structure: Both use the DISCOUNT loop variant of the given clause algorithm, both use variants of perfect discrimination trees as an indexing structure to speed up most unit operations, and both use similar heuristics to control the proof search.

Despite this similarity the two provers are built on very different core data structures. Waldmeister has been designed around a variant of perfect discrimination trees [McC92] with the aim of maximal efficiency in terms of memory and time. Consequently, its term representation is a flat term structure [Chr93] optimized for fast traversal and memory efficiency. E, on the other hand, is built around perfectly shared terms. Whole term sets are represented by a single directed acyclic graph, and all shared instances of a term are rewritten at once. To our knowledge, E is the only prover that performs rewriting directly on a shared term representation.

Both systems are among the most powerful provers for problems formulated in unit-equational logic as demonstrated at the recent CASC-JC [Sut01, SSPar]. Among general purpose provers, E enjoys a significant lead over the competing systems, and Waldmeister, which is specialized for unit-equality problems, has an approximately similar lead over E.

This paper is structured as follows: In section 2 we introduce the necessary background. The core of the paper is formed by sections 3 and 4, which analyze the static and dynamic aspects of the shared term representation, respectively. We conclude in section 5.

2 Unfailing Completion and Equational Theorem Proving

E and Waldmeister are based on different, but related calculi. E is a superposition [BG94] prover for full clausal logic with optional selection of negative literals. Waldmeister is based on unfailing completion [BDP89], extended with narrowing to be able to deal with existentially quantified variables in the goal. Since non-ground goals are handled differently by both provers, we restrict our discussion to the case of unit-equational theories with a single unit ground goal. For this case the superposition calculus degenerates into unfailing completion.

2.1 Proof search organization and heuristics

Unfailing completion is a *saturating* calculus, i. e. it systematically enumerates consequences of the input clauses. Most current high-performance theorem provers are based on such calculi, and most of those use a variant of the *given-clause algorithm* to achieve this enumeration. In this algorithm the set of all clauses is partitioned into a set $\mathcal{A}$ of *active* (or *processed*) clauses and a set $\mathcal{P}$ of passive (or *unprocessed*) clauses. Initially, $\mathcal{A}$ is empty and $\mathcal{P}$ contains the input. During each traversal of its main loop the algorithm picks a clause c (the *given clause*) from $\mathcal{P}$. This selection is controlled by an *evaluation heuristic* φ, which typically prefers small or old clauses. If c cannot be shown to be redundant, the algorithm inserts c into $\mathcal{A}$, generates all direct consequences between $\mathcal{A}$ and c, and finally adds them to $\mathcal{P}$. In this case we say that c has been *activated*. Another important parameter of the algorithm is the *reduction ordering* $>$ which can have a strong influence on the proof search.

The given-clause algorithm is used in two major variants, the *DISCOUNT loop* and the *Otter loop*, both named after the system that made the corresponding version popular. The difference lies in the treatment of simplification. In the DISCOUNT variant only the given clause and the active clauses are used for simplification. Newly generated clauses are simplified once after generation, and again if they are picked for processing. Active clauses are back-simplified with the given clause. Passive clauses, as their name implies, are not used for simplification. This leads to a very high inference rate and allows a variety of optimizations. In particular, since passive clauses do not participate in any inferences, they can be stored very efficiently[1]. Furthermore, the so-called orphan-criterion applies: A clause can be deleted if one of its "parents" has been interreduced.

Algorithm 1 The proof procedure of WALDMEISTER

FUNCTION $\textsc{Waldmeister}(\mathcal{I}np, >, \varphi, \mathit{strength})$: BOOLEAN
$(\mathcal{A}, \mathcal{P}, \mathcal{H}) := (\varnothing, \text{axioms}(\mathcal{I}np), \text{hypotheses}(\mathcal{I}np))$
WHILE $\neg\text{trivial}(\mathcal{H}) \wedge \mathcal{P} \neq \varnothing$ **DO**
  $e := \arg\min \varphi(\mathcal{P});\ \mathcal{P} := \mathcal{P} \backslash \{e\}$
  IF $\neg\text{orphan}(e)$ **THEN**
    $e := \text{Normalize}^{>}_{\mathcal{A}}(\mathit{full}, e)$
    IF $\neg\text{redundant}(e)$ **THEN**
      $(\mathcal{A}, P_1) := \text{Interred}^{>}(\mathcal{A}, e)$
      $\mathcal{A} := \mathcal{A} \cup \{e\}$
      $P_2 := \text{Superpos}^{>}(e, \mathcal{A})$
      $\mathcal{P} := \mathcal{P} \cup \text{Normalize}^{>}_{\mathcal{A}}(\mathit{strength}, P_1 \cup P_2)$
      $\mathcal{H} := \text{Normalize}^{>}_{\mathcal{A}}(\mathcal{H})$
    END
  END
END
RETURN $\text{trivial}(\mathcal{H})$

Both E and Waldmeister use this organization as can be seen in Alg. 1 and Alg. 2 respectively. The main (conceptional) difference between the provers lies in the handling of hypotheses. Waldmeister keeps the hypotheses in a separate set $\mathcal{H}$ whereas in E there is no special treatment. So the test predicate for

[1] In Waldmeister passive clauses are either stored in a compressed form or are entirely deleted and reconstructed by need.

Algorithm 2 The proof procedure of E (simplified)

FUNCTION $\textsc{eprover}(\mathcal{I}np, >, \varphi, \mathit{strength})$: BOOLEAN
1: $(\mathcal{A}, \mathcal{P}, P_0) := (\varnothing, \mathcal{I}np, \varnothing)$
2: **WHILE** $\Box \notin \mathcal{A} \cup P_0 \wedge \mathcal{P} \neq \varnothing$ **DO**
3: $\quad c := \arg\min \varphi(\mathcal{P});\ \mathcal{P} := \mathcal{P} \backslash \{c\}$
4: $\quad$ **IF** $\neg\text{orphan}(c)$ **THEN**
5: $\quad\quad c := \text{Normalize}^{>}_{\mathcal{A}}(\mathit{full}, c)$
6: $\quad\quad$ **IF** $\neg\text{redundant}(c)$ **THEN**
7: $\quad\quad\quad (\mathcal{A}, P_1) := \text{Interred}^{>}(\mathcal{A}, c)$
8: $\quad\quad\quad \mathcal{A} := \mathcal{A} \cup \{c\}$
9: $\quad\quad\quad P_2 := \text{Superpos}^{>}(c, \mathcal{A})$
10: $\quad\quad\quad P_0 := \text{Normalize}^{>}_{\mathcal{A}}(\mathit{strength}, P_1 \cup P_2)$
11: $\quad\quad\quad \mathcal{P} := \mathcal{P} \cup P_0$
12: $\quad\quad$ **END**
13: $\quad$ **END**
14: **END**
15: **RETURN** $\Box \in \mathcal{A} \cup P_0$

a successful proof are slightly different as well: E detects the generation of the empty clause $\Box$, Waldmeister tests whether the simplification of $\mathcal{H}$ joins a hypothesis to the trivial equation $t = t$. Besides that, the systems are very similar. As an example, both systems offer a parameter *strength*. This allows to specify how much effort shall be spent in the normalization of newly generated clauses (line 11 in Alg. 1 and Alg. 2), since most of the time in the proof process is consumed by this routine. Of course, the simplification at activation time (line 5) is always done with full strength.

In the Otter loop the simplification relation is induced by all clauses. Implementations vary in detail, but typically all unit clauses are used for both forward and backward rewriting. Thus, redundant clauses can be removed or simplified earlier. However, this implies that all clauses need to be stored in a way that makes them available for inferences. In particular, if indexing techniques are used, *all* clauses have to be indexed, and all clauses have to be stored explicitly. Furthermore, the orphan-criterion is not applicable.

2.2 The implementation of shared rewriting

All terms in E are stored in a *term bank*. They are inserted in a bottom-up fashion, i. e. a term's key in the term bank is computed from the function symbol and the list of pointers to its already shared arguments. The term bank itself is organized as a large hash table to allow the fast access of a term node via its key. Each term node carries, besides its key, information about its superterms and about external references. Term rewriting replaces a term cell wherever it is referenced and recursively traverses all superterms to ensure that the term bank has again only one unique node for each represented term. Hence, any single rewrite step potentially affects a large number of clauses. In addition to the reference pointers the term nodes also carry various information, e. g. about the time they were last rewritten to normal form.

Clauses are represented as lists of equational literals and contain various precomputed values. Passive clauses are associated with an arbitrary number of *heuristic evaluations* and are entered in an equivalent number of priority queues. Due to this very general design unit clauses carry a significant size overhead compared with Waldmeister's more specialized data structures.

The indexing data structure used in E is a perfect discrimination tree with age constraints. Each branch in the index is annotated with the age of the youngest clause it contains. These annotations and the normal form dates stored in each term cell enable the matching routine to cut off entire branches of the discrimination tree early if no useful match is possible. We found that for reasonably hard examples, this saves about 10% of overall time and about 30% of the time for matching and ordering tests for rewriting.

3 Experiments concerning the sharing factor

The benefits of the shared term approach obviously depend on the amount of sharing achieved, i. e. on the average number of standard term nodes each shared term cell represents. We call the ratio between the number of subterms in a collection of terms (in our case usually all terms in the proof state) and the shared terms nodes necessary to represent them the *sharing factor*. In this section, we analyze the behavior of E with respect to this central parameter.

There are several possible sources for the duplication of terms in the search state of a superposition based theorem prover. Typically, even in the initial clause set, small terms like variables or constant terms occur more than once. An analysis of the calculus reveals a number of potential reasons for an increase in the number of duplicated terms during the inference process. Ignoring the ordering restrictions, the superposition inference rule[2] can be expressed as follows (we write $u \dot{\simeq} v$ to cover both $u \simeq v$ and $u \not\simeq v$):

$$\frac{s \simeq t \vee C \quad u \dot{\simeq} v \vee D}{\sigma(u \dot{\simeq} v[t]_p \vee C \vee D)} \quad \text{where } \sigma = \mathrm{mgu}(s, v|_p)$$

As we can see, nearly all (sub)terms of the new clause are instantiated copies of terms from the parent clauses. This introduces duplication via three mechanisms.

1. The parts of the parent clauses containing no or only unbound variables are not affected by σ and are copied verbatim, leading to sharing between the new clause and the old ones.

2. The application of σ leads to sharing within the new clause whenever a bound variable occurs more than once in the uninstantiated new clause.

[2]The additional generating inference rules necessary for the non-unit case are applied very rarely in practice and are therefore insignificant for this analysis.

3. Often σ contains *copying bindings*, i. e. $\sigma(x)$ is an uninstantiated subterm from one of the parent clauses. This again leads to sharing between this parent and the new clause.

These considerations lead to the following hypothesis:

Hypothesis 1 The average sharing factor increases during the saturation process and is correlated to the size of the term set. Extremely high degrees of sharing are characterized by the presence of highly non-linear or permutative clauses like e. g. the AC axioms.

Experiment 1 The theorem prover E is retrofitted to determine the number of shared term cells and the number of standard cells they represent.

To establish enough diversity and therefore allow general conclusions, we use a large number of specifications with different basic parameters such as difficulty, size and structure of the signature, or number of axioms. The sharing factor is determined at the end of each run, when E has either solved the problem or has encountered a given resource limit. We use the set of all unit-equational problems with ground goals from TPTP 2.4.1 [SS98] as test problems, i. e. 380 proof tasks[3]. We use two different clause selection heuristics (symbol counting and symbol counting interleaved with age-based selection with a pick-given ratio of 5:1) and two term orderings, a lexicographic path ordering (LPO) and a Knuth-Bendix Ordering (KBO). We also use two different strengths for rewriting newly created clauses. Full rewriting uses all processed unit clauses to rewrite the new clauses. Alternatively, we perform rewriting with oriented units only, thus saving the potentially costly ordering tests of the instantiated equations. The combination of the three parameters results in 8 different parameter settings.

Additionally, we analyze selected examples in detail. At each iteration of the main saturation loop the given clause is recorded and the sharing factor is measured. This allows us to correlate the development with the structure of the given clauses selected and thus to partially explain the observations.

All tests were performed on SunBlade 1000 computers with 750 MHz processors, a time limit of 500 seconds and a memory limit of 192 MB.

Result 1 Figure 1 shows the sharing factor at the termination of the prover over the number of clauses generated during the run. The number of generated clauses is a good indicator for the amount of work done by the prover, and in most cases is strongly correlated to the total number of terms in the proof state. The average sharing factor increases from about 1 for trivial examples to over 15 for larger ones. About 5% of the runs show a sharing factor higher than 20. The diagram does not include a small number of outliers with values between 60 and 3200. There is no indication that search parameters or reduction ordering have a significant influence on the sharing factor.

[3] Waldmeister and E handle non-ground goals very differently, and no useful comparison is possible for this case.

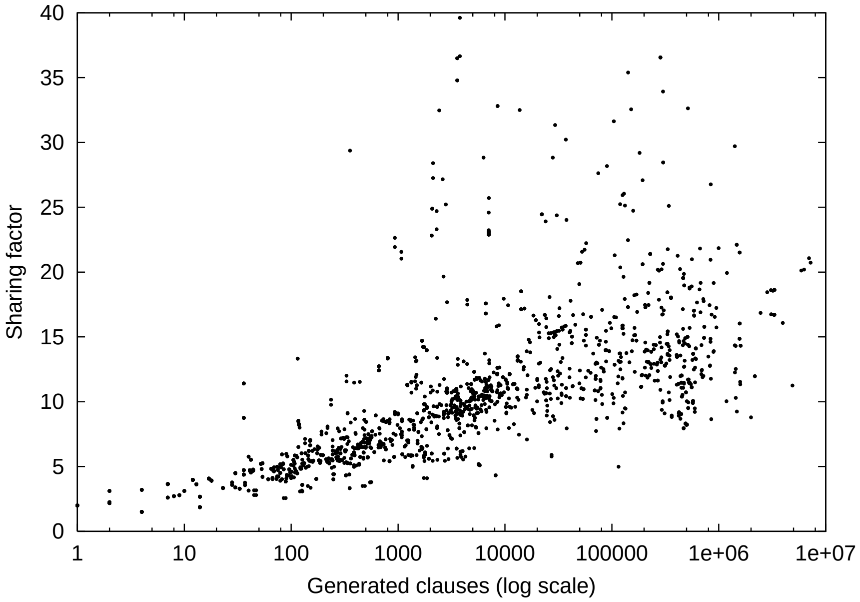


Figure 1: The sharing factor as a function of the length of the runs (measured as number of generated clauses).

The outliers are problems from group theory and from the ROB domain in TPTP. A detailed analysis of individual examples shows that the most extreme cases of sharing occur for problems with highly non-linear axiomatizations. For the problem GRP207-1 we observe variables with more than 30 occurrences in the right hand side of an equation after a very moderate number of activations.

A more interesting pattern can be observed if permutative equations, in particular AC axioms, are present. Figure 2 shows the development of the sharing factor for ROB005-1, a typical AC specification of medium difficulty. We can identify three periods with a strong monotonic increase of the sharing factor. During each of these periods all activated clauses belong to a new class of permutative equations. In these periods the sharing factor is increased due to the third mentioned mechanism: Overlaps with permutative equations result in substitutions with copying bindings only. If E's AC redundancy elimination mechanism (based on [AHL00]) is activated, the graph shows a much more stable behavior, with a sharing factor in the more moderate range of about 10 for most of the time. Thus, the high sharing factor is an indicator for the presence of a large number of very similar redundant clauses.

3.1 Sharing in the non-unit case.

To find out how these results generalize to the non-unit case we also performed experiments with Horn and non-Horn specifications. Due to the much larger

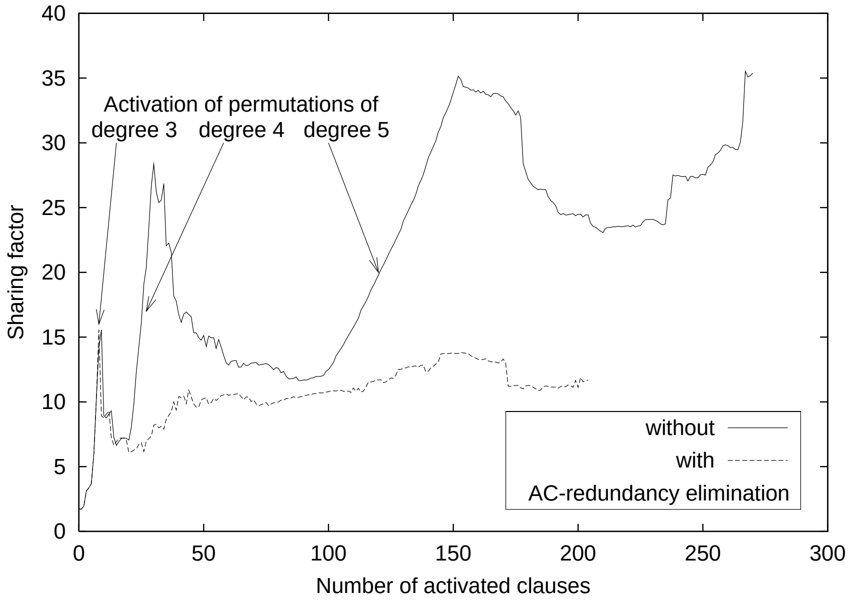


Figure 2: Development of the sharing factor over activated clauses for ROB005-1

amount of such problems contained in the TPTP, we were not able to perform the experiments as thoroughly as for the unit case. However, we performed two sets of runs, one without literal selection and one with literal selection. Both were run on SUN Ultra-10/300MHz machines, with a time limit of 300 seconds. This is roughly equivalent to 125 seconds on the faster machines used for the unit experiments.

For many problems with non-Horn specifications the sharing factor is very high – more than 100 for about 30% and about 10000 for roughly 10% of the problems. Generally, using literal selection leads to a somewhat lower sharing factors.

In the Horn case we observe a very interesting effect. If we run the prover with literal selection, which results in a positive unit strategy, the bulk of examples has sharing factors from 5 to 15. This is very similar to the unit case. Without literal selection the behavior is similar to the non-Horn case: Many examples exhibit a sharing factor of about 50, and a still significant number a factor of more than 200.

We interpret this results as follows: As we stated in the analysis at the beginning of section 3, there are three causes for the creation of shared terms: Copying of the unmodified context of an inference, instantiation of non-linear variables, and copying variable bindings. We believe that the first two effects are primarily responsible for the behavior in the non-unit case.

If we perform a generating inference between two non-unit clauses, we suffer from the so-called *duplication problem* [LP92]. All literals of the precondition clauses except for the inference literals are copied (in instantiated form) into the conclusion. If the clauses have more than two literals, this leads to an exponential growth in the number of literals in a clause. Thus, non-unit inferences lead to very large clauses – and inferences with large clauses lead to a lot of duplicated terms, both because there is more context copied in each inference, and because, on average, the degree of non-linearity increases.

In the Horn case, literal selection leads to a unit-strategy, i. e. no inferences involving more than one non-unit clause are performed, and hence the length of the clauses never increases. This explains why in this case the sharing factor behaves very similar to the unit case. Without literal selection, arbitrary inferences are possible, and thus we see both longer clauses and more sharing. As an example, consider the problem `BOO006-1` from TPTP, a problem of medium difficulty. If we use the simple clause weight heuristic without literal selection, the average number of literals per clause, measured at the time the total number of clauses in the proof state reaches 20000, is approximately 3.4, and the sharing factor is 475. With strict literal selection (always select the smallest negative literal), that numbers drop to 1.26 for the literals per clause and 78 for the sharing factor.

In the non-Horn case, literal selection no longer results in a unit strategy. However, it still results in positive strategy, and thus blocks all inferences between clauses with both positive and negative literals. Most long clauses have both positive and negative literals, and hence literal selection is able to limit the growth of clauses to some extend. Again, we look at an example of medium difficulty, `SWC398-1`. Without literal selection, the average literal number per clause is 5.4, and the sharing factor 244. With literal selection, we have 1.21 literals per clause, and a sharing factor of 6.2. As can be seen in both cases, a higher literal count (implying, on average, more inference context), also leads to a higher sharing factor.

4 Experiments concerning the shared rewriting

After analyzing the influence of the shared term implementation on the *static* representation of each consecutive search state, we will now discuss the effects of term sharing on more *dynamic* aspects, i. e. on the rewriting and matching operations and on the influence on the search behavior.

4.1 Influence on the search behavior

Comparing the number of rewrite steps performed by a system with shared rewriting with the number performed by a standard implementation is surprisingly hard. A first, conceptually simple, approach would be to implement both forms of rewriting in a single system, and to compare the two results. However, the shared rewriting has a profound influence on the design of many different parts of a proof system. Adapting all these parts for the conventional case amounts to reimplementing large parts of the prover. This is not feasible within reasonable resources.

Alternatively, one could analyze the behavior of E, and attempt to figure out how many conventional rewrite steps correspond to each shared inference performed by E. However, this is very hard: Due to sharing, simplification steps performed with new clauses affect as a side effect clauses in $\mathcal{P}$ a conventional prover would never touch. Furthermore, we gain no insights into the influence of shared rewriting on search behavior and running time.

We finally settled on the the following approach: Since E and Waldmeister are similar in their structure (except for shared rewriting!) we adjust them to behave as similar as possible. If this is achievable for a significant number of diverse problems, we can compare the number of conventional rewrite steps performed by Waldmeister with the number of shared steps performed by E.

Hypothesis 2 It is possible to make E and Waldmeister behave similar for a wide variety of proof tasks by adjusting search parameters of both systems, i. e. the introduction of shared rewriting does not significantly alter the search behavior.

Experiment 2 For both provers we fix the clause evaluation heuristic, the rewrite relation for forward simplification of newly generated clauses, the rewrite strategy (leftmost-innermost), and the term ordering to identical settings. We also disable techniques like AC-redundancy elimination or special goal handling that are only implemented in one prover or are likely to lead to significant differences.

We then compare the behavior of the provers for corresponding strategies. A good indicator for the similarity of two runs is the comparison of the number of clauses generated until a proof is found. Due to the known instability of most saturating proof techniques this number normally varies greatly for even minor differences in the search process.

We use the same problem set and machine configuration as in Experiment 1. Since both systems vary in inference speed it is not sensible to compare runs when one or both encounter a timeout. We therefore restrict our attention to the 983 runs where both systems find a proof within the given resource limits of 500 seconds and 192 MBytes.

Result 2 Figure 3 shows the number of steps performed by Waldmeister over the number of steps performed by E for each proof problem successfully proved by both systems. As we can see, a large majority of examples shows very similar behavior for both systems. If we allow a difference of 20% only, 630 out of 983 runs, i. e. 64% of all runs, are similar. Outliers are distributed quite evenly on both sides. Considering the above mentioned instability both provers show a remarkably similar behavior.

Thus, contrary to our initial expectation, we find no strong evidence that the choice of the term representation introduces a qualitatively different behavior in the unit-equational case.

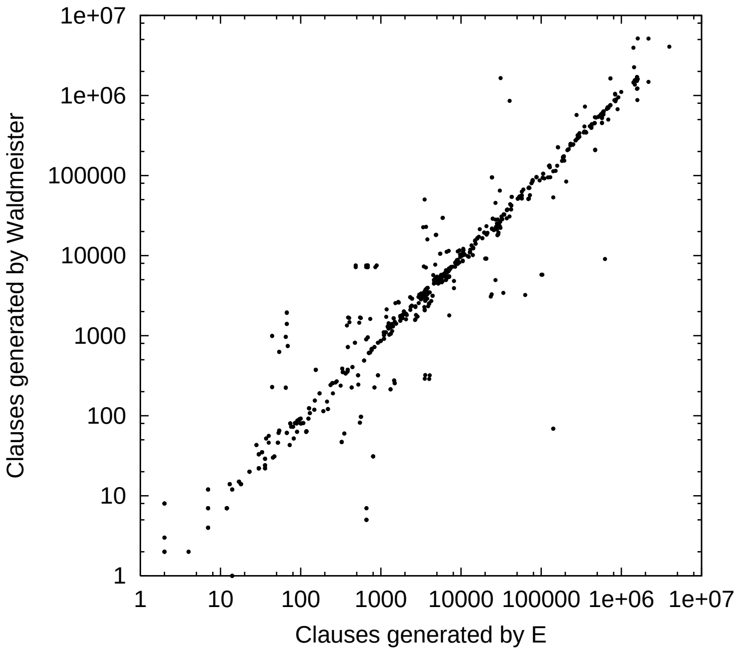


Figure 3: Clauses generated by Waldmeister over clauses generated by E for 983 runs

4.1.1 Stability of the clause count.

Our hypothesis, that even small disturbances in the inference process lead to vastly different numbers of generated clauses, is demonstrated by a different set of experiments. We also ran E and Waldmeister with an interleaved *symbol counting/first-in-first-out* heuristic. In this case, E counts activations of the goal as part of the clause selection process, while Waldmeister not, since it treats the goal as a distinct object. This small difference leads to a much wider spread of results. In this case, only 21% of the runs lie in the 20% difference interval.

4.2 The number of performed rewrite steps

For the influence of shared rewriting on normal form computations two main effects have to be considered. First, each rewrite step in E corresponds to multiple conventional rewrite steps, since all instances of the rewritten term are replaced at once. Second, the normal form procedure stores knowledge about previous match attempts in the term node and thus avoids redundant match attempts. This generalizes a typical optimization in innermost rewriting, namely the marking of irreducible subterms.

Hypothesis 3 The ratio between non-shared and shared rewrite steps and

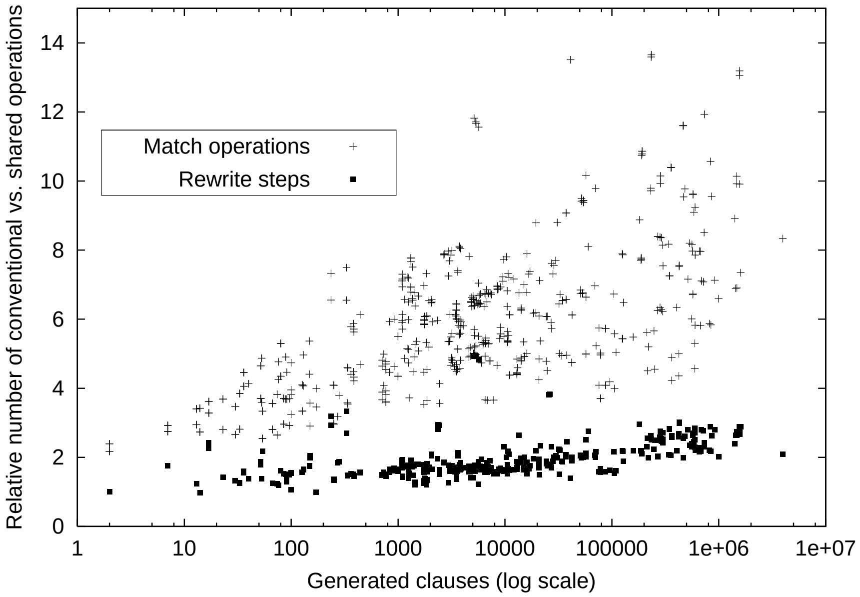


Figure 4: Quotient of conventional vs. shared rewriting steps resp. matching operations

matching operations increases for longer runs. This behavior is more pronounced for matching operations.

Experiment 3 We compare the number of rewrite steps and match operations performed by Waldmeister and E for the similar runs, i. e. where the number of generated clauses differs by at most 20% between both provers, determined in Experiment 2.

Result 3 Figure 4 shows both the relative proportion of unshared rewrite steps performed by Waldmeister versus shared rewrite steps performed by E and the relative proportion of matches attempted by both provers. Both values have been corrected for the small differences in the number of generated clauses.

For the rewrite steps we see a quite homogenous result, with the ratio of unshared to shared inferences rising from about 1 for easy problems to about 2.5 for harder problems. For the number of matches the spread of results is much bigger. The ratio of match attempts rises from 1 to about 9, with the spread for harder problems going from about 5 to about 14. A small number of outliers display a ratio of 4 or 5 in rewrite steps and a ratio from 15 to 25 for match attempts.

As expected, the ratio of conventional vs. shared operations rises with time. The small effect for rewrite steps is explained by the fact that the vast majority originates from normalizing the newly generated clauses. The number of such clauses typically only grows moderately over time. Most sharing, on the other hand, occurs in the much larger set $\mathcal{P}$. Since most clauses in $\mathcal{P}$ are never activated, the implicit rewriting of terms in these clauses has no counterpart in Waldmeister. For match attempts the normal form information at each shared note is crucial. If E encounters a term that has been normalized with the current rewrite relation, it can avoid matches on this term *and* on all of its subterms.

4.3 Comparison of the run times

Up to now, we have only measured and compared quantities that are implementation independent. This allows to asses the properties of a method or technology regardless of peculiarities of actual implementations. When comparing the run times of E and Waldmeister we have to take into account the disturbances due to different levels of optimizations in the actual code. Nevertheless, analyzing the ratio of run times over the problem complexity, that is, the time needed to solve it, gives insights into the suitability of shared rewriting for challenging proof tasks.

Hypothesis 4 There are no fundamental differences between shared and non-shared rewriting over the run time. Shared rewriting can cope better than non-shared rewriting with the additional costs for full simplification of newly generated clauses compared to simplification with oriented units only.

Experiment 4 We compare the run times of Waldmeister and E for the similar runs, i. e. where the number of generated clauses differs by at most 20% between both provers, determined in Experiment 2.

Result 4 As expected, Waldmeister is faster in most cases. But we find no significant correlation between the ratio of run times and the ratio of matching operations or rewrite steps.

An interesting pattern concerning the strength of initial simplification can be observed: If only orientable units are used, Waldmeister is about four to five times faster than E, independent of the magnitude of the run time. If full simplification is used, Waldmeister is only 2.5 times faster on the average, the variance is much higher, and there are even some examples, where E is faster than Waldmeister. Here too, no significant change over the run time is observed.

There are six runs where E is faster than Waldmeister and Waldmeister needs more than ten seconds. With one exception, they all belong to problems with one or two AC-symbols in the axiomatization. For Waldmeister, restricting the strength of the simplification relation is especially beneficial for this class of problems. For E, the effect is not so pronounced – in fact, in our experience E performs better with the full strenght rewrite relation on nearly all problems. This is consistent with the above findings.

Rewriting with orientable instances of unorientable equations, as done in the case of the full strength rewrite relation, requires an ordering comparison after each successful matching operation. This test leads to an increased costs for

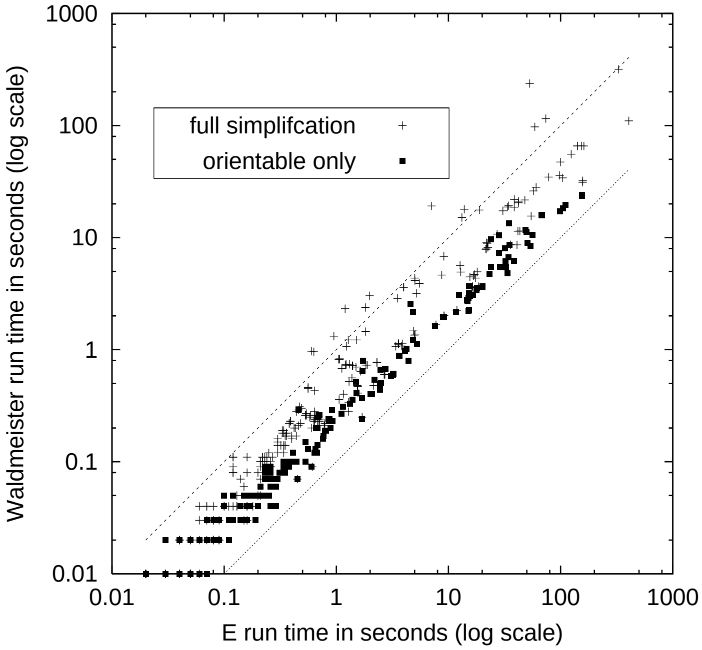


Figure 5: Comparison of actual running times of E and Waldmeister. The two boundary lines correspond to the case of equal speed for both provers and to the case where Waldmeister is 10 times faster. Note that the scales are logarithmic.

rewrite attempts with such equations. In the case of AC-symbols the following effect can be observed: The commutativity axiom matches at all positions where its function symbol occurs. Then the ordering constraints have to be checked. Most of the time they are not fulfilled. The situation is similar with other small permutative equations. As a result the normalization relation becomes notably more expensive. It seems that in this situation the normal form dates stored in the term nodes are especially profitable for E. We conclude that shared rewriting has advantages when the rewrite relation is costly.

5 Conclusion

Our analysis shows to our surprise that for unit equality problems there is no evidence for a significant difference caused by the choice of the term structure. The memory saving possible with sharing can also be achieved by much simpler compression techniques. Concerning rewriting, the positive effects of slightly fewer rewrite steps and noticeably fewer matches seem to be neutralized by the much higher cost for rewriting and term bank administration. When the rewrite relation is more expensive, the situation changes a little bit, as the analysis of the overall run times shows. There is some evidence that E can deal better with search situations with a high degree of redundant information. We also find that the shared term representation in E masks a number of weaknesses in the inference engine. In particular, term comparisons with KBO and LPO are much slower in E, and E does not use certain optimizations in indexing symmetric unit clauses. This seems to be compensated by the normal form dates in the term nodes.

Nevertheless, the shared rewriting paradigm requires a significantly higher implementation effort to get a working prover, and for the unit-equational case development resources should probably be invested into other areas.

We conjecture that shared rewriting promises more benefits for other settings. First, for provers using the Otter loop which have to represent all clauses explicitly, the space savings caused by the shared term representation can become important. Secondly, since the shared term implementation can save a significant amount of match operations, it should pay off in the case where matching is more expensive, e. g. in the case of AC rewriting. Finally, the much higher sharing factors observed in the non-unit case indicate that the real potential of the approach lies in this area. Unfortunately, we currently see no possibility of a detailed comparison for this case, since no prover with a structure sufficiently similar to E is known to us.

This study has had one positive side effect: Both authors have gained a far better understanding of both systems and of the implemented techniques. We found some minor bugs in one of E's heuristics and in some reporting code.

The outcome of these experiments has given us new insights about where to invest development effort. For example, the sharing factor seems to be a good indicator for the behavior of the search heuristic: A high sharing factor indicates that the prover generates a lot of redundant clauses and should probably change its search behavior.

References

[AHL00] J. Avenhaus, T. Hillenbrand, and B. Löchner. On Using Ground Joinable Equations in Equational Theorem Proving. In P. Baumgartner and H. Zhang, editors, *Proc. of the 3rd FTP, St. Andrews, Scottland*, Fachberichte Informatik. Universität Koblenz-Landau, 2000. (revised version to be published in the Journal of Symbolic Computation).

[BDP89] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[Chr93] J. Christian. Flat Terms, Discrimination Nets and Fast Term Rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. System Abstract: Waldmeister – Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, pages 232–236. Springer, 1999.

[LP92] S.-J. Lee and D.A. Plaisted. Eliminating Dupliction with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[McC92] W.W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 370–375. Springer, 2001.

[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.

[SSPar] G. Sutcliffe, C.B. Suttner, and J. Pelletier. The IJCAR ATP System Competition. *Journal of Automated Reasoning*, to appear.

[Sut01] G. Sutcliffe. The CASC-JC Web Site. `http://www.cs.miami.edu/~tptp/CASC/JC/`, 2001.

Implementation of Knowledge Bases for Natural Deduction

Dominique Pastre

Crip5 - Université René Descartes - Paris 5
45 rue des Saints Pères
F - 75270 Paris Cedex 06
pastre@math-info.univ-paris5.fr
http://www.math-info.univ-paris5.fr/~pastre

Abstract. This paper describes some methods of the MUSCADET theorem prover which is a knowledge-based system using natural deduction strategies. Detailed MUSCADET proofs are given, as well as many examples of rules.

1 Introduction

Most theorem provers nowadays are based on the resolution principle [12]. The properties of such provers have been studied from a theoretical point of view and much progress has also been made from a practical point of view. However, it is also useful to continue to improve theorem provers based on natural deduction, following the terminology of [1, 2]. The MUSCADET system [9–11] is such a prover [1]. Moreover, it is built as a knowledge-based system; all methods are expressed as rules which are either given to the system or automatically built by metarules. MUSCADET participated in the last three CADE Automated theorem proving System Competition (CASC-16/17/JC). It was the only prover based on natural deduction and the results show its complementarity with regard to resolution-based provers.

Section 2 presents MUSCADET's main methods and gives a detailed example of a proof. It also gives the principle of the rules' construction which is illustrated with an example. Section 3 presents the processing of existential properties which is a crucial strategy. A detailed proof of a theorem and an outline of another proof are given for theorems which could not be proved by resolution-based theorem provers in CASC-JC. The processing of negation is briefly mentionned in section 4. Section 6 concludes on the usefulness of cooperation between provers.

2 MUSCADET main methods

MUSCADET works with objects, hypotheses, a conclusion and rules. It has to prove the conclusion by applying rules which may, for example, add new hypotheses, modify the conclusion, create new objects, split the theorem into two

[1] MUSCADET is available at the address http://www.math-info.univ-paris5.fr/~pastre/muscadet/muscadet.html

or more sub-theorems. Rules are either logical rules or rules which have been built from definitions, lemmas or universal hypotheses.

2.1 An example of a proof

The MUSCADET proof of the following problem will illustrate this mechanism.

Theorem. *The union of the power set of two sets A and B is included in the power set of the union of A and B.*

Its first order statement is

$$\forall A \forall B(\mathcal{P}(A) \cup \mathcal{P}(B) \subset \mathcal{P}(A \cup B))$$

Its MUSCADET expression is

```
for_all(A, for_all(B, subset(union(power_set(A), power_set(B)),
                                                power_set(union(A,B)))))
```

Its TPTP expression is

```
! [A,B] : subset(union(power_set(A),power_set(B)),
                                                 power_set(union(A,B))))
```

The following definitions, which are used in this problem,

$$\forall A \forall B(A \subset B \Leftrightarrow \forall X(X \in A \Rightarrow X \in B))$$
$$\forall A \forall B \forall X(X \in A \cup B \Leftrightarrow X \in A \vee X \in B)$$
$$\forall A \forall X(X \in \mathcal{P}(A) \Leftrightarrow X \subset A)$$

are given to the system.

Their MUSCADET expression is

```
definition(subset(A,B)<=>for_all(X, member(X,A)=>member(X,B)))
definition(member(X,power_set(A))<=>subset(X,A))
definition(member(A, union(B, C))<=>member(A, B) or member(A, C))
```

In TPTP, it is not possible to use the keyword `definition`, and these definitions are given as axioms.

```
input_formula(subset,axiom,( ! [A,B] :
   ( subset(A,B) <=> ! [X] : ( member(X,A) => member(X,B) ) )    ))
input_formula(power_set,axiom,(
         ! [X,A] : ( member(X,power_set(A)) <=> subset(X,A) )    ))
input_formula(union,axiom,( ! [X,A,B] :
   ( member(X,union(A,B)) <=> ( member(X,A) | member(X,B) ) )    ))
```

MUSCADET has to recognize definitions among the axioms. Roughly, definitions are formulas containing an equivalence in which one argument is of the form $P(X1, ..., Xn)$ or $Q(X1, ..., F(X1, ..., Xn), ..., Xn)$ where P or F has no other occurrence in the formula.

From these definitions MUSCADET first builds the following rules, where the argument `N` is the number of the (sub-)theorem to which the rule will be applied.

```
rule(N, subset) :- hyp(N,subset(A,B)),hyp(N,member(X,A)),
                   not hyp(N,member(X,B)),
                   addhyp(N,member(X,B))
```

if $A \subset B$ and $X \in A$ are hypotheses,
then add the hypothesis $X \in B$ if it is not yet a hypothesis

```
rule(N, union) :- hyp(N,union(A,B):C),hyp(N,member(X,C)),
                  not hyp(N,member(X,A) or member(X,B)),
                  addhyp(N,member(X,A) or member(X,B)).
```

if $A \cup B : C$ [2] and $X \in C$ are hypotheses,
then add the hypothesis $X \in A \vee X \in B$ if it is not yet a hypothesis

```
rule(N, union1) :- hyp(N,union(A,B):C),hyp(N,member(X,A)),
                   not hyp(N,member(X,C)),
                   addhyp(N,member(X,C)).
```

if $A \cup B : C$ and $X \in A$ are hypotheses,
then add the hypothesis $X \in C$ if it is not yet a hypothesis

```
rule(N, union2) :- hyp(N,union(A,B):C),hyp(N,member(X,B)),
                   not hyp(N,member(X,C)),
                   addhyp(N,member(X,C)).
```

if $A \cup B : C$ and $X \in B$ are hypotheses,
then add the hypothesis $X \in C$ if it is not yet a hypothesis

```
rule(N, power_set) :- hyp(N,power_set(A):B),hyp(N,member(X,B)),
                      not hyp(N,subset(X,A)),
                      addhyp(N,subset(X,A)).
```

if $\mathcal{P}(A) : B$ and $X \in B$ are hypotheses,
then add the hypothesis $X \subset A$ if it is not yet a hypothesis

```
rule(N, power_set1) :- hyp(N,power_set(A):B),hyp(N,subset(X,A)),
                       not hyp(N,member(X,B)),
                       addhyp(N,member(X,B)).
```

if $\mathcal{P}(A) : B$ and $X \subset A$ are hypotheses,
then add the hypothesis $X \in B$ if it is not yet a hypothesis

The construction of these rules will be explained in the next sub-section.

[2] this means that C is the name of $A \cup B$

There are also universal logical rules, which are given to the system, and which are classic rules in natural deduction.

```
rule(N, for_all1) :- concl(N, for_all(X,C)), var(X),
                     create_object(N,x,X1), addobject(N,X1),
                     replace(C,X,X1,C1), newconcl(N, C1).
```

if the conclusion is $\forall X\, P(X)$
then create a new object $X1$ *and the new conclusion is* $P(X1)$

```
rule(N, =>) :- concl(N, A=>B), addhyp(N,A),
               newconcl(N,B).
```

if the conclusion is $H \Rightarrow C$
then add the hypothesis H *and the new conclusion is* C

Adding hypotheses (super-action `addhyp`) is defined by a pack a rules. Only elementary properties and existential properties are directly added. Conjunctive expressions are decomposed into elementary formulas. Instead of adding universal hypotheses and implications, rules are built.

```
rule(N, or) :- hyp(N, A or B), not  hyp_treated(N, A or B),
               concl(N, C),
               nouvconcl(N, (A=>C) and (B=>C)),
               addhyp_treated(N, A or B).
```

if $A \vee B$ *is a hypothesis and has not yet been treated, and the conclusion is* C
then the new conclusion is $(A \Rightarrow C) \wedge (B \Rightarrow C)$
and the fact that this hypothesis has been treated is memorized.

```
rule(N,stop1):- concl(N,C), clos(C), hyp(N,C),
                newconcl(N,true).
```

if the conclusion C *is totally instantiated and is already a hypothesis,*
then the (sub-)theorem is proved,
this is memorized by setting the conclusion to true

```
rule(N, elifun) :- concl(N, C),
                   elifun(N, C, C1), newconcl(N, C1).
```

The super-action $elifun$ "eliminates" functional symbols from the conclusion C and the new cocnlsuion is $C1$.
$elifun$ is a Prolog predicate which expresses procedural techniques to flatten deep terms. For example, if p is a predicate symbol, f and g are functional symbols and a, b, c are constants, $p(f(a), g(b, f(c)))$ will be replaced by $p(f\_a, g\_b\_f\_c)$ and the hypotheses $f(a) : f\_a$, $f(c) : f\_c$ and $g(b, f\_c) : g\_b\_f\_c$ added. f_a, f_c, $g\_b\_f\_c$ are constants automatically built by $elifun$ and the symbol ":" means that f_a, f_c and $g\_b\_f\_c$ are the names of $f(a)$, $f(c)$ and $g(b, f(a))$.
If X is a variable, $p(f(X))$ will be replaced by *only(f(X):Y, p(Y)* which means that for the only Y equal to $f(X)$, $p(Y)$ is true. This "new quantifier" *only* is later treated either as an existential quantifier or as a universal quantifier, de-

pending on the context (see [9]).

```
rule(N, defconcl1) :- concl(N, C),  definition(C <=> D),
                      newconcl(N, D).
```

if the conclusion is P(...) *and there is a definition* P(...) $\Leftrightarrow$ ...
then replace the conclusion by its definition.

```
rule(N, defconcl2) :- concl(N, C), C =.. [R, A, B], hyp(N, D:B),
                          definition(XRD <=> P),
                          XRD =.. [R,X,D1],not var(D1), D=D1,
                      replace(P, X, A, P1), newconcl(N, P1).
```

if the conclusion is R(A,B)
and there is a hypothesis F(...):B *and a definition* F(...) $\Leftrightarrow$ ...
then replace the conclusion by its definition

Now, here is the MUSCADET proof of the theorem.

MUSCADET first writes the address and the name of this problem in the TPTP library

```
tptp/Problems/SET/SET694+4.p
thI22
```

The problem statement is translated into the MUSCADET syntax

```
theorem to be proved
for_all(A, for_all(B, subset(union(power_set(A), power_set(B)),
                                           power_set(union(A, B)))))
```

and becomes the conclusion of the initial theorem numbered 0. At the begining there is no hypothesis and no object, but only this conclusion to be proved.

```
0 new conclusion
for_all(A, for_all(B, subset(union(power_set(A), power_set(B)),
                                           power_set(union(A, B)))))
```

Links are added for this theorem numbered 0 to the concepts which occur in the conjecture (and recursively to the concepts which occur in the definitions of the preceding concepts, if any).

```
0 add link subset
0 add link power_set
0 add link union
```

Then MUSCADET activates the rules that are pertinent for the conjecture, i.e. universal rules and rules which have been built from the definitions of the concepts linked to the conjecture. The order in which they are activated is heuristic.

```
active rules: for_all1 for_all2 elifun stop1 stop2 stop3 stop4 ...
not hypnot hypnotnot => <=> only ...
concl-exists1 concl-exists2 concl-exists3 ...
subset power_set power_set1 union union1 union2 ...
and defconcl1 exists or defconcl2 defconcl3 ...
```

Now rules are applied.
New objects x and $x1$ are created and the conclusion is instantiated with x and $x1$

```
0 add object x
0 new conclusion
for_all(A, subset(union(power_set(x), power_set(A)),
                                          power_set(union(x, A))))
--------------------------------------------------- rule for_all1
0 add object x1
0 new conclusion subset(union(power_set(x), power_set(x1)),
                                          power_set(union(x, x1)))
--------------------------------------------------- rule for_all1
```

Muscadet accepts statements with functional symbols but works as if there were predicate symbols. It creates objects, adds the definitions of theses objects as hypotheses and these names replaces the corresponding terms in the conclusion. The names are Prolog constants automatically built from the functional symbols. The symbol ":" means that $\mathit{union\_x\_x1}$, for example, is the name of the union of x and $x1$ The formula $\mathit{union}(x, x1) : \mathit{union\_x\_x1}$ will be manipulated as if it were a predicate $p(x, x1, \mathit{union\_x\_x1})$.

```
0 add object union_power_set_x_power_set_x1
0 add object power_set_x
0 add hypothesis power_set(x):power_set_x
0 add object power_set_x1
0 add hypothesis power_set(x1):power_set_x1
0 add hypothesis union(power_set_x,
                     power_set_x1):union_power_set_x_power_set_x1
0 add object power_set_union_x_x1
0 add object union_x_x1
0 add hypothesis union(x,x1):union_x_x1
0 add hypothesis power_set(union_x_x1):power_set_union_x_x1
0 new conclusion subset(union_power_set_x_power_set_x1,
                                             power_set_union_x_x1)
----------------------------------------------------- rule elifun
```

Then the conclusion is replaced by its definition

```
0 definition of the conclusion
0 new conclusion
for_all(X, member(X, union_power_set_x_power_set_x1) =>
```

```
                                  member(X, power_set_union_x_x1))
--------------------------------------------------- rule defconcl1
```

Creation of object x2

```
0 add object x2
0 new conclusion member(x2, union_power_set_x_power_set_x1) =>
                                  member(x2, power_set_union_x_x1)
---------------------------------------------------- rule for_all1
```

Rule => decomposes the statement into a hypothesis and a conclusion

```
0 add hypothesis member(x2, union_power_set_x_power_set_x1)
0 new conclusion member(x2, power_set_union_x_x1)
----------------------------------------------------------- rule =>
```

Rule *union* was built from the definition of union

```
0 add hypothesis member(x2, power_set_x)or member(x2, power_set_x1)
------------------------------------------------------- rule union
```

Rule *or* has a low priority. It is applied here because no other rule can be applied. Otherwise, rules with higher priority would be applied first in order to avoid useless splittings.

```
0 new conclusion (
  (member(x2,power_set_x) => member(x2,power_set_union_x_x1)) and
      (member(x2,power_set_x1) => member(x2,power_set_union_x_x1))
0 add hypothesis-treated member(x2, power_set_x) or
                                          member(x2, power_set_x1)
----------------------------------------------------------- rule or
```

Then rule *and* splits the theorem into two sub-theorems numbered 1 and 2.

```
********************************************* sub-theorem 1 *****
1 new conclusion member(x2, power_set_x) =>
                                  member(x2, power_set_union_x_x1)
----------------------------------------- creation of sub-theorem 1
1 add hypothesis member(x2, power_set_x)
1 new conclusion member(x2, power_set_union_x_x1)
----------------------------------------------------------- rule =>
1 add hypothesis subset(x2, x)
--------------------------------------------------- rule power_set
definition of the conclusion
1 new conclusion subset(x2, union_x_x1)
--------------------------------------------------- rule defconcl2
1 definition of the conclusion
1 new conclusion
for_all(X, member(X, x2)=>member(X, union_x_x1))
```

```
--------------------------------------------------- rule defconcl1
1 add object x3
1 new conclusion member(x3, x2)=>member(x3, union_x_x1)
---------------------------------------------------- rule for_all1
1 add hypothesis member(x3, x2)
1 new conclusion member(x3, union_x_x1)
---------------------------------------------------------- rule =>
1 add hypothesis member(x3, x)
------------------------------------------------------ rule subset
1 add hypothesis member(x3, union_x_x1)
------------------------------------------------------ rule union1
1 new conclusion true
------------------------------------------------------- rule stop1
theorem 1 proved
```

As the first sub-formula of the conjunctive conclusion of the initial theorem has been proved, it is removed from this conclusion

```
0 new conclusion
      member(x2, power_set_x1) => member(x2, power_set_union_x_x1)

******************************************** sub-theorem 2 *****
2 new conclusion
      member(x2, power_set_x1) => member(x2, power_set_union_x_x1)
---------------------------------------- creation of sub-theorem 2
2 add hypothesis member(x2, power_set_x1)
2 new conclusion member(x2, power_set_union_x_x1)
---------------------------------------------------------- rule =>
2 add hypothesis subset(x2, x1)
--------------------------------------------------- rule power_set
definition of the conclusion
2 new conclusion subset(x2, union_x_x1)
--------------------------------------------------- rule defconcl2
2 definition of the conclusion
2 new conclusion
-for_all(X, member(X, x2)=>member(X, union_x_x1)).
--------------------------------------------------- rule defconcl1
2 add object x4
2 new conclusion member(x4, x2)=>member(x4, union_x_x1)
---------------------------------------------------- rule for_all1
2 add hypothesis member(x4, x2)
2 new conclusion member(x4, union_x_x1)
---------------------------------------------------------- rule =>
2 add hypothesis member(x4, x1)
------------------------------------------------------ rule subset
2 add hypothesis member(x4, union_x_x1)
```

```
------------------------------------------------------ rule union2
2 new conclusion true
------------------------------------------------------- rule stop1
theorem 2 proved
```

As both sub-formulas of the conjunctive conclusion have been proved, the initial theorem is proved

```
0 new conclusion true
--------------------------------------------------------- rule and
theorem 0 proved ( thI22 ) in 0.79 seconds
==================================================================
```

2.2 Building rules

The construction of rules is performed by the super-action `buildrules` which is a recursive Prolog predicate and which builds one or more rules step by step from definitions and lemmas.

In the call `buildrules(E,N,Concept,Name,Corps)`,
- `E` is a (sub-)formula to be treated,
- `N` is the number of a (sub-)theorem if the rules being built are only local for this (sub-)theorem and any variable otherwise,
- `Concept` is the defined concept or the name of a lemma,
- `Corps` is the part of the rule which has already been built.
This predicate returns `true` if it has succeeded in building at least one rule. As a side effect the built rules are asserted as new clauses.

For a concept `P` defined by a formula `A<=>B`, where `A` is $P(X_1, X_2, ...)$, the first call is `buildrules(A=>B,_,P,Name,[])`
For a functional concept `F` defined by a formula $P(F(X_1, X_2, ...) \Leftrightarrow E$ the first call is `buildrules(FX:C => (PC <=> E))`
where `FX` is $F(X_1, X_2, ...)$ and `PC` is $P(C)$.

Here are some parts of the `buildrules` predicate

```
buildrules(E,N,Concept,Name,Corps) :-
      ( E = (A or B => C)
         -> buildrules((A=>C)and(B=>C),N,Concept,Name,Corps)
      ; E = (A=>B)
         -> ( A = A1 and A2 -> addcond(N,Corps,A1,Corps1),
                               buildrules(A2 => B,N,Concept,Name,Corps1)
              ; ...
              ; /* else */ addcond(N,Corps0,A,Corps1),
                           buildrules(B,N,Concept,Name,Corps1)
             )
      ; E = (A<=>B) -> buildrules((A=>B)and(B=>A),N,Concept,Name,Corps)
```

```
      ; E= A and B -> create_name_rule(Name, Name1),
                      buildrules(A,N,Concept,Name1,Corps),
                      create_name_rule(Name1, Name2),
                      buildrules(B,N,Concept,Name2,Corps)
      ; E = for_all(X,B), var(X),
         -> buildrules(B,N,Concept,Nom,Corps1)
      ; ...
      ; addendseq(Corps0, not hyp(N,E), Corps1) ,
        addendseq(Corps1, addhyp(N,E), Corps2),
        addrule(N, Concept, Name, Corps2),
      ).
```

The super-action `addcond` adds the new condition(s) `A` to the sequence `C` of the present conditions and returns the new sequence `CA`.

```
addcond(N,C,A,CA) :-
      ( A = A1 and A2 -> addcond(N,C,A1,C1), addcond(N,C1,A2,CA)
      ; ...
      ; addendseq(C,hyp(N,A),CA)
      ) .
```

`addendseq(C,C1,CA)` adds `C1` to the end of the sequence `C` and returns `CA`.

The super-action `addrule` asserts the rule as a new clause, except if `Arg` is a number. In this case, the rule is only added to the list of active rules (`addlocalrule(Arg,R,Name)`).

```
addrule(N, Arg, Name, Corps) :-
          ( number(Arg) ->
                      addlocalrule(Arg,(rule(N,Name):-Corps), Name)
          ; assert((rule(N,Name) :- Corps))
          ).
```

For the definition of *union*, the first call to `buildrules` is

```
buildrules(union(A,B):C => for_all(X, member(X,C) <=>
                                member(X,A) or member(X,B))),
           _, union, union, [])
```

Then the successive calls are the following

```
buildrules(for_all(X, member(X,C) <=> member(X,A) or member(X,B)),
           _,union, union,
           hyp(N, union(A,B):C)).

buildrules(member(X,C) <=> member(X,A) or member(X,B),
           _, union, union,
           hyp(N, union(A,B):C)).
```

```
buildrules(member(X,C) => member(X,A) or member(X,B))
              and (member(X,A) or member(X, B) => member(X,C)),
           _, union, union,
           hyp(N, union(A,B):C)).

   buildrules(member(X,C) => member(X,A) or member(X,B),
              _, union,union,
              hyp(N, union(A,B):C)).

   buildrules(member(X,A)or member(X,B),
              _, union, union,
              hyp(N, union(A,B):C), hyp(N,member(X,C))).
```

and the clause

```
rule(N, union) :- hyp(N,union(A,B):C), hyp(N,member(X,C)),
                  not hyp(N,member(X,A) or member(X,B)),
                  addhyp(N,member(X,A) or member(X,B)).
```

is added

```
   buildrules(member(X,A) or member(X,B) => member(X, C),
              _, union, union1,
              hyp(N, union(A,B):C)).

   buildrules(member(X,A) => member(X,C))
                    and (member(X,B) => member(X,C)),
              _, union, union1,
              hyp(N, union(A,B):C)).

      buildrules(member(X,A) => member(X,C), _, union, union1,
                 hyp(N, union(A,B):C)).

      buildrules(member(X,C), _, union, union1,
                 hyp(N, union(A,B):C), hyp(N, member(X,A))).
```

and the clause

```
rule(N, union1) :- hyp(N,union(A,B):C), hyp(N,member(X,A)),
                   not hyp(N,member(X,C)),
                   addhyp(N,member(X,C)).
```

is added

```
      buildrules(member(X,B) => member(X,C), _, union, union2,
              hyp(N, union(A,B):C)).

      buildrules(member(X,C), _, union, union2,
                 hyp(N, union(A,B):C), hyp(N, member(X,B))
```

and the clause

```
rule(N, union2) :- hyp(N,union(A,B):C),hyp(N,member(X,B)),
```

```
                    not hyp(N,member(X,C)),
                    addhyp(N,member(X,C)).
```
is added

3 Processing of existential properties

A crucial MUSCADET strategy is the processing of existential properties which I will illustrate with two theorems. These theorems were proposed at the last CADE competition (CASC-JC) and were proved only by MUSCADET. These results show the complementarity of MUSCADET with regard to resolution-based provers.

3.1 A detailed proof of a first theorem

This is problem SET722+4 in the TPTP library.
Theorem. *Consider two mappings f from A into B and g from B into C. If $g_o f$ is surjective then g is surjective.*

a) Here are the definitions used in this theorem

```
maps(F, A, B) <=> for_all(X, member(X,A) =>
         exists(Y, member(Y,B) and apply(F,X,Y)))
    and for_all(X, for_all(Y1, for_all(Y2,
         member(X,A) and member(Y1,B) and member(Y2,B)
                => (apply(F,X,Y1)and apply(F,X,Y2) => (Y1=Y2))))))

surjective(F, A, B) <=> for_all(Y,
             member(Y,B)=>exists(X, member(X,A)and apply(F,X,Y))))

for_all(G, for_all(F, for_all(A, for_all(B, for_all(C,
   for_all(X, for_all(Z, member(X,A) and member(Z,C) =>
      (apply(compose_function(G,F,A,B,C),X,Z) <=>
         exists(Y, member(Y,B) and
            apply(F,X,Y) and apply(G,Y,Z))))))))))
```

b) some of the automatically built rules

```
rule(N, maps) :- hyp(N, maps(F,A,B)), hyp(N, member(X,A)),
        not hyp(N, exists(Y, member(Y,B)and apply(F,X,Y))),
        addhyp(N, exists(Y, member(Y,B)and apply(F,X,Y))).
rule(N, maps1) :- hyp(N, maps(F,A,B)), hyp(N, member(X,A)),
        hyp(N, member(Y1,B)), hyp(N, member(Y2,B)),
        hyp(N, apply(F,X,Y1)), hyp(N, apply(F,X,Y2)), not Y1=Y2,
        addhyp(N,Y1=Y2).
rule(N, surjective) :- hyp(N, surjective(F,A,B)),
        hyp(N, member(Y,B)),
```

```
        not hyp(N, exists(X, member(X,A)and apply(F,X,Y))),
        addhyp(N, exists(X, member(X,A)and apply(F,X,Y))).
rule(N, compose_function_) :- hyp(N, member(X,A)),
        hyp(N, member(Y,C)),
        hyp(N, compose_function(G,F,A,B,C):H),
        hyp(N, apply(H,X,Y)),
        not hyp(N, exists(J, exists(Z, member(Z,B) and
                            apply(F,X,Z)and apply(G,Z,Y))),
        addhyp(N, exists(Z, member(Z,B) and
                            apply(F,X,Z)and apply(G,Z,Y))).
rule(N, compose_function_1) :- hyp(N, member(X,A)),
        hyp(N, member(Y,B)), hyp(N, member(Z,C)),
        hyp(N, apply(F,X,Y)), hyp(N, apply(G,Y,Z)),
        hyp(N, compose_function(G,F,A,B,C):H),
        not hyp(N, apply(H,X,Z)),
        addhyp(N, apply(H,X,Z)).
```

c) rules given to the system to treat existential hypotheses and conclusions,

```
rule(N, exists) :- hyp(N, exists(X,P)), not hyp_traite(N, exists(X,P)),
                   treat(N, exists(X,P)),
                   addhyp_treated(N, exists(X,P)).
```

if P is an existential hypothesis which has not yet been treated, then treat it.

As for rule *or*, this rule has a low priority. Existential hypotheses are first added such as they are. This rule is applied when no other rule with higher priority can be applied. Otherwise, it could create infinitely many useless elements.

```
rule(N, concl_exists2) :- concl(N, exists(X, B and C)),var(X),
                          hyp(N,B), hyp(N,C), newconcl(N,true).
```

This is one of the rules for existential conclusions in the trivial case.

d) here is the MUSCADET proof, where the constants `x`, `x1`, `x2`, `x3`, `x4`, `compose_function_x1_x_x2_x3_x4`, the names of which are automatically built, have been manually replaced by `f`, `g`, `a`, `b`, `c`, `h` in order to make this trace easier to read.

```
tptp/Problems/SET/SET722+4.p
thII13

theorem to be proved

for_all(F, for_all(G, for_all(A, for_all(B, for_all(C,
                 maps(F,A,B) and maps(G,B,C) and
                 surjective(compose_function(G,F,A,B,C),A,C)
         => surjective(G,B,C)))))).
```

```
0 new conclusion
for_all(F, for_all(G, for_all(A, for_all(B, for_all(C,
                 maps(F,A,B) and maps(G,B,C) and
                 surjective(compose_function(G,F,A,B,C),A,C)
         => surjective(G,B,C)))))).

0 add link maps
0 add link surjective

0 add object f
0 new conclusion
for_all(G, for_all(A, for_all(B, for_all(C,
                maps(f,A,B) and maps(G,B,C) and
                surjective(compose_function(G,f,A,B,C),A,C)
         => surjective(G,B,C))))).
---------------------------------------------------- rule for_all1
0 add object g
0 new conclusion
for_all(A, for_all(B, for_all(C,
                 maps(f,A,B) and maps(g,B,C) and
                 surjective(compose_function(g,f,A,B,C),A,C)
          => surjective(g,B,C)))).
---------------------------------------------------- rule for_all1
0 add object a
0 new conclusion
for_all(B, for_all(C, maps(f,a,B) and maps(g,B,C) and
              surjective(compose_function(g,f,a,B,C),a,C)
              => surjective(g,B,C))).
---------------------------------------------------- rule for_all1
0 add object b
0 new conclusion
for_all(C, maps(f,a,b) and maps(g,b,C) and
           surjective(compose_function(g,f,a,b,C),a,C)
           => surjective(g,b,C)).
---------------------------------------------------- rule for_all1
0 add object c
0 new conclusion maps(f,a,b) and maps(g,b,c) and
           surjective(compose_function(g,f,a,b,c),a,c)
           => surjective(g,b,c)
---------------------------------------------------- rule for_all1
0 add object h
0 add hypothesis
    compose_function(g,f,a,b,c):h
0 new conclusion maps(f,a,b) and maps(g,b,c) and
```

```
                 surjective(h,a,c)
                 => surjective(g,b,c)
-------------------------------------------------------- rule elifun
0 add hypothesis maps(f,a,b)
0 add hypothesis maps(g,b,c)
0 add hypothesis surjective(h,a,c)
0 new conclusion surjective(g,b,c)
------------------------------------------------------------ rule =>
0 definition of the conclusion
0 new conclusion
for_all(Y, member(Y,c) => exists(X,member(X,b) and apply(g,X,Y)))
------------------------------------------------------ rule defconcl1
0 add object x5
0 new conclusion
   (member(x5,c) => exists(X, member(X,b) and apply(g,X,x5)))
------------------------------------------------------ rule for_all1
0 add hypothesis member(x5,c)
0 new conclusion exists(X, member(X,b)and apply(g,X,x5))
------------------------------------------------------------ rule =>
0 add hypothesis
exists(X, member(X,a) and apply(h,X,x5))
---------------------------------------------------- rule surjective
0 add object x6
0 add hypothesis member(x6,a)
0 add hypothesis apply(h,x6,x5)
0 add hypothesis-treated
exists(X, member(X,a) and apply(h,A,x5)).
-------------------------------------------------------- rule exists
0 add hypothesis
exists(Y, member(Y,b) and apply(f,x6,Y)).
---------------------------------------------------------- rule maps
0 add hypothesis
exists(Z, member(Z,b) and apply(f,x6,Z) and apply(g,Z,x5)).
------------------------------------------- rule compose_function_
0 add object x7
0 add hypothesis member(x7,b)
0 add hypothesis apply(f,x6,x7)
0 add hypothesis-treated exists(Y, member(Y,b) and apply(f,x6,Y))
-------------------------------------------------------- rule exists
0 add hypothesis
exists(Y, member(Y,c)and apply(g,x7,Y))
---------------------------------------------------------- rule maps
0 add object x8
0 add hypothesis member(x8,b)
0 add hypothesis apply(f,x6,x8)
```

```
0 add hypothesis apply(g,x8,x5)
0 add hypothesis-treated
exists(Z, member(Z,b) and apply(f,x6,Z) and apply(g,Z,x5))
----------------------------------------------------- rule exists
0 new conclusion true
----------------------------------------------- rule concl_exists2
theorem 0 proved ( thII13 ) in 1.3 seconds
==================================================================
```

3.2 Another example

This is problem SET737+4 in the TPTP library.
Theorem. *Consider three mappings f from A into B, g from B into C and h from C into A. If $h_o g_o f$ and $f_o h_o g$ are injective and $g_o f_o h$ is surjective then h is one-to-one.*

There are several mappings and many elements to be created: images, pre-images and intermediary elements. As this process may be expansive, it is important not to develop one direction to the detriment of the others. The creation of elements is delayed to ensure that the elements necessary for the proof will be created.

Here is an outline of the proof.

MUSCADET applies the same types of rules as in the previous examples, and it has to prove that h is injective (first sub-theorem) and surjective (second sub-theorem).

For the first sub-theorem, it creates two elements $b1$ and $b2$ in B with the same image $c1$ by g and it has to prove that $b1 = b2$.

Then for each new element which is created, MUSCADET adds the existential hypothesis stating that it has an image, a pre-image in the case of surjective mapping, and an intermediary element in the case of composed mapping. They are treated one after the other and every time an existential property is treated and a new element introduced all the other rules are tried again. So, MUSCADET creates successively
- the image $a1$ of $c1$ by h (and it deduces that $h_o g(b1) : a1$ and $h_o g(b2) : a1$)
- the preimage $c2$ of $c1$ by $g_o f_o h$ (because $g_o f_o h$ is surjective)
- the image $b3$ of $a1$ by f (and it deduces that $f_o h_o g(b1) : b3$ and $f_o h_o g(b2) : b3$)
Now, because $f_o h_o g$ is injective, it deduces that $b1 = b2$ and sub-theorem 1 is proved.

For the second sub-theorem, it creates $c1$ in C and it has to prove that $\exists X (X \in B \wedge apply(g, X, c1))$. It creates successively
- the image $a1$ of $c1$ by h

- the pre-image $c2$ of $c1$ by $g_o f_o h$ (because $g_o f_o h$ is surjective)
- the image $b1$ of $a1$ by f (and it deduces that $f_o h(c1) : b1$)
- the image $a2$ of $c2$ by h
- the pre-image $c3$ of $c2$ by $g_o f_o h$ (useless)
- the intermediary element $b2$ such that $f_o h(c2) : b2$ and $g(b2) : c1$

The conclusion $\exists X(X \in B \wedge apply(g, X, c1))$ is now satisfied, so sub-theorem 2 is proved. As both sub-theorems are proved, the initial theorem is proved.

4 Processing of negation. Positive and negative properties

4.1 Processing of negation

MUSCADET works as far as possible without negations. Not only does it work with hypotheses and a conclusion instead of negative literals but it also eliminates the negations every time it is possible.

If the conclusion to be proved is a negation $\neg C$, it adds C as a new hypothesis and the new conclusion is $false$. This means that it will have to find a contradiction i.e. it will have to deduce the hypothesis $false$. If the conclusion is $\neg C1 \vee C2$, it adds $C1$ and the new conclusion is $C2$.

¿From definitions in which a negation $\neg P$ occurs, it builds several rules which are logically equivalent. Some rules contain the negative literal $\neg P$ in the same place as it appeared in the definition, and other rules contain the positive literal P as a condition or a conclusion, depending on whether $\neg P$ was on the left or on the right of an implication.

Here are some elementary rules and the proof of easy theorems in which we will see how negations are handled and also how and why local rules are built from implication hypotheses.

```
==================================================================
rule(N, not) :- concl(N,not A),
                addhyp(N,A), newconcl(N,false).
rule(N, hypnot) :- hyp(N,not A), concl(N,false),
                   retract(hyp(N,not A)), newconcl(N,A).
rule(N, hypnotnot) :- hyp(N,not not A),
                      addhyp(N, A).
rule(N,stop4):- hyp(N,A) , hyp(N,not A) ,
                newconcl(N,true).
==================================================================
theorem to be proved
for_all(A, not not A => A).

0 new conclusion
for_all(A, not not A => A).
```

```
0 add objet x
0 new conclusion not not x => x
---------------------------------------------------- rule for_all1
0 add hypothesis not not x
0 new conclusion x
--------------------------------------------------------- rule  =>
0 add hypothesis x
--------------------------------------------------- rule hypnotnot
0 new conclusion true
------------------------------------------------------- rule stop1
theorem 0 proved
==================================================================
theorem to be proved
for_all(A, for_all(B, A and not A => B)).

0 new conclusion
for_all(A, for_all(B, A and not A => B)).

0 add objet x
0 new conclusion
for_all(A, x and not x => A).
---------------------------------------------------- rule for_all1
0 add objet x1
0 new conclusion x and not x => x1
---------------------------------------------------- rule for_all1
0 add hypothesis x
0 add hypothesis not x
0 new conclusion x1
--------------------------------------------------------- rule  =>
0 new conclusion true
------------------------------------------------------- rule stop4
theorem 0 proved
==================================================================
theorem to be proved
for_all(A, for_all(B, (A => B) => (not B => not A))).

0 new conclusion
for_all(A, for_all(B, (A => B) => (not B => not A))).

0 add objet x
0 new conclusion
for_all(A, (x => A) => (not A => not x)).
---------------------------------------------------- rule for_all1
0 add objet x1
0 new conclusion (x => x1) => (not x1 => not x)
```

```
--------------------------------------------------- rule for_all1
0 add the local rule
(rule(N, rulhyp):- hyp(N,x), not hyp(N,x1),
                   addhyp(N,x1)).

0 new conclusion not x1 => not x
---------------------------------------------------------- rule =>
0 add hypothesis not x1
0 new conclusion not x
---------------------------------------------------------- rule =>
0 add hypothesis x
0 new conclusion false
--------------------------------------------------------- rule not
remove hypothesis not x1
0 new conclusion x1
------------------------------------------------------ rule hypnot
0 add hypothesis x1
------------------------------------------------------ rule rulhyp
0 new conclusion true
------------------------------------------------------- rule stop1
theorem 0 proved
==================================================================
```

4.2 Positive and negative properties

Negations occur every time a problem contains empty set ϕ or complements or set differences. Proofs are then often proofs by contradiction

For disjoint sets, as the definition is negative,

$$\forall A \forall B (disjoint(A,B) \Leftrightarrow \neg \exists X (X \in A \wedge X \in B)$$

it creates and uses a property *nondisjoint* which is a positive property.

$$\forall A \forall B (disjoint(A,B) \Leftrightarrow \neg nondisjoint(A,B))$$
$$\forall A \forall B (nondisjoint(A,B) \Leftrightarrow \exists X (X \in A \wedge X \in B)$$

5 Conclusion

Muscadet works in a manner which is quite different from resolution-based provers. It uses methods based on natural deduction and knowledge-based systems. Some of these methods are crucial and explain why Muscadet is able to prove some theorems that resolution-based provers are not able to prove. Muscadet is efficient for mathematical everyday problems which are expressed in a natural manner, for example in naive set theory (where *mappings* are primitive concepts, defined by their properties) and for problems which contain many axioms, definitions or lemmas. It is less efficient for problems which are defined axiomatically, from a logician's point of view, for example in axiomatic geometry or in axiomatic set theory (where *mappings* are particular *relations* which are subsets of *cross-products of sets*) and for problems which contain only one large conjecture and no intermediary definitions. In particular, it is not able to prove some theorems that all resolution-based provers are able to prove. Why can't we make them cooperate ? We, human beings, do not use the same methods for all problems that we have to solve. We can choose one method or another to solve a problem. In some cases, we begin with a method, then try another, and perhaps even another and sometimes we come back to the first one and succeed with it. To veritably improve theorem provers, it would be interesting to have several provers working together, in sequence, as we do when we successively try several methods or, in parallel, as computers are able to do, or better still to have a top-level mechanism which analyses the problem and chooses one of the provers. This cooperation is already present in some systems, for example in the OMEGA system [3–6, 8] or in OTTER [7] which incorporate human methods, specific knowledge, or higher order logic for specific fields of mathematics. I believe that cooperation could also benefit the basic methods in first order logic and self-contained problems such as those of the TPTP library. It seems to me that MUSCADET should be able by itself to select the most adapted prover if we give it some appropriate general knowledge.

References

1. Bledsoe, W.W. Splitting and reduction heuristics in automatic theorem proving, Journal of Artificial Intelligence 2 (1971), 55-77
2. Bledsoe, W.W., Non-resolution theorem proving, Artificial Intelligence 9 (1977), 1-35
3. Eisinger N., Siekmann J., Smolka G., The Markgraf Karl refutation procedure, IJCAI (1981), 511-518
4. Eisinger N., Ohlbach H.J., The Markgraf Karl Refutation Procedure (MKRP), in J.H.Siekmann, ed., Procedings of the 8th CADE, Springer Verlag, Berlin, 1986
5. Huang X., Kerber M., Kohlhase M., Melis E., Nesmith D., Richts J, Siekmann J., The Ω-MKRP proof development environnment, ECAI 94 Wokshop "From theorem provers to mathematical assistants: issues and possible solutions, 66-77
6. Kerber M., Pracklein A., Using tactics to reformulate formulae for resolution theorem proving, Annals of Mathematics and Artificial Intelligence, 18-2, 1996
7. Mc Cune W.W. and Padmanabhan R., Automated Deduction in Equational Logic and Cubic Curves, Lecture Notes in Artificial Intelligence, Springer-Verlag, 1996
8. Melis E., Siekmann J., Knowledge-based proof planning, Artificial Intelligence 115 (1999), 65-105
9. Pastre D., Muscadet: An Automatic Theorem Proving System using Knowledge and Metaknowledge in Mathematics, Artificial Intelligence 38-3 (1989), 257-318
10. Pastre D., Automated Theorem Proving in Mathematics, Annals on Artificial Intelligence and Mathematics 8-3/4 (1993), 425-447
11. Pastre D., MUSCADET version 2.3, User's manual, http://www.math-info.univ-paris5.fr/˜pastre/muscadet/manual-en.ps (2001)
12. Robinson J.A., A machine oriented logic based on the resolution principle, J.ACM 12, 1965, 23-41

The $.one$-Calculus: Towards An Efficient Implementation of Explicit Substitutions

Carsten Schürmann
Yale University
carsten@cs.yale.edu

November 22, 2001

1 Introduction

When implementing theorem provers, programming languages, and meta-logical frameworks where variable binding is a central and essential concept, many important design decisions with far reaching consequences arise early in the process. Among the most pressing are how to implement variables, binding constructs, and related operations such as substitutions, variable renaming, and lookup. And naturally, there is a variety of possible answers, ranging from named to de Bruijn representations with and without explicit substitutions. Each and every answer comes with different advantages and disadvantages. Some calculi have efficient implementations but are a nightmare from a programmers point of view. Many require lots of book-keeping. Some calculi support memoization, others do not. On the other hand, there are some calculi that offer some luxury of elegance but score relatively low on the performance scale.

The dilemma therefore is obvious: Implementors have to find a compromise between efficiency and maintainability. The choice is not easy since internal representation languages come in so many different flavors.

One popular option is the calculus of explicit substitutions $\lambda\sigma$ [ACCL90] and its variations used for example for Twelf [PS99], λProlog [Nad98], Flint [SLM98] or Omega [BCF$^+$97]. $\lambda\sigma$ has turned out to be an elegant and appropriate albeit hard to debug core structure for an implementation, and many algorithms such as weak head normalization, higher-order pattern unification [DHKP96] useful for higher-order logic programming, and automated theorem proving take advantage of the concise, efficient, non-redundant way explicit substitutions can be implemented. In this paper we sketch a refined but equivalent formulation of the $\lambda\sigma$-calculus, which is called $.one$-calculus. It allows compact representations of normal substitutions — especially pattern and pruning substitutions [Mil91].

Concretely, we have noticed that normal forms of explicit substitutions in $\lambda\sigma$ are asymmetric in the following sense. On the one hand, there are $\lambda\sigma$ substitutions such as $\uparrow^6$ that map empty contexts into objects valid in any arbitrary — here for illustration reasons of length 6 — context $\Gamma = A_1, A_2, A_3, A_4, A_5, A_6$. On the other, there is an identity substitution in its most explicit and frequently used form $1.2, 3.4.5.6.\uparrow^6$.

$$\Gamma \vdash \uparrow^6 : \cdot$$
$$\Gamma \vdash 1.2.3.4.5.6.\uparrow^6 : \Gamma$$

When implemented, the first substitution requires one cell and is parametrized by the length of the context Γ, whereas the second requires as many cells as the context contains declarations. If we choose, however, to introduce some syntax for the frequently used operation $1.\sigma\circ\uparrow$ and denote it with $\Downarrow \sigma$ (the so called *.one* operation), the encoding of the identity substitution above can be written equally concise as $\Downarrow^6$ id. Any sensible implementation of this substitution also requires only one cell. Let us write $\Uparrow \sigma$ for $\sigma\circ\uparrow$:

$$\Gamma \vdash \Uparrow^6 \mathrm{id} : \cdot$$
$$\Gamma \vdash \Downarrow^6 \mathrm{id} : \Gamma$$

This example already illustrates the main idea behind the *.one*-calculus. Its main benefits include a way to represent explicit substitutions compactly, and methods to lookup, compose, and normalize them efficiently. In our experience, the *.one*-calculus leads to elegant implementations, whose functions and procedures may be verified against their invariants (at least informally).

Another aspect of the *.one*-calculus is its performance benefits due to the compact representations of substitutions. We have conducted experiments and compared the results to the one used in the current implementation of the meta-logical framework Twelf [PS99]. First preliminary empirical results indicate that *.one*-substitutions increase the overall system performance.

The test bed for our performance measurements consists of a core reimplementation of Twelf based on the *.one*-calculus and a randomized substitution generation tool. It is, however, not yet as general as one would hope for; for example, the test bed lacks essential functionality such as unification indispensable for logical programming and automated theorem proving. Therefore, in its current state, our test bed is insufficient to experiment with real world data samples such as deductions in logic, typing derivations, compilation runs, and traces of operational semantics on abstract machines.

In this paper we present the *.one*-calculus of explicit substitutions. To this end, we briefly review explicit substitutions in Section 2, revisit the $\lambda\sigma$-calculus in Section 3, before we present the *.one*-calculus in Section 4. In Section 5 we discuss the implementation and some performance aspects. Finally, we assess results and mention future work in Section 6.

2 Explicit Substitutions

Substitutions are prevalent in many implementations ranging from proof assistants and theorem provers to programming languages and compilers. By definition, we understand a substitution as a set of bindings of variables names to terms. Traditionally, the application of a substitution to a term refers to a process that replaces every free variable in a term by its instantiation — and this in an eagerly fashion. But this design has drawbacks. The most serious is exhibited by the following unification problem. The unification of equation $(c\, U)[\sigma] \equiv (c'\, U')[\sigma']$, for example, requires that both sides are reduced to normal forms by a relatively expensive operation. The two terms are not unifiable unless $c = c'$. How can this be made more efficient? If only the application of substitutions σ and σ' could be delayed, the constant clash could have been detected almost immediately, and valuable time spent on computing $U[\sigma]$ and $U'[\sigma']$ could have been saved.

One way of delaying substitution application is to make substitutions explicit and admit closures of terms under substitutions as first-class objects. This idea underlies a large family of explicit substitution calculi, including the $\lambda\sigma$-calculus [ACCL90]. This calculus is based on a de Bruijn notation of variable names, and it is used in many implementations, such as Twelf [PS99], λProlog [Nad98], and Omega [BCF$^+$97]. Two important classes of substitutions that occur over and over are *pattern substitutions* and *pruning substitutions*:

Pattern substitutions arise in the context of higher-order unification. In general, this problem is undecidable, but when restricted to the fragment of higher-order patterns [Mil91] it becomes decidable. In this fragment, all logical variables (or meta variables) can only occur in form of patterns: $X\ x_1\ \ldots\ x_n$. Here, X is a logic variable and the x_i are pairwise disjoint parameters. Following Dowek et al. [DHKP96] higher-order pattern unification problems are turned into first-order unification problems by lowering logic variables to base type, and capturing the list of arguments in form of an explicit substitution.

Pruning substitutions occur frequently in the setting of higher-order patterns, logic programming, and automated theorem proving. They often arise during higher-order pattern unification and their job is it to map contexts into larger contexts. Every pruning substitution is also a pattern substitution and can be built in the *.one*-calculus solely from $\Uparrow$, $\Downarrow$, and id. But first we turn to the $\lambda\sigma$-calculus.

3 The $\lambda\sigma$-Calculus

The version of the $\lambda\sigma$-calculus which we will use in this paper is defined for general PTS [Bar92]. To take full advantage of explicit substitutions for the purpose of unification and weak head reduction, we make use of the spine notation of terms [CP97]. Instead of writing $U_1\ U_2\ U_3$ for the application of U_1 to U_2 and U_3, we write $U_1 \cdot (U_2; U_3; \mathrm{nil})$. U_1 is called *head*, and $(U_2; U_3; \mathrm{nil})$ form a *spine*. Heads and spines together form a *redex*, that may β-reduce if the head is of functional type.

Expressions:	U	$::=$	$X_{\Gamma;U} \mid 1 \mid c \mid U \cdot S \mid \lambda U_1 . U_2 \mid \Pi U_1 . U_2 \mid U[\sigma] \mid s$
Spines:	S	$::=$	$\mathrm{nil} \mid U; S \mid S[\sigma]$
Substitutions:	σ	$::=$	$\mathrm{id} \mid U.\sigma \mid \uparrow \mid \sigma_1 \circ \sigma_2$
Contexts:	Γ	$::=$	$\cdot \mid \Gamma, U$
Signatures:	Σ	$::=$	$\cdot \mid \Sigma, c : U$

$X_{\Gamma;U}$ is a logic variable of type U in context Γ. The same logic variable may occur several times in the same term, but it will always be defined in the same context and have the same type. c is a constant defined in Σ. We write $\lambda U_1 . U_2$ for λ-expressions, $\Pi U_1 . U_2$ for function types, and $U[\sigma]$ for the closure of an expression under a substitution σ. s denotes a sort.

1 is the de Bruijn index that refers to the innermost variable binder. Note, that 1 is the only construct that acts as variable in the $\lambda\sigma$-calculus. Variables that refer to other variable binders are closures of 1 with an appropriate substitution. $1[\uparrow^4]$, for example, refers to the fifth top variable binder, and represents therefore de Bruijn index 5.

Spines are lists of expressions, with the addition of the spine closure operation $S[\sigma]$ which closes a spine under a substitution. The definitions of contexts and signatures are standard. The three typing judgments [CP97] are defined below and appropriate sets of inference rules are given in Figure 1.

Valid expressions:	$\Gamma \vdash U_1 : U_2$
Valid spines:	$\Gamma \vdash S : U_1 > U_2$
Valid substitutions:	$\Gamma' \vdash \sigma : \Gamma$

The first judgment is standard, and so are most of its rules. We only comment on the one-rule where U must be closed under the $\uparrow$ substitution since otherwise, it would violate the representation invariant that says that types must be valid in Γ as well. For the validity judgment of spines, suppose that S is the argument list to a function of type U_1. Under these conditions the second judgment holds only if the resulting term has type U_2. nil represents the empty argument list and has therefore type $U > U$ (by nil). In the non-empty case, if S has already type $U_2 > U_3$, $U; S$ may serve as an argument list to a function of type $\Pi U_1 . U_2$ given that U has type U_1. A substitution is valid if all its bindings relate variables from Γ to terms that are valid in Γ'. The validity rules of the two basic substitutions are id and shift. The comp rule is standard. The closure $V[\sigma]$ in the left premiss of the dot-rule is necessary to preserve the invariant from above.

To the author's knowledge, up to this point of time, it is still unclear how much performance benefit can one expect from using explicit substitutions in an implementation over traditional techniques. But independent of performance considerations, explicit substitutions form a valuable organizing force, which has lead to an elegant implementation of the core of the Twelf system. Examples include modules for weak-head normalization, higher-order pattern unification, logic variable abstraction, logic programming, and proof search. We turn now to the definition of the *.one*-calculus.

$$\frac{}{\Gamma \vdash X_{\Gamma,U} : U}\,\mathsf{evar} \qquad \frac{}{\Gamma, U \vdash 1 : U[\uparrow]}\,\mathsf{one} \qquad \frac{c : U \in \Sigma}{\Gamma \vdash c : U}\,\mathsf{const}$$

$$\frac{\Gamma \vdash U_1 : s \qquad \Gamma, U_1 \vdash U : U_2}{\Gamma \vdash \lambda U_1.\, U : \Pi U_1.\, U_2}\,\mathsf{lam} \qquad \frac{\Gamma \vdash U_1 : s_1 \qquad \Gamma, U_1 \vdash U_2 : s_2}{\Gamma \vdash \Pi U_1.\, U_2 : s_2}\,\mathsf{pi}$$

$$\frac{\Gamma \vdash U : U_1 \qquad \Gamma \vdash S : U_1 > U_2}{\Gamma \vdash U \cdot S : U_2}\,\mathsf{redex} \qquad \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash U_1 : U_2}{\Gamma \vdash U_1[\sigma] : U_2[\sigma]}\,\mathsf{eclo}$$

$$\frac{}{\Gamma \vdash \mathrm{nil} : U > U}\,\mathsf{nil} \qquad \frac{\Gamma \vdash U : U_1 \qquad \Gamma \vdash S : U_2 > U_3}{\Gamma \vdash U; S : \Pi U_1.\, U_2 > U_3}\,\mathsf{app}$$

$$\frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash S : U_1 > U_2}{\Gamma \vdash S[\sigma] : U_1[\sigma] > U_2[\sigma]}\,\mathsf{sclo}$$

$$\frac{}{\Gamma \vdash \mathrm{id} : \Gamma}\,\mathsf{id} \qquad \frac{\Gamma' \vdash U : V[\sigma] \qquad \Gamma' \vdash \sigma : \Gamma}{\Gamma' \vdash U.\sigma : \Gamma, V}\,\mathsf{dot}$$

$$\frac{}{\Gamma, U \vdash \uparrow : \Gamma}\,\mathsf{shift} \qquad \frac{\Gamma'' \vdash \sigma_2 : \Gamma' \qquad \Gamma' \vdash \sigma_1 : \Gamma}{\Gamma'' \vdash \sigma_1 \circ \sigma_2 : \Gamma}\,\mathsf{comp}$$

Figure 1: Typing rules for $\lambda\sigma$.

4 The *.one*-Calculus

One basic design decision that implementors are facing when designing core data structures based on explicit substitutions calculi is the following: which of the substitution operations shall one assume to be primitive (that is uninterpreted, simply a constructor) and which should one interpret as a function? In $\lambda\sigma$, for example, composition of substitution could become a constructor or is it better to compose substitutions right away?

The *.one*-calculus carries this idea one step further. Its design is based on the observation that there are two basic operations that occur over and over in an implementation, especially in the context of higher-order pattern unification.

$$\begin{array}{lrcl} \textit{.one}\text{-operation:} & \Downarrow \sigma & = & 1.\, \sigma \circ \uparrow \\ \textit{shift}\text{-operation:} & \Uparrow \sigma & = & \sigma \circ \uparrow \end{array}$$

The first operation pushes substitutions under λ-binders:

$$(\lambda U_1.\, U_2)[\sigma] = \lambda U_1[\sigma].\, U_2[1.\, \sigma \circ \uparrow] = \lambda U_1[\sigma].\, U_2[\Downarrow \sigma]$$

The second arises frequently during unification.

Although both operations are defined in terms of the standard "." and composition operations we have collected enough evidence to justify turning

$.one$ and *shift* into primitive substitutions which is the main motivation of the $.one$-calculus presented here in this paper. Repeated applications of these new primitive operations are subsequently compounded into blocks.

$$
\begin{array}{llll}
\text{Expressions:} & U & ::= & X_{\Gamma;U} \mid 1 \mid c \mid U \cdot S \mid \lambda U_1 . U_2 \mid \Pi U_1 . U_2 \mid U[\tau] \\
\text{Spines:} & S & ::= & \text{nil} \mid U; S \mid S[\tau] \\
\text{Substitutions:} & \tau & ::= & \text{id} \mid U.\tau \mid \Downarrow^n \tau \mid \Uparrow^m \tau \mid \tau_1 \circ \tau_2 \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, U \\
\text{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma, c : U
\end{array}
$$

The standard shift substitution is definable in the $.one$ calculus as $\Uparrow$ id. Every expression in the $.one$-calculus can be mapped back into the $\lambda\sigma$-calculus using the embedding function $\ulcorner \cdot \urcorner$ that is defined as follows. It generalizes (polymorphically) to all syntactic categories above in a straightforward way.

$$
\begin{array}{rcl}
\ulcorner \text{id} \urcorner & = & \text{id} \\
\ulcorner U.\sigma \urcorner & = & \ulcorner U \urcorner . \ulcorner \sigma \urcorner \\
\ulcorner \Downarrow^n \sigma \urcorner & = & \begin{cases} \ulcorner \sigma \urcorner & \text{if } n = 0 \\ \ulcorner \Downarrow^{n-1} \sigma \urcorner \circ \uparrow & \text{if } n > 0 \end{cases} \\
\ulcorner \Uparrow^n \sigma \urcorner & = & \begin{cases} \ulcorner \sigma \urcorner & \text{if } n = 0 \\ 1 . (\ulcorner \Uparrow^{n-1} \sigma \urcorner \circ \uparrow) & \text{if } n > 0 \end{cases}
\end{array}
$$

The judgments are the same as in the $\lambda\sigma$ case except that the formulation of some rules has slightly changed and that we write τ instead of σ. The inference rules for expressions and spines are almost identical to the ones in Figure 1 except that the shift substitution in the one-rule must be replaced by "$\Uparrow$ id". The validity rules for substitutions are given in Figure 2. $\mathsf{down}_>$, down_0, $\mathsf{up}_>$, up_0 establish the meaning of repeated successive application of the $.one$ and the *shift* operation, respectively. In fact every derivation in the $.one$-calculus can be embedded into the $\lambda\sigma$-calculus.

Theorem 4.1 *1. If $\Gamma \vdash U_1 : U_2$ then $\ulcorner \Gamma \urcorner \vdash \ulcorner U_1 \urcorner : \ulcorner U_2 \urcorner$.*

2. If $\Gamma \vdash S : U_1 > U_2$ then $\ulcorner \Gamma \urcorner \vdash \ulcorner S \urcorner : \ulcorner U_1 \urcorner > \ulcorner U_2 \urcorner$.

3. If $\Gamma' \vdash \tau : \Gamma$ then $\ulcorner \Gamma' \urcorner \vdash \ulcorner \tau \urcorner : \ulcorner \Gamma \urcorner$.

¿From an implementors perspective, it is an important design decision to make which of the operations to keep as constructors, and which to implement as functions. For example, computing the closure $U[\tau]$ eagerly instead of forming it explicitly leads to the traditional implementation of substitution. Similar design questions arise when implementing $.one$, *shift*, and composition operations. They can be either be kept primitive or interpreted functionally. For the implementation discussed in Section 5, we have chosen to keep *shift* and $.one$ primitive, interpret substitution composition functionally, and join two consecutive blocks of $.one$ or *shift* operations into one whenever possible. Furthermore

$$\frac{}{\Gamma \vdash \mathrm{id} : \Gamma}\,\mathsf{id} \qquad \frac{\Gamma' \vdash U : V[\tau] \qquad \Gamma' \vdash \tau : \Gamma}{\Gamma' \vdash U.\tau : \Gamma, V}\,\mathsf{dot}$$

$$\frac{\Gamma' \vdash \tau : \Gamma}{\Gamma' \vdash \Downarrow^0 \tau : \Gamma}\,\mathsf{down}_0 \qquad \frac{\Gamma' \vdash \tau : \Gamma}{\Gamma' \vdash \Uparrow^0 \tau : \Gamma}\,\mathsf{up}_0$$

$$\frac{\Gamma' \vdash \Downarrow^{n-1} \tau : \Gamma}{\Gamma', U[\tau] \vdash \Downarrow^n \tau : \Gamma, U}\,\mathsf{down}_> \qquad \frac{\Gamma' \vdash \Uparrow^{n-1} \tau : \Gamma}{\Gamma', U \vdash \Uparrow^n \tau : \Gamma}\,\mathsf{up}_>$$

$$\frac{\Gamma'' \vdash \tau_2 : \Gamma' \qquad \Gamma' \vdash \tau_1 : \Gamma}{\Gamma'' \vdash \tau_1 \circ \tau_2 : \Gamma}\,\mathsf{comp}$$

Figure 2: Typing rules for substitutions.

we push the $U.\tau$ operation as far into the substitution as possible. The resulting substitutions are called *compact*.

$$\begin{array}{rcl} \tau_u & = & \Uparrow^n \tau_d \mid U.\tau_u \mid \mathrm{id} \\ \tau_d & = & \Downarrow^n \tau_u \mid U.\tau_d \mid \mathrm{id} \end{array}$$

Compact substitutions are unique for the fragment where the U's are de Bruijn numbers of the form $1[\Uparrow^n \mathrm{id}]$. We call these substitutions *pure*. For arbitrary terms U however, the situation is slightly different, because it takes some computational effort to decide on how far to move U into the substitution. For example $U.(\Uparrow^n \sigma) = \Uparrow^n (U'.\sigma)$ only if $U = U'[\Uparrow^n \mathrm{id}]$. Therefore, compact substitutions that are not pure are in general not unique.

The following table defines the composition of two $.one$-substitutions. The compactification of the intermediate results is implicitly assumed.

$\tau_1 \circ \tau_2$	id	$U'.\tau_2'$	$\Downarrow^m \tau_2'$	$\Uparrow^m \tau_2'$
id	τ_2			
$U.\tau_1'$	$U[\tau_2].(\tau_1' \circ \tau_2)$			
$\Downarrow^n \tau_1'$	$\Downarrow^n \tau_1'$	$U'.(\Downarrow^{n-1} \tau_1' \circ \tau_2')$	$\Downarrow^n (\tau_1' \circ \Downarrow^{m-n} \tau_2')$ if $n \leq m$ $\Downarrow^m (\Downarrow^{n-m} \tau_1' \circ \tau_2')$ if $n > m$	$\Uparrow^m (\tau_1' \circ \tau_2')$
$\Uparrow^n \tau_1'$	$\Uparrow^n \tau_1'$	$\Uparrow^{n-1} \tau_1' \circ \tau_2'$	$\tau_1' \circ \Uparrow^n (\Downarrow^{m-n} \tau_2')$ if $n \leq m$ $(\Uparrow^{n-m} \tau_1') \circ (\Uparrow^m \tau_2')$ if $n > m$	$\Uparrow^m (\tau_1' \circ \tau_2')$

With the internal data structures and basic functionality in place, it is now possible to define a reduction semantics on $.one$-expressions and spines. The following set of equations defines how explicit substitutions can be pushed structurally inside a term.

$$
\begin{array}{rcll}
1[\tau] & = & U & \text{(where } \tau(1) = U\text{)} \\
c[\tau] & = & c & \\
(U \cdot S)[\tau] & = & U[\tau] \cdot S[\tau] & \\
(\lambda U_1 . U_2)[\tau] & = & \lambda U_1[\tau] . U_2[\Uparrow \tau] & \\
(\Pi U_1 . U_2)[\tau] & = & \Pi U_1[\tau] . U_2[\Uparrow \tau] & \\
(U[\tau_1])[\tau_2] & = & U[\tau_1 \circ \tau_2] & \\
\multicolumn{4}{l}{\text{no simplification for } X_{\Gamma;U}[\tau] \text{ possible}}
\end{array}
$$

$$
\begin{array}{rcl}
\text{nil}[\tau] & = & \text{nil} \\
(U; S)[\tau] & = & (U[\tau]; S[\tau]) \\
(S[\tau_1])[\tau_2] & = & S[\tau_1 \circ \tau_2]
\end{array}
$$

Most of these equations are standard. We only comment on a few. $\tau(1) = U$ stands for the operation of looking up de Bruijn index 1 in τ. It is formally defined in the Section 5. The fourth and fifth equation demonstrate the elegance of the *.one* operation. The recommended subsequent compactification step on $\Uparrow \tau$ has been omitted.

5 Implementation

How good is the performance of the *.dot*-calculus really? Is it faster than an implementation of $\lambda\sigma$-calculus? In order to shed some light on this question, we have began with the implementation of the *.dot*-calculus as a core data structure for the Twelf system. Although the implementation project is still underway, it has matured to a state that allowed us to use it as a testbed for the performance comparision of substitution composition.

The testbed consists of a direct implementation of the *.one*-calculus, which is as direct as the the current implementation of the $\lambda\sigma$-calculus in Twelf. The only optimization that is common to both platforms is that they use integer numbers for de Bruijn indices. The current version of Twelf uses integer $k+1$ to refer to $1[\uparrow^k]$, and similarly, in the new version $k+1$ refers to $1[\Uparrow^k \text{id}]$. In this setting the lookup function $\tau(k)$ returns a closure $U[\tau]$ and is defined as follows.

$$
\begin{array}{rcl}
\text{id}(k) & = & k \\[1ex]
(U.\tau)(k) & = & \left\{ \begin{array}{ll} U[\text{id}] & \text{if } k = 1 \\ \tau(k-1) & \text{otherwise} \end{array} \right. \\[2ex]
(\Uparrow^n \tau)k & = & \left\{ \begin{array}{ll} k + k' & \text{if } \tau(k) = k' \\ U[\Uparrow^n \text{id}] & \text{if } \tau(k) = U[\text{id}] \\ U[\Uparrow^{n+m} \text{id}] & \text{if } \tau(k) = U[\Uparrow^m \text{id}] \end{array} \right. \\[2ex]
(\Downarrow^n \tau)(k) & = & \left\{ \begin{array}{ll} k & \text{if } 1 \leq k \leq n \\ (\Uparrow^n \tau)(k-n) & \text{if } n < k \end{array} \right.
\end{array}
$$

The experiments conducted, however, are not yet as general as one would hope for; for example, this test bed lacks essential functionality such as unifi-

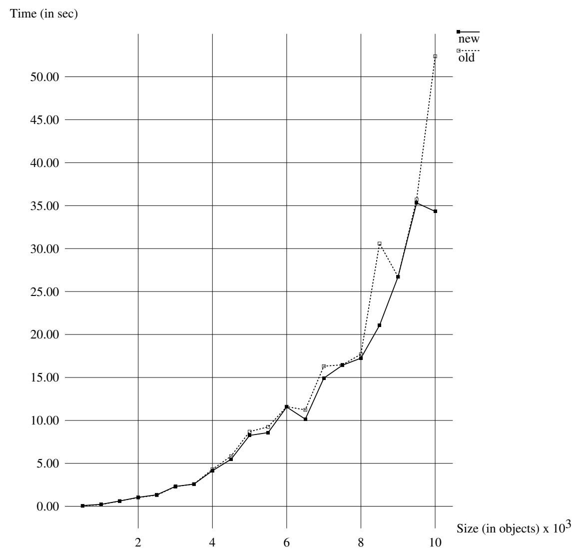


Figure 3: Pure substitutions.

cation indispensable for logical programming and automated theorem proving. Therefore, in its current state, the test bed is insufficient to experiment with concrete data samples such as deductions in logic, typing derivations, compilation runs, and traces of operational semantics on abstract machines.

Using this new implementation, we have randomly (using largely uniform distributions) generated substitutions samples, and compared the performance of the new test bed implementation (marked as "new" in the graphs) with the Twelf implementation (marked as "old" in the graphs). Although substitutions had to be compactified over and over again, the new implementation compares well to the old Twelf implementation for pure and pattern substitutions. On pruning substitutions, the new implementation outperforms the old by far. Each datapoint represents an average of 100 runs. The maximal size of a substitution is 10000 elements.

Pure substitutions. The performance graph is depicted in Figure 3. The x-axis shows the maximal size of elements in a substitution, the y-axis the time it took to compose to random substitutions. In this experiment the *.one*-calculus performs at least as good as the current Twelf implementation.

Pattern substitutions. The performance graph is depicted in Figure 4. Twelf outperforms the *.one*-calculus by a slight margin, which may be justified by the overhead cost of compactification.

Pruning substitutions. The performance graph is depicted in Figure 5. This graph clearly shows how the new implementation outperforms the old.

In summary, judging from these experiments, the *.one*-calculus is perfor-

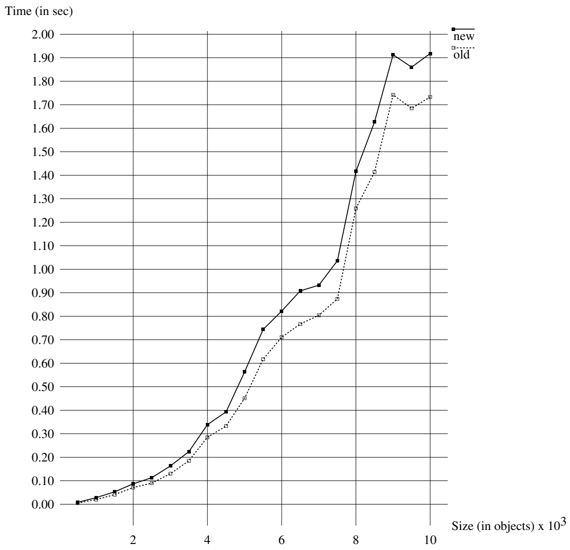


Figure 4: Pattern substitutions.

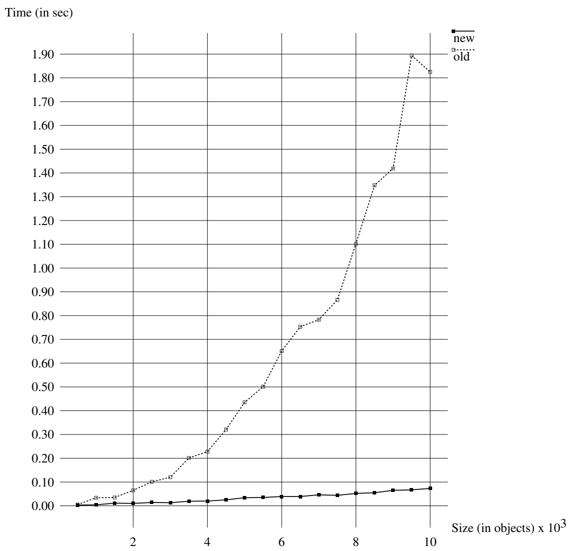


Figure 5: Pruning substitutions.

mancewise and elegancewise a valuable alternative to the $\lambda\sigma$-calculus. We expect the performance results to be even more convincing once we experiment with real data and take measures of the different components of Twelf separately, including unification, weak head reduction, abstraction, logic programming, and theorem proving.

6 Conclusion

Good implementations of logic programming languages, meta-logical frameworks and automated theorem provers are typically correlated with efficient, elegant, and concise representation languages for core datastructures. For example, explicit substitutions and spines complement each other nicely in the implementation of Twelf [PS99]. In this paper we have defined the $.one$-calculus that leads to space efficient implementations of explicit substitutions and provides time efficient implementations of substitution composition and lookup. An experimental study following in the near future will elaborate on which operations among $\Uparrow^n \sigma$, $\Downarrow^n \sigma$, or $\sigma_1 \circ \sigma_2$ are best implemented as functions or kept as primitives.

References

[ACCL90] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California, January 1990. ACM Press.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[BCF+97] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karten Konrad, Andreas Meier, Erica Melis, Wolf Schaarschmidt, Jörg Siekmann, and Volker Sorge. Omega: Towards a mathematical assistant. In *Proceedings of the Conference on Automated Deduction (CADE)*, 1997.

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, April 1997.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International*

Conference and Symposium on Logic Programming, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Nad98] Gopalan Nadathur. An explicit substitution notation in a lambdaProlog implementation. In *Proceedings of the First International Workshop on Explicit Substitutions*, Tsukuba, Japan, 1998. Available as Technical Report TR-98-01, Department of Computer Science, University of Chicago.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, Baltimore, Maryland, September 1998.

Phase Model Checking for some Linear Logic Calculi

Sylvain Soliman
Sylvain.Soliman@inria.fr

Abstract

Building upon previous work in the logical semantics of linear concurrent constraint programming languages (LCC), an example of linear logic based calculus, we design an original phase model checking method for proving safety properties of programs. We describe our implementation of the phase model checker using constraint programming techniques and provide first experimental results.

1 Introduction

The class of Concurrent Constraint programming languages (CC) was introduced a decade ago by Vijay Saraswat as a unifying framework for constraint logic programming and concurrent logic programming [11]. The CC paradigm constitutes a representative abstraction of constraint programming languages that are used nowadays in a wide variety of application domains, most notably for solving combinatorial search problems.

One can give a simple translation of CC programs in Jean-Yves Girard's Linear Logic [6] that is sound and complete for observing the entailment-closed sets of successes and accessible stores [5, 4, 12]. This, among other reasons, led to a generalization of CC languages, called Linear Concurrent Constraint languages (LCC), that is used throughout the paper for the presentation of our results. The class LCC is obtained from CC simply by allowing constraint systems based on linear logic instead of classical logic. This is a generalization which greatly extends the expressive power of CC, as non monotonic evolutions of the store are possible through the consumption of constraints by ask agents [3, 12] but it still enjoys a simple linear logic semantics with the same soundness and completeness properties. These results have been used in [4, 5] to prove safety properties of LCC programs simply by exhibiting special phase models of programs.

This paper presents, after some preliminary definitions, an application of these results to the proof of safety properties of LCC programs, through the implementation of a model checker based on the phase semantics of linear logic, as first proposed in [4]. We provide first experimental results on the efficiency of the phase model checker for proving the safety of some simple LCC programs for specifying protocols. Finally we conclude on the perspectives of this work and the generalizations of this procedure to other calculi.

2 Preliminaries

Let us briefly recall the definitions of the LCC language, of the phase semantics of linear logic, and the way they can be used for proving safety properties.

2.1 LCC

2.1.1 Syntax and operational semantics

In this paper, a set of variables is denoted by X, Y,..., the set of free variables occurring in a formula A is denoted by $\mathrm{fv}(A)$, a sequence of variables is denoted by $\vec{x}$, $A[\vec{t}/\vec{x}]$ denotes the formula A in which the free occurrences of variables $\vec{x}$ have been replaced by terms $\vec{t}$ (with the usual renaming of bound variables for avoiding variable clashes).

For a set S, $S^\star$ denotes the set of finite sequences of elements in S. For a transition relation $\rightarrow$, $\rightarrow^\star$ denotes the transitive and reflexive closure of $\rightarrow$.

The essential difference between LCC and CC is that constraints are formulae of linear logic and that communication (the *ask* rule) consumes information.

Definition 2.1 (Linear constraint system) *A* linear constraint system *is a pair* $(\mathcal{C}, \vdash_{\mathcal{C}})$*, where:*

- $\mathcal{C}$ *is a set of formulae (the* linear constraints*) built from a set V of variables, a set Σ of function and relation symbols, with logical operators: the multiplicative conjunction $\otimes$, its neutral element* 1*, and the existential quantifier $\exists$;*

- $\Vdash_{\mathcal{C}}$ *is a subset of $\mathcal{C} \times \mathcal{C}$ which defines the non-logical axioms of the constraint system.*

- $\vdash_{\mathcal{C}}$ *is the least subset of $\mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the following rules of intuitionistic linear logic:*

$$c \vdash c \qquad \frac{\Gamma, c \vdash d \quad \Delta \vdash c}{\Gamma, \Delta \vdash d} \qquad \vdash 1 \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

$$\frac{\Gamma \vdash c_1 \quad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \qquad \frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma \vdash \exists x\ c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, \exists x\ c \vdash d}\ x \notin fv(\Gamma, d)$$

The syntax of LCC *agents* is given in table 1, where $||$ stands for parallel composition, $+$ for non-deterministic choice, $\exists$ for variable hiding and $\rightarrow$ for blocking ask. The atomic agents $p(\vec{x})\ldots$ are called *process calls* or *procedure calls*, we assume that the arguments in the sequence $\vec{x}$ are all distinct variables. The *ask* agent in LCC is written with a universal quantifier in order to make explicit the variables which are bound in the guard.

Recursion is obtained by declarations. We make the usual hypothesis that in a declaration $p(\vec{x}) = A$, all the free variables occurring in A occur in $\vec{x}$. The set of declarations of an LCC program, denoted by $\mathcal{D}$, is the closure by variable renaming of a set of declarations given for distinct procedure names p. A *program* $D.A$ is a declaration D together with an initial agent A.

The operational semantics is defined on configurations where the store is distinguished from agents. A *configuration* is a triple $(X; c; A)$, where c is a constraint called the store, A is an agent or $\emptyset$ if empty, and X is a set of variables,

Agents	$A ::= p(\vec{x}) \mid tell(c) \mid (A \parallel A) \mid A + A \mid \exists x A \mid \forall \vec{x}(c \rightarrow A)$
Declarations	$D ::= \epsilon \mid p(\vec{x}) = A \mid D, D$
Program	$P ::= D.A$
α-Conversion	$\dfrac{z \notin fv(A)}{\exists y A \equiv \exists z A[z/y]}$
Parallel comp.	$A \parallel B \equiv B \parallel A$ $A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$
Equivalence	$\dfrac{(X;c;\Gamma) \equiv (X';c';\Gamma') \longrightarrow (Y';d';\Delta') \equiv (Y;d;\Delta)}{(X;c;\Gamma) \longrightarrow (Y;d;\Delta)}$
Tell	$(X;c;tell(d),\Gamma) \longrightarrow (X;c \otimes d;\Gamma)$
Ask	$\dfrac{c \vdash_{\mathcal{C}} d[\vec{t}/\vec{y}] \otimes e}{(X;c;\forall \vec{y}(d \rightarrow A),\Gamma) \longrightarrow (X;e;A[\vec{t}/\vec{y}],\Gamma)}$
Hiding	$\dfrac{y \notin X \cup fv(c,\Gamma)}{(X;c;\exists y A,\Gamma) \longrightarrow (X \cup \{y\};c;A,\Gamma)}$
Procedure calls	$\dfrac{(p(\vec{y}) = A) \in D}{(X;c;p(\vec{y}),\Gamma) \longrightarrow (X;c;A,\Gamma)}$
Blind choice	$(X;c;A+B,\Gamma) \longrightarrow (X;c;A,\Gamma)$ $(X;c;A+B,\Gamma) \longrightarrow (X;c;B,\Gamma)$

Table 1: LCC syntax and operational semantics.

called the *hidden* variables of c and A. The operational semantics is defined in the style of the CHAM [2] by a transition system which does not take into account specific evaluation strategies. The *structural congruence* $\equiv$ is the least congruence satisfying the rules of table 1. For convenience here, and unlike in [5], the logical equivalence of constraints is not built-in in the congruence. We write $\Gamma, \Delta, ...$ for multisets of agents in configurations. Congruence is extended to multisets of agents in the obvious way: $\Gamma \equiv \Gamma'$ iff $\Gamma = \{A_1, \dots, A_n\}$, $\Gamma' = \{A'_1, \dots, A'_n\}$ and $\forall i = 1, \dots, n$, $A_i \equiv A'_i$. Two configurations are said *congruent*, $(X;c;\Gamma) \equiv (X';c';\Gamma')$, when the sets X and X' are equal, the constraints c and c' are $\mathcal{C}$-equivalent, and the multisets of agents Γ and Γ' are congruent. The *transition* relation $\longrightarrow$ is the least transitive relation on configurations satisfying the rules of table 1.

2.1.2 LCC logical semantics

LCC programs have a simple semantics in linear logic. In this section we give translations of LCC agents by logical formulae which are sound and complete for the observations of success stores, accessible stores and terminal stores.

Let us fix a constraint system $(\mathcal{C}, \Vdash_{\mathcal{C}})$ and a set of declarations $\mathcal{D}$.

Definition 2.2 *LCC agents are translated into intuitionistic linear logic formulas in the following way:*

$$\begin{array}{ll} tell(c)^\dagger = c & (A+B)^\dagger = A^\dagger \,\&\, B^\dagger \\ p(\vec{x})^\dagger = p(\vec{x}) & (\exists x A)^\dagger = \exists x A^\dagger \\ \forall x(c \to A)^\dagger = \forall x(c \multimap A^\dagger) & (A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \end{array}$$

If Γ *is the multiset of agents* $(A_1 \dots A_n)$*, one defines* $\Gamma^\dagger = A_1^\dagger \otimes \cdots \otimes A_n^\dagger$*. If* $\Gamma = \emptyset$ *then* $\Gamma^\dagger = 1$*.*

The translation $(X; c; \Gamma)^\dagger$ *of a configuration* $(X; c; \Gamma)$ *is the formula* $\exists X \Gamma^\dagger$*.*

ILL$(\mathcal{C}, \mathcal{D})$ denotes the deduction system obtained by adding to ILL:

- the non-logical axiom $c \vdash d$ for every $c \vdash_{\mathcal{C}} d$ in $\vdash_{\mathcal{C}}$,
- the non-logical axiom $p(\vec{x}) \vdash A^\dagger$ for every declaration $p(\vec{x}) = A$ in $\mathcal{D}$.

Theorem 2.3 (Soundness) *Let* $(X; c; \Gamma)$ *and* $(Y; d; \Delta)$ *be LCC configurations.*
If $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *then* $(X; c; \Gamma)^\dagger \dashv\vdash_{ILL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\dagger$*.*
If $(X; c; \Gamma) \longrightarrow^* (Y; d; \Delta)$ *then* $(X; c; \Gamma)^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\dagger$*.*

The reader can refer to [5, 13] for the results concerning completeness with respect to different observables (stores, successes, suspensions, etc...) and for the proofs of all the theorems.

2.2 Phase semantics based proofs

2.2.1 Phase semantics of intuitionistic linear logic

Phase semantics is the natural provability semantics of linear logic [6]. We only need here a fragment of intuitionistic linear logic ($\otimes$, $\&$ and $\multimap$, which correspond respectively to the parallel, choice and blocking ask operators, as shown in section 2.1). Nevertheless it is simpler to recall Okada's definition of the phase semantics for full intuitionistic LL [9] and to extend it to constants ($\mathbf{1}$, $\mathbf{0}$, $\top$).

Definition 2.4 *A* phase space $\mathbf{P} = (P, \cdot, 1, \mathcal{F})$ *is a commutative monoid* $(P, \cdot, 1)$ *together with a set* $\mathcal{F}$ *of subsets of* P*, whose elements are called* facts*, satisfying the following closure properties:*

– $\mathcal{F}$ *is closed under arbitrary intersection,*

– for all $A \subset P$*, for all* $F \in \mathcal{F}$*, the set* $\{x \in P : \forall a \in A, a \cdot x \in F\}$ *is a fact of* $\mathcal{F}$*, noted* $A \multimap F$*.*

As we shall see, facts correspond to ILL formulas and thus to LCC agents (cf. section 2.1).

Note that facts are closed under linear implication $\multimap$. Here are a few noticeable facts: the greatest fact $\top = P$, the smallest fact $\mathbf{0}$, and $\mathbf{1} = \bigcap\{F \in \mathcal{F} : 1 \in F\}$.

A *parametric fact* A is a total function from V to $\mathcal{F}$ assigning to each variable x a fact $A(x)$. Any fact can be seen as a constant parametric fact, and any operation defined on facts can be extended to parametric facts: $(A \star B)(x) = A(x) \star B(x)$.

Let A, B be (parametric) facts, define the following facts:

$$\begin{aligned}
A \mathbin{\&} B &= A \cap B, \\
A \otimes B &= \bigcap\{F \in \mathcal{F} : A \cdot B \subset F\}, \\
A \oplus B &= \bigcap\{F \in \mathcal{F} : A \cup B \subset F\}, \\
\exists x A &= \bigcap\{F \in \mathcal{F} : (\bigcup_{x \in V} A(x)) \subset F\}, \\
\forall x A &= \bigcap\{F \in \mathcal{F} : (\bigcap_{x \in V} A(x)) \subset F\}.
\end{aligned}$$

Definition 2.5 *An* enriched phase space *is a phase space* $(P, \cdot, 1, \mathcal{F})$ *together with a subset* O *of* $\mathcal{F}$*, whose elements are called* open facts*, such that:*

– O *is closed under arbitrary* $\oplus$ *(in particular there is a greatest open fact),*

– $\mathbf{1}$ *is the greatest open fact,*

– O *is closed under finite* $\otimes$*,*

– $\otimes$ *is idempotent on* O *(if* $A \in O$ *then* $A \otimes A = A$*).*

$!A$ *is defined as the greatest open fact contained in* A*.*

The set of facts has been provided with operators corresponding to ILL connectives (and therefore to LCC operators), we now translate formulas into facts.

Definition 2.6 *Given an enriched phase space, a* valuation *is a mapping* η *from atomic formulas to facts such that* $\eta(\top) = \top$*,* $\eta(\mathbf{1}) = \mathbf{1}$ *and* $\eta(\mathbf{0}) = \mathbf{0}$*.*

The interpretation $\eta(A)$ (resp. $\eta(\Gamma)$) of a formula A (resp. of a context Γ) is defined inductively in the obvious way:

$$\begin{aligned}
\eta(A \otimes B) &= \eta(A) \otimes \eta(B), \\
\eta(A \multimap B) &= \eta(A) \multimap \eta(B), \\
\eta(!A) &= !\eta(A), \\
\eta(A \mathbin{\&} B) &= \eta(A) \mathbin{\&} \eta(B), \\
\eta(A \oplus B) &= \eta(A) \oplus \eta(B), \\
\eta((\Gamma, \Delta)) &= \eta(\Gamma) \otimes \eta(\Delta), \\
\eta(\forall x A) &= \forall x \eta(A), \\
\eta(\exists x A) &= \exists x \eta(A), \\
\eta(\Gamma) &= \mathbf{1} \text{ if } \Gamma \text{ is empty.}
\end{aligned}$$

Sequents are interpreted as follows: $\eta(\Gamma \vdash A) = \eta(\Gamma) \multimap \eta(A)$. This brings one to defining a notion of validity:

Definition 2.7 (Validity) *Define:*

$\mathbf{P}, \eta \models (\Gamma \vdash A)$ *iff* $1 \in \eta(\Gamma \vdash A)$, *i.e.* $\eta(\Gamma) \subset \eta(A)$,
$\mathbf{P} \models (\Gamma \vdash A)$ *iff for every valuation* η: $\mathbf{P}, \eta \models (\Gamma \vdash A)$,
$\models (\Gamma \vdash A)$ *iff for every phase space* $\mathbf{P}$: $\mathbf{P} \models (\Gamma \vdash A)$.

This semantics of ILL formulas enjoys the following main properties:

Theorem 2.8 (Soundness [6, 9]) *If there is a sequent calculus proof of* $\Gamma \vdash A$ *then* $\models (\Gamma \vdash A)$.

Theorem 2.9 (Completeness [6, 9]) *If* $\models (\Gamma \vdash A)$ *then there is a sequent calculus proof of* $\Gamma \vdash A$.

2.2.2 Proving safety properties of LCC programs with the phase semantics

Using the phase semantics presented above we can prove safety properties of LCC programs. The theorem 2.8 of soundness of the phase semantics w.r.t. ILL can easily be extended to $ILL_{\mathcal{C},\mathcal{D}}$ by imposing to any valuation η to satisfy the inclusions coming from the non-logical axioms (the axiom $c \vdash d$ imposes $\eta(c) \subset \eta(d)$).

By contrapositive we get:

$$\exists \mathbf{P}, \eta, \text{ s.t. } \mathbf{P}, \eta \not\models (\Gamma \vdash A) \text{ implies } \Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A,$$

which is equivalent to:

$$\exists \mathbf{P}, \eta, \text{ s.t. } \eta(\Gamma) \not\subset \eta(A) \text{ implies } \Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A.$$

As the contrapositive of the theorem 2.3 of soundness from LCC to $ILL_{\mathcal{C},\mathcal{D}}$ is:

$$(X; c; \Gamma)^{\dagger} \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} (Y; d; \Delta)^{\dagger} \text{ implies } (X; c; \Gamma) \not\longrightarrow (Y; d; \Delta)$$

We have:

Proposition 2.10 *To prove a safety property of the kind:* $(X; c; \Gamma) \not\longrightarrow (Y; d; \Delta)$, *it is enough to show that:*
$\exists$ *a phase space* $\mathbf{P}$, *a valuation* η, *and an element* $a \in \eta((X; c; \Gamma)^{\dagger})$ *such that* $a \notin \eta((Y; d; \Delta)^{\dagger})$.

This proposition allows to reduce the problem of proving safety properties of LCC programs, i.e. proving the non-existence of some derivation, to an existence problem: finding a phase structure, an interpretation and a counter-example for the above inclusion, or even, only proving their existence. Note that only soundness theorems are used, the second part of the correspondence (completeness) gives a certain certitude that when looking for a semantical proof of a true safety property, it exists!

3 Singleton Based Phase Model Checking

In [4] we used the phase semantics of Linear Logic to devise the new method that we just summarized and it revealed useful, especially for proving safety properties of LCC programs. We now describe how it can be implemented with some restrictions using efficient constraint programming techniques.

Recall that, to prove a property such as A never reaches a store containing c, the soundness of the linear logic semantics allowed us to look for a countermodel of $A \vdash c \otimes \top$. Any calculus where properties can be expressed in that way can then rely on the soundness of the phase semantics to ensure that it is enough to exhibit a phase space and an interpretation of formulae (agents and constraints) into facts, compatible with the non-logical axioms, and such that $[\![A]\!] \not\subset [\![c]\!]$ (see section 2.2).

Our approach to finding the phase space and the valuation is now based on a few remarks:

- We want to find everything *statically*, so we will not use any of the methods like those of Okada and Terui [10], which generate the phase space from a proof search, thus an execution.
- We want to enumerate some spaces, but we cannot (and do not want to) try them all (recall that one of the spaces is that of the proofs, so using it is equivalent to looking at the proof directly). We thus restrict our search to some *special* kinds of phase spaces, where we will be able to reason easily (to prove the non-inclusion). This process can be seen as some kind of abstract interpretation of our proof search.
- In [14] Yilma tried this kind of approach, restricting himself to *small* and *cyclic* structures. But then he had to enumerate all the possible valuations to find the counter-example for the inclusion.
- Almost all the examples that we treated (in [4] and later) have solutions in very simple phase structures, where all sets are facts and the valuation associates only singletons to formulae.

These remarks, and especially the last one, which transforms inclusion conditions to equalities, led us to use the constraint logic programming tools as a basis for our *phase model checker*.

3.1 Implementation

Our implementation of the phase model checker uses $\mathbb{N}$ as the basic monoid for all our phase structures, not only because it is what was used in the examples of [4], but also because it builds, in some sense, the free monoid based on the conditions of the non-logical axioms. The prime numbers will thus be a good basis for our interpretation.

The system then searches for a valuation, i.e. it associates an integer to each atomic formula, such that the equalities coming from the non-logical axioms are respected. In other words, these conditions are equality constraints on the variables associated with each atomic formula. The non-inclusion being the only dis-equality constraint.

An implementation of these ideas was realized, using the constraint logic programming language GNU-Prolog with constraint solvers over finite domains.

3.2 Examples

Here is a non-recursive version of the dining philosophers program in LCC.

```
goal :- phil1,phil2,phil3,phil4,phil5,fork1,fork2,fork3,fork4,fork5.
phil1 :- fork1,fork2 -> eat1,(eat1 -> fork1,fork2,phil1).
phil2 :- fork2,fork3 -> eat2,(eat2 -> fork2,fork3,phil2).
...
phil5 :- fork5,fork1 -> eat5,(eat5 -> fork5,fork1,phil5).
```

We note `','` for $\otimes$ and $\|$, depending on position, and `'->'` for $\rightarrow$. This ASCII version of LCC syntax is parsed directly by the model checker.

Here is the result of a simple execution, checking that philosophers 1 and 2 can never eat at the same time:

```
| ?- find_phase('philo_5',[goal],[([eat1,eat2],top)]).
eat5=1 eat3=1 eat1=2  fork4=1 fork2=2 phil5=1 phil3=1 phil1=1
eat4=1 eat2=2 fork5=1 fork3=1 fork1=1 phil4=1 phil2=1 goal=2
```

Actually here is a more detailed explanation of what happens: 16 variables corresponding to atomic constraints (5 for each `phil`, `fork` and `eat` plus one for `goal`) are declared; 10 other variables corresponding to *asks* are declared; 16 equality constraints are given to the solver: one for each *ask* and one for each declaration; finally the dis-equality constraint corresponding to the safety property is added.

If we had n philosophers, we would thus have $5n+1$ variables, with $3n+1$ equality constraints and one dis-equality. You can see at the end of this paper results for bigger instances of this problem.

Note that a simple constraint such as `A #\= B` would only allow to check properties as $A \not\vdash B$, however we often want to check if $A \not\vdash B \otimes \top$. Therefore we have to rely on integer arithmetic to notice that if m, n and p are integers $\forall p, m \neq n \cdot p$ is equivalent to m *modulo* $n \neq 0$, and GNU-Prolog's FD solver nows how to handle these as `A rem B #\= 0`.

4 Going farther

4.1 Coping with incompleteness

The method described here is however not complete for different reasons:

One is that if we are based on finite domains, we will never be able to handle properly programs that have an unbounded parameter. In some cases we will be able to collapse all the values after some bound, but for instance if you take the dining philosophers with `N` not given, it is very hard to find a phase space as that given in [4], as it relies on an infinite number of primes.

Another reason is linked with the choice to consider only singletons. This has the important drawback of creating confusions. For instance with the following program: `P = tell(d)`, `Q = c` $\rightarrow$ `P`, a singleton-based phase structure does not allow one to prove that `c` is not accessible from `P`, i.e. `P` $\not\vdash$ `c` $\otimes \top$, as we can deduce $[\![\mathtt{P}]\!] = [\![\mathtt{c}]\!] \cdot [\![\mathtt{Q}]\!]$ from the second declaration.

Finally, all integers are invertible, that is $m \cdot p = n \cdot p$ implies $m = n$. Thus we cannot capture the idea that in the first case the *presence* of p is necessary to go from m to n. Therefore we cannot prove much for programs such as Peterson's mutual exclusion algorithm given below, as there is a *check* on the value of a variable that may not change its value but allows a process to go from `off` to `on` (see below).

```
p1 :- off1,turn1 -> on1,turn2,wait1.
p1 :- off1,turn2 -> on1,turn2,wait1.
wait1 :- off2 -> off2,cs1,(cs1,on1 -> off1,p1).
wait1 :- turn1 -> turn1,cs1,(cs1,on1 -> off1,p1).
(same for p2 and wait2)
init :- off1,off2,turn1,p1,p2.
```

It is important to note however that one can cope with all these problems to show some important properties, by simply using a more precise way of expressing what is to be shown. Let us consider again the previous example of Peterson's mutual exclusion algorithm. It is shown in [14] that even if it is not provable directly that $\mathtt{init} \not\longrightarrow^* \mathtt{cs1} \otimes \mathtt{cs2} \otimes \top$, one can use the phase semantics to prove it by remarking that the only four ways to reach such a state are states of the following forms: $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{turn1} \otimes \mathtt{turn2} \otimes \top$, $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{turn1} \otimes \mathtt{off1} \otimes \top$, $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{off2} \otimes \mathtt{turn2} \otimes \top$ and $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{off1} \otimes \mathtt{off2} \otimes \top$.

The phase model checker can then compute:

```
| ?- find_phase('peterson',[init],[([on1,on2,turn1,turn2],top),
| ?- ([on1,on2,turn1,off1],top),([on1,on2,off2,turn2],top),
| ?- ([on1,on2,off1,off2],top)]).
init=8 wait2=1 p2=1  off2=2  turn2=2 off1=2  p1=1
cs2=1  on2=2   cs1=1 wait1=1 on1=2   turn1=2
```

4.2 Doubletons and finite sets

We have seen above that almost all the limits of our method can be coped with, however the singleton restriction might sometimes be too restrictive, for instance in the very simple producer/consumer protocol given below:

```
p :- dem -> pro,p.                    c :- pro -> dem,c.
init :- dem,dem,dem,p,p,p,p,p,c,c.
```

One can check that it is impossible to prove with singletons that a producer cannot consume, i.e. $p \parallel pro \not\longrightarrow dem \parallel \Gamma$, as the definition of p gives $p \cdot pro = p \cdot dem$.

Solvers on sets would allow us to solve this problem, it is however noticeable that a simple modification of our solver allows it to handle *doubletons* by simply using GNU-Prolog's list constructor, and that we thus already have a more general solver.

```
| ?- find_dbl('prod_cons',[p,pro],[([dem],top)]).
p=[1,1] pro=[2,1] dem=[2,2] c=[0,0]
```

Such a modification should allow us to use bounded lists of a size much bigger than two, however our solver gets really slow and the lack of a specialized solver is severely felt.

5 Experimental results

The following table sums up our experimental results. The size of the example indicates the number of initial agents. The numbers of variables and constraints indicate the size of the constraint satisfaction problem issued from the phase model checker.

protocol	size	number of variables	number of constraints	time for proving safety
prod./cons. (doubletons)	2	12	5	3850 ms
dining philosophers	5	26	16	480 ms
dining philosophers	10	51	31	3470 ms
dining philosophers	20	101	61	44730 ms
Peterson's algorithm	2	25	25	1420 ms

6 Conclusion

To be efficient, program verification needs abstractions in order to get rid of useless execution details. The semantics of concurrent constraint programs in linear logic and in its model theory of provability, namely phase semantics, provides an abstract interpretation of LCC programs which is given by the logic of LCC agents that is built-in inside a phase model checker that we have implemented with some restrictions.

We have shown that these restrictions, while making it possible to use the constraint programming technology to search for phase models with arithmetic constraints, still allow us to find automatically some non-trivial proofs of safety properties of LCC programs. The efficiency issue will of course need more work than our first prototype implementation, which did not even try to guide the interpreter during the constraint solving phase, but the technology already available in constraint solvers gives us great hope for improvement.

Finally, as the only part really relevant to LCC in this study was the expressibility, through its semantics, of safety properties in linear logic, we believe that our method can thus be used for other calculi based on linear logic and hope to extend concretely the prover in that direction in the future. The first extension is that of handling the family of logic programming languages based on linear logic, like LO [1], Lolli [8], or Lygon [7], where only a new parser is needed (the properties are already expressed as linear logic formulae). Other coordination languages or rule based languages may also have a sound linear logic semantics, so there is still a lot of experimentation to do.

References

[1] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.

[3] E. Best, F.S. de Boer, and C. Palamidessi. Concurrent constraint programming with information removal. In *Proceedings of Coordination*, LNCS. Springer-Verlag, 1997.

[4] F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proc. 13th Annual IEEE Symposium on Logic in Computer Science, Indianapolis*, 1998.

[5] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 164, 2001. Available from http://pauillac.inria.fr/~fages/Papers/FRS01ic.ps.

[6] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[7] J. Harland, D. Pym, and M. Winikoff. Programming in lygon: An overview. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, Munich*, pages 391–405, July 1996.

[8] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[9] M. Okada. Girard's phase semantics and a higher-order cut-elimination proof. Technical report, Institut de Mathématiques de Luminy, 1994.

[10] M. Okada and K. Terui. Completeness proofs for linear logic based on the proof search method (preliminary report). In J. Garrigue, editor, *Type theory and its applications to computer systems*, pages 57–75. Research Institute for Mathematical Sciences, Kyoto University, 1998.

[11] V.A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[12] V.A. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

[13] S. Soliman. *Programmation concurrente avec contraintes et logique linéaire*. PhD thesis, Université Paris 7, Denis Diderot, 2001.

[14] Y. Yilma. Non-monotonic concurrent constraint programming verification using phase semantics. Master's thesis, School of Information Technology and Engineering, University of Ottawa, 1999.

The Design and Implementation of a Compositional Competition-Cooperation Parallel ATP System

Geoff Sutcliffe
Department of Computer Science, University of Miami, USA
Email: geoff@cs.miami.edu

Abstract

Key concerns in the development of more powerful ATP systems are to provide *breadth* of coverage – an ability to solve a large range of problems, and to provide greater *depth* of coverage – an ability to solve more difficult problems, within the same resource limits. This work describes the design and implementation of CSSCPA, a compositional competition-cooperation parallel ATP System. CSSCPA combines existing high performance ATP systems in a framework that allows them to work independently, but also allows communication of intermediate results. The performance data shows that CSSPCA has high breadth and depth of coverage.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development and use of systems that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. Current ATP systems are capable of solving non-trivial problems, e.g., EQP [McC00a] solved the Robbins problem [McC97]. However, the search complexity of most interesting problems is enormous, which has two consequences for ATP. First, in order to solve certain types of hard problems, it is typically necessary to tune an ATP system for the problems. Such tuning almost inevitably has the consequence that the system can no longer solve some other problems, i.e., gain in one direction is at the cost of loss in another.[1] Second, there are many problems that still cannot currently be solved within realistic resource limits. Therefore, key concerns in the development of more powerful ATP systems are to provide *breadth* of coverage – an ability to

[1]It would be marvelous if the characteristics of ATP problems were sufficient to correctly identify every type of problem for which tuning has resulted in successful solution, for then the appropriately tuned features of an ATP system could be automatically invoked when the type of problem is recognized. However, thus far such recognition seems impossible.

solve a large range of problems, and to provide greater *depth* of coverage – an ability to solve more difficult problems, within the same resource limits.

One approach to providing breadth of coverage is the development of *compositional* ATP systems. Compositional systems are built from multiple component systems, and use one or more of the components when attempting to solve a problem. Compositional systems may be characterized by the way in which the components are run: *Time slicing* systems select one or more components, allocate some fraction of the available CPU time to each of the selected components, and then run the components one after the other until a solution is found. Examples of time slicing systems are Gandalf [Tam97] and more recent versions of Vampire [RV99]. *Competition* systems similarly select components and allocate CPU time, but then run the components in parallel (or at least concurrently, according to the number of CPUs available) until a solution is found. Examples of competition systems are RCTHEO [Ert92] and SSCPA [SS99].

Both time slicing and competition systems rely on the phenomenon that components can be selected so that there is a significant difference in the set of problems that each can solve quickly – the properties of sub-linearity and complementarity [SW99]. Time slicing systems have the advantage that they are inherently well suited to single CPU machines, and the components can be given 'dedicated' access to the CPU in the order of likelihood of solving the problem. Competition systems have the advantage that they can take advantage of multiple CPU architectures (especially SMP machines), and there is no need to decide which components are more likely to solve the problem. For both time slicing and competition systems, greater diversity across the components provides greater breadth of coverage.

The components of a simple compositional system do not cooperate, with no communication of control information or intermediate results. Such compositional systems can solve at most the union of the problems that the individual components can solve within the same total CPU time limit, i.e., there is no gain in depth of coverage (and some problems that can be solved by individual components within the full CPU time limit may not be solved within the fraction of the CPU time limit allocated to the components in the compositional setting). The capabilities of a compositional system can be significantly affected by the addition of cooperation. The communication of control information is problematic if the components are diverse (as recommended above), because they have different search spaces and the control information from one component is typically inappropriate for another. One coarse grained way of effecting the communication of control information is to have components with different control strategies, and to start and stop the components according to evidence of their success in the context of the overall system. This approach is taken in the DISCOUNT system [ADF95].

The communication of intermediate results is significantly easier. For time slicing systems, the intermediate results generated by an unsuccessful component in the sequence are passed on to subsequent components. Examples of time-slicing-cooperation systems are Gandalf [Tam97] and the e-iterator strategy within E-SETHEO [SW99]. For competition systems, a protocol has to be established to allow communication of intermediate results during runtime. For competition systems in particular, communication between diverse component systems often has synergistic effects, leading to "super linear" speed ups. This is due to cross fertilization between the components, as a component may receive useful intermediate results that it would not generate itself. Examples of competition-cooperation systems are HPDS [Sut92] and TECHS [DF99]. The addition of cooperation to a compositional system is an effective way of increasing the depth of coverage of the system. The synergistic effects allow the system to solve problems that none of the component systems are able to solve independently. As before, diversity is important, as this provides more extreme cross fertilization.

The above survey suggests that a *C*ompetition-*C*ooperation *C*ompositional system, running genuinely in *P*arallel on a multi-CPU machine (a *CCCP* system), has high prospects for attaining both breadth and depth of coverage.[2] The following sections of this paper examine a particular instance of the design and implementation of a CCCP system. Section 2 looks at the issues and choices in the design of such a system. Sections 3 and 4 describe the design and implementation of the CSSCPA[3] system, highlighting the benefits of the design decisions made, and difficulties encountered. Section 5 provides performance data, and Section 6 concludes the paper.

2 Design Issues for a CCCP ATP System

There are three main issues that need to be addressed in the design of a CCCP system:

- How the components will be controlled and monitored. Aspects of this include how the components will be started, how the resource usage of the components will be allocated and limited, and how the components will be stopped when a solution is found or the resource limits are exceeded.
- What data formats will be used. The data formats for the input problem, the intermediate results that are communicated, and the output, all need to be considered.
- The granularity and mode of communication between the components. Options here range from fine grained point-to-point communication, where small intermediate results are transferred immediately and directly to other components, through to coarse grained bulk transfer of many intermediate results at widely spaced intervals.

The choices for each of these issues are constrained to a large degree by the nature of the component systems. There are three common types of component systems, each with quite different characteristics, which affect the range of options available in the design.

[2] Other approaches to using parallelism in ATP are surveyed in [SS94] and [Bon00].

[3] Pronounced "sea skipper".

The highest degree of design flexibility is available when the components are designed and developed in-house. In this case there is access to and understanding of the design and implementation of the components, and it is easily possible to make adaptations in the components specifically for the CCCP system. From the control perspective, it is possible to directly manage the component processes, and to have the components internally limit their resource usage. A single data format can be designed and used consistently for input, communication, and output. Both fine and coarse grained communication of intermediate results are possible. In particular, if a common data format is used, there is no overhead of format conversion, which is particularly attractive if the communication between components is fine grained. The drawback of using in-house components is that significant effort has to be expended in order to design and implement components that have sufficiently high performance. In many cases there is simply not enough expertise and programmer-power to achieve this. HPDS is an example of a CCCP system that benefited from the advantages and also suffered from the disadvantages of this approach.

An intermediate degree of design flexibility is available when using high performance components developed elsewhere, but for which the source code is available and understandable. Such components can be modified as required to run in the CCCP system. Modifications can be made to make component control easily possible, internal translation of data formats can be implemented, and communication hooks can be inserted. The main advantage of this option is the adoption of existing high performance systems, which may have required significant effort and expertise to develop. However, the effort required to make the necessary modifications to someone else's code is often prohibitive. Further, as new versions of the components are released by their developers, it is necessary to port the modifications to the new versions. As a result, it is difficult to keep such a CCCP system upgraded to the most recent component technology. TECHS is an example of such a CCCP system.

The least design flexibility is available when using high performance components developed elsewhere, without any intention of making modifications. There are significant advantages and disadvantages of this approach. Controlling the execution and monitoring the CPU usage of a component may be difficult, especially if the component runs multiple processes. Although starting a component may be easy, it may then be difficult to monitor and limit its resource usage, or to stop all processes of the component.[4] The data formats of the components are likely to be incompatible, and it is necessary to implement external format translation for at least the input and communication data. Almost certainly the components will not accept intermediate results during runtime, thus making a coarse grained communication model necessary. These drawbacks require solutions at both the logical and practical levels. If solutions

[4]For example, under UNIX, if a process in the middle of a three level process hierarchy terminates, leaving the bottom level process to communicate with the top level process using files, the bottom process is no longer in the process hierarchy of the top process. It is then difficult to stop the bottom level process, and the CPU time of the bottom process is not accumulated as child CPU usage in the top level process.

can be found, however, there are some attractive aspects to this approach, all stemming from the fact that the components are used without modification. First, the highest performance components can be used, and their performance will not be degraded through modifications required to fit them into the CCCP system. Second, it is not necessary to have access to the components' sources. This makes it possible to use components that for some reason, e.g., proprietary constraints, can be distributed only in binary form. Third, as the CCCP system can be concerned with only the external presentation of the components, it is likely to be easy to replace a component by a newer version. This is because the external presentation of a (component) system often remains the same while internal structures are (possibly significantly) changed.

3 The Design of CSSCPA

CSSCPA is a CCCP system for problems in the CNF of first order classical logic, expressed in the TPTP CNF syntax [SS98]. CSSCPA uses existing high performance components, without any modification. The components must have an option to produce, on their standard output, intermediate clauses that are logical consequences of the input problem. In addition to the component ATP systems, CSSCPA employs a formula librarian (the FLi) that can do subsumption and also detect unit contradictions in the clauses it holds.

When given an ATP problem, CSSCPA first sends a copy of the problem to the FLi, where it is stored. CSSCPA then selects components to use, based on a database of information about the eligible components' strengths for various problem types. CSSCPA starts the components, and parses their standard outputs for logical consequences. Each logical consequence is forwarded to the FLi. If at any time a component finds a solution, then CSSCPA is stopped and success is reported.

The FLi keeps an incoming clause only if keeping it improves the overall quality of its clause set, in the sense that a better clause set is one that is easier to refute. For example, the FLi's clause set is improved if an incoming logical consequence subsumes a clause in the set (see Section 4 for further details of the clause set evaluation). Whenever the quality of clause set in the FLi improves, the FLi reports the improvement to CSSCPA. When the clause set in the FLi has improved significantly relative to the original input problem, CSSCPA stops the components. CSSCPA then collects the improved clause set from the FLi, and restarts with the improved clause set as the input problem.

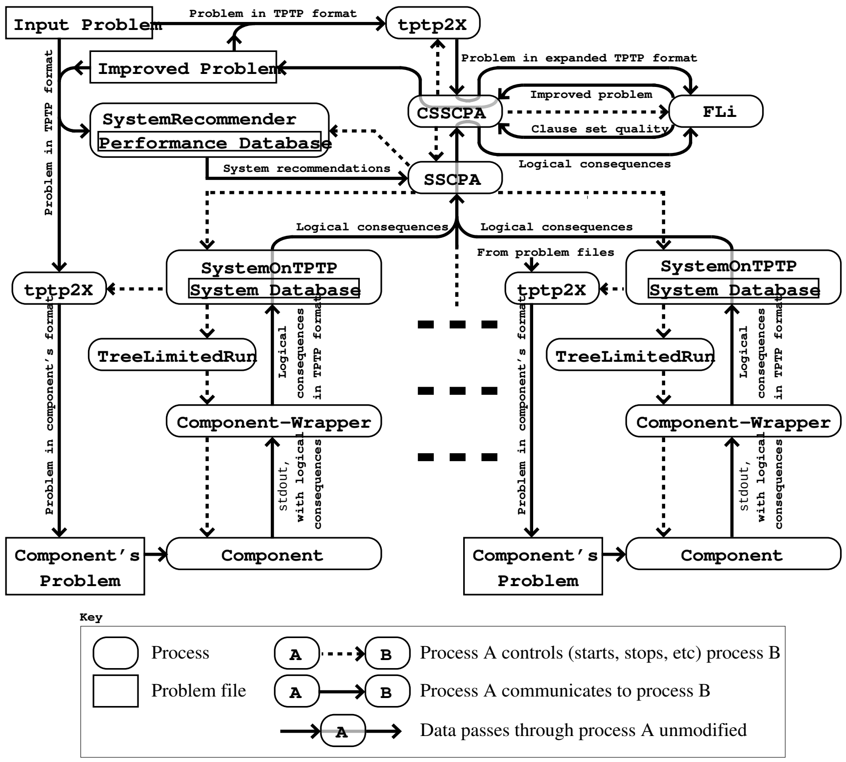
When a proof is found, CSSCPA outputs the original problem file, the sequence of improved problem files, and the component system's proof. The clauses in the improved problem files are annotated to indicate their source, either from the original input problem, or from one of the component ATP systems. This provides sufficient information to construct a monolithic proof, which can be checked using standard techniques.

CSCCPA creates a sequence of successively easier problems to solve. This technique is called *iterative easing*. The arrangement is clearly sound, provided that the component systems output only logical consequences. On a macro level CSSCPA may be viewed as a time-slicing compositional system, in which each CSSCPA iteration is one component system.

4 The Implementation of CSSCPA

The overall process architecture of the CSSCPA implementation is shown in Figure 1.

Figure 1: The CSSCPA Architecture



When given an ATP problem, CSSCPA uses the `tptp2X` utility to expand any `include` statements in the problem, and forwards the input problem clauses to the FLi. The FLi is an external module of the E ATP system [Sch01], and thus employs the efficient data structures and formula manipulation routines in the E implementation. When the FLi has received and stored all the input clauses, it reports the number of clauses, number of literals, and sum weight of the clause set, on its standard output. This information is captured by CSSCPA as the quality of the input problem clauses. The FLi then sits in a loop, reading clauses (the logical consequences from the component ATP systems) on its standard input, processing them, and reporting any improvements of the clause set quality on its standard output. CSSCPA monitors the standard output of the FLi. If the reported quality of the FLI's clause set passes a threshold, CSSCPA stops the execution of the component systems and collects the improved clause set from the standard output of the FLi. If the clause set does not contain a unit contradiction, CSSCPA restarts using the improved clause set as the input problem.

The selection and execution of components for CSSCPA is done by the SSCPA system [SS99]. To select components, SSCPA runs a `SystemRecommender` program. The `SystemRecommender` accesses a database of information about the eligible components' strengths. The database is built from an evaluation of performance data for the ATP systems on TPTP problems [SS01]. The database assigns problems to one of 16 Specialist Problem Classes (SPCs) based on syntactic problem characteristics, and ranks the systems within each SPC. When a new problem is presented, its SPC is identified and the best performing systems for the SPC are then known. SSCPA divides the CPU time remaining equally between the selected systems. SSCPA then invokes SystemOnTPTP [Sut00] to run each of the selected components on the problem. Note that the set of selected components may change between iterations in CSSCPA, due to changes in the characteristics of the clause set.

Each instance of SystemOnTPTP accesses a database of information about the ATP systems to determine the format in which its ATP system requires the problem. The SystemOnTPTP then uses the `tptp2X` utility to do the necessary transformations and formatting.

SystemOnTPTP uses a control process (called `TreeLimitedRun`) to control and monitor its ATP system. The control process starts the ATP system, monitors the resource usage of the system, imposes resource usage limits, and has sufficient information to be able to stop all the processes that the system has running when a resource limit is exceeded (the CPU time allocated, a wall clock time limit, and a memory limit). The control program monitors the CPU usage of the ATP system's processes by scanning the `/proc` file system. The wall clock limit is implemented by an `alarm` system call within the control process. The memory limit is imposed through use of the `setrlimit` system call. When the CPU or wall clock time limit is reached, the control process scans the `/proc` file system for the system's processes, and uses a `kill` system call to stop them all.

The control process runs the specified ATP system inside a component wrapper. The wrapper scans the standard output of the ATP system and extracts clauses that are logical consequences of the input problem. The wrapper then translates the clauses to TPTP format before writing them to its standard output. The standard output of the wrapper is captured by the corresponding instance of SystemOnTPTP, which echoes the information to its standard output. SSCPA collates the standard outputs from the SystemOnTPTP instances, and writes them to its standard output. This is captured by CSSCPA, and the logical consequences are then forwarded to the standard input of the FLi.

In the current implementation of CSSCPA, the quality of the FLi's clause set is measured in two ways: the total weight (symbol count) of the clause set, and the average clause weight. The quality of the clause set improves when either of these decreases. The total weight decreases when an incoming clause subsumes clauses whose sum weight is greater than that of the incoming clause. The average clause weight decreases when an incoming clause's weight is less than the current average clause weight. The clause set in the FLi is considered to have "improved significantly" when either of the quality measures goes down by some fraction of the input clause set's measures. The fractions are parameters to CSSCPA.

CSSCPA, SSCPA, SystemOnTPTP, and the component wrappers are all implemented in `perl`. The FLi and `TreeLimitedRun` are implemented in C. `tptp2X` is implemented in Prolog. The `perl` implementation of key components may be a bottleneck in the communication, but at this stage it seems to be acceptable. It is noteworthy that all inter-process communication uses the standard input and output streams. At the bottom level, this is necessary for capturing the logical consequences from the component ATP systems, given the commitment to using unmodified components. The decision to use standard IO streams for the other levels followed as a consequence.

5 Performance

Initial testing of CSSCPA has been done using the 1745 TPTP problems that are non-Horn, have some (but not only) equality literals, and have an infinite Herbrand universe (i.e., a very general class of problems). The same three components were selected all the time, they being SPASS 1.03 [WAB+99], E 0.62 [Sch01], and Otter 3.0.6 [McC00b], all running in their default "auto" modes (splitting was turned off in SPASS, so that only logical consequences were generated). A 300 second time limit was imposed, individually when testing the individual component systems, and as a total in the CSSCPA setting. Table 1 summarizes the results. The SSCPA column shows the results for the naive mode of SSCPA, in which the three components are run in competition parallel with a CPU time limit of 100 seconds for each component. The SSCPA* column shows the SSCPA results with a CPU time limit of 300 seconds for each component.

Table 1: CSSCPA Results

	CSSCPA	E	Otter	SPASS	SSCPA	SSCPA*
Solved by	686	616	364	630	673	723
CSSCPA, not by other		131	329	91	75	52
other, not by CSSCPA		65	11	39	62	89

CSSCPA solved 52 problems that none of the components solved within 300 seconds, and 25 `SWC` problems with a TPTP difficulty rating of 1.00, i.e., 25 problems that no existing ATP system is known to be able to solve. These results show that CSSCPA has high depth of coverage. CSSCPA solves more problems than any component, and 13 more problems than their composition in SSCPA. These results show that CSSCPA has high breath of coverage (although it would be even better if CSSCPA subumed (solved a superset of the problems solved by) the components and SSCPA).

It is interesting that there are 62 problems solved by SSCPA but not by CSSCPA. The essential differences between SSCPA and CSSCPA are the communication of intermediate results and a reduction of the CPU time limit on each component in successive CSSCPA iterations. Clearly the "improvements" in the problem and the reduced time limits affect the components' abilities to solve those problems.

It should be noted that CSSCPA's performance on a given problem can change from run to run, due to changes in the operating system's scheduling of the component ATP systems, which affects the order in which logical consequences are forwarded to the FLi.

6 Conclusion

The need for powerful ATP systems that have both breadth and depth of coverage has motivated the design and implementation of the compositional competition-cooperation parallel ATP system CSSCPA. CSSCPA combines existing high performance ATP systems in a framework that allows them to work independently, but also allows communication of intermediate results. The performance data shows that CSSPCA has high breadth and depth of coverage.

It is planned to extend the range of component systems available to CSSCPA. In particular, the use of analytic provers, e.g., model elimination or tableau based provers, with lemma generation capabilities, seems attractive. It is expected that there will be strong cross fertilization between saturation systems and analytic systems, due to their different deduction and search strategies.

The soundness of CSSCPA is dependent on the soundness of the components, and also (from a practical viewpoint) the correct capturing and forwarding of logical consequences. It is planned to independently verify CSSCPA proofs by converting the sequence of improved problems into a monolithic proof, and applying standard proof checking techniques.

Acknowledgement: Thanks to Stephan Schulz for implementing the FLi.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In Hsiang J., editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, pages 397–402. Springer-Verlag, 1995.

[Bon00] M.P. Bonacina. A Taxonomy of Parallel Strategies for Deduction. *Annals of Mathematics and Artificial Intelligence*, 29:223–257, 2000.

[DF99] J. Denzinger and D. Fuchs. Cooperation of Heterogeneous Provers. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence* , pages 10–15. Morgan Kaufmann, 1999.

[Ert92] W. Ertel. OR-Parallel Theorem Proving with Random Competition. In Voronkov A., editor, *Proceedings of the 1992 Conference on Logic Programming and Automated Reasoning* , number 624 in Lecture Notes in Artificial Intelligence, pages 226–237. Springer-Verlag, 1992.

[McC97] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McC00a] W.W. McCune. EQP: Equational Prover. http://www-unix.mcs.anl.gov/AR/eqp/, 2000.

[McC00b] W.W. McCune. Otter: An Automated Deduction System. http://www-unix.mcs.anl.gov/AR/otter/, 2000.

[RV99] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 292–296. Springer-Verlag, 1999.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 370–375. Springer-Verlag, 2001.

[SS94] C.B. Suttner and J. Schumann. Parallel Automated Theorem Proving. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence 1*, pages 209–257. Elsevier Science, 1994.

[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.

[SS99] G. Sutcliffe and D. Seyfang. Smart Selective Competition Parallelism ATP. In A. Kumar and I. Russell, editors, *Proceedings of the 12th Florida Artificial Intelligence Research Symposium*, pages 341–345. AAAI Press, 1999.

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

[Sut92] G. Sutcliffe. A Heterogeneous Parallel Deduction System. In R. Hasegawa and M.E. Stickel, editors, *Proceedings of the Workshop on Automated Deduction: Logic Programming and Parallel Computing Approaches, FGCS'92*, 1992.

[Sut00] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.

[SW99] G. Stenz and A. Wolf. E-SETHEO: Design, Configuration and Use of a Parallel Automated Theorem Prover. In N. Foo, editor, *Proceedings of AI'99: The 12th Australian Joint Conference on Artificial Intelligence*, number 1747 in Lecture Notes in Artificial Intelligence, pages 231–243. Springer-Verlag, 1999.

[Tam97] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[WAB+99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Tpoic. System Description: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 378–382. Springer-Verlag, 1999.