First-order Theorem Proving FTP 2009

International Workshop on First-Order Theorem Proving
Oslo, Norway, July 2009
Proceedings

Nicolas Peltier and Viorica Sofronie-Stokkermans (eds.)

Preface

This volume contains the papers presented at the International Workshop on First-Order Theorem Proving (FTP 2009) held in Oslo, Norway, on July 6–7, 2009. First-order theorem proving is widely recognized as a core theme of automated deduction and has achieved considerable successes in the last decades. FTP 2009 is the seventh in a series of workshops intended to focus effort on first-order theorem proving by providing a forum for presentation of recent work and discussion of research in progress. The FTP workshop is held since 1997; its aim is to bring together researchers interested in all aspects of first-order theorem proving. It welcomes original contributions on theorem proving in first-order classical, many-valued, modal and description logics. Previous editions of FTP took place in Schloss Hagenberg, Austria (1997); Vienna, Austria (1998); St Andrews, Scotland (2000); Valencia, Spain (2003); Koblenz, Germany (2005) and Liverpool, UK (2007).

FTP 2009 was held together with the 18th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2009, web page: tableaux09.ifi.uio.no). On July 7, 2009 there was a joint session with TABLEAUX 2009 with Peter Jeavons as (joint) invited speaker.

The technical program of FTP 2009 consists of two invited talks on by Silvio Ghilardi on "Model-Checking of Array-Based Systems: from Foundations to Implementation" and one by Peter Jeavons on "Presenting Constraints" (joint with Tableaux 2009), eight regular papers and two position papers. The topics of these papers match very well those of the workshop, ranging from the theoretical foundations of first-order theorem proving to practical applications, e.g. in verification and web technology.

Many people contributed to make this workshop possible and we sincerely thank all of them. First of all, we would like to thank all the scientists who submitted interesting papers and abstracts to FTP 2009 and the invited speakers for agreeing to speak at the workshop. Many thanks also to all the attendees for contributing to the intensive exchange of ideas in the workshop. We also thank all the members of the Program Committee and the additional reviewers for their excellent job and for their thorough and quick reviews. We are very grateful to the local organisers (in particular Roger Antonsen, Martin Giese and Arild Waaler) for their numerous advices, constant support and for taking care of practical matters. We thank the Norwegian Research Council and the Dept. of Informatics at the University of Oslo for their generous financial support. We also would like to thank the steering committee, in particular Ullrich Hustadt for their strong support to the FTP workshop series.

Nicolas Peltier and Viorica Sofronie-Stokkermans

Programme committee chairs

Nicolas Peltier (CNRS - Laboratory of Informatics of Grenoble)
Viorica Sofronie-Stokkermans (MPI für Informatik, Saarbrücken)

Programme committee

Alessandro Armando (DIST - University of Genova, Italy)
Franz Baader (TU Dresden, Germany)
Peter Baumgartner (National ICT Australia)
Bernhard Beckert (University of Koblenz, Germany)
Maria Paola Bonacina (Università degli Studi di Verona, Italy)
Ricardo Caferra (Grenoble INP - Laboratory of Informatics of Grenoble, France)
Martin Giese (University of Oslo, Norway)
Ullrich Hustadt (University of Liverpool, UK)
Alexander Leitsch (Vienna University of Technology, Austria)
Christopher Lynch (Clarkson University, USA)
Nicola Olivetti (LSIS - Université Paul Cézanne, Marseille, France)
Nicolas Peltier (CNRS - Laboratory of Informatics of Grenoble, France)
David Plaisted (University of North Carolina, Chapel Hill, USA)
Silvio Ranise (Università degli Studi di Verona, Italy)
Michael Rusinowitch (LORIA - INRIA Lorraine, France)
Renate Schmidt (University of Manchester, UK)
Viorica Sofronie-Stokkermans (MPI für Informatik, Saarbrücken, Germany)
Arild Waaler (University of Oslo, Norway)
Christoph Weidenbach (MPI für Informatik, Saarbrücken, Germany)

Additional reviewers

Stephane Demri
Jinbo Huang
William McCune
Ammar Mohammed
Andrei Paskevich
Rafael Penaloza
Serena Elisa Ponta
Geoff Sutcliffe

Local organization

Roger Antonsen (University of Oslo)

FTP Steering Committee

Table of Contents

Model Checking of Array-Based Systems: from Foundations to Implementation

Silvio Ghilardi

Dipartimento di Scienze dell'Informazione,
Università degli Studi di Milano (Italy)

Abstract. We are interested in automatically proving safety properties of infinite state systems, by combining the classical algebraic approach of [4] with deductive techniques exploiting, off-the-shelf, SMT solvers. After briefly recalling the main contributions in [4] leading to the use of backward reachability analysis to prove safety properties and overviewing the long line of works stemming from that seminal paper (such as [9, 8, 5–7]), we present the notion of *array based systems* [10]. Such systems are declarative abstractions of several classes of parametrised systems and (sequential) programs manipulating arrays. In the framework of array based systems, key notions from [4] (such as configuration, configuration ordering, and monotonic transition) can be adapted and reused in a uniform and simple way. A by-product of this approach is to make readily available deductive techniques (like the synthesis and the use of invariants [11]) in the context of the algorithmic verification technique of backward reachability. This is so because the framework retains the modularity and the flexibility typical of logic-based approaches to model-checking (in the same spirit of, e.g., [14]).

The key feature of array-based systems is that a suitable format for initial/unsafe states and transition formulae can be designed: this format is sufficiently expressive to cover interesting classes of infinite state systems and, at the same time, generates proof obligations (during backward analysis) that can be discharged by instantiation and SMT solving techniques for quantifier-free formulae.

To make the theoretical framework useful in practice, powerful heuristics are required to obtain adequate performances: these heuristics concern optimization of the computation of the pre-image [13], (static and dynamic) filtration of the instantiations that current SMT solvers cannot yet handle efficiently, as well as forward/backward simplification routines [12].

In the last part of the talk, we report our experimental experience with a prototype tool called MCMT [1], currently under development: we discuss its architecture (especially the interplay between the generation of proof obligations, the computation of pre-images, and the various heuristics) and its integration with the SMT solver YICES [3]; finally we compare MCMT with some state-of-the-art model checkers based on dedicated techniques like PFS [2].

This is joint work with Silvio Ranise (Università di Verona).

References

1. MCMT. http://homes.dsi.unimi.it/∼ghilardi/mcmt.
2. PFS. http://www.it.uu.se/research/docs/fm/apv/tools/pfs.
3. YICES. http://yices.csl.sri.com.
4. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.
5. P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*, volume 4424 of *LNCS*, pages 721–736, 2007.
6. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, volume 4590 of *LNCS*, pages 145–157, 2007.
7. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. of VMCAI*, volume 4905 of *LNCS*, pages 22–36, 2008.
8. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. In *Proc. of CSL*, volume 1683 of *LNCS*, pages 50–66, 1999.
9. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS*, pages 352–359. IEEE Computer Society, 1999.
10. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model-Checking of Array-based Systems. In *Proc. of IJCAR*, LNCS, 2008. Full version available as a Technical Report at http://homes.dsi.unimi.it/∼ghilardi/allegati/GhiNiRaZu-RI318-08.pdf.
11. S. Ghilardi and S. Ranise. Goal-Directed Invariant Synthesis in Model Checking Modulo Thoeries. In *Proc. of TABLEAUX 09*, LNCS, 2009. Full version available as a Technical Report at http://homes.dsi.unimi.it/∼ghilardi/allegati/GhRa-RI325-09.pdf.
12. S. Ghilardi and S. Ranise. Model Checking Modulo Theories at work: the integration of Yices with MCMT. In *Proc. of AFM 09*, 2009. Available from MCMT web page.
13. S. Ghilardi, S. Ranise, and T. Valsecchi. Light-Weight SMT-based Model-Checking. In *Proc. of AVOCS 07-08*, ENTCS, 2008. Available from MCMT web page.
14. T. Rybina and A. Voronkov. A logical reconstruction of reachability. In *Revised Papers of the 5th Int. A. Ershov Mem. Conf. on Perspectives of Systems Informatics (PSI 2003)*, volume 2890 of *LNCS*, pages 222–237, 2003.

Presenting Constraints

Peter Jeavons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford UK

Abstract. We describe the constraint satisfaction problem and show that it unifies a very wide variety of computational problems. We discuss the techniques that have been used to analyse the complexity of different forms of constraint satisfaction problem, focusing on the algebraic approach, explaining the basic ideas and highlighting some of the recent results in this area.

The above abstract belongs to a joint invited talk at FTP & Tableaux 2009. The full version of the paper is included in [1].

References

1. P. Jeavons. Presenting constraints. In M. Giese and A. Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, 18th International Conference, TABLEAUX 2009*, volume 5607 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

Constraint Modelling: A Challenge for Automated Reasoning

Peter Baumgartner and John Slaney
{*firstname.secondname*}@nicta.com.au

NICTA[⋆] and Australian National University, Canberra, Australia

Abstract. Cadoli *et al* [BCM04,MC05,CM04] noted the potential of first order automated reasoning for the purpose of analysing constraint models, and reported some encouraging initial experimental results. We are currently pursuing a very similar research program with a view to incorporating deductive technology in a state of the art constraint programming platform. Here we outline our own view of this application direction and discuss new empirical findings on a more extensive range of problems than those considered in the previous literature. While the opportunities presented by reasoning about constraint models are indeed exciting, we also find that there are formidable obstacles in the way of a practically useful implementation.

1 Constraint Programming

A constraint satisfaction problem (CSP) is normally described in the following terms: given a finite set of decision variables $v_1, \ldots, v_n$ with associated domains $D_1, \ldots, D_n$, and a relation $C(v_1, \ldots v_n)$ between the variables, a *state* is an assignment to each variable v_i of a value d_i from D_i. A state is a *solution* to the CSP iff $C(d_1, \ldots, d_i)$ holds. In practice, C is the conjunction of a number of constraints each of which relates a small number of variables. It is common to seek not just any solution, but an optimal one in the sense that it minimises the value of a specified *objective function*.

Logically, C is a theory in a language in which the v_i are proper names ("constants" in the usual terminology of logic). A state is an interpretation of the language over a domain (or several domains, if the language is many-sorted) corresponding to the domains of the variables, and a solution is an interpretation that satisfies C. On this view, CSP reasoning is the dual of theorem proving: it is seeking to establish possibility (satisfiability) rather than necessity (unsatisfiability of the negation).

Techniques used to solve CSPs range from the purely logical, such as SAT solving, through finite domain (FD) reasoning which similarly consists of a backtracking search over assignments, using a range of propagators appropriate to

[⋆] NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative.

different constraints to force some notion of local consistency after each assignment, to mixed integer programming using a variety of numerical optimisation algorithms. Hybrid solution methods, in which different solvers are applied to sub-problems, include SMT (satisfiability modulo theories), column generation, large neighbourhood search and many more or less *ad hoc* solver combinations for specific purposes. The whole area has been researched intensively over the last half century, generating an extensive literature from the automated reasoning, artificial intelligence and operations research communities. The reader is referred to [DC03,MS98] for an introduction to the field.

Constraint programming is an approach to designing software for CSPs, whereby the search is controlled by a program written in some high-level language (sometimes a logic programming language, but in modern systems often C++ or something similar) and specific solvers may be used to evaluate particular predicates or perform propagation steps, or may be passed the entire problem after some preprocessing. The constraint programming paradigm gives a great deal of flexibility, allowing techniques to be tailored to problems, while at the same time accessing the power and efficiency of high-performance CSP solvers.

1.1 Separating Modelling from Solving

Engineering a constraint program for a given problem is traditionally a two-phase process. First the problem must be *modelled.* This is a matter of determining what are the decision variables, what are their domains of possible values and what constraints they must satisfy. Then a program must be produced to *evaluate* the model by using some solver or combination of solvers to search for solutions. This program may be written by a human programmer, or derived automatically from the model, or some combination of the two. Most of the Constraint Programming (CP) and Operations Research (OR) literature concerns problem solving, assuming that “the problem” resulting from the modelling phase is given.

In recent years, there has been a growing realisation of the importance of modelling as part of the overall process, so modern CP or Mathematical Programming (MP) platforms feature a carefully designed modelling language such as ILOG’s OPL [Hen99] or AMPL from Bell Labs [FGK02]. Contemporary work on modelling languages such as ESRA [FPg04], ESSENCE [FGJ$^+$07] and Zinc [MNR$^+$08] aims to provide a rich representation tool, with primitives for manipulating sets, arrays, records and suchlike data structures and with the full expressive power of (at least) first order quantification. It also aims to make the problem representation independent of the solver(s) so that one and the same conceptual model can be mapped to a form suitable for solution by mixed integer programming, by SAT solving or by local search.

1.2 Zinc

In the present report, the modelling language used will be Zinc, which is part of the G12 platform currently under development by NICTA (Australia).[1]

The G12 platform provides a series of languages: Mercury, Cadmium and Zinc. Mercury is a constraint logic programming language, Cadmium a rather specialised programming language for syntax transformations based on term rewriting, and Zinc a modelling language in which problems are specified in an algorithm-independent way [MNR$^+$08]. It is a typed (mostly) first order language, with basic types `int`, `float` and `bool`, and user-defined finite enumerated types. To these are applied the `set-of`, `array-of`, `tuple`, `record` and subrange type constructors. These may be nested, with some restrictions mainly to avoid such things as infinite arrays and explicitly higher order types (functions with functional arguments). Zinc also allows a certain amount of functional programming, which is not of present interest. It provides facilities for declaring decision variables of most types and constants (parameters) of all types. Standard mathematical functions such as `+` and `sqrt` are built in. Constraints may be written using the expected comparators such as $==$ and $\leq$ or user-defined predicates to form atoms, and the usual boolean connectives and quantifiers (over finite domains) to build up compounds. Assignments are special constraints whereby parameters are given their values. The values of decision variables are not normally fixed in the Zinc specification, but have to be found by some sort of search.

It is normal to place the Zinc model in one file, and the data (parameters, assignments and perhaps some enumerations) in another. The model tends to stay the same as the data vary. For example, without changing any definitions or general specifications, a new schedule can be designed for each day as fresh information about orders, jobs, customers and prices becomes available.

The user support tools provided by the G12 development environment should facilitate debugging and other reasoning about models independently of any data. However, since the solvers cannot evaluate a model until at least the domains are specified, it is unclear how this can be done. Some static visualisation of the problem, such as views of the Zinc-level constraint graph, can help a little, but to go much further we need a different sort of reasoning: we need first order deduction.

[1] See `http://nicta.com.au/research/projects/constraint_programming_platform`. We have benefited greatly from being in a team that has included Michael Norrish, Rajeev Gore, Jeremy Dawson, Jia Meng, Anbulagan and Jinbo Huang, and from the presence in the same laboratory of an AI team including Phil Kilby, Jussi Rintanen, Sylvie Thiébaux and others. The G12 project involves well over 20 researchers, including Peter Stuckey, Kim Marriott, Mark Wallace, Toby Walsh, Michael Maher, Andrew Verden and Abdul Sattar. The details of our indebtedness to these people and their colleagues are too intricate to be spelt out here.

2 Deductive Tasks

There is no good reason to expect a theorem prover to be used as one of the solvers for the purposes of a constraint programming platform such as G12. In many practical cases the main issue is optimality, the existence of solutions being obvious, and it is not clear how theorem proving can help with this. Moreover, the reasoning required to solve CSPs typically amounts to propagation of constraints over finite domains rather than to chaining together complex inferences, and for this purpose SAT solvers and the like are useful, but traditional first order provers are not.[2] However, for analysing the models before they have been grounded by data, first order deduction is the only option. Previous work [BCM04,MC05,CM04] has identified some tasks and practical experiences using a first-order theorem prover. A serious deficiency of the previous accounts, however, is the absence of numerical reasoning. Zinc, like other modelling languages, supports integer domains, and even floating point ones. These are crucial: there is no hope of dealing adequately with industrial problems of scheduling and resource management without numbers. However, as we show below, even very simple integer arithmetic poses major difficulties for first-order theorem provers.

We are interested in the following problems, which are all capable of automation.

2.1 Proof that the Model is Inconsistent

Inconsistency can indicate a bug, or merely a problem overconstrained by too many requirements. It can arise in "what if" reasoning, where the programmer has added speculative conditions to the basic description or it can arise where partial problem descriptions from different sources have been combined without ensuring that their background assumptions mesh.

A traditional debugging move, also useful in the other cases of inconsistency, is to find and present a [near] minimal inconsistent core: that is, a minimally inconsistent subset of the constraints. The problem of "axiom pinpointing" in reasoning about large databases is logically similar, but in the constraint programming case the number of possible axioms tends to be comparatively small and the proofs of inconsistency comparatively long. The advantage of finding a first order proof of inconsistency, rather than merely analysing nogoods from a backtracking search, is that a proof can be presented to a programmer, thus answering the question of *why* the particular subset of constraints is inconsistent.

[2] The "typical" case is not the only case, of course. The satisfiability problem for Zinc is undecidable, since the language can express Diophantine equations over the unbounded domain of the integers. For Zinc models (without data) it is even easier to find undecidable theories, since the problem of deciding whether an arbitrary first order formula has a finite model is easily encoded, as are special cases like the word problem for semigroups. Sometimes, therefore, theorem proving may be the best we can do, but such cases do not arise in industrial process scheduling or other common CP applications.

2.2 Proof of Symmetry

The detection and removal of symmetries is of enormous importance to finite domain search. Where there exist isomorphic solutions, there exist also isomorphic subtrees of the search tree. In some cases almost all of the search can be eliminated if the symmetries are detected early enough. A standard technique is to introduce "symmetry breakers", which are extra constraints imposing conditions satisfied by some but not all (preferably by exactly one) of the solutions in a symmetry class. Symmetry breakers prevent entry to subtrees of the search tree isomorphic to the canonical one.

It may be evident to the constraint programmer that some transformation gives rise to a symmetry. Rotating or reflecting the board in the N Queens problem would be an example. However, other cases may be less obvious, especially where there are side constraints that could interfere with symmetry. Moreover, it may be unclear whether the intuitively obvious symmetry has been properly encoded or whether in fact every possible solution can be transformed into one which satisfies all of the imposed symmetry breakers.

It is therefore important to be able to show that a given transformation defined over the state space of the problem does actually preserve the constraints, and therefore that it transforms solutions into solutions. Since symmetry breakers may be part of the model rather than part of the data, we may wish to prove such a property independently of details such as domain sizes. There is an example in the next section.

2.3 Redundancy Tests

A redundant constraint is one that is a logical consequence of the rest. It is common to add redundant constraints to a problem specification, usually in order to increase the effect of propagation at each node of the search tree. Sometimes, however, redundancy may be unintentional: this may indicate a bug—perhaps an intended symmetry-breaker which in fact changes nothing—or just a clumsy encoding.

Where redundant constraints are detected, either during analysis of the model or during preprocessing of the problem including data, this might usefully be reported to the constraint programmer who can then decide whether such redundancy is intentional and whether the model should be adjusted in the light of this information. It may also be useful to report irredundancy where a supposedly redundant constraint has been added: the programmer might usefully be able to request a redundancy proof in such a case.

2.4 Functional Dependency

Functions may also be redundant, in the sense that the values of certain functions may completely determine the value of another for all possible arguments. As in the case of constraint redundancy, functional dependence may be intentional or accidental, and either way it may be useful to the constraint programmer to know whether a function is dependent or not.

Consider graph colouring as an example. It is obvious that in general (that is, independently of the graph in question) the extensions of all but one of the colours are sufficient to fix the extension of the final one, but that this is not true of any proper subset of the "all but one". In the presence of side constraints, however, and especially of symmetry breakers, this may not be obvious at all. In such cases, theorem proving is the appropriate technology.

2.5 Equivalence of Models

It is very common in constraint programming that different approaches to a given problem may result in *very* different encodings, expressing constraints in different forms and even using different signatures and different types. The problem of deciding whether two models are equivalent, even in the weak sense that solutions exist for the same values of some parameters such as domain sizes, is in general hard. Indeed, in the worst case, it is undecidable. However, hardness in that sense is nothing new for theorem proving, so there is reason to hope that equivalence can often enough be established by the means commonly used in automated reasoning about axiomatisations.

Concrete applications of proving equivalence stem from all sorts of transformations of constraint models. For instance, one might (automatically) detect that certain variables must receive different values according to the current model and pose a global `all_different` constraint instead. Other transformations are inspired by optimising compiler technology, such as loop-invariants hoisting (exchange "forall" and "exists" loops), common subexpression elimination, algebraic rewriting (theory specific equational rewriting) and partial evaluation (see [MKB+05]).

2.6 Simplification

A special case of redundancy, which in turn is a special case of model equivalence, occurs in circumstances where the full strength of a constraint is not required. A common example is that of a biconditional ($\Leftrightarrow$) where in fact one half of it ($\Rightarrow$) would be sufficient. Naïve translation between problem formulations can easily lead to unnecessarily complicated constraints such as $a < \sup(S)$ which is naturally rendered as
$\exists y(\forall z((\forall x \in S(x \leq z)) \leftrightarrow y \leq z) \wedge a < y)$,
while the simpler $\exists y \in S(x < y)$ would do just as well. Formal proofs of the correctness of simplifications can usefully be offered to the programmer at the model analysis stage.

```
int: N;
array[1..N] of var 1..N: q;
constraint forall (x in 1..N, y in 1..x-1)
    (q[x] != q[y]
∧   (q[x]+x != q[y]+y
∧   (q[x]-x != q[y]-y);
solve satisfy;
```

Fig. 1. Zinc model for the N Queens problem

```
int: N;
array[1..N] of var 1..N: q;
constraint forall (x in 1..N, y in 1..x where x != y)
    (q[x] != q[y]
∧   (q[x]+x != q[y]+y
∧   (q[x]-x != q[y]-y);
solve satisfy;
```

Fig. 2. Alternative model for the N Queens problem

3 Experiments

We conducted some experiments in order to evaluate the feasibility of state of the art automated reasoning technology to solve deductive proof tasks as explained in Section 2.

3.1 N-Queens

We consider the N Queens problem, a staple of CSP reasoning. N queens are to be placed on an a chessboard of size $N \times N$ in such a way that no queen attacks any other along any row, column or diagonal. The model is given in Figure 1 and the data consists of one line giving the value of N (e.g. '`N = 8;`').

Index Refinement As a very simple example of equivalence of models, consider the formulation of the n-queens problem in Figure 2. Notice how it differs slightly from the one in Figure 1 in the use of indexing. One may expect that re-formulations like these occur frequently and their correctness should be rather straightforward to establish automatically.

Alldifferent Constraint. The `alldifferent` constraint on a set of variables requires them to take pairwise different values. Because specialized, efficient constraint solving techniques have been developed for `alldifferent`, it may make sense to replace or enrich parts of a given constraint model by an `alldifferent` constraint. Clearly, in our example, any solution of the n-queens problem obviously satisfies the `alldifferent` constraint for $\{\texttt{q}[1], \ldots, \texttt{q}[N]\}$. It is easy to formulate this as a proof task: simply add the constraint `not(forall (x in 1..N, y in 1..x-1) (q[x] != q[y]))` to the constraint model and prove unsatisfiability. Of course, a sufficiently rich set of axioms for the underlying theories (integer arithmetic, e.g.) has to be provided to the prover as well.

Detecting Symmetries Suppose that as a result of inspection of this problem for small values of N it is conjectured, either automatically or by the programmer, that the transformation $s[x] = q[n+1-x]$ is a symmetry. We wish to prove this for all values of N. That is, we need a first order proof that the constraints with s substituted for q follow from the model as given and the definition of s. Intuitively, this is obvious, as it corresponds to the operation of reflecting the board, but intuitive obviousness is not proof and we wish to see what a standard theorem prover makes of it.

One prover we took off the shelf for this experiment was Prover9 by McCune [McC].[3] A certain amount of numerical reasoning is required, for which additional axioms must be supplied. The full theory of the integers is not needed: algebraic properties of addition and subtraction, along with numerical order, suffice. All of this is captured in the theory of totally ordered abelian groups (see e.g. [MA88]) which is quite convenient for first order reasoning [Wal01]. We tried two encodings: one in terms of the order relation $\leq$ and the other an equational version in terms of the lattice operations `max` and `min`.

The first three goals:
$(1 \leq x \wedge x \leq n) \Rightarrow 1 \leq s(x)$
$(1 \leq x \wedge x \leq n) \Rightarrow s(x) \leq n$
$s(x) = s(y) \Rightarrow x = y$
are quite easy for Prover9 when $s(x)$ is defined as $q(n+1-x)$. By contrast, the other two
$(1 \leq x \wedge x \leq n) \wedge (1 \leq y \wedge y \leq n) \Rightarrow s(x) + x \neq s(y) + y$
$(1 \leq x \wedge x \leq n) \wedge (1 \leq y \wedge y \leq n) \Rightarrow s(x) - x \neq s(y) - y$
are not provable inside a time limit of 30 minutes, even with numerous helpful lemmas and weight specifications. It makes little difference to these results whether the abelian l-group axioms are presented in terms of the order relation or as equations.

To push the investigation one more step, we also considered the transformation obtained by setting s to q^{-1}. This is also a symmetry, corresponding to reflection of the board about a diagonal. This time, it is necessary to add an axiom to the Queens problem definition, as the all-different constraint on q is not inherited by s. The reason is that for all we can say in the first order vocabulary, N might be infinite—it could be any infinite number in a nonstandard model of the integers—and in that case a function from $\{1 \ldots N\}$ to $\{1 \ldots N\}$ could be injective without being surjective.

The immediate fix is to add surjectivity of the '`q`' function to the problem definition, after which in the relational formulation Prover9 can easily deduce

[3] Previous work [CM04,CM05] user Otter for similar problems in graph coloring; its successor Prover9 is similar but generally superior.

the three small goals and the first of the two diagonal conditions. The second is beyond it, until we add the redundant axiom

$$x_1 - y_1 = x_2 - y_2 \Rightarrow x_1 - x_2 = y_1 - y_2$$

With this, it finds a proof in a second or so. In the equational formulation, no proofs are found in reasonable time.

We also tried the Vampire prover (version 8) and came to the same conclusions as with Prover9. With redundant axioms the proof is found easily, without them, not. One message from this experiment is that care must be taken to avoid implicit appeal to the fact that domains are finite. Another is that a range of arithmetical reasoning tricks and transformations will have to be identified and coded into the system. The above transformation of equalities between differences (and its counterparts for inequalities) illustrates this.

An encouraging feature is that a considerable amount of the reasoning turns only on algebraic properties of the number systems, and so may be amenable to treatment by standard first order provers.

A perhaps even more natural idea is to try theorem provers with native support for arithmetic reasoning instead of "general" first-order logic theorem provers. The development of such provers is still an active research topic, see, e.g., [BFT08,KV07,WP06,Rüm08], but some of the available SMT-solvers (Satisfiability Modulo Theories [RT06]) already support the logic and theories that we need. To explain, the proof obligation in the example is an entailment between two universally quantified formulas with free functions symbols, over the theory of linear integer arithmetic. SMT solvers are not full-fledged theorem provers for first-order logic, and on proof tasks of that form, current SMT solvers need to rely on (incomplete) instantiation heuristics to remove the universal quantifiers in the premise of an entailment. Despite that, in the example, the two SMT solvers that we tried, CVC3 [BT07] and Yices, had no difficulties even with the *original* problem formulation, the one without any additional redundant axioms (runtimes: less than one second). We found this a very encouraging result.

We also tried the default solver that comes with the G12 platform. Like any constraint solver, it is not a theorem prover and cannot prove that the symmetry property holds for all values of the board size N. However, we found it instructive to prove, with G12, the symmetry property with specific values for N. The rationale behind this exercise is the methodology to first try some small instances, to see if a conjecture is trivially falsified. Of course this is pointless in this example, but in general it may help to find bugs in the coding, or counterexamples for non-valid conjectures.

Table 1 summarizes the experimental results, for all problems described above. The results indicate that the SMT solvers, YICES and CVC3, perform much better on these problems than the theorem provers. Moreover, the formulations for the theorem provers are highly sensitive to the axiomatization of the background theory. Without a minimal "right" set of axioms, the proof will not be found or proof times increase drastically.

Problem	E	E-Darwin	SPASS	Vampire	YICES	CVC3	G12
`alldifferent` implied	-	< 1	1	< 1	< 1	< 1	< 1 ($N = 10$) 5 ($N = 12$) 137 ($N = 14$) > 600 ($N = 16$)
Index refinement	-	-	- (⇔) 6 (⇐) 12 (⇒)	-	< 1	< 1	< 1 ($N = 10$) 8 ($N = 12$) 242 ($N = 14$) > 600 ($N = 16$)
N-Queens symmetry	- (-)	- (3)	- (-)	- (6)	< 1	< 1	< 1 ($N = 10$) 6 ($N = 12$) 182 ($N = 14$) > 600 ($N = 16$)

Table 1. Systems on N-Queens related problems. All times in seconds. An entry "-" means "no solution found within 100 seconds". *N-Queens symmetry:* entries in parenthesis "(·)" refer to "tweaked" problem formulations, with redundant axioms; *Index refinement:* (⇔): proof obligation is equivalence; (⇐) and (⇒): one direction only. E, E-Darwin and Vampire can't prove the latter either.

3.2 Puzzle

A toy example of redundant constraints is found in the following logic puzzle [Ano]:

> Five couples celebrate their wedding anniversaries. Their surnames are Johnstone, Parker, Watson, Graves and Shearer. The husbands' given names are Russell, Douglas, Charles, Peter and Everett. The wives' given names are Elaine, Joyce, Marcia, Elizabeth and Mildred.
>
> 1. Joyce has not been married as long as Charles or the Parkers, but longer than Douglas and the Johnstones.
> 2. Elizabeth married twice as long ago as the Watsons, but half as long as Russell.
> 3. The Shearers married ten years before Peter and ten years after Marcia.
> 4. Douglas and Mildred have been married for 25 years less than the Graves who, having been married for 30 years, are the couple who have been married the longest.
> 5. Neither Elaine nor the Johnstones married most recently.
> 6. Everett has been married for 25 years
>
> Who is married to whom, and how long have they been married?

Parts of clue 1, that Joyce has been married longer than Douglas and also longer than the Johnstones, are deducible from the other clues. Half of clue 5, that Elaine has not been married the shortest amount of time, is also redundant. The argument is not very difficult, and is left for the reader's amusement. A finite domain constraint solver has no difficulty with it.

Presenting the problem of deriving any of these redundancies to Prover9 is not easy. The small amount of arithmetic involved is enough to require a painful amount of axiomatisation, and even when the addition table for the natural numbers up to 30 is completely spelt out, the derivation is beyond the abilities of the prover.

If the fact that the numbers of years are all in the set $\{5, 10, 15, 20, 25, 30\}$ is given as an axiom, *and* extra arguments are given to all function and relation symbols to prevent unification across sorts, then of course the redundancy proofs become easy for the prover. However, it is unreasonable to expect that so much help will be forthcoming in general. Even requiring just a little of the numerical reasoning to be carried out by the prover takes the problem out of range.

Part of the difficulty is due to the lack of numerical reasoning, but as before, forcing the problem statement into a single-sorted logic causes dramatic inefficiency. It is also worth noting that the proofs of redundancy are long (some hundreds of lines) and involve nearly all of the assumptions, indicating that axiom pinpointing is likely to be useless for explaining overconstrainedness at least in some range of cases.

3.3 Radiation

The background for this example was given in [BBBS07], which considers the problem of decomposing an integer matrix into a positively weighted sum of binary matrices that have the so-called consecutive-ones property. We do not need the details of this problems here. Instead it suffices to say that the problem is well-known and of practical relevance. It has an important application in cancer radiation therapy treatment planning: the sequencing of multileaf collimators to deliver a given radiation intensity matrix, representing (a component of) the treatment plan.

The proof task here is along the lines as described in Section 2.6 above. It requires to show that an occurrence of the "max" function between integers can be replaced by stipulating the existence of a lower bound instead. This leads to more efficient constraint solving. The solutions are preserved, essentially, because the objective is to compute *minimal* solutions, and maxima and upper bounds leading to minimal solutions coincide then (in this example).

Besides having to prove that minimal solutions are preserved, an additional complication comes from a rather syntactically deep embedding of the max function in the constraint model. Furthermore, it occurs within a summation formula, and this way stands for a *parametric* number n, the summation bound, of usages. In addition, it occurs within a predicate definition, whose arguments are unary arrays, and the predicate is "invoked" by taking sub-arrays of certain globally defined non-unary arrays. Because it is a non-trivial exercise already to recast this constraint model in a predicate logic formula we started with a coarse abstraction of the model. We defined five proof tasks, whose differences are shown in the following table:

Problem#	**model_max**$(x, y, n) \Leftrightarrow$	**model_ub**$(x, y, n) \Leftrightarrow$
(1)	$n = \max(x, y)$	$\mathrm{ub}(x, y, n)$
(2)	$\max(x, y) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \wedge z \leq n)$
(3)	$\mathrm{sum}(\max(x, y)) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \wedge \mathrm{sum}(z) \leq n)$
(4)	$\mathrm{c} + \max(x, y) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \wedge \mathrm{c} + z \leq n)$
(5)	$\mathrm{c} + \mathrm{sum}(\max(x, y)) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \wedge \mathrm{c} + \mathrm{sum}(z) \leq n)$

The second and third column define different abstractions of the original constraint model, models in terms of "max" and of "upper bound", respectively. It is not difficult to define minimal solutions. For the "max" version, for instance, one defines:

$$\forall x, y, z\ \mathrm{minsol\_model\_max}(x, y, n) \Leftrightarrow \\ (\mathrm{model\_max}(x, y, n) \wedge \forall z\ (\mathrm{model\_max}(x, y, z) \Rightarrow n \leq z))\ .$$

The definition for "minsol_model_ub(x, y, n)", the minimal solutions in terms of upper bounds, is given analogously. The proof task then is to show that together with a (straightforward) axiomatization of "max" and "ub", and possibly more axioms, the equivalence

$$\forall x, y, z\ (\mathrm{minsol\_model\_max}(x, y, n) \Leftrightarrow \mathrm{minsol\_model\_ub}(x, y, n))$$

follows. Table 3.3 contains the results.

Conclusions

While, as noted, the investigation is still preliminary, some conclusions can already be drawn. Notably, work is required on expanding the capacities of conventional automatic theorem provers:

1. Numerical reasoning, both discrete and continuous, is essential. The theorems involved are not deep—showing that a simple transformation like reversing the order $1 \ldots N$ is a homomorphism on a model or restricting attention to numbers divisible by 5—but are not easy for standard theorem proving technology either. Theorem provers will not succeed in analysing constraint models until this hurdle is cleared.
2. Other features of the rich representation language also call for specialised reasoning. Notably, the vocabulary of set theory is pervasive in CSP models, but normal theorem provers have difficulties with the most elementary of set properties. Some first order reasoning technology akin to SMT, whereby specialist modules return information about sets, arrays, tuples, numbers, etc. which a resolution-based theorem prover can use, is strongly indicated. Reasoning modulo a background theory, which originated with the introduction of theory resolution, is the obvious starting point, but is it enough?

Problem		E	E-Darwin	SPASS	YICES	CVC3
	$\Leftrightarrow$	13	7	1	-	-
(1)	$\Rightarrow$	2	4	1	-	-
	$\Leftarrow$	51	2	1	1	1
	$\Leftrightarrow$	-	-	-	-	-
(2)	$\Rightarrow$	-	-	2	-	-
	$\Leftarrow$	-	-	2	-	-
	$\Leftrightarrow$	-	-	-	-	-
(3)	$\Rightarrow$	-	-	23	-	-
	$\Leftarrow$	-	-	-	-	-
	$\Leftrightarrow$	-	-	-	-	-
(4)	$\Rightarrow$	-	-	2	-	-
	$\Leftarrow$	-	-	-	-	-
	$\Leftrightarrow$	-	-	-	-	-
(5)	$\Rightarrow$	-	-	-	-	-
	$\Leftarrow$	-	-	-	-	-

Table 2. Systems on abstractions of the radiation problem. All times in seconds. $\Leftrightarrow$: proof obligation is equivalence, as stated above; $\Leftarrow$ and $\Rightarrow$: one direction only. An entry "-" means "no solution found within 100 seconds".

3. Many-sorted logic is absolutely required. There are theorem provers able to exploit sorts, but despite decades of literature on the subject, many still do not. A telling point is that TPTP still does not incorporate sorts in its notation or its problems.
4. Constraint models sometimes depend on the finiteness of parameters. Simple facts about them may be unprovable without additional constraints to capture the effects of this, as illustrated by the case of the symmetries of the N Queens problem. This is not a challenge for theorem provers as such but rather for the process of preparing constraint models for first order reasoning.
5. In some cases, proofs need to be presented to human programmers who are not working in the vocabulary of theorem proving, who are not logicians, and who are not interested in working out the details of complicated paramodulation inferences. Despite some efforts, the state of the art in proof presentation remains unsatisfactory. This *must* be addressed somehow.[4]

Despite the above challenges, and perhaps in a sense because of them, constraint model analysis offers an exciting range of potential rôles for automated deduction. Constraint-based reasoning has far wider application than most canvassed uses of theorem provers, such as software verification, and certainly connects with practical concerns much more readily than most of [automated] pure mathematics. Reasoning about constraint models without their data is a niche

[4] The IPV tool associated with TPTP marks a good recent step towards addressing it. More in the same style needs to be the object of wider research: any serious theorem prover should routinely come with an advanced proof presentation package.

that only first (or higher) order deductive systems can fill. Those of us who are concerned to find practical applications for automated reasoning should be working to help them fill it.[5]

References

[Ano] Anonymous. Anniversaries: Logic puzzle. `http://genealogyworldwide.com/genealogy_fun.php`.

[BBBS07] Davaatseren Baatar, Natashia Boland, Sebastian Brand, and Peter J. Stuckey. Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In Pascal Van Hentenryck and Laurence A. Wolsey, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings*, volume 4510 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

[BCM04] Lucas Bordeaux, Marco Cadoli, and Toni Mancini. Exploiting fixable, removable, and implied values in constraint satisfaction problems. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2004.

[BFT08] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. ME(LIA) – Model Evolution With Linear Integer Arithmetic Constraints. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 258–273. Springer, November 2008.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[CM04] Marco Cadoli and Toni Mancini. Exploiting functional dependencies in declarative problem specifications. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 628–640. Springer, 2004.

[CM05] Marco Cadoli and Toni Mancini. Using a theorem prover for reasoning on constraint problems. In *AI*IA*, pages 38–49, 2005.

[DC03] Rina Dechter and David Cohen. *Constraint Processing.* Morgan Kaufmann, 2003.

[FGJ+07] A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 80–87, 2007.

[5] This research was supported by NICTA (National ICT Australia) and by the Australian National University. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

[FGK02] Robert Fourer, David Gay, , and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002. `http://www.ampl.com/`.

[FPg04] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR'03, Revised Selected Papers (LNCS 3018)*, pages 214–232. Springer-Verlag, 2004.

[Hen99] Pascal Van Hentenryck. *The OPL optimization programming language.* MIT Press, Cambridge, MA, 1999.

[KV07] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic (CSL'07)*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

[MA88] Todd Feil Marlow Anderson. *Lattice-ordered Groups: An Introduction.* Springer-Verlag, 1988.

[MC05] Toni Mancini and Marco Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In Jean-Daniel Zucker and Lorenza Saitta, editors, *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings*, volume 3607 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2005.

[McC] William McCune. Prover9. `http://www.cs.unm.edu/ mccune/mace4/`.

[MKB$^+$05] Darko Marinov, Sarfraz Khurshid, Suhabe Bugrara, Lintao Zhang, and Martin Rinard. Optimizations for compiling declarative models into boolean formulas. In *Computer Aided Verification (CAV)*, LNAI. Springer, 2005.

[MNR$^+$08] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints, Special Issue on Abstraction and Automation in Constraint Modelling*, 13(3), 2008.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[RT06] Silvio Ranise and Cesare Tinelli. Satisfiability modulo theories. *Trends and Controversies - IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[Rüm08] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 274–289. Springer, November 2008.

[Wal01] Uwe Waldmann. Superposition and chaining for totally ordered divisible abelian groups. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*, pages 226–241, 2001.

[WP06] Uwe Waldmann and Virgile Prevosto. Spass+t. In Stephan Schulz Geoff Sutcliffe, Renate Schmidt, editor, *ESCoREmpirically Successful Computerized Reasoning*, CEUR Workshop Proceedings, pages 18–33, Seattle, WA, USA, 2006.

Toward an Efficient Equality Computation in Connection Tableaux: A Modification Method without Symmetry Transformation[1] — A Preliminary Report—

Koji Iwanuma§1, **Hidetomo Nabeshima**§1, **and Katsumi Inoue**§2

§1**University of Yamanashi**, 4-3-11 Takeda, Kofu-shi, Yamanashi, 400-8511, Japan
Email: {iwanuma,nabesima}@yamanashi.ac.jp

§2**National Institute of Informatics**, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan. Email: ki@nii.ac.jp

Abstract

In this paper, we study an efficient equality computation in connection tableaux, and give a new variant of Brand, Bachmair-Ganzinger-Voronkov and Paskevich's modification methods, where the symmetry elimination rule is never applied. As is well known, effective equality computing is very difficult in a top-down theorem proving framework such as connection tableaux, due to a strict restriction to re-writable terms. The modification method with ordering constraints is a well-known remedy for top-down equality computation, and Paskevich adapted the method to connection tableaux. However the improved modification method still causes essentially redundant computation which originates in a symmetry elimination rule for equational clauses. The symmetry elimination may produce an exponential number of clauses from a given single clause, which inevitably causes a huge amount of redundant backtracking in connection tableaux. In this paper, we study a simple but effective remedy, that is, we abandon such symmetry elimination for clauses and instead introduce new equality inference rules into connection tableaux. These new inference rules have a possibility of achieving efficient equality computation, without losing the symmetry property of equality, which never cause redundant backtracking nor redundant contrapositive computation. We implemented the proposed methods in a sophisticated prover SOLAR which is originally designed to finding logical consequences, and show a preliminary experimental results for TPTP benchmark problems. This research is now in progress, thus the experimental results provided in this paper are tentative ones.

1 Introduction

In this paper, we study an efficient equality computation in connection tableaux, and give a variant of modification methods investigated by Brand [2], Bachmair-Ganzinger-Voronkov [1] and Paskevich [9]. We investigate a novel modification method such that a symmetry elimination rule is never applied.

[1]This research was partially supported by the Grant-in-Aid from The Ministry of Education, Science and Culture of Japan ((A) No.20240016)

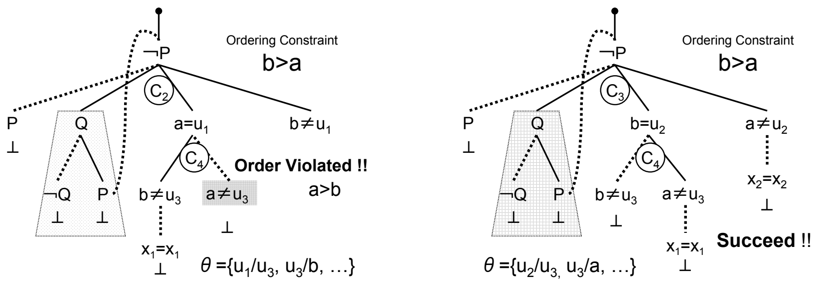


Figure 1: Connection Tableaux for Modification with ordering constraints

As is well known, effective equality [1, 3] computing is very difficult in a top-down theorem proving framework such as connection tableaux [6], due to a strict restriction to re-writable terms [12]. The modification method proposed by Brand has the great possibility for improving top-down equality computation. Bachmair, Ganzinger and Voronkov improved Brand's method with ordering constraints, and Paskevich adapted connection tableau calculus to the modification method using ordering. However the improved connection tableaux still causes redundant computation which is essentially involved by a symmetry elimination rule for equational clauses. The symmetry elimination may produce an exponential number of clauses from a given single clause, which inevitably causes a huge amount of redundant backtracking in Connection Tableaux.

Let $\mathcal{S}_1$ be a set of clauses $\{\, \neg P,\ P \vee Q \vee a \approx b,\ b \not\approx a,\ \neg Q \vee P\}$. The modification method transforms $\mathcal{S}_1$ into the following set of clauses with ordering constraints:

$$\begin{array}{ll}
C_1: & \neg P \\
C_2: & (P \ \vee\ Q \ \vee\ a \simeq u_1 \ \vee\ b \not\simeq u_1) \cdot (a \succ u_1 \wedge b \succeq u_1) \\
C_3: & (P \ \vee\ Q \ \vee\ b \simeq u_2 \ \vee\ a \not\simeq u_2) \cdot (b \succ u_2 \wedge a \succeq u_2) \\
C_4: & (b \not\simeq u_3 \ \vee\ a \not\simeq u_3) \cdot (b \succeq u_3 \wedge a \succeq u_3) \\
C_5: & \neg Q \vee P \\
\text{Ref}: & x \simeq x \qquad \text{(Reflexivity Axiom)}
\end{array}$$

A clause with ordering constraints takes the form of $D \cdot \delta$ where D is an ordinary clause and δ is a conjunction of ordering constraints $s \succ t$, $s \succeq t$ or $s = t$. The ordering constraint δ of $D \cdot \delta$ is expected to be satisfiable together with D. Notice that the above two clauses C_2 and C_3 are produced from the single clause $P \vee Q \vee a \approx b$ by symmetry elimination rule (more precisely, together with transitivity elimination rule). Figure 1 depicts two consecutive tableaux in a connection tableaux derivation, where we assume the ordering $b \succ a$ over constants. The left tableau fails to be closed because the goal $a \not\simeq u_2$ violates the ordering constraint $a \succeq u_3$, where the variable u_3 is substituted with b. The failure of derivation invokes backtracking, and eventually replaces the tableau clause C_2 below the top clause C_1 with the clause C_3. The right tableau in Fig. 1 succeeded in being closed, and simultaneously satisfies the ordering constraints. Notice that there are identical subtableaux below the goal Q in both left and right tableaux. Unfortunately, none of well-known pruning methods, such as folding-up/C-reduction or local failure caching, can prevent the redundant duplicated computation, because the clause C_2 containing Q in the left tableau is replaced with C_3 in the right tableau. Such a redundant computation essentially originates in the duplication of a given clause by symmetry elimination.

In this paper, we study a simple but effective remedy, that is, we abandon such symmetry elimination of clauses, and instead introduce new equality inference rules into connection tableaux. These new inference rules can achieve efficient equality computation, without losing the symmetry property of equality, which never cause redundant backtracking nor redundant contrapositive computation. Finally, we evaluate the proposed method through experiments with TPTP benchmark problems. Paskevich [9] also gave a new connection tableau calculus which uses lazy paramodulation instead of symmetry elimination. Paskevich's paramodulation-based connection calculus is very sophisticated, but seems to be a bit complicated and difficult in efficient implementation. Although the calculus proposed in this paper is superficially a little bit complicated, the underlying principle is very simple, and is easy to implement. At last, we emphasize that this research is now in progress, In this paper we show just some tentative results.

2 Preliminaries

We give some preliminaries according to Paskevich [9]. A language considered in this paper is first-order logic with equality in clausal form. A *clause* is a multi-set of literals, usually written as a disjunction $L_1 \vee \ldots \vee L_n$. The empty clause is denoted as $\perp$.

The equality predicate is denoted by the symbol $\approx$. We abbreviate the negation $\neg(s \approx t)$ as $s \not\approx t$. We consider equalities as unordered pairs of terms; that is, $a \approx b$ and $b \approx a$ stand for the same formula. As is well known, the equality is characterized by the congruence axioms $\mathcal{E}$ consisting of four axioms, i.e., *reflexivity*, *symmetry*, *transitivity* and *monotonicity*. The symbol $\simeq$ will denote "pseudo-equality", i.e., a binary predicate without any specific semantics. We utilize $\simeq$ in order to replace the symbol $\approx$ when we transform a clause set into a logic without equality. The order of arguments becomes significant here: $a \simeq b$ and $b \simeq a$ denote different formulas. The expression $s \not\simeq t$ stands for $\neg(s \simeq t)$.

We denote non-variable terms by $\mathbf{nv}$, $\mathbf{nv_1}$ and $\mathbf{nv_2}$, and also arbitrary terms by l, r, s, t, u and v. Variables are denoted by x, y and z. Substitutions are denoted by σ and θ. The result applying a substitution σ to an expression E is denoted by $E\sigma$. We write $E[s]$ to indicate that a term s occurs in E, and also write $E[t]$ to denote the expression obtained from E by replacing one occurrence of s with t.

We use an ordering constraint as defined in Bachmair et al. [1]. A *constraint* is a conjunction of *atomic constraints* $s = t$, $s \succ t$ or $s \succeq t$. The letters γ and δ denote constraints. A compound constraint $(a = b \wedge b \succ c)$ can be written in an abbreviated form $a = b \succ c$. A substitution σ *solves* an atomic constraint $s = t$ if the terms $s\sigma$ and $t\sigma$ are syntactically identical. It is a solution of an atomic constraint $s \succ t$ $(s \succeq t)$ if $s\sigma > t\sigma$ $(s\sigma \geq t\sigma$, respectively) with respect to a given term ordering $>$. Throughout this paper, we assume that a term ordering $\succ$ is a *reduction ordering* which is total over ground terms.[2] We say that σ is a solution of a constraint γ if it solves all atomic constraints in γ; γ is called *satisfiable* whenever it has a solution.

[2]A *reduction ordering* $>$ is an ordering over terms such that: (1) $>$ is well-founded; (2) for any terms s, t, u and any substitution θ, if $s > t$ then $u[s\theta] > u[t\theta]$ holds.

Expansion (Exp): $$\frac{\mathcal{S}, (L_1 \vee \cdots \vee L_k) \ || \ \Gamma}{L_1 \quad \cdots \quad L_k}$$

Strong Connection (SC):

$$\frac{\mathcal{S} \ || \ \Gamma, \neg P(r), P(s)}{\bot \cdot (r = s)} \qquad \frac{\mathcal{S} \ || \ \Gamma, P(r), \neg P(s)}{\bot \cdot (r = s)}$$

Weak Connection (WC):

$$\frac{\mathcal{S} \ || \ \Gamma, \neg P(r), \Delta, P(s)}{\bot \cdot (r = s)} \qquad \frac{\mathcal{S} \ || \ \Gamma, P(r), \Delta, \neg P(s)}{\bot \cdot (r = s)}$$

Figure 2: Connection calculus **CT** for a set $\mathcal{S}$ of clauses

Let $\mathcal{S}$ be a set of clauses. A *constrained clause tableau* for $\mathcal{S}$ is a finite tree $\mathcal{T}$ (See Fig. 1 as an example). Each node except for a root node is a pair $L \cdot \gamma$ where L is a literal and γ is a constraint. Any branch that contains the literal $\bot$, which represents the false, is *closed*. A tableau is *closed*, whenever every branch in it is closed and the overall of constraints in it is satisfiable.

Each inference step grows some branch in the tableau by adding new leaves under the leaf of the branch in question. Initially, an inference starts from the single root node. Symbolically, we describe an inference rule as follows:

$$\frac{\mathcal{S} \ || \ \Gamma}{L_1 \cdot \gamma_1 \quad \cdots \quad L_n \cdot \gamma_n}$$

where $\mathcal{S}$ is an initial given set of clauses, Γ is the branch being augmented (with constraints not mentioned), and $(L1 \cdot \gamma_1), \ldots, (L_n \cdot \gamma_n)$ are the added nodes. Whenever we choose some clause C in $\mathcal{S}$ to participate in the inference, we implicitly rename all variable in C to some fresh variables. The standard connection tableau calculus [6, 9], denoted by **CT**, for a set $\mathcal{S}$ of clauses has inference rules depicted in Fig. 2.

Any clause tableau built by the rules of **CT** can be considered as a tree of inference steps. Every tableau of **CT** always starts with an expansion step; also that first expansion step can be followed only by another expansion, since connection step requires at least two literals in a branch. In a tableau, an *expansion clause* is the added clause in an expansion step.

Let $\mathcal{T}$ be a tableau of **CT** for a set $\mathcal{S}$ of clauses. We say that $\mathcal{T}$ is *strongly connected* whenever every strong connection step in a tableau follows an expansion step, and every expansion step except for the first (or top) one is followed by exactly one strong connection step. Moreover, $\mathcal{T}$ is said to be a *refutation* for $\mathcal{S}$ if $\mathcal{T}$ is strongly connected and closed.

Theorem 1 (Letz et.al [6]) *The* **CT** *calculus is sound and complete in first-order logic without equality.*

Ordered paramodulation is a well-known efficient equality inference rule. It is well known that top-down (or linear) deduction systems, including connection tableaux, are difficult frameworks for efficient equality computation because of hard restriction of redexes, i.e., subterms allowed to rewrite. For example, Snyder and Lynch [12] showed that: paramodulation into a variable is necessary for completeness; ordering constraints is incompatible with top-down theorem proving even if paramodulation into a variable is allowed. As a remedy, the modification proposed by Brand [2] has been investigated by many researchers.

2.1 Modification Method and Connection Tableaux

In this subsection, we firstly, show the modification method given by Bachmair, Ganzinger and Voronkov [1] which uses ordering constraints. Secondly, we show Paskevich's connection tableau calculus [9], denoted as $\mathbf{CT}^{\simeq}$,[3] for refuting a set of clauses generated by the modification method.

2.1.1 Elimination of Congruence Axioms

Given a set $\mathcal{S}$ of equational clauses, we apply three kinds of elimination rules and replace the equality predicate $\approx$ by the predicate $\simeq$ to obtain a modified clause set $\mathcal{S}$', such that $\mathcal{S}$' is satisfiable iff $\mathcal{S}$ is equationally satisfiable. If R is a set of such elimination rules, we say a constrained clause is in R-normal form if no rule in R is applicable to it. We denote by $R(\mathcal{S})$ the set of all R-normal forms of a clause in $\mathcal{S}$.

We first show S-modification rules which replaces the equality symbol $\approx$ with the pseudo-equality $\simeq$, and generates several clauses which can simulate computational effects of symmetry axiom.

- *Positive S-modification*:
$$s \approx t \ \vee\ C \quad \Rightarrow \quad s \simeq t \ \vee\ C \ \text{ and } \ t \simeq s \ \vee\ C$$

- *Negative S-modification*:
$$\begin{array}{rcl} \mathbf{nv} \not\approx t \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq t \ \vee\ C \\ x \not\approx \mathbf{nv} \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq x \ \vee\ C \\ x \not\approx y \ \vee\ C & \Rightarrow & C\theta \end{array}$$

  where θ is a substitution $\{x/y\}$.

Remark: Positive S-modification rule is quite problematic, because one equation is *duplicated* to two equations each of which has converse directions. We shall give a remedy for it in the next section.

Secondly we give M-modification rules which flatten clauses by abstracting subterms via introduction of new variables as follows:

$$\begin{array}{rcl} P(\dots, \mathbf{nv}, \dots) \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq z \vee P(\dots, z, \dots) \ \vee\ C \\ \neg P(\dots, \mathbf{nv}, \dots) \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq z \vee \neg P(\dots, z, \dots) \ \vee\ C \\ f(\dots, \mathbf{nv}, \dots) \simeq t \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq z \vee f(\dots, z, \dots) \simeq t \ \vee\ C \\ f(\dots, \mathbf{nv}, \dots) \not\simeq t \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq z \vee f(\dots, z, \dots) \not\simeq t \ \vee\ C \\ s \simeq f(\dots, \mathbf{nv}, \dots) \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq z \vee s \simeq f(\dots, z, \dots) \ \vee\ C \\ s \not\simeq f(\dots, \mathbf{nv}, \dots) \ \vee\ C & \Rightarrow & \mathbf{nv} \not\simeq z \vee s \not\simeq f(\dots, z, \dots) \ \vee\ C \end{array}$$

where z is a new variable, called an *abstraction variable*.

The third one is T-modification rule for generating clauses which can simulate effects of transitivity axiom.

[3]Notice that $\mathbf{CT}^{\simeq}$ was introduced to prove the completeness of the *lazy paramodulation calculus* in [9].

Expansion (Exp):
$$\frac{\text{SMT}(\mathcal{S}), (L_1 \vee \cdots \vee L_k) \ \|\ \Gamma}{L_1 \quad \cdots \quad L_k}$$

$\mathbf{Equality\ Resoution(ER)}$:
$$\frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma,\ l \not\simeq r}{\bot \cdot (l = r)}$$

Strong Connection (SC):
$$\frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, \neg P(r), P(s)}{\bot \cdot (r = s)} \qquad \frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, P(r), \neg P(s)}{\bot \cdot (r = s)}$$

$$\frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, \mathbf{nv} \not\simeq r, s \simeq t}{\bot \cdot (\mathbf{nv} = s \succ t = r)} \qquad \frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, s \simeq t, \mathbf{nv} \not\simeq r}{\bot \cdot (\mathbf{nv} = s \succ t = r)}$$

Weak Connection (WC):
$$\frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, \neg P(r), \Delta, P(s)}{\bot \cdot (r = s)} \qquad \frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, P(r), \Delta, \neg P(s)}{\bot \cdot (r = s)}$$

$$\frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, \mathbf{nv} \not\simeq r, \Delta, s \simeq t}{\bot \cdot (\mathbf{nv} = s \succ t = r)} \qquad \frac{\text{SMT}(\mathcal{S}) \ \|\ \Gamma, s \simeq t, \Delta, \mathbf{nv} \not\simeq r}{\bot \cdot (\mathbf{nv} = s \succ t = r)}$$

Figure 3: Connection tableaux **CT**$^{\simeq}$ for SMT($\mathcal{S}$)

- *Positive T-modification:*
$$s \simeq \mathbf{nv} \ \vee\ C \quad \Rightarrow \quad s \simeq z \vee \mathbf{nv} \not\simeq z \ \vee\ C$$
- *Negative T-modification:*
$$s \not\simeq \mathbf{nv} \ \vee\ C \quad \Rightarrow \quad s \not\simeq z \vee \mathbf{nv} \not\simeq z \ \vee\ C$$
where z is a new variable, called a *link variable*.

Notice that if the term t in $s \simeq t$ is a variable, then T-modification does *nothing*.

Let SMT($\mathcal{S}$) denote a set T(M(S($\mathcal{S}$))), i.e., the set of normal clauses obtained from $\mathcal{S}$ by consecutively applying S, M and T-modification. Notice that the size of SMT($\mathcal{S}$) is *exponential* to the one of S.

Theorem 2 (Bachmair et al. [1]) *$\mathcal{S} \cup \mathcal{E}$ is unsatisfiable iff* SMT($\mathcal{S}$) $\cup \{x \simeq x\}$ *is unsatisfiable, where $\simeq$ is a new symbol for simulating the equality.*

Bachmair et al. [1] studied weak ordering constraints for modification. An atomic ordering constraint $s \succ t$ ($s \succeq t$) is assigned to each positive (or respectively, negative) literal $s \simeq t$ (or respectively, $s \not\simeq t$) in SMT($\mathcal{S}$), except for the negative equality $x \not\simeq y$ for any variables x and y.

CEE($\mathcal{S}$) denote the set of clauses of SMT($\mathcal{S}$) with ordering constraints.

Theorem 3 (Bachmair et al. [1]) *$\mathcal{S} \cup \mathcal{E}$ is unsatisfiable iff* CEE($\mathcal{S}$) $\cup \{x \simeq x\}$ *is unsatisfiable, where $\simeq$ is a new symbol for simulating the equality.*

2.1.2 Connection Tableaux for Modification with Ordering Constraints

Paskevich [9] adapted the calculus **CT** for computing CEE($\mathcal{S}$), and gave the connection tableau calculus **CT**$^{\simeq}$ for modification with ordering constraints, which is described in Fig. 3. Notice that $\mathbf{nv}$ denotes a non-variable term in **CT**$^{\simeq}$.

Theorem 4 (Paskevich [9]) *The calculus* $\mathbf{CT}^{\simeq}$ *is sound and complete. That is,* $\mathcal{S} \cup \mathcal{E}$ *is unsatisfiable iff there is a closed and strongly connected tableau in* $\mathbf{CT}^{\simeq}$ *for SMT*($\mathcal{S}$)*.*

3 Connection Tableaux for Modification without S-Modification

The size of SMT($\mathcal{S}$) is unfortunately *exponential* to the one of S, which is truly problematic and causes a huge amount of redundant computation. The positive S-modification, hence, should be abandoned. We alternatively introduce new inference rules for simulating the effects of symmetry axiom and construct a new connection tableau calculus **CTwS** (Connection Tableaux for modification Without S-modification).

Definition 1 Let *P-modification* be a transformation rule of clauses, which just replaces the equality symbol $\approx$ with the pseudo symbol $\simeq$ in positive equalities. We define nSMT($\mathcal{S}$) to be a a set of normal clauses obtained from $\mathcal{S}$ by just successively applying P-modification, negative S-modification, M-modification and negative T-modification.

Notice that the size of nSMT($\mathcal{S}$) is *linear* to the one of $\mathcal{S}$ because positive S-modification is never applied.

Once the positive S-modification is abandoned, no symmetry formula $t \simeq s$ of an initial equality $s \simeq t$ is generated in the modification process, which means that the succeeding positive T-modification is not accomplished either. Therefore, we need a mechanism compensating such a deficit of clause transformation. In this paper, we introduce new inference rules which can simulate not only positive S-modification but also *positive T-modification* for keeping transitivity properties of a positive equality.

We propose the following new rules, called *symmetry and transitivity splitting rules*, abbreviated as *ST-splitting*, which can simultaneously simulate the computational effects of symmetry and transitivity axioms.

Naive ST-Splitting Rule:

$$\frac{\text{nSMT}(\mathcal{S}) \ \ || \ \ \Gamma, s \simeq \mathbf{nv}}{s \simeq z \quad \mathbf{nv} \not\simeq z} \qquad\qquad \frac{\text{nSMT}(\mathcal{S}) \ \ || \ \ \Gamma, s \simeq x}{s \simeq x}$$

$$\frac{\text{nSMT}(\mathcal{S}) \ \ || \ \ \Gamma, \mathbf{nv} \simeq t}{t \simeq z \quad \mathbf{nv} \not\simeq z} \qquad\qquad \frac{\text{nSMT}(\mathcal{S}) \ \ || \ \ \Gamma, x \simeq t}{t \simeq x}$$

where $\mathbf{nv}$ is a non-variable term and x is a variable.

3.1 Controlling ST-Splitting I: A Raw Equality

ST-Splitting should be applied to each positive equality *at most one time*, because more than two times applications of these rules are clearly redundant. Therefore we need a controlling mechanism.

In this paper, we firstly give a *raw positive equality*, denoted as $\boxed{s \simeq t}$, which is introduced into a tableau by the expansion rule. Some of raw positive equalities $\boxed{s \simeq t}$

are changed to *ordinary equality literals* by ST-splitting. Conversely ST-Splitting rule is restricted to apply only to a raw positive equality. Moreover, the strong connection rule for a negative equality is also restricted to apply only to raw positive equalities. Furthermore, we force every raw positive equality to be followed either by ST-Splitting or by new strong contraction rules shown below.

Given a literal L, we write $[L]$ to denote a framed literal $\boxed{s \simeq t}$, called a *raw positive literal* if L is a positive equality $s \simeq t$; otherwise $[L]$ denotes L itself. We modify the expansion rule into the one which produces a raw literal for a positive equality.

Expansion for nSMT$(\mathcal{S})$**:**

$$\frac{\text{nSMT}(\mathcal{S}),\ (L_1 \vee \cdots \vee L_k)\ \ ||\ \ \Gamma}{[L_1]\ \ \cdots\ [L_k]}$$

ST-Splitting Rule should be changed to treat only raw positive literals.

ST-Splitting Rule:

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{s \simeq \mathbf{nv}}}{s \simeq z \quad \mathbf{nv} \not\simeq z} \qquad \frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{s \simeq x}}{s \simeq x}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{\mathbf{nv} \simeq t}}{t \simeq z \quad \mathbf{nv} \not\simeq z} \qquad \frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{x \simeq t}}{t \simeq x}$$

Example 1 Consider the set $\mathcal{S}_1$ of clauses in Section 1. The set nSMT$(\mathcal{S})$ of normal clauses is:

$$\begin{array}{ll} C_1: & \neg P. \\ C_6: & P \vee Q \vee a \simeq b \\ C_4': & (b \not\simeq u_3\ \vee\ a \not\simeq u_3) \\ C_5: & \neg Q \vee P \end{array}$$

Figure 4 shows two connection tableaux in **CTwS** for $\mathcal{S}_1$, each of which corresponds with the one in Fig. 1. Notice that no backtracking occurs for undoing the expansion introducing the clause C_6 in the derivation from the left tableau to the right one. Therefore none of duplicated computations invoked for the subgoal Q in **CT**$^{\simeq}$ occur in the calculus **CTwS**.

3.2 Controlling ST-Splitting II: Strong Connection

The original form of strong connection for negative equality is no longer appropriate, because it cannot deal with raw positive equalities nor corporate with ST-splitting rule. The new calculus **CTwS** has to simulate all valid inferences involving the strong connection in **CT**$^{\simeq}$ for SMT($\mathcal{S}$) in order to preserve completeness. Let $C \in \mathcal{S}$ be a clause $s \simeq t \vee K_1 \vee \cdots \vee K_m$. There are four possible clauses obtained by S-modification and T-modification from C with respect to $s \simeq t$:

$$\begin{array}{lrl} D_1: & s \simeq z \vee \mathbf{nv_2} \not\simeq z \vee K_1' \vee \cdots \vee K_m' & \text{if } t \text{ is a non-variable term } \mathbf{nv_2} \\ D_2: & s \simeq x \vee K_1' \vee \cdots \vee K_m' & \text{if } t \text{ is a variable } x \\ D_3: & t \simeq z \vee \mathbf{nv_2} \not\simeq z \vee K_1' \vee \cdots \vee K_m' & \text{if } s \text{ is a non-variable term } \mathbf{nv_2} \\ D_4: & t \simeq x \vee K_1' \vee \cdots \vee K_m' & \text{if } s \text{ is a variable } x \end{array}$$

where z is a fresh variable. All of these clauses have possibilities to be used as an expansion clause for the strong connection in **CT**$^{\simeq}$. Next we consider new strong connection rules for **CTwS** in order to simulate these inferences in **CT**$^{\simeq}$.

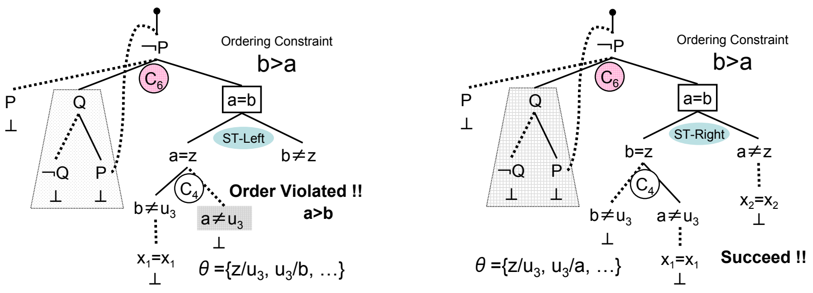


Figure 4: Two connection tableaux in **CTwS** for nSMT($\mathcal{S}_1$)

Firstly, we study a simulation of strong connection using the clause D_1 in SMT($\mathcal{S}$). Consider an expansion inference for D_1 in $\mathbf{CT}^{\simeq}$.

$$\dfrac{L}{s \simeq z \quad \mathbf{nv_2} \not\simeq z \quad K'_1 \quad \cdots \quad K'_m}\ (\mathrm{Exp})$$

If L is a negative equality $\mathbf{nv_1} \not\simeq r$ such that $\mathbf{nv_1}$ is non-variable and is unifiable with s, then the following strong connection is available in $\mathbf{CT}^{\simeq}$:

(1) $$\dfrac{\mathbf{nv_1} \not\simeq r}{\dfrac{s \simeq z}{\bot \cdot (\mathbf{nv_1} = s \succ z = r)}\ (\mathrm{SC}) \quad \mathbf{nv_2} \not\simeq z \quad K'_1 \ \cdots \ K'_m}\ (\mathrm{Exp})$$

On the other hand, if L is a positive equality $u \simeq v$ such that u is unifiable with $\mathbf{nv_2}$, then we have the following strong connection in $\mathbf{CT}^{\simeq}$:

(2) $$\dfrac{u \simeq v}{s \simeq z \quad \dfrac{\mathbf{nv_2} \not\simeq z}{\bot \cdot (\mathbf{nv_2} = u \succ v = z)}\ (\mathrm{SC}) \quad K'_1 \ \cdots \ K'_m}\ (\mathrm{Exp})$$

The above first inference (1) in $\mathbf{CT}^{\simeq}$ can be simulated in nSMT($\mathcal{S}$) with the new expansion rule and ST-splitting for a raw equality $\boxed{s \simeq \mathbf{nv_2}}$ and the *weak connection* rule as follows:

$$\dfrac{\mathbf{nv_1} \not\simeq r}{\dfrac{\boxed{s \simeq \mathbf{nv_2}}}{\dfrac{s \simeq z}{\bot \cdot (\mathbf{nv_1} = s \succ z = r)}\ (\mathrm{WC}) \quad \mathbf{nv_2} \not\simeq z}\ (\mathrm{ST}) \quad K'_1 \ \cdots \ K'_m}\ (\text{new Exp})$$

However, it is definitely better to use a sort of strong connection rule instead of the weak connection, because a connection constraint for a tableau becomes much simpler and more effective to drastically reduce the search space. We, hence, introduce a new strong connection rule which can perform the above inference steps as an integrated one-step inference in **CTwS**. The following is a naive form for directly simulating the inference (1):

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq \mathbf{nv_2}}}{\bot \cdot (\mathbf{nv_1} = s \succ z = r) \qquad \mathbf{nv_2} \not\simeq z}$$

where $\mathbf{nv_1}$ and $\mathbf{nv_2}$ are non-variable terms. We can eliminate the link variable z because z never occurs elsewhere in a tableau, and moreover we can add an ordering constraint. The final form of the above rule is:

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq \mathbf{nv_2}}}{\bot \cdot (\mathbf{nv_1} = s \succ r) \quad \mathbf{nv_2} \not\simeq r \cdot (\mathbf{nv_2} \succeq r)}$$

Remark: The above ordering constraint $\mathbf{nv_2} \succeq r$ is not explicitly used in the strong connection in **CT**$^{\simeq}$, as shown in the inference (1). Thus this additional constraint can reduce the alternative choices of expansion rules, compared with **CT**$^{\simeq}$. Recall the term $\mathbf{nv_2}$ initially occurs as an argument of the equality $s \simeq \mathbf{nv_2}$ of the original clause $s \simeq \mathbf{nv_2} \vee K_1 \vee \cdots \vee K_m$ in $\mathcal{S}$. Thus we can say, **CTwS** directly uses full information of the equality $s \simeq \mathbf{nv_2}$ for strong connection and thus expansion, while **CT**$^{\simeq}$ just uses this information indirectly through variable binding for a linked variable.[4] This difference is a rather important point because several state-of-arts top-down provers, such as SETHEO [6] and SOLAR [8], often reorder goals for improving the efficiency of inferences.

Similarly, the above inference (2) can also be simulated in nSMT$(\mathcal{S})$ with a raw positive equality $\boxed{s \simeq \mathbf{nv_2}}$ as follows:

$$\cfrac{u \simeq v}{\cfrac{\boxed{s \simeq \mathbf{nv_2}}}{s \simeq z \qquad \cfrac{\mathbf{nv_2} \not\simeq z}{\bot \cdot (\mathbf{nv_2} = u \succ v = z)}\ \text{(WC)}}\ \text{(ST)} \qquad K_1' \ \cdots\ K_m'}\ \text{(new Exp)}$$

This observation leads to the following rule, which can achieve the above inference steps as a single inference.

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \Gamma,\ u \simeq v,\ \boxed{s \simeq \mathbf{nv_2}}}{s \simeq z \quad \bot \cdot (\mathbf{nv_2} = u \succ v = z)}$$

We can also eliminate the link variable z and add an additional ordering for $s \simeq z$ without losing completeness. Finally, we obtain the following new rule:

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \Gamma,\ u \simeq v,\ \boxed{s \simeq \mathbf{nv_2}}}{s \simeq v \cdot (s \succ v) \quad \bot \cdot (\mathbf{nv_2} = u \succ v)}$$

Notice that this rule superficially requires a *positive* raw literal $\boxed{s \simeq \mathbf{nv_2}}$ as a partner of strong connection of a *positive* literal $u \simeq v$.

Next we study a simulation of strong connection using the clause D_2. Consider the following inference involving expansion and strong connection of D_2 in **CT**$^{\simeq}$.

(3) $$\cfrac{\mathbf{nv_1} \not\simeq r}{\cfrac{s \simeq x}{\bot \cdot (\mathbf{nv_1} = s \succ x = r)}\ \text{(SC)} \qquad K_1' \ \cdots\ K_m'}\ \text{(Exp)},$$

[4]See the variable binding of z in the inference (1), for example.

where $\mathbf{nv_1}$ is a non-variable term. The above (3) can simply be simulated in nSMT$(\mathcal{S})$ with the raw equality $\boxed{s \simeq x}$ as follows:

$$\cfrac{\cfrac{\cfrac{\mathbf{nv_1} \not\simeq r}{\boxed{s \simeq x}}\ (\text{new Exp}),}{s \simeq x}\ (\text{ST}) \qquad K'_1 \ \cdots \ K'_m}{\bot \cdot (\mathbf{nv_1} = s \succ x = r)}\ (\text{WC})$$

This observation derives the following strong connection rule in **CTwS**:

$$\frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq x}}{\bot \cdot (\mathbf{nv_1} = s \succ x = r)}$$

Moreover, we have to investigate inferences using strong connections with the clauses D_3 and D_4 of SMT$(\mathcal{S})$, and can derive additional three rules for nSMT$(\mathcal{S})$ by similar discussions. Eventually, we obtain the following set of strong connection rules for nSMT$(\mathcal{S})$:

Strong Connection for Negative Equality in nSMT$(\mathcal{S})$:

$$\frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq \mathbf{nv_2}}}{\bot \cdot (\mathbf{nv_1} = s \succ r) \quad \mathbf{nv_2} \not\simeq r \cdot (\mathbf{nv_2} \succeq r)} \qquad \frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq x}}{\bot \cdot (\mathbf{nv_1} = s \succ x = r)}$$

$$\frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{\mathbf{nv_2} \simeq t}}{\bot \cdot (\mathbf{nv_1} = t \succ r) \quad \mathbf{nv_2} \not\simeq r \cdot (\mathbf{nv_2} \succeq r)} \qquad \frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{x \simeq t}}{\bot \cdot (\mathbf{nv_1} = t \succ x = r)}$$

Strong Connection for Positive Equality in nSMT$(\mathcal{S})$:

$$\frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ u \simeq v,\ \boxed{s \simeq \mathbf{nv_2}}}{s \simeq v \cdot (s \succ v) \quad \bot \cdot (\mathbf{nv_2} = u \succ v)} \qquad \frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ u \simeq v,\ \boxed{\mathbf{nv_2} \simeq t}}{t \simeq v \cdot (t \succ v) \quad \bot \cdot (\mathbf{nv_2} = u \succ v)}$$

where $\mathbf{nv_1}$ and $\mathbf{nv_2}$ denote non-variable terms, x is a variable.

We show a total view of the connection tableaux **CTwS** for nSMT$(\mathcal{S})$ in Fig. 5. The following is the first main theorem of this paper:

Theorem 5 *The calculus* **CTwS** *is sound and complete. That is, $\mathcal{S} \cup \mathcal{E}$ is unsatisfiable iff there is a closed and strongly connected tableau in* **CTwS** *for nSMT$(\mathcal{S})$.*

3.3 Yet another Connection Tableaux for Modification

In this section, we consider yet another connection tableaux, called **CTwST**, where the strong connection for positive equality is further improved with a more strict ordering constraint. As was shown in the previous subsection, one of the strong connection for a positive equality for nSMT$(\mathcal{S})$ is:

SC-PosE-1: $$\frac{\text{nSMT}(\mathcal{S}) \ || \ \Gamma,\ s \simeq t,\ \mathbf{nv_1} \not\simeq r}{\bot \cdot (\mathbf{nv_1} = s \succ t = r)}$$

Expansion (Exp):

$$\frac{\text{nSMT}(\mathcal{S}),\ (L_1 \vee \cdots \vee L_k)\ \ ||\ \ \Gamma}{[L_1]\ \ \cdots\ [L_k]}$$

Equality Resolution (ER)

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ l \not\simeq r}{\bot \cdot (l = r)}$$

ST Splitting (ST):

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{s \simeq \mathbf{nv_1}}}{s \simeq z \quad \mathbf{nv_1} \not\simeq z}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{s \simeq x}}{s \simeq x}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{\mathbf{nv_1} \simeq t}}{t \simeq z \quad \mathbf{nv_1} \not\simeq z}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \boxed{x \simeq t}}{t \simeq x}$$

Strong Connection for Non-Equality **(SC–NonE):**

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \neg P(r), P(s)}{\bot \cdot (r = s)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, P(r), \neg P(s)}{\bot \cdot (r = s)}$$

Strong Connection for Neg. Equality **(SC–NegE):**

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq \mathbf{nv_2}}}{\bot \cdot (\mathbf{nv_1} = s \succ r) \quad \mathbf{nv_2} \not\simeq r \cdot (\mathbf{nv_2} \succeq r)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{s \simeq x}}{\bot \cdot (\mathbf{nv_1} = s \succ x = r)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{\mathbf{nv_2} \simeq t}}{\bot \cdot (\mathbf{nv_1} = t \succ r) \quad \mathbf{nv_2} \not\simeq r \cdot (\mathbf{nv_2} \succeq r)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ \mathbf{nv_1} \not\simeq r,\ \boxed{x \simeq t}}{\bot \cdot (\mathbf{nv_1} = t \succ x = r)}$$

Strong Connection for Pos. Equality **(SC–PosE):**

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ s \simeq t,\ \mathbf{nv_1} \not\simeq r}{\bot \cdot (\mathbf{nv_1} = s \succ t = r)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ u \simeq v,\ \boxed{s \simeq \mathbf{nv_1}}}{s \simeq v \cdot (s \succ v) \quad \bot \cdot (\mathbf{nv_1} = u \succ v)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma,\ u \simeq v,\ \boxed{\mathbf{nv_1} \simeq t}}{t \simeq v \cdot (t \succ v) \quad \bot \cdot (\mathbf{nv_1} = u \succ v)}$$

Weak Connection (WC):

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \neg P(r), \Delta, P(s)}{\bot \cdot (r = s)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, P(r), \Delta, \neg P(s)}{\bot \cdot (r = s)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, \mathbf{nv_1} \not\simeq r, \Delta, s \simeq t}{\bot \cdot (\mathbf{nv_1} = s \succ t = r)}$$

$$\frac{\text{nSMT}(\mathcal{S})\ \ ||\ \ \Gamma, s \simeq t, \Delta, \mathbf{nv_1} \not\simeq r}{\bot \cdot (\mathbf{nv_1} = s \succ t = r)}$$

Figure 5: Connection tableaux **CTwS** for nSMT($\mathcal{S}$)

Recall that T-modification splits a given negative equality $l \not\simeq r$ into the disjunction $l \not\simeq z \vee r \not\simeq z$ if r is not a variable. Thus, the literal $l \not\simeq z$ (or $r \not\simeq z$) loses the information about the initial partner term r (or respectively, l). Thus the above strong connection rule cannot utilize full information provided by negative equalities in a clause set $\mathcal{S}$ which is initially given. As a remedy, we omit T-modification for negative equality literals as well, and instead give a set of new connection rules for preserving transitivity

Definition 2 We define nSM($\mathcal{S}$) to be a a set of normal clauses obtained from $\mathcal{S}$ by just applying negative S-modification and M-modification.

The calculus **CTwST** differs from **CTwS** in the following points; firstly **CTwST** accepts nSM($\mathcal{S}$) as an input set of clauses, not nSMT($\mathcal{S}$); secondly we add a new expansion rule and T-splitting rules for treating a *raw negative* equality; thirdly we replace the strong connection **SC-PosE-1** with new three rules with *raw negative* equalities. We modify the expansion rule to the one which produces raw literals both for positive and negative equalities. Given a literal L, we write $[[L]]$ to denote the framed literal $\boxed{L}$, called a *raw literal* if L is a positive equality $s \simeq t$ or a negative equality $s \not\simeq t$; otherwise $[[L]]$ denotes L itself.

Expansion Rule for nSM($\mathcal{S}$):

$$\frac{\text{nSM}(\mathcal{S}),\ (L_1 \vee \cdots \vee L_k)\ \ ||\ \ \Gamma}{[[L_1]]\ \ \cdots\ \ [[L_k]]}$$

We add the following T-splitting rules in order to treating raw negative equalities, which naturally correspond with T-modification.

T-Splitting for Negative Equality for nSM($\mathcal{S}$):[-0.5ex]

$$\frac{\text{nSM}(\mathcal{S})\ \ ||\ \ \Gamma,\ \boxed{s \not\simeq \mathbf{nv_1}}}{s \not\simeq z \quad \mathbf{nv_1} \not\simeq z} \qquad\qquad \frac{\text{nSM}(\mathcal{S})\ \ ||\ \ \Gamma,\ \boxed{s \not\simeq y}}{s \not\simeq y}$$

At last, we replace the rule **SC-PosE-1** by the following three rules:

Strong Connection for Positive Equality for nSM($\mathcal{S}$):

$$\frac{\text{nSM}(\mathcal{S})\ \ ||\ \ \Gamma, l \simeq r,\ \boxed{\mathbf{nv_1} \not\simeq \mathbf{nv_2}}}{\bot \cdot (\mathbf{nv_1} = l \succ r) \quad \mathbf{nv_2} \not\simeq r \cdot (\mathbf{nv_2} \succeq r)} \qquad\qquad \frac{\text{nSM}(\mathcal{S})\ \ ||\ \ \Gamma, l \simeq r,\ \boxed{s \not\simeq \mathbf{nv_2}}}{s \not\simeq r \cdot (s \succeq r) \quad \bot \cdot (\mathbf{nv_2} = l \succ r)}$$

$$\frac{\text{nSM}(\mathcal{S})\ \ ||\ \ \Gamma, l \simeq r,\ \boxed{\mathbf{nv_1} \not\simeq y}}{\bot \cdot (\mathbf{nv_1} = l \succ r = y)}$$

4 Extended SOLAR and Experimental Evaluation

In this section, we show some tentative experimental results with **SOLAR** [8], which is an efficient consequence finding program based on Skipping Ordered Linear Resolution [4] by using Connection Tableaux technology [6, 5]. At first we show the basic

Table 1: Basic performance comparison of theorem provers

	SOLAR	Otter	E 1.0	E 1.0 (A)*
# of solved unit EQ.	170	474	589	630
# of solved non-unit EQ.	676	727	907	2013
# of solved non-EQ.	1163	1044	1131	1640

* Note: (A) means that E system uses the option"-xAuto -tAuto".

performance of SOLAR compared with state-of-the-art theorem provers Otter 3.0 [7] and E 1.0 [11]. Table 1 shows the numbers of problems of TPTP library v.3.5.0 which each theorem prover can solve within the time limit of 60 CPU seconds. The first row is for unit equation problems; the second is non-unit equational ones; the third is for non-equational ones. SOLAR is competitive for the class of non-equational problems, but is not for equational problems.[5]

Table 2 shows the performances of several kinds of equality computation methods in connection tableaux.[6] The first "Axioms" indicates a naive use of the congruence axioms, and the second "M-mod" represents a method for using just M-modification together with reflexivity, symmetry and transitivity axioms. Each row denoted by "infer." is the sum total of the numbers of inferences needed for equational problems which can commonly be solved by all of $\mathbf{CT}^{\simeq}$, **CTwS** and **CTwST**. The upper half of Table 2 shows the results obtained by using ordinary M-modification, while the lower half is for the ones obtained by using a semi-optimized M-modification, given in [1], such that the flattening never applies to any occurrences of an ordering-minimal constant symbol. Regretfully, the best performance is provided by the naive use method of the congruence axioms. Modification methods commonly inherit a disadvantage caused by M-modification which increases the length of each clause by flattening. $\mathbf{CT}^{\simeq}$ and **CTwS**, however, significantly decrease the number of inference steps from M-modification method. With the semi-optimized M-modification, **CTwS** is superior to $\mathbf{CT}^{\simeq}$. Certainly, **CTwS** decreases the amount of inference steps compared with $\mathbf{CT}^{\simeq}$, which means that **CTwS** succeeds to prevent redundant computations originating in S-modification. By comparison between the upper part and the lower one in Table 2, we can understand the importance of optimization of M-modification for avoiding redundant computations, which are invoked by long disjunctions of *thin* negative equalities produced by flattening operations.

5 Conclusion and Future Work

We investigated Paskevich's connection tableaux for equality computation, and pointed out that a naive use of S-modification is problematic. We proposed, as some remedies, improved connection tableau calculi for efficient equality computation. We also showed tentative experimental results of evaluating the proposed methods using SOLAR. This research is now in progress. For example, we are still studying a further

[5]TPTP library v.3.5.0 has 2,175 non-equational problems and 4,171 equational problems, where there are 863 unit equational problems.

[6]Throughout experiments, we used non-recursive Knuth-Bendix ordering given by Riazanov and Voronkov [10]. as a reduction ordering.

Table 2: Comparison of equality computation methods in connection tableaux

	Axioms	M-mod	**CT**$^{\simeq}$	**CTwS**	**CTwST**
# of unit EQ.	170	161	180	183	179
# of non-unit EQ.	636	490	507	499	489
# of infer. of unit EQ.	4,883K	12,900K	8,903K	1,403K	2,367K
# of infer. of non-unit EQ.	38,621K	251,244K	86,837K	78,339K	119,094K
# of unit EQ.	—	—	183	185	183
# of non-unit EQ.	—	—	518	540	512
# of infer. of unit EQ.	—	—	5,545K	5,212K	8,397K
# of infer. of non-unit EQ.	—	—	66,529K	58,588K	86,253K

improvement of M-modification. Moreover, we found that the dynamic term-binding to variables in derivations frequently gives ill effects on the behaviors of **CTwS** and **CTwST**. In order to improve this situation, we will re-formalize our methods in the context of the basic method and the closure mechanism in the near future. Furthermore, one of anonymous referees suggested that the effects of ST-splitting can be achieved by the following clause transformation:

$$s \approx t \vee C \quad \Rightarrow \quad P_{new}(\vec{x}) \vee C,\ \neg P_{new}(\vec{x}) \vee s \simeq t \text{ and } \neg P_{new}(\vec{x}) \vee t \simeq s$$

where P_{new} is a new predicate symbol and $\vec{x}$ denotes the list of variables occurring in s and t. Notice that the literal $P_{new}(\vec{x})$ corresponds to a raw equality in our framework. This rule can be used for simulating ST-splitting instead of positive S-modification rule. This method seems to have a great possibility in several aspects. We are now conducting some theoretical studies and experimental evaluations.

References

[1] L. Bachmair, H. Ganzinger and A. Voronkov: Elimination of equality via transformation with ordering constraints. *Proc. CADE-15, LNCS*, Vol.1421, pp.175-190 (1998)

[2] D. Brand: Proving theorems with the modification method. *SIAM Journal of Computing*, Vol.4, pp.412-430 (1975)

[3] P. Baumgartner and C. Tinelli: The model evolution calculus with equality. *Proc. CADE-20, LNAI.* Vol.3632, pp.392–408 (2005)

[4] K. Inoue: Linear resolution for consequence finding. *Artificial Intelligence* Vol.56 pp.301–353 (1992)

[5] K. Iwanuma, K. Inoue and K. Satoh: Completeness of pruning methods for consequence finding procedure SOL. *Proc. Int. Workshop on First-order Theorem Proving (FTP2000)*, pp.89-100 (2000).

[6] R. Letz, C. Goller and K. Mayr: Controlled integration of the cut rule into connection tableau calculi. *J. Automated Reasoning*, Vol.13, pp.297-338 (1994).

[7] W. McCune: Skolem functions and equality in automated deduction. *Proc. AAAI-90*, pp.246–251 (1990)

[8] H. Nabeshima, K. Iwanuma and K. Inoue: SOLAR: a consequence finding system for advanced reasoning. *Proc. Tableaux'03. LNAI* Vol.2796, pp.257-263 (2003)

[9] A. Paskevich: Connection tableaux with lazy paramodulation. *J. Automated Reasoning* (2008).

[10] A. Riazanov: Implementing an efficient theorem prover. PhD thesis, University of Manchester, 2002.

[11] St. Schulz: System description E 0.81. *Proc IJCAR-2004, LNAI*, Vol.3097, pp.223-228 (2004)

[12] W. Snyder and C. Lynch: Goal directed strategies for paramodulation, *Proc. RTA-91, LNCS* Vol.448, pp.150–111 (1991).

Redundancy Elimination in Monodic Temporal Reasoning

Michel Ludwig[1] and Ullrich Hustadt[1]

Department of Computer Science, University of Liverpool, United Kingdom
{Michel.Ludwig,U.Hustadt}@liverpool.ac.uk

Abstract. The elimination of redundant clauses is an essential part of resolution-based theorem proving in order to reduce the size of the search space. In this paper we focus on ordered fine-grained resolution with selection, a sound and complete resolution-based calculus for monodic first-order temporal logic. We define a subsumption relation on temporal clauses, show how the calculus can be extended with reduction rules that eliminate redundant clauses, and we illustrate the effectiveness of redundancy elimination with some experiments.

1 Introduction

Monodic first-order temporal logic [9] is a fragment of first-order temporal logic (without equality) which, in contrast to first-order temporal logic itself, has a semi-decidable validity and satisfiability problem. Besides semi-decidability, monodic first-order temporal logic enjoys a number of other beneficial properties, e.g. the existence of non-trivial decidable subclasses, complete reasoning procedures, etc.

In addition to a tableaux-based calculus [13] for monodic first-order temporal logic, several resolution-based calculi have been proposed for the logic, starting with (monodic) temporal resolution [3]. A more machine-oriented version, the fine-grained first-order temporal resolution calculus, was described in [12]. Subsequently, a refinement of fine-grained temporal resolution, the ordered fine-grained temporal resolution with selection calculus, was presented in [10].

Essentially, the inference rules of ordered fine-grained resolution with selection can be classified into two different categories. The majority of the rules are based on standard first-order resolution between different types of temporal clauses. The remaining inference rules, the so-called eventuality resolution rules, reflect the induction principle that holds for monodic temporal logic over a flow of time isomorphic to the natural numbers. The applicability of the rules in this second category is only semi-decidable, making the construction of fair derivations, that is, derivations in which every non-redundant clause that is derivable from a given clause set is eventually derived, a non-trivial problem. A new inference procedure solving this problem has been recently been described in [15] and is implemented in the theorem prover TSPASS [14].

In this paper we focus on a different aspect of ordered fine-grained temporal resolution with selection, namely, redundancy elimination. The use of an ordering and a selection function which restricts inferences to literals which are selected or, in the absence of selected literals, to (strictly) maximal literals, already reduces the possible inferences considerably. However, it cannot prevent the derivation of redundant clauses, e.g. tautological clauses or clauses which are subsumed by other, simpler, clauses. Redundancy elimination is therefore an important ingredient for practical resolution calculi and for theorem provers based on such calculi.

The paper is organised as follows. In Section 2 we briefly recall the syntax and semantics of monodic first-order temporal logic. The ordered fine-grained resolution with selection calculus is presented in Section 3. In Sections 4 and 5 we show how redundancy elimination can be added to the calculus. Finally, in Section 6 we briefly discuss how redundancy elimination fits with our implementation of the calculus and present some experimental results which show the effectiveness of redundancy elimination.

2 First-Order Temporal Logic

We assume the reader to be familiar with first-order logic and associated notions, including, for example, terms and substitutions.

Then, the language of First-Order (Linear Time) Temporal Logic, FOTL, is an extension of classical first-order logic by temporal operators for a discrete linear model of time (i.e. isomorphic to $\mathbb{N}$). The vocabulary of FOTL (without equality and function symbols) is composed of a countably infinite set X of *variables* x_0, x_1, …, a countably infinite set of *constants* c_0, c_1, …, a non-empty set of *predicate symbols* P, P_0, …, each with a fixed arity ≥ 0, the *propositional operators* $\top$ (**true**), $\neg$, $\vee$, the *quantifiers* $\exists x_i$ and $\forall x_i$, and the *temporal operators* $\Box$ ('always in the future'), $\Diamond$ ('eventually in the future'), $\bigcirc$ ('at the next moment'), U ('until') and W ('weak until') (see e.g. [7]). We also use $\bot$ (**false**), $\wedge$, and $\Rightarrow$ as additional operators, defined using $\top$, $\neg$, and $\vee$ in the usual way. The set of FOTL formulae is defined as follows: $\top$ is a FOTL formula; if P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are variables or constants, then $P(t_1, \ldots, t_n)$ is an *atomic* FOTL formula; if φ and ψ are FOTL formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\exists x\varphi$, $\forall x\varphi$, $\Box\varphi$, $\Diamond\varphi$, $\bigcirc\varphi$, $\varphi \,\mathsf{U}\, \psi$, and $\varphi \,\mathsf{W}\, \psi$. Free and bound variables of a formula are defined in the standard way, as well as the notions of open and closed formulae. For a given formula φ, we write $\varphi(x_1, \ldots, x_n)$ to indicate that all the free variables of φ are among $x_1, \ldots, x_n$. As usual, a *literal* is either an atomic formula or its negation, and a *proposition* is a predicate of arity 0.

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

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

Fig. 1. Truth-Relation for First-Order Temporal Logic

definition of the *truth relation* $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$ (only for those $\mathfrak{a}$ such that $\mathfrak{a}(x) \in D_n$ for every variable x) is given in Fig. 1.

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

A structure $\mathfrak{M} = (D_n, I_n)_{n \in \mathbb{N}}$ is said to be a *model* for a formula φ if and only if for every assignment $\mathfrak{a}$ with $\mathfrak{a}(x) \in D_0$ for every variable x it holds that $\mathfrak{M}_0 \models^{\mathfrak{a}} \varphi$. A formula is *satisfiable* if and only there exists a model for φ. A formula φ is *valid* if and only if every temporal structure $\mathfrak{M} = (D_n, I_n)_{n \in \mathbb{N}}$ is a model for φ.

The set of valid formulae of this logic is not recursively enumerable. However, the set of valid *monodic* formulae is known to be finitely axiomatisable [19]. A formula φ of FOTL is called *monodic* if any subformula of φ of the form $\bigcirc\psi$, $\Box\psi$, $\Diamond\psi$, $\psi_1 \mathsf{U} \psi_2$, or $\psi_1 \mathsf{W} \psi_2$ contains at most one free variable. For example, the formulae $\exists x \Box \forall y P(x, y)$ and $\forall x \Box P(c, x)$ are monodic, whereas the formula $\forall x \exists y (Q(x, y) \Rightarrow \Box Q(x, y))$ is not monodic.

Every monodic temporal formula can be transformed into an equi-satisfiable normal form, called *divided separated normal form (DSNF)* [12].

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

We associate with each monodic temporal problem $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ the monodic FOTL formula $\mathcal{I} \wedge \Box\mathcal{U} \wedge \Box\forall x \mathcal{S} \wedge \Box\forall x \mathcal{E}$. When we talk about particular properties of a temporal problem (e.g., satisfiability, validity, logical consequences, etc) we refer to properties of this associated formula.

The transformation to DSNF is based on a renaming and unwinding technique which substitutes non-atomic subformulae by atomic formulae with new predicate symbols and replaces temporal operators by their fixed point definitions as described, for example, in [8].

Theorem 1 (see [4], Theorem 3.4). *Any monodic formula in first-order temporal logic can be transformed into an equi-satisfiable monodic temporal problem in DSNF with at most a linear increase in the size of the problem.*

The main purpose of the divided separated normal form is to cleanly separate different temporal aspects of a FOTL formula from each other. For the resolution calculus in this paper we will need to go one step further by transforming the universal and initial part of a monodic temporal problem into clause normal form.

Definition 2. *Let $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ be a monodic temporal problem. With every eventuality $\Diamond L(x) \in \mathcal{E}$ and constant c occurring in P we uniquely associate a propositional symbol p_c^L. Then the* classification $\mathrm{Cls}(\mathsf{P})$ of P *is a quadruple $\langle \mathcal{U}', \mathcal{I}', \mathcal{S}', \mathcal{E}' \rangle$ such that $\mathcal{U}'$ is a set of clauses*[1]*, called* universal clauses*, consisting of the clausification of $\mathcal{U}$ and clauses $\neg p_c^L \vee L(c)$ for every $\Diamond L(x) \in \mathcal{E}$ and constant c occurring in P; $\mathcal{I}'$ is a set of clauses, called* initial clauses*, obtained by clausification of $\mathcal{I}$; $\mathcal{S}'$ is the smallest set of step clauses such that all step clauses from $\mathcal{S}$ are in $\mathcal{S}'$ and for every non-ground step clause $P(x) \Rightarrow \bigcirc L(x)$ in $\mathcal{S}$ and every constant c occurring in P, the clause $P(c) \Rightarrow \bigcirc L(c)$ is in $\mathcal{S}'$; $\mathcal{E}'$ is the smallest set of eventuality clauses such that all eventuality clauses from $\mathcal{E}$ are in $\mathcal{E}'$ and for every non-ground eventuality clause $\Diamond L(x)$ in $\mathcal{E}$ and every constant c occurring in P, the eventuality clause $\Diamond p_c^L$ is in $\mathcal{E}'$.*

One has to note that new constants and, especially, function symbols of an arbitrary arity can be introduced during the Skolemization process. As a consequence it is not possible in general to instantiate every variable that occurs in the original problem with all the constants and function symbols. On the other hand, the variables occurring in the step and eventuality clauses have to be instantiated with the constants that are present in the *original* problem (before Skolemization) in order to ensure the completeness of the calculus presented in Section 3.

Note further that more general step clauses with more than one atom on the left-hand side and more than one literal on the right-hand side can be derived by the calculus introduced in Section 3. In what follows $\mathcal{U}$ denotes the (current) universal part of a monodic temporal problem P.

3 Ordered Fine-Grained Resolution with Selection

We assume that we are given an *atom ordering* $\succ$, that is, a strict partial ordering on ground atoms which is well-founded and total, and a *selection function* S

[1] Clauses, as well as disjunctions and conjunctions, will be considered as multisets.

which maps any first-order clause C to a (possibly empty) subset of its negative literals and which is instance compatible:

Definition 3. *We say that a selection function S is* instance compatible *if and only if for every clause C, for every substitution σ and for every literal $l \in C\sigma$ it holds that $\in S(C\sigma)$ iff there exists a literal $l' \in S(C)$ such that $l'\sigma = l$.*

The atom ordering $\succ$ is extended to ground literals by $\neg A \succ A$ and $(\neg)A \succ (\neg)B$ if and only if $A \succ B$. The ordering is extended on the non-ground level as follows: for two arbitrary literals L and L', $L \succ L'$ if and only if $L\sigma \succ L'\sigma$ for every grounding substitution σ. A literal L is called (strictly) maximal w.r.t. a clause C if and only if there is no literal $L' \in C$ with $L' \succ L$ ($L' \succeq L$). A literal L is *eligible* in a clause $L \vee C$ for a substitution σ if either it is selected in $L \vee C$, or otherwise no literal is selected in C and $L\sigma$ is maximal w.r.t. $C\sigma$.

The atom ordering $\succ$ and the selection function S are used to restrict the applicability of the deduction rules of fine-grained resolution as follows.

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

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

if σ is a most general unifier of the atoms A and B, *A is eligible in $(C_1 \vee A)$ for σ, and $\neg B$ is eligible in $(\neg B \vee C_2)$ for σ.* The result is a universal clause.

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

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

if σ is a most general unifier of the atoms A and B, and *A is eligible in $(C_1 \vee A \vee B)$ for σ.* The result is again a universal clause.

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

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

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

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

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

where $C_1 \Rightarrow \bigcirc(D_1 \vee A)$ is a step clause, $D_2 \vee \neg B$ is a universal clause, and σ is a most general unifier of the atoms A and B such that σ does not map variables from C_1 into a constant or a functional term, *A is eligible in $(D_1 \vee A)$ for σ, and $\neg B$ is eligible in $(D_2 \vee \neg B)$ for σ.* There also exists an analogous rule where the positive literal A is contained in a universal clause and the negative literal $\neg B$ in a step clause.

(5) *Ordered fine-grained positive step factoring with selection.*

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

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

(6) *Clause conversion.* A step clause of the form $C \Rightarrow \bigcirc\bot$ is rewritten to the universal clause $\neg C$.
Step clauses of the form $C \Rightarrow \bigcirc\bot$ will also be called *terminating* or *final* step clauses.

(7) *Duplicate literal elimination in left-hand sides of terminating step clauses.* A clause of the form $(C \wedge A \wedge A) \Rightarrow \bigcirc\bot$ yields the clause $(C \wedge A) \Rightarrow \bigcirc\bot$.

(8) *Eventuality resolution rule w.r.t.* $\mathcal{U}$*:*

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

where $\forall x(\mathcal{A}_i(x) \Rightarrow \bigcirc\mathcal{B}_i(x))$ are formulae computed from the set of step clauses such that for every i, $1 \leq i \leq n$, the *loop* side conditions $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \neg L(x))$ and $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \bigvee_{j=1}^{n}(\mathcal{A}_j(x)))$ are valid.[2]
The set of full merged step clauses, satisfying the loop side conditions, is called a *loop in* $\Diamond L(x)$ and the formula $\bigvee_{j=1}^{n} \mathcal{A}_j(x)$ is called a *loop formula.* More details can be found in [10].

(9) *Ground eventuality resolution rule w.r.t.* $\mathcal{U}$*:*

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

where $\mathcal{A}_i \Rightarrow \bigcirc\mathcal{B}_i$ are ground formulae computed from the set of step clauses such that for every i, $1 \leq i \leq n$, the *loop* side conditions $\mathcal{U} \wedge \mathcal{B}_i \models \neg l$ and $\mathcal{U} \wedge \mathcal{B}_i \models \bigvee_{j=1}^{n} \mathcal{A}_j$ are valid. The notions of *ground loop* and *ground loop formula* are defined similarly to the case above.

Rules (1) to (7), also called rules of *fine-grained step resolution*, are either identical or closely related to the deduction rules of ordered first-order resolution with selection; a fact that we exploit in our implementation of the calculus. The condition in rules (4) and (5) that a unifier σ may not map variables from the antecedent into a constant or a functional term is a consequence of the expanding domain assumption. Without this restriction, the calculus would be unsound [12, Example 5].

Loop formulae, which are required for applications of the rules (8) and (9), can be computed by the fine-grained breadth-first search algorithm (FG-BFS), depicted in Fig. 2. In this algorithm, LT($\mathcal{S}$) is the minimal set of clauses containing $\mathcal{S}$ such that for every non-ground step clause $(P(x) \Rightarrow \bigcirc M(x)) \in \mathcal{S}$, the set LT($\mathcal{S}$) contains the clause $P(c^l) \Rightarrow \bigcirc M(c^l)$ (c^l is a constant used only for loop search). The process of running the FG-BFS algorithm is called *loop search.* A variant of the FG-BFS algorithm for handling ground eventualities also exists.

[2] In the case $\mathcal{U} \models \forall x \neg L(x)$, the *degenerate clause*, $\top \Rightarrow \bigcirc\top$, can be considered as a premise of this rule; the conclusion of the rule is then $\neg\top$.

```
Function FG-BFS
Input:   A set of universal clauses $\mathcal{U}$ and a set of step clauses $\mathcal{S}$, saturated
         by ordered fine-grained resolution with selection, and an eventuality
         clause $\Diamond L(x) \in \mathcal{E}$.
Output:  A loop formula $H(x)$ with at most one free variable.
Method:  (1) Let $H_0(x) = \mathbf{true}$; $\mathcal{M}_0 = \emptyset$; $i = 0$
         (2) Let $\mathcal{N}_{i+1} = \mathcal{U} \cup \mathrm{LT}(\mathcal{S}) \cup \{\mathbf{true} \Rightarrow \bigcirc(\neg H_i(c^l) \vee L(c^l))\}$. Apply
             the rules of ordered fine-grained resolution with selection except
             the clause conversion rule to $\mathcal{N}_{i+1}$. If we obtain a contradiction,
             then return the loop true (in this case $\forall x \neg L(x)$ is implied by the
             universal part).
             Otherwise let $\mathcal{M}_{i+1} = \{C_j \Rightarrow \bigcirc\bot\}_{j=1}^{n}$ be the set of all new
             terminating step clauses in the saturation of $\mathcal{N}_{i+1}$.
         (3) If $\mathcal{M}_{i+1} = \emptyset$, return false; else let $H_{i+1}(x) = \bigvee_{j=1}^{n}(\tilde{\exists} C_j)\{c^l \rightarrow x\}$
         (4) If $\forall x(H_i(x) \Rightarrow H_{i+1}(x))$, return $H_{i+1}(x)$.
         (5) i = i + 1; goto 2.
Note: The constant $c^l$ is a fresh constant used for loop search only
```

Fig. 2. Breadth-First Search Algorithm Using Fine-Grained Step Resolution.

Let *ordered fine-grained resolution with selection* be the calculus consisting of the rules (1) to (7) above, together with the ground and non-ground eventuality resolution rules described above, i.e. rules (8) and (9). We denote this calculus by $\mathfrak{I}_{FG}^{S,\succ}$.

Definition 4 (Derivation). *A (linear)* derivation *Δ (in $\mathfrak{I}_{FG}^{S,\succ}$) from the clausification* $\mathrm{Cls}(\mathsf{P}) = \langle \mathcal{U}_1, \mathcal{I}_1, \mathcal{S}_1, \mathcal{E} \rangle$ *of a monodic temporal problem P is a sequence of tuples $\Delta = \langle \mathcal{U}_1, \mathcal{I}_1, \mathcal{S}_1, \mathcal{E} \rangle, \langle \mathcal{U}_2, \mathcal{I}_2, \mathcal{S}_2, \mathcal{E} \rangle, \ldots$ such that each tuple at an index $i+1$ is obtained from the tuple at the index i by adding the conclusion of an application of one of the inference rules of $\mathfrak{I}_{FG}^{S,\succ}$ to premises from one of the sets $\mathcal{U}_i$, $\mathcal{I}_i$, $\mathcal{S}_i$ to that set, with the other sets as well as $\mathcal{E}$ remaining unchanged*[3].

*A derivation Δ such that the empty clause is an element of a $\mathcal{U}_i \cup \mathcal{I}_i$ is called a ($\mathfrak{I}_{FG}^{S,\succ}$-)*refutation *of $\langle \mathcal{U}_1, \mathcal{I}_1, \mathcal{S}_1, \mathcal{E} \rangle$.*

A derivation Δ is fair *if and only if for each clause C which can be derived from premises in $\langle \bigcup_{i\geq 1}\mathcal{U}_i, \bigcup_{i\geq 1}\mathcal{I}_i, \bigcup_{i\geq 1}\mathcal{S}_i, \mathcal{E} \rangle$ there exists an index j such that C occurs in $\langle \mathcal{U}_j, \mathcal{I}_j, \mathcal{S}_j, \mathcal{E} \rangle$.*

Ordered fine-grained resolution with selection is sound and complete for constant flooded monodic temporal problems over expanding domains as stated in the following theorem.

Theorem 2 (see [10], Theorem 5). *Let P be a monodic temporal problem. Let $\succ$ be an atom ordering and S an instance compatible selection function. Then P is unsatisfiable iff there exists a $\mathfrak{I}_{FG}^{S,\succ}$-refutation of* $\mathrm{Cls}(\mathsf{P})$. *Moreover, P is unsatisfiable iff any fair $\mathfrak{I}_{FG}^{S,\succ}$-derivation is a refutation of* $\mathrm{Cls}(\mathsf{P})$.

[3] In an application of ground eventuality or eventuality resolution rule, the set $\mathcal{U}$ in the definition of the rule refers to $\mathcal{U}_i$.

To prove Theorem 2, we show that any refutation of a temporal problem by the $\mathfrak{I}_e$ calculus of [12] there is a corresponding refutation by $\mathfrak{I}_{FG}^{S,\succ}$. Rule (7), which allows the elimination of duplicate literals in the left-hand sides of step clauses, is required to establish this correspondence.

4 Adding Redundancy Elimination

Given that our calculus uses an ordering refinement, it seems natural to establish that the calculus admits redundancy elimination by using the approach in [1, Section 4.2]. To do so, we would first need to define a model functor I that maps any (not necessarily satisfiable) temporal problem P not containing the empty clause to an interpretation $\mathfrak{M}_{\mathsf{P}}$ and then show that $\mathfrak{I}_{FG}^{S,\succ}$ has the reduction property for counterexamples with respect to the model functor I and the ordering $\succ$, that is, for every temporal problem P and minimal clause $\mathcal{C}$ in P which is false in $\mathfrak{M}_{\mathsf{P}}$, there exists an inference with (main) premise $\mathcal{C}$ and conclusion $\mathcal{D}$ that is also false in P but smaller than $\mathcal{C}$ wrt. $\succ$. We could then define a clause $\mathcal{C}$ to be redundant wrt. P if there exists clauses $\mathcal{C}_1, \ldots, \mathcal{C}_k$ in P such that $\mathcal{C}_1, \ldots, \mathcal{C}_k \models \mathcal{C}$ and $\mathcal{C} \succ \mathcal{C}_i$ for all i, $1 \leq i \leq k$, and it would be straightforward to show that $\mathfrak{I}_{FG}^{S,\succ}$ remains complete if redundant clauses are eliminated from derivations.

However, due to the presence of eventualities in temporal problems, defining an appropriate model functor is a non-trivial and open problem. For example, consider the satisfiable propositional temporal problem $\mathsf{P} = \langle \{p \vee q\}, \emptyset, \{p \Rightarrow \bigcirc\neg l\}, \{\Diamond l\}\rangle$ and an ordering $\succ$ such that $p \succ q$. Applying the standard model functor defined in [1] to the clause $p \vee q$ results in a model in which p is true. Given that $p \vee q$ is a universal clause, it would be natural to define $\mathfrak{M}_{\mathsf{P}}$ in such way that p is true at every moment of time. However, due to the step clause $p \Rightarrow \bigcirc\neg l$, $\Box\Diamond l$ is not true in $\mathfrak{M}_{\mathsf{P}}$ which means that $\mathfrak{M}_{\mathsf{P}}$ is not a model of P. Thus, this simplistic approach to defining a model functor is not correct for temporal problems containing eventualities.

We have recently introduced a model functor I for propositional temporal problems [16], which is able to associate a model $\mathfrak{M}$ of P with every satisfiable temporal problem P, but $\mathfrak{I}_{FG}^{S,\succ}$ is not reductive wrt. I. Thus, this model functor is not suitable for establishing that $\mathfrak{I}_{FG}^{S,\succ}$ admits redundancy elimination.

Thus, we have to follow a different approach in order to show how ordered fine-grained resolution with selection can be extended with redundancy elimination rules. In the following we will define the notions of a tautological clause and of a subsumed clause. In order to show that $\mathfrak{I}_{FG}^{S,\succ}$ is still complete if such clauses are eliminated during a derivation, we need to show that for every refutation without redundancy elimination there exists a refutation with redundancy elimination. It turns out that in order to be able to do so, we need to add two inference rules to our calculus and impose a restriction on the selection function.

First of all, we consider tautological clauses. As a tautological clause is defined to be a clause that is true in every structure $\mathfrak{M} = (D_n, I_n)_{n\in\mathbb{N}}$, we obtain the following lemma:

Lemma 1. *Let $\mathcal{C}$ be a initial, universal or step clause. Then:*

(i) If $\mathcal{C}$ is an initial or universal clause, then $\mathcal{C}$ is a tautology iff $\mathcal{C} = \neg L \vee L \vee C'$, for some possibly empty disjunction of literals C'.

(ii) If $\mathcal{C} = C_1 \Rightarrow \bigcirc C_2$ is a step clause, then $\mathcal{C}$ is a tautology iff $C_2 = \neg L \vee L \vee C_2'$, for some possibly empty disjunction of literals C_2'.

It has be noted that for point (ii) of the lemma above C_1 is assumed to be **true** or a non-empty conjunction of atoms.

Thus, just as in the non-temporal first-order case, there is again a syntactic criterion for characterising tautologies, namely the presence of complementary literals. For a set of clauses $\mathcal{N}$ (or a temporal problem) we denote by $\mathrm{taut}(\mathcal{N})$ the set of all the tautological clauses contained in the set $\mathcal{N}$.

The subsumption relation on initial, universal and step clauses is now defined as follows.

Definition 5. *We define a subsumption relation $\leq_s$ on initial, universal and step clauses as follows:*

(i) For two initial clauses $\mathcal{C}$ and $\mathcal{D}$, two universal clauses $\mathcal{C}$ and $\mathcal{D}$, or a universal clause $\mathcal{C}$ and an initial clause $\mathcal{D}$ we define

$$\mathcal{C} \leq_s \mathcal{D} \textit{ iff there exists a substitution } \sigma \textit{ with } \mathcal{C}\sigma \subseteq \mathcal{D}.$$

(ii) For two step clauses $\mathcal{C} = C_1 \Rightarrow \bigcirc C_2$ and $\mathcal{D} = D_1 \Rightarrow \bigcirc D_2$ we define

$$\mathcal{C} \leq_s \mathcal{D} \textit{ iff there exists a substitution } \sigma \textit{ with } C_1\sigma \subseteq D_1, C_2\sigma \subseteq D_2 \textit{ and}$$
$$\textit{for every } x \in \mathit{var}(C_1) \cap \mathit{var}(C_2) \colon \sigma(x) \in X.$$

(iii) For a universal clause $\mathcal{C}$ and a step clause $\mathcal{D} = D_1 \Rightarrow \bigcirc D_2$ we define

$$\mathcal{C} \leq_s \mathcal{D} \textit{ iff there exists a substitution } \sigma \textit{ with } \mathcal{C}\sigma \subseteq \neg D_1 \textit{ or } \mathcal{C}\sigma \subseteq D_2.$$

By $\mathcal{N} \leq_s \mathcal{N}'$ we denote that all clauses in $\mathcal{N}'$ are subsumed by clauses in $\mathcal{N}$.

Thus, subsumption between two initial, two universal or an initial and a universal clause is defined analogously to the subsumption on regular first-order clauses. However, we can only allow a universal clause to subsume a initial clause, but not conversely, as an initial clause only holds in the initial moment of time while a universal clause is true at every moment of time. We also allow subsumption between a universal and a step clause if and only if the universal either subsumes the negated left-hand side or the right-hand side of the step clause.

For subsumption between two step clauses $C_1 \Rightarrow \bigcirc C_2$ and $D_1 \Rightarrow \bigcirc D_2$, we have to impose an additional constraint on the substitution that is used for multiset inclusion: in analogy to inference rules (4) and (5), it has to be ensured that variables occurring in the left-hand sides C_1 and C_2 are only mapped to variables. While for the two inference rules this restriction is imposed to ensure soundness, here the motivation is completeness.

To see that, consider a temporal problem P with universal clauses $P(x)$ and $\neg Q(c)$ and a step clause $P(x) \Rightarrow \bigcirc Q(x)$. The clausification of P will then also contain a step clause $P(c) \Rightarrow \bigcirc Q(c)$. This additional step clause can be resolved with $\neg Q(c)$ using rule (4) with the identity substitution as unifier to obtain $P(c) \Rightarrow \bigcirc\bot$ which, using the conversion rules, gives us a new universal clause $\neg P(c)$. Another inference step with $P(x)$ results in a contradiction. Now, without a restriction on the substitution that can be used in subsumption, $P(x) \Rightarrow \bigcirc Q(x)$ would subsume $P(c) \Rightarrow \bigcirc Q(c)$. We could then try to derive a contradiction by resolving $P(x) \Rightarrow \bigcirc Q(x)$ with $\neg Q(c)$. However, the unifier of $Q(x)$ and $Q(c)$ maps the variable x, which also occurs on the left-hand side of the step clause to the constant c. Thus, an inference by rule (4) using these two premises is not possible and a contradiction can no longer be derived.

Definition 6. *Let $\mathcal{C}$ and $\mathcal{D}$ be initial, step or universal clauses. Then we say that $\mathcal{C}$* properly *subsumes $\mathcal{D}$, written $\mathcal{C} <_s \mathcal{D}$, if and only if $\mathcal{C}$ subsumes $\mathcal{D}$ but not vice-versa, i.e. $\mathcal{C} <_s \mathcal{D}$ iff $\mathcal{C} \leq_s \mathcal{D}$ and $\mathcal{D} \not\leq_s \mathcal{C}$.*

Lemma 2. *Let $\mathcal{C}$ and $\mathcal{D}$ be initial, step or universal clauses such that $\mathcal{C} \leq_s \mathcal{D}$. Then it holds for an initial clause $\mathcal{D}$ that the formula $[(\Box)\tilde{\forall}\mathcal{C}] \Rightarrow [\tilde{\forall}\mathcal{D}]$ is valid, and for a step or universal clause $\mathcal{D}$ that the formula $[\Box\tilde{\forall}\mathcal{C}] \Rightarrow [\Box\tilde{\forall}\mathcal{D}]$ is valid, where $\tilde{\forall}\mathcal{C}$ denotes the universal closure of $\mathcal{C}$.*

Having defined criteria for identifying tautological and subsumed clauses, we could now try to prove that for every refutation without redundancy elimination there exists a refutation with redundancy elimination. However, it turns out that such a correspondence is difficult to establish if the refutation contains applications of the duplicate literal elimination rule whose premise is subsumed.

For example, consider the step clause $\mathcal{D}_1 = P(x) \wedge P(x) \Rightarrow \bigcirc\bot$ which is subsumed by $\mathcal{C}_1 = P(x) \wedge P(y) \Rightarrow \bigcirc\bot$. From $\mathcal{D}_1$ we can derive $\mathcal{D}_2 = P(x) \Rightarrow \bigcirc\bot$ using the duplicate literal elimination rule. But our calculus does not contain a rule which allows us to derive a clause $\mathcal{C}_2$ from $\mathcal{C}_1$ that subsumes $\mathcal{D}_2$ nor does $\mathcal{C}_1$ itself subsume $\mathcal{D}_2$. Similarly, the universal clause $\mathcal{C}_3 = \neg P(x) \vee \neg P(y)$ would also subsume $\mathcal{C}_1$. But again, $\mathcal{C}_3$ does not subsume $\mathcal{D}_2$ nor can we derive a clause from $\mathcal{C}_3$ which subsumes $\mathcal{D}_2$ using the rules of our calculus.

In order to deal with these two cases we need additional factoring rules, in particular, we need to extend our calculus by the following two rules:

– (Arbitrary) Factoring in left-hand sides of terminating step clauses:

$$\frac{C \wedge A \wedge B \Rightarrow \bigcirc\bot}{(C \wedge A)\sigma \Rightarrow \bigcirc\bot} \ ,$$

where σ is a most general unifier of the atoms A and B.

– (Arbitrary) Factoring in (at most) monadic negative universal clauses:

$$\frac{\neg A_1 \vee \cdots \vee \neg A_n \vee \neg A_{n+1}}{(\neg A_1 \vee \cdots \vee \neg A_n)\sigma} \ ,$$

where every atom $A_1, \ldots, A_{n+1}$ contains at most one free variable and σ is a most general unifier of the atoms A_n and A_{n+1}.

The calculus ordered fine-grained resolution with selection extended by the two rules introduced above will be called *subsumption complete* ordered fine-grained resolution with selection and will be denoted by $\mathfrak{I}^{S,\succ}_{FG,Sub}$.

Finally, for our completeness proof we also need to require that the selection function is *subsumption compatible*, as defined below.

Definition 7. *We say that a selection function S is* subsumption compatible *if and only if for every substitution σ and for every two clauses C, D with $C\sigma \subseteq D$ it holds for every literal $l \in D$ that $l \in S(D)$ iff $l\sigma \in S(C)$*

We now have everything in place to show that subsumption complete ordered fine-grained resolution with selection allows the elimination of tautological and subsumed clauses.

Lemma 3. *Let $\mathcal{C}_1$, $\mathcal{C}_2$ be initial, universal or step clauses such that $\mathcal{C}_1$ is a tautology. Then it holds that every resolvent $\mathcal{C}$ of $\mathcal{C}_1$ and $\mathcal{C}_2$ is either a tautology or subsumed by $\mathcal{C}_2$.*

Lemma 4. *Let $\mathcal{C}_1$ be a tautology. Then it holds that every factor $\mathcal{C}$ of $\mathcal{C}_1$ is a tautology.*

Lemma 5. *Let $\mathcal{U}$ be a set of universal clauses and let $\mathcal{N}, \tilde{\mathcal{N}}$ be sets of step clauses such that $\mathcal{N} \leq_s \tilde{\mathcal{N}}$. Additionally, let $\tilde{\Delta}$ be a derivation of a step clause $\tilde{\mathcal{C}}$ by subsumption complete ordered fine-grained resolution with selection without the clause conversion rule from clauses in $\mathcal{U} \cup \tilde{\mathcal{N}}$.*

Then there exists a derivation Δ of a step clause $\mathcal{C}$ by subsumption complete ordered fine-grained resolution with selection from clauses in $\mathcal{U} \cup \mathcal{N}$ such that $\mathcal{C} \leq_s \tilde{\mathcal{C}}$.

Proof. Lemma 5 is shown by induction on the length of the derivation $\tilde{\Delta}$. For the base case we assume that $\tilde{\mathcal{C}}$ is a step clause in $\tilde{\mathcal{N}}$. Then there is a clause $\mathcal{C}$ in $\mathcal{N}$ with $\mathcal{C} \leq_s \tilde{\mathcal{C}}$. For the induction step we consider a step clause $\tilde{\mathcal{C}}$ that is derived by one of the rules of fine-grained step resolution excluding the clause conversion rule, that is, rules (1) to (5) and (7), from premises $\tilde{\mathcal{C}_1}$ and $\tilde{\mathcal{C}_2}$ which are either elements of $\mathcal{U} \cup \tilde{\mathcal{N}}$ or previously derived clauses. By the induction hypothesis there are clauses $\mathcal{C}_1$ and $\mathcal{C}_2$ with $\mathcal{C}_1 \leq_s \tilde{\mathcal{C}_1}$ and $\mathcal{C}_2 \leq_s \tilde{\mathcal{C}_2}$ which are either elements of $\mathcal{U} \cup \mathcal{N}$ or previously derived, and either $\mathcal{C}_1 \leq_s \tilde{\mathcal{C}}$, $\mathcal{C}_2 \leq_s \tilde{\mathcal{C}}$ or we can derive a clause $\mathcal{C}$ with $\mathcal{C} \leq_s \tilde{\mathcal{C}}$ from $\mathcal{C}_1$ and $\mathcal{C}_2$.

Lemma 6. *Let $\mathcal{N}$ and $\tilde{\mathcal{N}}$ be sets of initial, universal clauses or step clauses such that $\mathcal{N} \leq_s \tilde{\mathcal{N}}$. Additionally, let $\tilde{\Delta}$ be a derivation of a clause $\tilde{\mathcal{C}}$ by subsumption complete ordered fine-grained resolution with selection from clauses in $\tilde{\mathcal{N}}$.*

Then there exists a derivation Δ of a clause $\mathcal{C}$ by subsumption complete ordered fine-grained resolution with selection from clauses in $\mathcal{N}$ such that $\mathcal{C} \leq_s \tilde{\mathcal{C}}$.

The previous statement still holds if $\mathcal{N} \leq_s \tilde{\mathcal{N}} \setminus \mathrm{taut}(\tilde{\mathcal{N}})$ and $\tilde{\mathcal{C}}$ is not a tautology.

```
Function Subsumption-Restricted-FG-BFS
Input:   A set of universal clauses $\mathcal{U}$ and a set of step clauses $\mathcal{S}$, saturated
         by ordered fine-grained resolution with selection, and an eventuality
         clause $\Diamond L(x) \in \mathcal{E}$.
Output:  A formula $R(x)$ with at most one free variable.
Method:  (1) Let $R_0(x) = \mathbf{true}$; $M_0 = \emptyset$; $i = 0$
         (2) Let $\mathcal{N}'_{i+1} = \mathcal{U} \cup \mathrm{LT}(\mathcal{S}) \cup \{\mathbf{true} \Rightarrow \bigcirc(\neg R_i(c^l) \vee L(c^l))\}$. Apply
             the rules of ordered fine-grained resolution with selection *except
             the clause conversion rule* to $\mathcal{N}'_{i+1}$, together with the *removal of
             tautological and subsumed clauses*. If we obtain a contradiction,
             then return the loop true (in this case $\forall x \neg L(x)$ is implied by the
             universal part).
             Otherwise let $\mathcal{M}'_{i+1} = \{C_j \Rightarrow \bigcirc\bot\}_{j=1}^{n}$ be the set of all new
             *terminating* step clauses in the saturation of $\mathcal{N}'_{i+1}$.
         (3) If $\mathcal{M}'_{i+1} = \emptyset$, return false; else let $R_{i+1}(x) = \bigvee_{j=1}^{n}(\tilde{\exists} D_j)\{c^l \rightarrow x\}$
         (4) If $\forall x(R_i(x) \Rightarrow R_{i+1}(x))$, return $R_{i+1}(x)$.
         (5) i = i + 1; goto 2.
```

Fig. 3. Restricted Breadth-First Search Using Ordered Fine-Grained Step Resolution with Selection

Proof. Let $\tilde{\Delta} = \tilde{\mathcal{D}}_1, \ldots, \tilde{\mathcal{D}}_{n-1}, \tilde{\mathcal{C}}(= \tilde{\mathcal{D}}_n)$. If $\mathcal{N} \leq_s \tilde{\mathcal{N}}$, then one can show the existence of the derivation Δ by induction on the length of the derivation $\tilde{\Delta}$ in analogy to Lemma 5.

In the case where $\mathcal{N} \leq_s \tilde{\mathcal{N}} \setminus \mathrm{taut}(\tilde{\mathcal{N}})$ holds, it can be shown inductively for every clause $\tilde{\mathcal{D}}_i$ $(1 \leq i \leq n)$ which is not a tautology that there exists a derivation Δ of a clause $\mathcal{D}_i$ with $\mathcal{D}_i \leq_s \tilde{\mathcal{D}}_i$ by subsumption complete ordered fine-grained resolution with selection from clauses in $\mathcal{N}$.

Theorem 3. *Let P be the clausification of a monodic temporal problem. Let $\succ$ be an atom ordering and S a subsumption compatible selection function. Then P is unsatisfiable iff there exists a $\mathfrak{I}^{S,\succ}_{FG,Sub}$-refutation of* $\mathrm{Cls}(\mathsf{P})$. *Moreover, P is unsatisfiable iff any fair $\mathfrak{I}^{S,\succ}_{FG,Sub}$-derivation is a refutation of* $\mathrm{Cls}(\mathsf{P})$.

Proof. Given a $\mathfrak{I}^{S,\succ}_{FG}$-refutation Δ of P we establish by induction on Δ that there also exists a $\mathfrak{I}^{S,\succ}_{FG,Sub}$-refutation of P using Lemmata 3 to 6.

5 Subsumption and Loop Search

Theorem 3 shows that we can eliminate tautologies and subsumed clauses during the construction of a derivation at the level of inference rule applications of the $\mathfrak{I}^{S,\succ}_{FG,Sub}$ calculus. However, the rules of the calculus are also applied within the fine-grained breadth-first search algorithm FG-BFS which is used to find loop formulae for the application of the eventuality resolution rules. Naturally, the question arises whether tautological and subsumed clauses can also be eliminated within FG-BFS.

The answer to that is positive. Figure 3 shows the so-called *subsumption restricted* breadth-first search algorithm using ordered fine-grained step resolution with selection, a modification of FG-BFS which removes tautological and subsumed clauses during the saturation process by ordered fine-grained resolution with selection in step (2) of the algorithm. In the way in which the algorithm shown in Figure 3 is defined the constructed sets $\mathcal{M}'_i$ will not contain terminating step clauses $C \Rightarrow \bigcirc\bot$ and $D \Rightarrow \bigcirc\bot$ such that $C \leq_s D$.

We are then able to prove the following result:

Theorem 4. *Let P be a monodic temporal problem. Let $\succ$ be an atom ordering and S a subsumption compatible selection function. Then P is unsatisfiable iff there exists a $\mathfrak{I}^{S,\succ}_{FG,Sub}$-refutation of* $\mathrm{Cls}(\mathsf{P})$ *with applications of the eventuality resolution rule restricted to loop formulae found by the function Subsumption-Restricted-FG-BFS. Moreover, P is unsatisfiable iff any fair $\mathfrak{I}^{S,\succ}_{FG,Sub}$-derivation with applications of the eventuality resolution rule restricted to loop formulae found by the function Subsumption-Restricted-FG-BFS is a refutation of* $\mathrm{Cls}(\mathsf{P})$.

6 Implementation and Experimental Results

The subsumption complete ordered fine-grained resolution calculus including the restricted breath-first loop search procedure have been implemented in the theorem prover TSPASS [4], which is based on the first-order resolution prover SPASS 3.0. In order to be able to construct fair derivations, the loop search procedure has been integrated with the remainder of the calculus as described in [14, 15].

The main procedure of TSPASS uses a given-clause algorithm [18] in which a clause selected from the set $\mathcal{US}$ of usable clauses, which initially contains all clauses from a given temporal problem, is used to derive all consequences $\mathcal{NEW}$ by resolving the selected clause with all the clauses in the set $\mathcal{WO}$ of worked-off clauses, which is initially empty, after the selected clause has been moved from $\mathcal{US}$ to $\mathcal{WO}$. Based on the results in the previous two sections, we can apply tautology elimination (to $\mathcal{NEW}$), forward subsumption and backward subsumption [18]. With forward subsumption all clauses in $\mathcal{NEW}$ that are subsumed by a clause in $\mathcal{WO}$ or $\mathcal{US}$ are deleted from $\mathcal{NEW}$. With backward subsumption clauses in $\mathcal{WO}$ and $\mathcal{US}$ subsumed by a clause in $\mathcal{NEW}$ are deleted from these sets. After redundancy elimination the remaining clauses in $\mathcal{NEW}$ are added to $\mathcal{US}$. This process continues until either a contradiction is derived or $\mathcal{US}$ becomes empty.

To show the effectiveness of tautology elimination, forward subsumption, and backward subsumption for the subsumption complete ordered fine-grained resolution calculus, we have applied TSPASS 0.92-0.16, for example, on the specification of the game Cluedo [2]. Due to lack of space we cannot present other examples, but we obtained similar results. The Cluedo problem specifications

[4] Available at `http://www.csc.liv.ac.uk/~michel/software/tspass/`

	-F/-B/-T		-F/+B/-T		-F/-B/+T		-F/+B/+T	
	Clauses	Time	Clauses	Time	Clauses	Time	Clauses	Time
1	—	TO	—	TO	239202	328.938s	17664	0.193s
2	—	TO	—	TO	—	TO	694574	10.959s
3	—	TO	—	TO	—	TO	—	TO
4	—	TO	—	TO	—	TO	529244	12.566s
5	—	TO	—	TO	—	TO	—	TO
6	—	TO	—	TO	—	TO	—	TO

	+F/-B/-T		+F/+B/-T		+F/-B/+T		+F/+B/+T	
	Clauses	Time	Clauses	Time	Clauses	Time	Clauses	Time
1	481	0.035s	480	0.039s	445	0.033s	444	0.038s
2	2354	0.130s	2262	0.129s	1926	0.115s	1892	0.124s
3	11065	1.375s	9912	1.310s	10102	1.350s	9170	1.278s
4	1460	0.087s	1559	0.097s	1125	0.074s	1343	0.093s
5	594	0.051s	594	0.052s	488	0.044s	488	0.049s
6	765	0.059s	765	0.055s	645	0.050s	645	0.054s

Table 1. Results Obtained for the Cluedo Examples

consist of six valid, i.e. unsatisfiable, assertions that can be made in an example Cluedo game. The full details can be found in [5, 6]. Problem 4 is the only specification that contains eventuality formulae.

The experiments were run on a PC with an Intel Core 2 Duo E6400 CPU and 2 GB of main memory with a timeout (TO) of 1 CPU hour for each problem. The results are shown in Table 1. Here, '+B', '+F', and '+T' indicate that backward subsumption, forward subsumption, and tautology elimination, respectively, have been enabled while '-B', '-F', and '-T' indicate that they have been disabled. Given that all six assertions are valid, proofs can theoretically be found by a complete reasoner without the need for redundancy elimination. As the experiments indicate this is clearly not the case within a reasonable amount of time. On the other hand with all options for redundancy elimination enabled even the most difficult problem can be solved in little more than one second. As one might expect, forward subsumption is the most effective of the three options, followed by tautology elimination, while backward subsumption can on occasion slow down the process of finding a proof rather than speeding it up. Overall, the experiments confirm that redundancy elimination is crucial for effective resolution-based theorem proving in monodic first-order temporal logic.

7 Conclusion

In this paper we have considered redundancy elimination in the context of ordered fine-grained resolution with selection, a sound and complete resolution-based calculus for monodic first-order temporal logic. We have shown that a slight modification of the calculus is compatible with the elimination of tautologies and subsumed clauses.

Our results can be used to show that the calculus can also be extended with additional rules, for example, condensation and matching replacement resolution [18] with suitable restrictions on the substitutions and on the orderings of the literals that are to be removed, which reduce to a sequence of inference and redundancy elimination steps. Such rules can be useful to further increase the effectiveness of the calculus and for the construction of decision procedures for decidable fragments of monodic first-order temporal logic [10].

In addition, we have presented experimental results which confirm that the elimination of redundant clauses is essential for effective resolution-based theorem proving in monodic first-order temporal logic.

References

1. L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [17], chapter 2, pages 19–99.
2. Cluedo. http://www.hasbro.com.
3. A. Degtyarev, M. Fisher, and B. Konev. Monodic temporal resolution. In *Proc. CADE-19*, volume 2741 of *LNCS*, pages 397–411. Springer, 2003.
4. A. Degtyarev, M. Fisher, and B. Konev. Monodic temporal resolution. *ACM Transactions On Computational Logic*, 7(1):108–150, 2006.
5. C. Dixon. Specifying and verifying the game Cluedo using temporal logics of knowledge. Technical Report ULCS-04-003, University of Liverpool, 2004.
6. C. Dixon, M. Fisher, B. Konev, and A. Lisitsa. Practical first-order temporal reasoning. In *Proc. TIME'08*, pages 156–163. IEEE Comp. Soc., 2008.
7. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.
8. M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, 2001.
9. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.
10. U. Hustadt, B. Konev, and R. A. Schmidt. Deciding monodic fragments by temporal resolution. In *Proc. CADE-20*, volume 3632 of *LNCS*, pages 204–218. Springer, 2005.
11. B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt. Towards the implementation of first-order temporal resolution: the expanding domain case. In *Proc. TIME-ICTL 2003*, pages 72–82. IEEE Comp. Soc., 2003.
12. B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt. Mechanising first-order temporal resolution. *Information and Computation*, 199(1-2):55–86, 2005.
13. R. Kontchakov, C. Lutz, F. Wolter, and M. Zakharyaschev. Temporalising tableaux. *Studia Logica*, 76(1):91–134, 2004.
14. M. Ludwig and U. Hustadt. Implementing a fair monodic temporal logic prover. *AI Communications.* To appear.
15. M. Ludwig and U. Hustadt. Fair monodic temporal reasoning. In *Proc. CADE-22*, 2009. To appear.
16. M. Ludwig and U. Hustadt. Resolution-based model construction for PLTL. In *Proc. of TIME'09*, 2009. To appear.
17. J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes).* Elsevier and MIT Press, 2001.
18. C. Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [17], pages 1965–2013.
19. F. Wolter and M. Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic. *Annals of Pure and Applied Logic*, 118:133–145, 2002.

Minimal Model Generation with respect to an Atom Set

Miyuki Koshimura[1]*, Hidetomo Nabeshima[2],
Hiroshi Fujita[1], and Ryuzo Hasegawa[1]

[1] Kyushu University, Motooka 744, Nishi-ku, Fukuoka, 819-0395 Japan,
{koshi,fujita,hasegawa}@ar.is.kyushu-u.ac.jp,
[2] University of Yamanashi, Takeda 4-3-1, Kofu, 400-8511 Japan,
nabesima@yamanashi.ac.jp

Abstract. This paper studies minimal model generation for SAT instances. In this study, we minimize models with respect to an atom set, and not to the whole atom set. In order to enumerate minimal models, we use an arbitrary SAT solver as a subroutine which returns models of satisfiable SAT instances. In this way, we benefit from the year-by-year progress of efficient SAT solvers for generating minimal models. As an application, we try to solve job-shop scheduling problems by encoding them into SAT instances whose minimal models represent optimum solutions.

1 Introduction

The notion of minimal Herbrand models is important in a wide range of areas such as logic programming, deductive database, software verification, and hypothetical reasoning. Some applications would actually need to generate minimal models of a given formula.

In this work, we consider the problem of automating propositional minimal model generation with respect to an atom set. Some earlier works [3, 14, 9] considered minimal model generation with respect to the whole atom set.

Bry and Yahya [3] presented a sound and complete procedure for generating minimal models. They incorporate complement splitting and constrained search into positive unit hyper-resolution in order to reject nonminimal models. Niemelä [14] also gave a sound and complete procedure. His method is based on a generate and test method: generate a sequence of minimal model candidates and reject nonminimal models by groundedness test which passes minimal models. Hasegawa et al. [9] presented an minimal model generation method employing branching assumptions and lemmas so as to prune branches that lead to nonminimal models, and to reduce minimality tests on obtained models.

However, these earlier works do not make use of some pruning techniques such as non-chronological or intelligent backtracking, and generating lemmas. These techniques make reasoning systems practical ones. In recent years, the

* This work was supported by KAKENHI (20240003).

propositional satisfiability (SAT) problem has been studied actively [1]. Specially, many works for implementing efficient SAT solvers have been performed in the last decade. The state-of-the-art SAT solvers can solve SAT problems consisting of millions of clauses in a few minutes. Then, it has been realized that we solve several kinds of problems by encoding them into SAT problems [7, 2].

This paper shows a method to generate minimal models with a SAT solver. Thus, the method benefits from the year-by-year progress of SAT solvers implementing the pruning techniques efficiently. We also try to solve the job-shop scheduling problems (JSSP) in the minimal model generation framework, in which, minimal models represent optimum, namely, the shortest schedules.

The remaining part of this paper is organized as follows: First we present a characterization of minimal models that is the key to our method to handle minimal model generation. Section 3 gives minimal model inference procedures with a SAT solver. Section 4 describes the job-shop scheduling problem and encodes it as a SAT instance. Section 5 demonstrates that the procedures are successfully implemented with the SAT solver MiniSat 2 by solving several JSSPs. We end the paper with a short summary and a discussion of future works.

2 Properties of Minimal Models

Models of a propositional formula can be represented by a set of propositional variables (or atoms); namely, each model is represented by the set of propositional variables to which it assigns true. For example, the model assigning true to a, false to b, and true to c is represented by the set $\{a, c\}$. In this representation, we can compare two models by set inclusion. For example, model $\{a, c\}$ is smaller than model $\{a, b, c\}$. In this study, we focus on minimality of models in the representation.

Definition 1. *Let P, M_1 and M_2 be atom sets. Then, M_1 is said to be smaller than M_2 with respect to P if $M_1 \cap P$ is a proper subset of $M_2 \cap P$.*

Example 1. Let $M_1 = \{p_1, p_2, p_3, a\}$, $M_2 = \{p_1, p_3, b, c, e, f\}$ and $P = \{p_1, p_2, p_3\}$. Then, M_2 is smaller than M_1 with respect to P.

Definition 2 (Minimal model). *Let A be a propositional formula, P be an atom set, and M be a model of A. Then, M is said to be a minimal model of A with respect to P when there is no model smaller than M with respect to P.*

Example 2. Let A be a propositional formula and $P = \{p_1, p_2, p_3\}$. And, A has three models $M_1 = \{p_1, p_2, p_3, a\}$, $M_2 = \{p_1, p_2, b\}$, and $M_3 = \{p_3, c\}$. Then, M_2 and M_3 are minimal models with respect to P while M_1 is not minimal.

This definition is the same as that of circumscription $Circum(A(P, Z); P; Z)$ [10] when variable predicates $Z = \overline{P}$, i.e. no fixed predicate. Note that $\overline{P}$ denotes the set complement of P. In this sense, our study is a specialized one of circumscription. There is a little difference between our work and circumscription for the treatment of models. We are interested only in truth values of atoms in P.

Therefore, we regard two models M_1 and M_2 as equal when $M_1 \cap P = M_2 \cap P$, while these two are distinguished in the framework of circumscription when $M_1 \neq M_2$.

The following theorem is a straight extension of **Proposition 6** in Niemelä's work [14]. This theorem gives the basis of the computational treatment of minimal models as **Proposition 6** does.

Theorem 1. *Let A be a propositional formula, P be an atom set, and M be a model of A. Then, M is a minimal model of A with respect to P iff a formula $A \wedge \neg(a_1 \wedge a_2 \wedge \ldots \wedge a_m) \wedge \neg b_1 \wedge \neg b_2 \wedge \ldots \wedge \neg b_n$ is unsatisfiable, where $\{a_1, a_2, \ldots, a_m\} = M \cap P$ and $\{b_1, b_2, \ldots, b_n\} = \overline{M} \cap P$.*

Proof. Let G be $A \wedge \neg(a_1 \wedge a_2 \wedge \ldots \wedge a_m) \wedge \neg b_1 \wedge \neg b_2 \wedge \ldots \wedge \neg b_n$.
Assume that M is not a minimal model. Then, there is a model N smaller than M with respect to P. Thus, the following properties hold: $\forall j(1 \leq j \leq n)(N \models \neg b_j)$ and $\exists i(1 \leq i \leq m)(N \models \neg a_i)$. Of course, $N \models A$ because N is a model of A. Therefore, $N \models G$; namely G is satisfiable.
Conversely, we assume G is satisfiable. Then, there is a model N such that $\forall j(1 \leq j \leq n)(N \models \neg b_j)$ and $\exists i(1 \leq i \leq m)(N \models \neg a_i)$. This implies N is smaller than M with respect to P. That is, M is not a minimal model with respect to P.

Example 3. Let A be a propositional formula and $P = \{p_1, p_2, p_3, p_4\}$. Then, a model $\{p_1, p_4, c, d\}$ of A is minimal with respect to P iff $A \wedge \neg(p_1 \wedge p_4) \wedge \neg p_2 \wedge \neg p_3$ is unsatisfiable.

3 Procedures

This section gives procedures for generating minimal models of a SAT instance with a SAT solver based on the generate and test method: generating a sequence $M_1, \ldots, M_i, \ldots$ of models and performing minimality test on each M_i. In these procedures, we use a single SAT solver as both generator and tester where we assume the SAT solver returns a model of a satisfiable SAT instance. Almost all SAT solvers satisfy this assumption.

Figure 1 (a) shows a minimal model generation with respect to an atom set P which is implicitly given to the procedure. We call this *the naive version.* `A0` is a SAT instance to be proved. The function `solve(A)` denotes the core part of the SAT solver. The function returns `false` when a SAT instance `A` is unsatisfiable and `true` when `A` is satisfiable. In the latter case, a model M of `A` is obtained through an array from which we construct two formulas `F1` and `F2` for a minimality test on M with respect to P where `F1` $= \neg(a_1 \wedge \ldots \wedge a_m)$ and `F2` $= \neg b_1 \wedge \ldots \wedge \neg b_n$. A boolean variable `exhaustive` indicates whether the procedure generates all minimal models or only one minimal model. If `exhaustive` is set to *true*, all minimal models are generated.

If `solve(A)` in line (2) returns *true*, the body of the while statement is executed. In this case, as a model M of `A` is obtained, we perform a minimality test on M (in (4)). If the test passes, that is `solve(A)` in (4) returns $false$, we conclude M is minimal with respect to P. If the test fails or `exhaustive` is *true*, $\mathtt{F}_1$ is added to $\mathtt{A}_1$ as a conjunct in order to avoid generating the same model or larger models in succeeding search. Thus, the role of the conjunct $\mathtt{F}_1$ is pruning redundant models.

```
(1)  $A = A_0$; $A_1 = A_0$; // $A_0$: a SAT instance to be proved
(2)  while (solve(A)) { // Found a model $M$ where $M \cap P = \{a_1, \ldots, a_m\}$
                           // and $\overline{M} \cap P = \{b_1, \ldots, b_n\}$
(3)      $A = A \wedge F_1 \wedge F_2$;   //  $F_1 = \neg(a_1 \wedge \ldots \wedge a_m)$, $F_2 = \neg b_1 \wedge \ldots \wedge \neg b_n$
(4)      if (!solve(A)) { // Perform minimality test
(5)          "minimal model found";
(6)          if (!exhaustive) break;
(7)      }
(8)      $A_1 = A_1 \wedge F_1$; $A = A_1$; // continue searching minimal models
                                 // without generating larger models.
(9)  }
```

(a) Naive version

```
(1)  $A = A_0$;
(2)  while (solve(A)) {      // Found a model M,
(3)      MM = minimize(A,M); // minimize M, and obtain a minimal model MM
(4)      if (!exhaustive) break;
(5)      $A = A \wedge F_1$;        // $MM \cap P = \{a_1, \ldots, a_m\}$ and $F_1 = \neg(a_1 \wedge \ldots \wedge a_m)$
(6)  }

(7)  function minimize(A,M) {
     // returns a minimal model small than or equal to M
     // where $M \cap P = \{a_1, \ldots, a_m\}$ and $\overline{M} \cap P = \{b_1, \ldots, b_n\}$
(8)      $A = A \wedge F_1 \wedge F_2$;   // $F_1 = \neg(a_1 \wedge \ldots \wedge a_m)$, $F_2 = \neg b_1 \wedge \ldots \wedge \neg b_n$
(9)      if(!solve(A)) { // Perform minimality test
(10)         return M;    // M is a minimal model
(11)     } else {                  // Found a new model SM smaller than M
(12)         minimize(A, SM); // and minimize SM
(13) }   }
```

(b) Normal version

Fig. 1. Procedures for minimal model generation

Figure 1 (b) shows a modified procedure of the naive version. We call this *the normal version*. In this version, when a model is found, we minimize it with the function `minimize`. Its definition is shown from the line (7) to (13). This uses the result of the minimality test on `A` in (9). When the test fails, in other words, `solve(A)` in (9) returns *true*, we obtain a model `SM` of `A`. `SM` is smaller than `M` because of the conjuncts $\mathtt{F}_1$ and $\mathtt{F}_2$. Thus, `SM` is the next target of `minimize`. Note that $\exists i(1 \leq i \leq m)(a_i \notin \texttt{SM})$. Therefore, at least one current $\neg a_i$ participates in $\texttt{F}_2$ of the next `minimize`.

The major difference between the naive version and the normal version is the use of `SM` obtained from the minimality test. The naive version ignores it while the normal version uses it. Therefore, we expect that the normal version is more efficient than the naive version for enumerating minimal models.

3.1 Lemma Reusing

Many state-of-the-art SAT solvers learn *lemmas* called conflict clauses to prune redundant search space, but lemmas deduced from a certain SAT instance can not apply to solve other SAT instances. Therefore, a function call `solve(A)` in Figure 1 (both (a) and (b)) can not use lemmas deduced from previous `solve(A)` in general.

However, every SAT instance `A` in `solve(A)` satisfies the following lemma-reusability condition [13] if the conjunct $\texttt{F}_1$ is not added to `A`, when the SAT solver uses Chaff-like lemma generation mechanism [12].

Definition 3 (Lemma-reusability condition [13]). *Suppose that A and B are SAT instances. The lemma-reusability condition between A and B is as follows: If A includes a non-unit clause x, then B contains x.*

If both A and B satisfy the condition, we can use lemmas generated by `solve(A)` for `solve(B)`. This is justified by the following proposition which is a paraphrase of Theorem 1 in [13].

Proposition 1. *If A is a SAT instance and c is any lemma generated by `solve(A)`, then c is a logical consequence of a set of some non-unit clauses in A.*

This proposition is true when we use the SAT solver MiniSat for implementing `solve(A)`, because MiniSat does not use any unit clause for generating lemmas.

$\texttt{F}_1$ is a non-unit clause and violates the lemma-reusability condition. However, the only role of $\texttt{F}_1$ is excluding models larger than the model causing $\texttt{F}_1$. Then, lemmas depending on $\texttt{F}_1$ can be used for succeeding minimal model generation. It follows from what has been said that every call `solve(A)` shares lemmas each other.

3.2 An Implementation with MiniSat

We have implemented the minimal model generation procedures with the SAT solver MiniSat [5] version 2.1 which is written in C++. MiniSat 2.1 took the first place in the main track of SAT-Race 2008.

The `solve` method of MiniSat is declared as follows:

```
bool solve(const vec<Lit>& assumps)
```

The method determines the satisfiability of a set of clauses under an assumption `assumps`. It returns `true` if the set is satisfiable; otherwise `false`. The clause set is realized by a vector `clauses` and initialized to a SAT instance ($\mathtt{A_0}$ in Figure 1). The assumption `assumps` is a vector of literals which means the conjunction of the literals.

In our implementation, the clause $\mathtt{F_1}$ is appended to `clauses` and the formula $\mathtt{F_2}$ is set to `assumps`. Then, the `solve` method is invoked. We don't need to remove $\mathtt{F_1}$ from `clauses` before the next `solve` invocation because the role of $\mathtt{F_1}$ is excluding models larger than the model causing $\mathtt{F_1}$. If $\mathtt{F_2}$ is appended to `clauses`, we need to remove $\mathtt{F_2}$ from `clauses` before the next invocation. Therefore, we add $\mathtt{F_2}$ to `assumps` instead of `clauses`. Thus, removing $\mathtt{F_2}$ is not necessary.

When we need only one minimal model[3] rather than all minimal models, we can append $\mathtt{F_2}$ to `clauses` without removing $\mathtt{F_2}$ afterward. We also implement such solver based on the normal version and call it *the single-solution version.*

4 Solving the JSSP

A JSSP consists of a set of jobs and a set of machines. Each job is a sequence of operations. Each operation requires the exclusive use of a machine for an uninterrupted duration, i.e. its processing time. A schedule is a set of start times for each operation. The time required to complete all the jobs is called the makespan. The objective of the JSSP is to determine the schedule which minimizes the makespan.

In this study, we follow a variant of the SAT encoding proposed by Crawford and Baker [4]. In the SAT encoding, we assume there is a schedule whose makespan is at most i and generate a SAT instance S_i. If S_i is satisfiable, then the JSSP can complete all the jobs by the makespan i. Therefore, if we find a positive integer k such that S_k is satisfiable and S_{k-1} is unsatisfiable, then the minimum makespan is k.

For minimizing the makespan, Nabeshima et al. [13] applied two kinds of methods, *incremental search and binary search.* One can easily estimate the upper bound L_{up} of the minimum makespan by serialising all the operations of all the jobs [4]. The lower bound L_{low} is also easily estimated by taking the maximum length of each job in which we assume every job is performed independently. In the incremental search, we start from L_{low} and increase the makespan by 1 until we encounter the satisfiable instance S_t. If such S_t is found, then the minimum makespan is t. We explain the binary search by an example of $L_{up} = 393$ and $L_{low} = 49$. Firstly, we try to solve S_{221} because 221 is the midpoint between 48 and 393. If S_{221} is satisfiable, then try S_{135}. If S_{135} is unsatisfiable, then try S_{178}. We continue this binary search until we encounter the satisfiable instance S_t and unsatisfiable instance S_{t-1}.

[3] The JSSP is such a problem.

[4] In this study, we use a modified estimation a bit cleverer than this obvious estimation.

In order to solve the JSSP in the minimal model generation framework, we introduce a set $P_u = \{p_1, p_2, \ldots, p_u\}$ of new atoms when $L_{up} = u$. The intended meaning of $p_i = true$ is that we found a schedule whose makespan is i or longer than i. To realize the intention, the formulas $F_i(i = 1, \ldots, u)$, which represent "if all the operations complete at i, then p_i becomes true," are introduced. Besides, we introduce a formula $T_u = (\neg p_u \vee p_{u-1}) \wedge (\neg p_{u-1} \vee p_{u-2}) \wedge \cdots \wedge (\neg p_2 \vee p_1)$ which implies that $\forall l(1 \leq l < k)(p_l = true)$ must hold if $p_k = true$ holds.

In this setting, if we obtain a model M of $G_u(= S_u \wedge F_1 \wedge \cdots \wedge F_u \wedge T_u)$ and k is the maximum integer such that $p_k \in M$, that is, $\forall j(k < j \leq u)(p_j \notin M)$, then we must have $\forall l(1 \leq l \leq k)(p_l \in M)$, namely, $M \cap P_u = \{p_1, \ldots, p_k\}$. The existence of such k is guaranteed by F_k and T_u, and indicates that there is a schedule whose makespan is k. If k is the minimum makespan, there is no model of G_u smaller than M with respect to P_u. Thus, a minimal model of G_u with respect to P_u represents a schedule which minimizes the makespan.

Example 4. Given a JSSP with $L_{up} = 10$. Then, we make S_{10} according to Crawford encoding, $P_{10} = \{p_1, \ldots, p_{10}\}$, and $T_{10} = (\neg p_{10} \vee p_9) \wedge \cdots \wedge (\neg p_2 \vee p_1)$. Let M be a minimal model of $G_{10}(= S_{10} \wedge F_1 \wedge \cdots \wedge F_{10} \wedge T_{10})$ with respect to P_{10} and $M \cap P_{10} = \{p_1, p_2, p_3\}$. Then, the minimum makespan of the JSSP is 3.

This SAT encoding technique, in which a minimal model represents an optimum solution, is applicable to several problems such as graph coloring problem, open-shop scheduling problem, two dimensional strip packing problem, and so on. Thus, the technique gives a framework to solve these problems.

The encoding is easily adapted for a partial Max-SAT encoding by adding some unit clauses. Max-SAT is the optimization version of SAT where the goal is to find a model satisfying the maximum number of clauses. In order to solve the JSSP in the partial Max-SAT framework[5], we introduce u unit clauses $\neg p_i(i = 1, \ldots, u)$. Then, we solve $MAX_u(= G_u \wedge \neg p_1 \wedge \ldots \wedge \neg p_u)$ with a partial Max-SAT solver where all clauses in G_u are treated as hard clauses and $\neg p_i(i = 1, \ldots, u)$ are as soft clauses. A Max-SAT model of MAX_u represents a optimum schedule.

Example 5. Let P_{10}, G_{10}, and M be the same as in *Example 4*. Then $MAX_{10} = G_{10} \wedge \neg p_1 \wedge \ldots \wedge \neg p_{10}$ has a (Max-SAT) model M which falsifies only three soft clauses $\neg p_1$, $\neg p_2$, and $\neg p_3$. Note that every model of G_{10} falsifies at least these three clauses.

5 Experiments

We executed the three versions(naive/normal/single-solution), a Max-SAT solver MiniMaxSat[6], and a SAT-based JSSP solver SATSHOP which is a successor of

[5] A partial Max-SAT solver can handle hard clauses and soft clauses. The hard clauses must be satisfied while the soft clauses need not be necessarily satisfied. The goal is to find a model satisfying the all hard clauses and the maximum number of soft clauses.

the JSSP solver proposed in [13]. The MiniMaxSat took the third place in the partial Max-SAT category (industrial) of Max-SAT Evaluation 2008.

The SATSHOP tries to solve a JSSP in the following way. First, making a relaxed problem to improve the upper bound L_{up}. The relaxed problem is an approximation of the original problem. It is obtained by rounding up every operation time. Its optimum solution gives a new upper bound L_{up}^{new} which satisfies $L_{up}^{new} \leq L_{up}$. The relaxed problem is solved with the SAT encoding technique using binary search.

Next, solving the problem with L_{up}^{new} by *decremental search*. Basically, decremental search is a dual of incremental search. We start from L_{up}^{new} and decrease the makespan until we encounter the unsatisfiable instance.

We try to solve 82 JSSPs in OR-Library [15]. The problems are abz5–abz9, ft06, ft10, ft20, la01–la40, orb01–orb10, swv01–swv20, and yn1–yn4. We limited the execution time of each problem to 2 CPU hours. The single-solution version and SATSHOP succeed to solve 33 problems out of 82 problems. The naive version, normal version, and MiniMaxSAT succeed to solve 32, 31, and 14 problems, respectively. Table 1 shows the experimental results of 33 problems solved.

All experiments were conducted on a Pentium M 753(1.20GHz) machine with 1GB memory running Linux 2.6.16. Each problem is encoded to a CNF (conjunctive normal form)[6]. The second and third columns show statistics of CNFs. The fourth column "$|P|$" shows the size of an atom set P with respect to which we minimize a model. The fifth column "Optimum" shows the minimum makespan. "Single" is the single-solution version. The "Total" row shows the total CPU time for the single-solution version or SATSHOP. The "Ratio" row shows (total time of SATSHOP)/(total time of Single).

The single-solution version usually beats other two versions as expected. On average it solves problems 1.6 times faster than the naive version and 1.3 times faster than the normal version for the 31 problems solved by these three versions.

On the other hand, the SATSHOP beats these three versions on almost all problems. On average it solves problems about 1.7 times faster than the single-solution version. The main reason for the domination of the SATSHOP is that it tries to solve a relaxed problem first. The relaxed one is easy to solve by orders of magnitude. In order to eliminate the effect of the relaxation, we also run the SATSHOP in a non-relaxation mode where it try to solve JSSPs without relaxation. This causes an increase of the runtime of the SATSHOP. The single-solution version, then, is almost comparable with the SATSHOP: the former sometimes beats the latter, and vice versa. On average, the latter solves problems about 1.2 times faster than the former.

The MiniMaxSAT is the worst solver in our experience. It can solve only half of problems solved by others within 2 CPU hours. It seems to be several hundred times slower than others. We may need to develop a SAT encoding tailored for MaxSAT solvers.

[6] We also use the SATSHOP as an encoder. Thus, the core part of a SAT instance solved by the three versions is the same one solved by the SATSHOP.

Table 1. Experimental results of OR-Library

Prob-	No. of	No. of	$\|P\|$	Opti-	runtime in seconds				
lem	Variables	Clauses		mum	Naive	Normal	Single	MaxSAT	SATSHOP
abz5	103,440	1,111,236	1374	1234	55.1	47.1	25.5	time-out	27.5
abz6	81,995	879,864	1075	943	9.2	6.6	6.8	2640.8	6.7
ft06	1,847	11,744	63	55	0.0	0.0	0.0	0.1	0.0
ft10	98,278	1,057,688	1211	930	180.9	263.9	106.5	time-out	63.1
la01	41,288	435,549	921	666	13.6	9.4	6.4	630.1	2.2
la02	36,328	382,368	815	655	50.9	35.7	10.3	1414.1	4.8
la03	37,038	390,635	825	597	21.1	7.0	8.3	1414.6	4.3
la04	37,473	395,009	838	590	5.5	3.3	3.4	1031.9	2.4
la05	28,360	297,253	651	593	7.4	10.2	8.7	1402.5	6.4
la16	94,286	1,014,336	1171	945	32.0	22.7	18.8	5448.9	8.1
la17	71,549	767,296	926	784	12.3	5.8	5.7	1537.5	3.8
la18	73,387	786,292	963	848	14.1	6.7	6.7	1341.9	3.2
la19	92,303	992,462	1162	842	28.4	30.4	14.0	5189.8	10.1
la20	91,828	986,657	1162	902	12.0	6.5	8.7	1473.4	5.9
la22	158,923	2,493,700	1275	927	1815.0	1625.1	1246.2	time-out	878.7
la23	157,152	2,461,589	1279	1032	2239.0	1810.0	1408.4	time-out	827.9
la24	162,299	2,545,886	1318	935	1657.4	1401.4	1280.9	time-out	1238.6
la25	146,928	2,300,907	1196	977	2271.5	2167.6	1343.0	time-out	1180.4
la36	265,072	4,178,965	1546	1268	996.2	770.0	641.5	time-out	210.1
la37	334,107	5,277,206	1868	1397	4975.2	4129.2	3284.7	time-out	1749.7
la38	289,006	4,560,370	1624	1196	time-out	time-out	4797.2	time-out	2511.2
la39	277,638	4,379,775	1584	1233	1738.9	1458.6	892.4	time-out	457.2
la40	279,616	4,411,308	1591	1222	6856.2	time-out	5532.0	time-out	2498.5
orb01	106,638	1,148,842	1303	1059	1822.4	1658.1	1351.08	time-out	1302.4
orb02	107,190	1,154,808	1295	888	25.5	14.5	13.0	2673.5	6.5
orb03	123,706	1,334,614	1461	1005	1530.1	749.0	552.1	time-out	554.2
orb04	113,489	1,223,253	1369	1005	80.6	85.6	63.1	time-out	45.5
orb05	94,014	1,010,346	1152	887	50.1	46.9	31.2	time-out	27.9
orb06	126,502	1,364,933	1500	1010	431.3	294.5	193.4	time-out	105.2
orb07	45,996	492,810	563	397	22.5	17.3	13.6	1414.0	9.8
orb08	101,159	1,089,539	1209	899	98.9	62.2	49.6	time-out	40.0
orb09	96,905	1,043,343	1189	934	106.3	83.9	68.9	time-out	40.2
orb10	125,675	1,356,366	1503	944	49.7	27.9	33.2	time-out	14.9
Total [seconds]					-	-	23025.3	-	13847.4
Ratio					-	-	(1.00)	-	0.60

Turning now to the 49 problems unsolved within 2 CPU hours, even their 48 relaxed problems can not be solved by SATSHOP. Furthermore, some SAT instances are huge [7] for our experimental environment. Ten of the 49 instances require more than 1GB memory, and five of the ten require more than 4GB memory which a 32-bits CPU can not manipulate any more.

6 Conclusions and Future Work

In this paper we presented a characterization of a minimal model with respect to an atom set. Based on this characterization, we gave minimal model generation procedures using a SAT solver as a subroutine. The only function we require from the SAT solver is to compute a model of a satisfiable SAT instance. Thus, our implementation benefits from efficiency of state-of-the-art SAT solvers.

We implemented the naive, normal, and single-solution versions with the SAT solver MiniSat 2. We have performed an experimental evaluation with 82 JSSPs. It shows that the single-solution version usually beats the other two versions. Unfortunately, it rarely beats the SAT-based JSSP solver SATSHOP which performs several optimizations concerning the problem domain. It is for this reason that the SATSHOP generally beats others. In spite of the domination of the SATSHOP, the minimal model generation approach still has an advantage over the SATSHOP in the sense that the former is more general than the latter: the latter solve only JSSP while the former can solve not only JSSP but also several problems such as graph coloring problem, two dimensional strip packing problem, and so on. Stochastic SAT solvers, such as WalkSAT [17], may be useful for increasing performance of the three versions.

We have also applied the Max-SAT solver MiniMaxSAT to the 82 JSSPs. The experimental results show that the MiniMaxSAT is definitely inefficient for solving the JSSP in our SAT encoding though it is a state-of-the-art Max-SAT solver. Implementing a Max-SAT solver based on our approach looks like interesting future work.

Some problems can not be solved because of memory capacity. In order to solve these problems in our framework, we have to purchase a 64-bits CPU and memory, or develop methods to manipulate the problem on the available memory. Encoding the problem into a first order formula seems to be a promising approach to save memory [16].

Answer set programming launched out into the new paradigm of logic programming in 1999, in which a logic program represents the constraints of a problem and its answer sets correspond to the solutions of the problem [11]. Computing answer sets is realized by generating minimal models and checking whether they satisfy some conditions for *negation as failure* [8]. We plan to extend this work to computing answer sets.

[7] SWV13 has the hugest instance in our experiment. It has 2.4 million variables and 121.6 million clauses. And its DIMACS file in gzip format occupies 596 MB.

References

1. L. Bordeaux, Y. Hamadi, and L. Zhang: Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Computing Surveys,* Vol.38, No.4, Article 12 (2006)
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu: Symbolic Model Checking without BDDs. In Proc. of TACAS'99, pp.193–207 (1999)
3. F. Bry and A. Yahya: Minimal Model Generation with Positive Unit Hyper-resolution Tableaux. In Proc. of TABLEAUX'96, pp.143–159 (1996)
4. J. M. Crawford, A. B. Baker: Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In Proc. of AAAI-94, pp.1092–1097 (1994)
5. N. Eén and N. Sörensson: An Extensible SAT-solver. In Proc. of SAT-2003, pp.502–518 (2003)
6. F. Heras, J. Larrosa, and A. Oliveras: MiniMaxSAT: An Efficient Weighted Max-SAT Solver. *J. of Artificial Intelligence Research,* Vol.31, pp.1–32 (2008)
7. H. Kautz and B. Selman: Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In Proc. of AAAI-96, pp.1194–1201 (1996)
8. K. Inoue, M. Koshimura, and R. Hasegawa: Embedding Negation as Failure into a Model Generation Theorem Prover. In Proc. of CADE-11, pp.400–415 (1992)
9. R. Hasegawa, H. Fujita, and M. Koshimura: Efficient Minimal Model Generation Using Branching Lemmas. In Proc. of CADE-17, pp.184–199 (2000)
10. V. Lifschitz: Computing Circumscription. In Proc. of IJCAI-85, pp.121–127 (1985)
11. V. Lifschitz: Answer Set Planning. In Proc. of ICLP-99, pp.23–37 (1999)
12. M. V. Moskewicz, C. F. Madigan, Y. Zhao, and L. Zhang: Chaff: Engineering an Efficient SAT Solver. In Proc. of DAC'01, pp.530–535 (2001)
13. H. Nabeshima, T. Soh, K. Inoue, and K. Iwanuma: Lemma Reusing for SAT based Planning and Scheduling. In Proc. of ICAPS'06, pp.103–112 (2006)
14. I. Niemelä: A Tableau Calculus for Minimal Model Reasoning. In Proc. of TABLEAUX'96, pp.278–294 (1996)
15. OR-Library. `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`
16. J. A. Navarro-Pérez and A. Voronkov: Encodings of Bounded LTL Model Checking in Effectively Propositional Logic. In Proc. of CADE-21, pp.346–361 (2007)
17. B. Selman, H. Kautz, and B. Cohen: Local Search Strategies for Satisfiability Testing. Discrete Mathematics and Theoretical Computer Science, vol. 26, AMS, (1996)

Literal Projection and Circumscription

Christoph Wernhard

Technische Universität Dresden
christoph.wernhard@tu-dresden.de

Abstract. We develop a formal framework intended as a preliminary step for a single knowledge representation system that provides different representation techniques in a unified way. In particular we consider first-order logic extended by techniques for second-order quantifier elimination and non-monotonic reasoning. Background of the work is literal projection, a generalization of second-order quantification which permits, so to speak, to quantify upon an arbitrary sets of ground literals, instead of just (all ground literals with) a given predicate symbol. In this paper, an operator raise is introduced that is only slightly different from literal projection and can be used to define a generalization of circumscription in a straightforward and compact way. Some properties of this operator and of circumscription defined in terms of it, also in combination with literal projection, are then shown. A previously known characterization of consequences of circumscribed formulas in terms of literal projection is generalized from propositional to first-order logic. A characterization of answer sets according to the stable model semantics in terms of circumscription is given. This characterization does not recur onto syntactic notions like *reduct* and fixed-point construction. It essentially renders a recently proposed "circumscription-like" characterization in a compact way without involvement of a specially interpreted connective.

1 Introduction

We develop a formal framework intended as a preliminary step for a single knowledge representation system that provides different representation techniques in a unified way. In particular we consider first-order logic extended by techniques for second-order quantifier elimination and non-monotonic reasoning.

Second-order quantifier elimination permits to express a large number of knowledge representation techniques (see for example [5]), including abduction, modularization of knowledge bases and the processing of circumscription. It is also closely related to knowledge compilation [13]. Variants of second-order quantifier elimination also appear under names such as *computation of uniform interpolants*, *forgetting*, and *projection*. Restricted to propositional formulas it is called *elimination of Boolean quantified variables*.

We focus here on a particular generalization of second-order quantifier elimination, the *computation of literal projection* [10, 11]. Literal projection generalizes second-order quantification by permitting, so to speak, to quantify upon an *arbitrary set of ground literals*, instead of just (all ground literals with) a given predicate symbol. Literal projection allows, for example, to express predicate quantification upon a predicate just in positive or negative polarity. Eliminating such a quantifier from a formula in negation normal form results in a formula that might still contain the quantified predicate, but only in literals whose polarity is complementary to the quantified one. This polarity dependent behavior of literal projection is essential for the relationship to non-monotonic reasoning that is investigated in this paper.

In particular, we consider circumscription and, based on it, the stable model semantics, which underlies many successful applications developed during the last decade. It is well-known that the processing of circumscription can be expressed as a second-order quantifier elimination task [1]. The formalization of circumscription investigated here does not just rely on literal projection as a generalization of second-order quantification, but utilizes the polarity dependent behavior of literal projection to obtain a particular straightforward and compact characterization. The concrete contributions of this paper are:

- The introduction of an operator raise that is only slightly different from literal projection and can be used to define a generalization of parallel circumscription with varied predicates in a straightforward and compact way.

  Like literal projection, the raise operator is defined in terms of semantic properties only, and is thus independent of syntactic properties or constructions. Some properties of this operator and circumscription, also in interaction with literal projection, are then shown (Sect. 3–6).

- The characterization of consequences of circumscribed formulas in terms of literal projection. We make a known result given in [7] more precise and generalize it from propositional to first-order formulas. In the extended report version of this paper [12] we provide a thorough proof (Sect. 6).

- A definition of answer sets according to the stable model semantics in terms of circumscription. Unlike the common definitions of stable models, it does not recur onto syntactic notions like *reduct* and fixed-point construction. It is essentially an adaption of the "circumscription-like" definition recently proposed in [3, 4]. In contrast to that definition, it does not involve a specially interpreted rule forming connective (Sect. 7).

The paper is structured as follows: Preliminaries are given in Section 2, including a description of the used semantic framework and a summary of background material on literal projection. In Sections 3–7 the proper contributions of this paper are described and formally stated. Proofs of propositions and theorems as well as more details on the relationship of the introduced definition of stable models to other characterizations can be found in the extended report version of this paper [12].

2 Notation and Preliminaries

Symbols. We use the following symbols, also with sub- and superscripts, to stand for items of types as indicated in the following table (precise definitions of these types are given later on in this section). They are considered implicitly as universally quantified in definition, theorem and proposition statements.

F, G	– Formula
A	– Atom
L	– Literal
S	– Set of ground literals (also called *literal scope*)
M	– Consistent set of ground literals
I, J, K	– Structure
β	– Variable assignment

Notation. Unless specially noted, we assume that a *first-order formula* is constructed from first-order literals, truth value constants $\top, \bot$, the unary connective $\neg$, binary connectives $\land, \lor$ and the first-order quantifiers $\forall$ and $\exists$. We write the positive (negative) literal with atom A as $+A$ $(-A)$. Variables are x, y, z, also with subscripts. As meta-level notation with respect to this syntax we use implication $\rightarrow$, biconditional $\leftrightarrow$ and n-ary versions of the binary connectives.

A clause is a sentence of the form $\forall x_1 \ldots \forall x_n (L_1 \lor \ldots \lor L_m)$, where $n, m \geq 0$ and the L_i for $i \in \{1, \ldots, m\}$ are literals. Since all variables in a clause are universally quantified, we sometimes do not write its quantifier prefix.

We assume a fixed first-order signature with at least one constant symbol. The sets of all ground terms and all ground literals, with respect to this signature, are denoted by TERMS and ALL, respectively.

The Projection Operator and Literal Scopes. A *formula* in general is like a first-order formula, but in its construction two further operators, $\mathsf{project}(F, S)$ and $\mathsf{raise}(F, S)$, are permitted, where F is a formula and S specifies a set of ground literals. We call a set of ground literals in the role as argument to $\mathsf{project}$ or raise a *literal scope*. We do not define here a concrete syntax for specifying literal scopes and just speak of a *literal scope*, referring to the actual literal scope in a semantic context as well as some expression that denotes it in a syntactic context. The formula $\mathsf{project}(F, S)$ is called the *literal projection* of F onto S. Literal projection generalizes existential second-order quantification [10] (see also Sect. 4 below). It will be further discussed in this introductory section (see [10, 11] for more thorough material). The semantics of the raise operator will be introduced later on in Sect. 3.

Interpretations. We use the notational variant of the framework of Herbrand interpretations described in [10]: An *interpretation* $\mathfrak{I}$ is a pair $\langle I, \beta \rangle$, where I is a *structure*, that is, a set of ground literals that contains for all ground atoms A exactly one of $+A$ or $-A$, and β is a *variable assignment*, that is, a mapping of the set of variables into TERMS.

Satisfaction Relation and Semantics of Projection. The satisfaction relation between interpretations $\mathfrak{I} = \langle I, \beta \rangle$ and formulas is defined by the clauses in Tab. 1, where L matches a literal, F, F_1, F_2 match a formula, and S matches a literal scope. In the table, two operations on variable assignments β are used: If F is a formula, then $F\beta$ denotes F with all variables replaced by their image in β; If x is a variable and t a ground term, then $\beta\frac{t}{x}$ is the variable assignment that maps x to t and all other variables to the same values as β. Entailment and equivalence are straightforwardly defined in terms of the satisfaction relation. Entailment: $F_1 \models F_2$ holds if and only if for all $\langle I, \beta \rangle$ such that $\langle I, \beta \rangle \models F_1$ it holds that $\langle I, \beta \rangle \models F_2$. Equivalence: $F_1 \equiv F_2$ if and only if $F_1 \models F_2$ and $F_2 \models F_1$.

Intuitively, the literal projection of a formula F onto scope S is a formula that expresses about literals in S the same as F, but expresses nothing about other literals. The projection is equivalent to a formula without the projection operator, in negation normal form, where all ground instances of literals occurring in it are members of the projection scope. The semantic definition of literal projection in Tab. 1 can be alternatively expressed as: An interpretation $\langle I, \beta \rangle$ satisfies $\mathsf{project}(F, S)$ if and only if there is a structure J such that $\langle J, \beta \rangle$ satisfies F and I can be obtained from J by replacing literals that are not in S with their complements. This includes the special case $I = J$, where no literals are replaced.

Table 1. The Satisfaction Relation with the Semantic Definition of Literal Projection

$\langle I, \beta \rangle \models L$	iff$_{\text{def}}$	$L\beta \in I$
$\langle I, \beta \rangle \models \top$		
$\langle I, \beta \rangle \not\models \bot$		
$\langle I, \beta \rangle \models \neg F$	iff$_{\text{def}}$	$\langle I, \beta \rangle \not\models F$
$\langle I, \beta \rangle \models F_1 \land F_2$	iff$_{\text{def}}$	$\langle I, \beta \rangle \models F_1$ and $\langle I, \beta \rangle \models F_2$
$\langle I, \beta \rangle \models F_1 \lor F_2$	iff$_{\text{def}}$	$\langle I, \beta \rangle \models F_1$ or $\langle I, \beta \rangle \models F_2$
$\langle I, \beta \rangle \models \forall x\, F$	iff$_{\text{def}}$	for all $t \in \mathsf{TERMS}$ it holds that $\langle I, \beta\frac{t}{x} \rangle \models F$
$\langle I, \beta \rangle \models \exists x\, F$	iff$_{\text{def}}$	there exists a $t \in \mathsf{TERMS}$ such that $\langle I, \beta\frac{t}{x} \rangle \models F$
$\langle I, \beta \rangle \models \mathsf{project}(F, S)$	iff$_{\text{def}}$	there exists a J such that $\langle J, \beta \rangle \models F$ and $J \cap S \subseteq I$

Relation to Conventional Model Theory. Literal sets as components of interpretations permit the straightforward definition of the semantics of literal projection given in the last clause in Tab. 1. The set of literals I of an interpretation $\langle I, \beta \rangle$ is called *"structure"*, since it can be considered as representation of a structure in the conventional sense used in model theory: The domain is the set of ground terms. Function symbols f with arity $n \geq 0$ are mapped to functions f' such that for all ground terms $t_1, ..., t_n$ it holds that $f'(t_1, ..., t_n) = f(t_1, ..., t_n)$. Predicate symbols p with arity $n \geq 0$ are mapped to $\{\langle t_1, ..., t_n \rangle \mid +p(t_1, ..., t_n) \in I\}$. Moreover, an interpretation $\langle I, \beta \rangle$ represents a conventional second-order interpretation [2] (if predicate variables are considered as distinguished predicate symbols): The structure in the conventional sense corresponds to I, as described above, except that mappings of predicate variables are omitted. The assignment is β, extended such that all predicate variables p are mapped to $\{\langle t_1, ..., t_n \rangle \mid +p(t_1, ..., t_n) \in I\}$.

Some More Notation. The following table specifies symbolic notation for (i) the complement of a literal, (ii) the set of complement literals of a given set of literals, (iii) the set complement of a set of ground literals, (iv) the set of all positive ground literals, (v) the set of all negative ground literals, (vi) the set of all ground literals whose predicate symbol is from a given set, and (vii, viii) a structure that is like a given one, except that it assigns given truth values to a single given ground atom or to all ground atoms in a given set, respectively.

(i) $\widetilde{+A} \overset{\text{def}}{=} -A$; $\widetilde{-A} \overset{\text{def}}{=} +A$. The literal $\widetilde{L}$ is called the *complement* of L.
(ii) $\widetilde{S} \overset{\text{def}}{=} \{\widetilde{L} \mid L \in S\}$.
(iii) $\overline{S} \overset{\text{def}}{=} \mathsf{ALL} - S$.
(iv) $\mathsf{POS} \overset{\text{def}}{=} \{+A \mid +A \in \mathsf{ALL}\}$.
(v) $\mathsf{NEG} \overset{\text{def}}{=} \{-A \mid -A \in \mathsf{ALL}\}$.
(vi) $\hat{P}$ is the set of all ground literals whose predicate is P or is in P, resp., where P is a predicate symbol, or a tuple or set of predicate symbols.
(vii) $I[L] \overset{\text{def}}{=} (I - \{\widetilde{L}\}) \cup \{L\}$.
(viii) $I[M] \overset{\text{def}}{=} (I - \widetilde{M}) \cup M$.

Literal Base and Related Concepts. The *literal base* $\mathcal{L}(F)$ of a first-order formula F in negation normal form is the set of all ground instances of literals in F. The following formal definition generalizes this notion straightforwardly for formulas that are not in negation normal form and possibly include the $\mathsf{project}$ and raise operator: $\mathcal{L}(L)$ is the set of all ground instances of L; $\mathcal{L}(\top) \overset{\text{def}}{=} \mathcal{L}(\bot) \overset{\text{def}}{=} \{\}$; $\mathcal{L}(\neg F) \overset{\text{def}}{=} \widetilde{\mathcal{L}(F)}$; $\mathcal{L}(F \otimes G) \overset{\text{def}}{=} \mathcal{L}(F) \cup \mathcal{L}(G)$ if $\otimes$ is $\land$ or $\lor$; $\mathcal{L}(\otimes x F) \overset{\text{def}}{=} \mathcal{L}(\otimes(F, S)) \overset{\text{def}}{=} \mathcal{L}(F)$ if $\otimes$ is a quantifier or the $\mathsf{project}$ or raise operator, respectively.

We call the set of ground literals "about which a formula expresses something" its *essential literal base*, made precise in Def. 1 (see [10, 11] for a more thorough discussion). It can be proven that essential literal base of a formula is a subset of its literal base. The essential literal base is independent of syntactic properties: equivalent formulas have the same essential literal base.

Definition 1 (Essential Literal Base). The *essential literal base* of a formula F, in symbols $\mathcal{L}_{\mathcal{E}}(F)$, is defined as $\mathcal{L}_{\mathcal{E}}(F) \overset{\text{def}}{=} \{L \mid L \in \mathsf{ALL}$ and there exists an interpretation $\langle I, \beta \rangle$ such that $\langle I, \beta \rangle \models F$ and $\langle I[\widetilde{L}], \beta \rangle \not\models F\}$.

Properties of Literal Projection. A summary of properties of literal projection is displayed in Tab. 2 and 3. Most of them follow straightforwardly from the semantic definition of $\mathsf{project}$ shown in Tab. 1 [11]. The more involved proof of Tab. 2.xxi (and the related Tab. 3.v) can be found in [10, 11]. The properties in Tab. 3 strengthen properties in Tab. 2, but apply only to formulas that satisfy a condition related to their essential literal base. These formulas are called $\mathcal{E}$-formulas and are defined as follows:

Definition 2 ($\mathcal{E}$-Formula). A formula F is called *$\mathcal{E}$-formula* if and only if for all interpretations $\langle I, \beta \rangle$ and consistent sets of ground literals M such that $\langle I, \beta \rangle \models F$ and $M \cap \mathcal{L}_{\mathcal{E}}(F) = \emptyset$ it holds that $\langle I[\widetilde{M}], \beta \rangle \models F$.

First-order formulas in negation normal form without existential quantifier – including propositional formulas and first-order clausal formulas – are $\mathcal{E}$-formulas. Being an $\mathcal{E}$-formula is a property that just depends on the semantics of a formula, that is, an equivalent to an $\mathcal{E}$-formula is also an $\mathcal{E}$-formula. See [10, 11] for more discussion.[1]

Table 2. Properties of Literal Projection

(i)	$F \models \mathsf{project}(F, S)$
(ii)	*If* $F_1 \models F_2$, *then* $\mathsf{project}(F_1, S) \models \mathsf{project}(F_2, S)$
(iii)	*If* $F_1 \equiv F_2$, *then* $\mathsf{project}(F_1, S) \equiv \mathsf{project}(F_2, S)$
(iv)	*If* $S_1 \supseteq S_2$, *then* $\mathsf{project}(F, S_1) \models \mathsf{project}(F, S_2)$
(v)	$\mathsf{project}(\mathsf{project}(F, S_1), S_2) \equiv \mathsf{project}(F, S_1 \cap S_2)$
(vi)	$F_1 \models \mathsf{project}(F_2, S)$ *if and only if* $\mathsf{project}(F_1, S) \models \mathsf{project}(F_2, S)$
(vii)	$\mathsf{project}(F, \mathsf{ALL}) \equiv F$
(viii)	$\mathsf{project}(F, \mathcal{L}(F)) \equiv F$
(ix)	$\mathsf{project}(\top, S) \equiv \top$
(x)	$\mathsf{project}(\bot, S) \equiv \bot$
(xi)	F *is satisfiable if and only if* $\mathsf{project}(F, S)$ *is satisfiable*
(xii)	$\mathcal{L}_{\mathcal{E}}(\mathsf{project}(F, S)) \subseteq S$
(xiii)	$\mathcal{L}_{\mathcal{E}}(\mathsf{project}(F, S)) \subseteq \mathcal{L}_{\mathcal{E}}(F)$
(xiv)	*If* $\mathsf{project}(F, S) \models F$, *then* $\mathcal{L}_{\mathcal{E}}(F) \subseteq S$
(xv)	$\mathsf{project}(F, S) \equiv \mathsf{project}(F, \mathcal{L}(F) \cap S)$
(xvi)	$F_1 \models F_2$ *if and only if* $\mathsf{project}(F_1, \mathcal{L}(F_2)) \models F_2$
(xvii)	*If no instance of* L *is in* S, *then* $\mathsf{project}(L, S) \equiv \top$
(xviii)	*If all instances of* L *are in* S, *then* $\mathsf{project}(L, S) \equiv L$
(xix)	$\mathsf{project}(F_1 \lor F_2, S) \equiv \mathsf{project}(F_1, S) \lor \mathsf{project}(F_2, S)$
(xx)	$\mathsf{project}(F_1 \land F_2, S) \models \mathsf{project}(F_1, S) \land \mathsf{project}(F_2, S)$
(xxi)	*If* $\mathcal{L}(F_1) \cap \widetilde{\mathcal{L}(F_2)} \subseteq S \cap \widetilde{S}$ *then* $\mathsf{project}(F_1 \land F_2, S) \equiv \mathsf{project}(F_1, S) \land \mathsf{project}(F_2, S)$
(xxii)	$\mathsf{project}(\exists x F, S) \equiv \exists x\ \mathsf{project}(F, S)$
(xxiii)	$\mathsf{project}(\forall x F, S) \models \forall x\ \mathsf{project}(F, S)$

Table 3. Properties of Literal Projection for $\mathcal{E}$-Formulas E

(i)	$\mathsf{project}(E, \mathcal{L}_{\mathcal{E}}(E)) \equiv E$	(strengthens Tab. 2.viii)
(ii)	$\mathcal{L}_{\mathcal{E}}(E) \subseteq S$ *if and only if* $\mathsf{project}(E, S) \equiv E$	(strengthens Tab. 2.xiv)
(iii)	$\mathsf{project}(E, S) \equiv \mathsf{project}(E, \mathcal{L}_{\mathcal{E}}(E) \cap S)$	(strengthens Tab. 2.xv)
(iv)	$F \models E$ *if and only if* $\mathsf{project}(F, \mathcal{L}_{\mathcal{E}}(E)) \models E$	(strengthens Tab. 2.xvi)
(v)	*If* $\mathcal{L}_{\mathcal{E}}(E_1) \cap \widetilde{\mathcal{L}_{\mathcal{E}}(E_2)} \subseteq S \cap \widetilde{S}$ *then* $\mathsf{project}(E_1 \land E_2, S) \equiv \mathsf{project}(E_1, S) \land \mathsf{project}(E_2, S)$	(strengthens Tab. 2.xxi)

[1] An example that is not an $\mathcal{E}$-formula is the sentence $F \overset{\text{def}}{=} \forall x\ \ +\mathsf{r}(x, \mathsf{f}(x)) \land \forall x \forall y(-\mathsf{r}(x, y) \lor +\mathsf{r}(x, \mathsf{f}(y))) \land \exists x \forall y(-\mathsf{r}(x, y) \lor +\mathsf{p}(y))$. Let the domain be the set of all terms $\mathsf{f}^n(\mathsf{a})$ where $n \geq 0$. For each member T of the domain it can be verified that $+\mathsf{p}(T) \notin \mathcal{L}_{\mathcal{E}}(F)$. On the other hand, an interpretation that contains $-\mathsf{p}(T)$ for all members T of the domain cannot be a model of F.

3 The Raise Operator

The following operator raise is only slightly different from literal projection and, as we will see later on, can be used to define a generalization of parallel circumscription with varied predicates in a straightforward and compact way.

Definition 3 (Raise).

$$\langle I, \beta \rangle \models \mathsf{raise}(F, S) \text{ iff}_{\text{def}} \begin{array}{l} \text{there exists a } J \text{ such that} \\ \langle J, \beta \rangle \models F \text{ and} \\ J \cap S \subset I \cap S. \end{array}$$

The definition of raise is identical to that of literal projection (Tab. 1), with the exception that $J \cap S$ and $I \cap S$ are related by the *proper subset* instead of the *subset* relationship.

The name "raise" suggests that a model $\langle I, \beta \rangle$ of $\mathsf{raise}(F, S)$ is not "the lowest" model of F, in the sense that there exists another model $\langle J, \beta \rangle$ of F with the property $J \cap S \subset I \cap S$. An equivalent specification of the condition $J \cap S \subset I \cap S$ in the definition of raise provides further intuition on its effect: A literal scope S can be partitioned into three disjoint subsets S_p, S_n, S_{pn} such that S_p (S_n) is the set of positive (negative) literals in S whose complement is not in S, and S_{pn} is the set of literals in S whose complement is also in S. Within Def. 3, the condition $J \cap S \subset I \cap S$ can then be equivalently expressed by the conjunction of $J \cap (S_p \cup S_n) \subset I \cap (S_p \cup S_n)$ and $J \cap S_{pn} = I \cap S_{pn}$. That is, with respect to members of S whose complement is not in S, the structure J must be a proper subset of I, and with respect to the other members of S it must be identical to I.

Proposition 1 below shows some properties of the raise operator: It is monotonic (Prop. 1.i). From this follows that it is a "semantic" operator in the sense that for equivalent arguments the values are equivalent too (Prop. 1.ii). Like projection, the raise operator distributes over disjunction (Prop. 1.iii). Proposition 1.iv follows from monotonicity. Proposition 1.v shows that for scopes that contain exactly the same atoms positively as well as negatively, raise is inconsistent. Propositions 1.vi and 1.vi show the interplay of raise with projection onto the same scope. Proposition 1.viii provides a characterization of *literal* projection in terms of raise and *atom* projection [10], a restricted form of projection where the polarity of the scope members is not taken into account, which can be expressed as literal projection onto scopes S constrained by $S = \widetilde{S}$.

Proposition 1 (Properties of Raise).

(i) *If* $F_1 \models F_2$, *then* $\mathsf{raise}(F_1, S) \models \mathsf{raise}(F_2, S)$.
(ii) *If* $F_1 \equiv F_2$, *then* $\mathsf{raise}(F_1, S) \equiv \mathsf{raise}(F_2, S)$.
(iii) $\mathsf{raise}(F_1 \lor F_2, S) \equiv \mathsf{raise}(F_1, S) \lor \mathsf{raise}(F_2, S)$.
(iv) $\mathsf{raise}(F_1 \land F_2, S) \models \mathsf{raise}(F_1, S) \land \mathsf{raise}(F_2, S)$.
(v) *If* $S = \widetilde{S}$, *then* $\mathsf{raise}(F, S) \equiv \bot$.
(vi) $\mathsf{raise}(\mathsf{project}(F, S), S) \equiv \mathsf{raise}(F, S)$.
(vii) $\mathsf{project}(\mathsf{raise}(F, S), S) \equiv \mathsf{raise}(F, S)$.
(viii) $\mathsf{project}(F, S) \equiv \mathsf{project}(F, S \cup \widetilde{S}) \lor \mathsf{raise}(F, S)$.

4 Definition of Circumscription in Terms of Raise

The following definition specifies a formula $\mathsf{circ}(F, S)$ that provides a characterization of circumscription in terms of raise, as we will first outline informally and then show more precisely.

Definition 4 (Circumscription).

$$\mathsf{circ}(F, S) \stackrel{\text{def}}{=} F \land \neg\mathsf{raise}(F, S).$$

In this notation, the *parallel circumscription of predicate constants P in sentence F with varied predicate constants Z* [8] is expressed as $\mathsf{circ}(F, S)$, where S is the set of all ground literals L such that either

1. L is positive and its predicate is in P, or
2. The predicate of L is neither in P nor in Z.

In other words, the scope S contains the circumscribed predicates just positively (the positive literals according to item 1.), and the "fixed" predicates in full (all positive as well as negative literals according to item 2.). Since the literal scope in $\mathsf{circ}(F, S)$ can be an arbitrary sets of literals, $\mathsf{circ}(F, S)$ is more general than parallel circumscription with varied predicates: Model maximization conditions can be expressed by means of scopes that contain negative literals but not their complements. Furthermore, it is possible to express minimization, maximization and variation conditions that apply only to a subset of the instances of a predicate.

We now make precise how circ relates to the established definition of circumscription by means of second-order quantification [8, 1, 5]. The following definition specifies a second-order sentence $\mathsf{CIRC}[F; P; Z]$ that is called *parallel circumscription of predicate constants P in F with varied predicate constants Z* in [8] and is straightforwardly equivalent to the sentence called *second-order circumscription of P in F with variable Z* in [1, 5]:

Definition 5 (Second-Order Circumscription). Let F be a first-order sentence and let P, P', Z, Z' be mutually disjoint tuples of distinct predicate symbols such that: $P = p_1, \ldots, p_n$ and $P' = p'_1, \ldots, p'_n$ where $n \geq 0$; both Z and Z' have the same length ≥ 0; members of P' and P with the same index, as well as members of Z' and Z with the same index, are of the same arity; and P' and Z' do not contain predicate symbols in F. Let F' be the formula that is obtained from F by replacing each predicate symbol that is in P or Z by the predicate symbol with the same index in P' or Z', respectively. For $i \in \{1, \ldots, n\}$ let $\overline{x}_i$ stand for $x_1, \ldots, x_k$, where k is the arity of predicate symbol p_i. Let $P' < P$ stand for

$$\bigwedge_{i=1}^{n} \forall \overline{x}_i (p'_i(\overline{x}_i) \rightarrow p_i(\overline{x}_i)) \land \neg \bigwedge_{i=1}^{n} \forall \overline{x}_i (p'_i(\overline{x}_i) \leftrightarrow p_i(\overline{x}_i))).$$

Considering the predicate symbols in P' and Z' as predicate variables, the *second-order circumscription of P in F with variable Z*, written $\mathsf{CIRC}[F; P; Z]$, is then defined as:

$$\mathsf{CIRC}[F; P; Z] \stackrel{\text{def}}{=} F \land \neg\exists P', Z' \, (F' \land P' < P).$$

Existential second-order quantification can be straightforwardly expressed with literal projection: $\exists p\ G$ corresponds to $\mathsf{project}(G, S)$, where S is the set of all ground literals with a predicate other than p. From Tab. 2.xv it can be derived that also a smaller projection scope is sufficient: $\mathsf{project}(G, S)$ is equivalent to $\mathsf{project}(G, S')$ for all subsets S' of S that contain those literals of S whose predicate symbol occurs in G. Accordingly, $\mathsf{CIRC}[F; P; Z]$ can be expressed straightforwardly in terms of literal projection instead of the second-order quantification:

Definition 6 (Second-Order Circumscription in Terms of Projection). Let F be a first-order formula and let P, P', Z, Z' be tuples of predicate symbols as specified in the definition of CIRC (Def. 5). Let Q be the set of predicate symbols in F that are neither in P nor in Z. Then $\mathsf{CIRC\text{-}PROJ}[F; P; Z]$ is a formula with the projection operator, defined as:

$$\mathsf{CIRC\text{-}PROJ}[F; P; Z] \overset{\text{def}}{=} F \land \neg\mathsf{project}(F' \land P' {<} P,\ \hat{P} \cup \hat{Q}).$$

The Q parameter in Def. 6 is the set of the "fixed" predicates. The set of literals $(\hat{P} \cup \hat{Q})$ suffices as projection scope, since the quantified body of the right conjunct of $\mathsf{CIRC}[F; P; Z]$, that is, $(F' \land P' {<} P)$, contains – aside of the quantified predicate symbols from P', Z' – just predicate symbols that are in P or in Q.

The following theorem makes precise how second-order circumscription can be expressed with circ. Its proof in [12] formally relates second-order circumscription expressed by projection (Def. 6) with circumscription defined in terms of of the raise operator (Def. 4).

Theorem 1 (Second-Order Circumscription Expressed by circ). *Let F be a first-order formula and let P, P', Z, Z' be tuples of predicate symbols as specified in the definition of* CIRC *(Def. 5). Let Q be the set of predicate symbols in F that are neither in P nor in Z. Then*

$$\mathsf{CIRC\text{-}PROJ}[F; P; Z] \equiv \mathsf{circ}(F, (\hat{P} \cap \mathsf{POS}) \cup \hat{Q}).$$

5 Well-Foundedness

As discussed in [8], circumscription can in general only be applied usefully to a sentence F if all models of F extend some model of F that is minimal with respect to the circumscribed predicates. The concept of well-foundedness [8] makes this property precise. We show that it can be expressed in a compact way in terms of projection. This characterization facilitates to prove properties of circumscription that have well-foundedness as a precondition, as for example Prop. 3 and Theorem 2 below.

Definition 7 (Well-Founded Formula). F is called *well-founded with respect to S* if and only if

$$F \models \mathsf{project}(\mathsf{circ}(F, S), S).$$

In this definition, the literal scope S can be an arbitrary set of literals, corresponding to variants of circumscription as indicated in Sect. 4. We now explicate how this definition renders the definition of well-foundedness in [8], which is defined for the special case of circumscription of a single predicate p with varied predicates Z. That definition is as as follows (adapted to our notation): Let F be a first-order sentence, p be predicate symbol and Z be a tuple of predicate symbols. The sentence F is called *well-founded with respect to* $(p; Z)$ if for every model $\mathfrak{I}$ of F there exists a model $\mathfrak{J}$ of $\mathsf{CIRC}[F; p; Z]$ such that $\mathfrak{I}$ and $\mathfrak{J}$ differ only in how they interpret p and Z and the extent of p in J is a (not necessarily strict) subset of its extent in I. We can convert this definition straightforwardly into our semantic framework: Let Q be the set of predicate symbols in F that are different from p and not in Z. The sentence F is then well-founded with respect to $(p; Z)$ if for all interpretations $\langle I, \beta\rangle$ such that $\langle I, \beta\rangle \models F$ there exists an interpretation $\langle J, \beta\rangle$ such that (1.) $\langle J, \beta\rangle \models \mathsf{CIRC\text{-}PROJ}[F; p; Z]$, (2.) $J \cap \hat{p} \cap \mathsf{POS} \subseteq \cap I$, and (3.) $J \cap \hat{Q} = I \cap \hat{Q}$. The $\mathsf{project}$ operator allows to express this converted definition compactly: Let S be the literal scope $((\hat{p} \cap \mathsf{POS}) \cup \hat{Q})$. By Theorem 1, $\mathsf{CIRC\text{-}PROJ}[F; p; Z]$ is equivalent to $\mathsf{circ}(F, S)$. Furthermore, given that I and J are structures and $\hat{Q} = \widetilde{\hat{Q}}$, the conjunction of items (2.) and (3.) above is equivalent to $J \cap S \subseteq I$. By the definition of $\mathsf{project}$ (Tab. 1), the statement that there exists an interpretation $\langle J, \beta\rangle$ satisfying items (1.)–(3.) can be expressed as $\langle I, \beta\rangle \models \mathsf{project}(\mathsf{circ}(F, S), S)$.

6 Interplay of Projection and Circumscription

The following proposition shows properties of projection nested within circumscription. It is independent of the *well-founded* property.

Proposition 2 (Circumscribing Projections).

(i) $\mathsf{circ}(F, S) \models \mathsf{circ}(\mathsf{project}(F, S), S)$.

(ii) $\mathsf{circ}(\mathsf{project}(F, S), S) \models \mathsf{circ}(\mathsf{project}(F, S \cup \widetilde{S}), S)$.

In the special case where $S \cup \widetilde{S} = \mathsf{ALL}$, which holds for example if $S = \mathsf{POS}$, the two entailments Prop. 2.i and Prop. 2.ii can be combined to the equivalence $\mathsf{circ}(\mathsf{project}(F, S), S) \equiv \mathsf{circ}(F, S)$. From this equivalence, it can be derived that two formulas which express the same about positive literals (that is, have equivalent projections onto POS) have the same minimal models (that is, have equivalent circumscriptions for scope POS).

The following proposition concerns circumscription nested within projection. It is a straightforward consequence of the definition of *well-founded* along with Tab. 2.vi and 2.ii.

Proposition 3 (Projecting Circumscriptions). *If F is well-founded with respect to S, then*

$$\mathsf{project}(\mathsf{circ}(F, S), S) \equiv \mathsf{project}(F, S).$$

From this proposition follows that if two well-founded formulas have equivalent circumscriptions for some scope, then also their projections onto that scope are equivalent. With properties of projection, it also follows that if S is a positive literal scope (that is, $S \subseteq \mathsf{POS}$) then $\mathsf{project}(\mathsf{circ}(F, \mathsf{POS}), S) \equiv \mathsf{project}(F, S)$. This equivalence can be applied to provide a straightforward justification for applying methods to compute minimal models also to projection computation onto positive scopes: We consider methods that compute for a given input formula F an output formula F' that satisfies syntactic criteria (for example correspondence to a tableau) which permit projection computation with low computational effort, such that projection computation is in essence already performed by computing F'. Assume that the output formula has the same minimal models as the input formula, that is, $\mathsf{circ}(F', \mathsf{POS}) \equiv \mathsf{circ}(F, \mathsf{POS})$. If F' is well-founded, for positive literal scopes S it then follows that $\mathsf{project}(F', S) \equiv \mathsf{project}(F, S)$. A tableau constructed by the hyper tableau calculus can indeed be viewed as representation of such a formula F' [13].

Literal forgetting is a variant of literal projection that can be defined as $\mathsf{forget}(F, S) \overset{\text{def}}{=} \mathsf{project}(F, \overline{S})$ and is investigated for propositional logic in [7]. It is shown there that circumscription, or more precisely the formulas that are entailed by circumscriptions, can be characterized in terms of literal forgetting. Two such characterizations are given as Proposition 22 in [7], where the simpler one applies if the literal base of the entailed formula is restricted in a certain way.

These characterizations are rendered here in terms of literal projection as Theorem 2.ii and 2.iii below, where we generalize and make more precise the statements given in [7] in the following four respects: (1.) We use weaker requirements on the entailed formulas by referring to the *essential* literal base instead of the (syntactic) literal base. The respective requirements on the syntactic literal base follow, since it is a subset of the essential literal base (see Sect. 2). (2.) We provide a thorough proof in [12]. The proof given in [7] just shows the characterizations as straightforward consequence of [9, Theorems 2.5 and 2.6], for which in turn no proof is given, neither in [9], nor in [6] which is referenced by [9]. (3.) We generalize the characterizations to first-order logic. (4.) We add a third basic variant (Theorem 2.i) for consequents that are stronger restricted than in Theorem 2.ii.

This basic variant is actually a straightforward generalization of Proposition 12 in [8], which is introduced as capturing the intuition that, under the assumption of well-foundedness, a circumscription provides no new information about the fixed predicates, and only "negative" additional information about the circumscribed predicates.

Theorem 2 (Consequences of Circumscription). *If F is well-founded with respect to S, then*

$$\mathsf{circ}(F, S) \models G$$

is equivalent to (at least) one of the following statements, depending on $\mathcal{L}_{\mathcal{E}}(G)$:

(i) $F \models G$, *if* $\mathcal{L}_{\mathcal{E}}(G) \subseteq S$;

(ii) $F \models \mathsf{project}(F \land G, S)$, *if* $\mathcal{L}_{\mathcal{E}}(G) \subseteq (S \cup \widetilde{S})$;

(iii) $F \models \mathsf{project}(F \land \neg\mathsf{project}(F \land \neg G, S), S)$.

7 Answer Sets with Stable Model Semantics

In [3, 4] a characterization of stable models in terms of a formula translation that is *similar* to the second-order circumscription has been presented. Roughly, it differs from circumscription in that only certain *occurrences* of predicates are circumscribed, which are identified by their position with respect to a non-classical rule forming operator. We develop a variant of this characterization of stable models that is *in terms of* circumscription. It involves no non-classical operators. Instead, to indicate occurrences be circumscribed, it puts atoms into term position, wrapped by one of two special predicates.

We let the symbols $\circ$ and $\bullet$ denote these predicates. They are unary, and we write them without parentheses – for example $\bullet\mathsf{p}(\mathsf{a})$. With them, we now formally define a notion of *logic program*. Its correspondence to the more conventional view of a logic program as formed by non-classical operators will then be indicated.

Definition 8 (Logic Program).

(i) A *rule clause* is a clause[2] of the form

$$\bigvee_{i=1}^{m} -\circ A_i \;\lor \bigvee_{i=1}^{n} +\bullet B_i \;\lor \bigvee_{i=1}^{o} +\circ C_i \;\lor \bigvee_{i=1}^{p} -\bullet D_i,$$

where $k, m, n, o, p \geq 0$ and the subscripted A, B, C, D are terms.

(ii) For a rule clause of the form specified in (8.i), the rule clause $(\bigvee_{i=1}^{m} -\circ A_i \lor \bigvee_{i=1}^{n} +\bullet B_i)$ is called its *negated body*, and the rule clause $(\bigvee_{i=1}^{o} +\circ C_i \lor \bigvee_{i=1}^{p} -\bullet D_i)$ is called its *head*.

(iii) A *logic program* is a conjunction of rule clauses.

(iv) The symbol SYNC stands for the formula $\forall x(+\bullet x \leftrightarrow +\circ x)$.

Conventionally, logic programs are typically notated by means of a special syntax with truth value constants $(\top, \bot)$, conjunction (,), disjunction (;), negation as failure (not) and rule forming ($\rightarrow$) as connectives. A rule clause according to (Def. 8.i) is then written as a rule of the form

$$A_1, \ldots, A_m, \mathsf{not}\, B_1, \ldots, \mathsf{not}\, B_n \rightarrow C_1; \ldots; C_o; \mathsf{not}\, D_1; \ldots; \mathsf{not}\, D_p, \qquad \text{(i)}$$

where $m, n, o, p \geq 0$ and the subscripted A, B, C, D are atoms. If $m = n = 0$, then the left side of the rule is $\top$; if $o = p = 0$, then the right side is $\bot$.

[2] Recall that a *clause* as specified in Sect. 2 may contain universally quantified variables. The implicit quantifier prefixes of clauses are not written in this definition.

The following definition specifies a formula $\mathsf{ans}(F)$ whose models are exactly those interpretations that represent an answer set of F according to the stable model semantics.

Definition 9 (Answer Set). For all logic programs F:

$$\mathsf{ans}(F) \stackrel{\text{def}}{=} \mathsf{circ}(F, \mathsf{POS} \cup \hat{\bullet}) \land \mathsf{SYNC}.$$

In the definition of $\mathsf{ans}(F)$, the scope of the circumscription, $(\mathsf{POS} \cup \hat{\bullet})$, is equal to $((\hat{\circ} \cap \mathsf{POS}) \cup \hat{\bullet})$ which matches the right side of Theorem 1, indicating that $\mathsf{ans}(F)$ can also be expressed in terms of second-order circumscription.

We now explicate the relationship of the characterization of stable models by $\mathsf{ans}(F)$ to the characterization in [3, 4], and justify in this way that $\mathsf{ans}(F)$ indeed characterizes stable models. More detailed proofs and relationships to reduct based characterizations of answer sets can be found in [12]. We limit our considerations to logic programs according to Def. 8.iii, which are clausal sentences (the characterization in [3, 4] applies also to nonclausal sentences).

Let F be a logic program. Let $P = p_1, \ldots, p_n$ be the function symbols of the principal terms in F (that is, the predicate symbols if the wrapper predicates $\circ$ and $\bullet$ are dropped). Let $P' = p'_1, \ldots, p'_n$ and $Q = q_1, \ldots, q_n$ be tuples of distinct predicate symbols which are disjoint with each other and with P. We use the following notation to express variants of F that are obtained by replacing predicate symbols:

– We write F also as $F[\circ, \bullet]$, to indicate that $\circ$ and $\bullet$ occur in it.
– The formula $F[U, V]$, where $U = u_1, \ldots, u_n$ and $V = v_1, \ldots, v_n$ are tuples of predicate symbols is $F[\circ, \bullet]$ with all atoms $\circ(p_i(\overline{t}))$ replaced by $u_i(\overline{t})$ and all atoms $\bullet(p_i(\overline{t}))$ replaced by $u_i(\overline{t})$, where $\overline{t}$ matches the respective argument terms. As a special case, $F[P, P]$ is then $F[\circ, \bullet]$ with all atoms of the form $\circ A$ or $\bullet A$ replaced by A.

Let $\mathsf{cnv}(F)$ denote F converted into the syntax of logic programs with non-classical operators used by [3, 4] (see [12] for an explicit such conversion). Let $\mathsf{SM}(\mathsf{cnv}(F))$ be the second-order sentence that characterizes the stable models of $\mathsf{cnv}(F)$ according to [3, 4]. The following equivalence can be verified, where $P' {<} P$ has the same meaning as in Def. 5:

$$\mathsf{SM}(\mathsf{cnv}(F)) \equiv F[P, P] \land \neg\exists P'(F[P', P] \land P' {<} P). \tag{ii}$$

The right side of equivalence (ii) is not a second-order circumscription, since P occurs in $F[P', P]$ as well as in $P' {<} P$. To fit it into the circumscription scheme, we replace the occurrences of P in $F[P', P]$ by Q and add the requirement that P and Q are equivalent: Let $(P \leftrightarrow Q)$ stand for $\bigwedge_{i=1}^{n}(p_i(\overline{x}_i) \leftrightarrow q_i(\overline{x}_i))$, where $\overline{x}_i$ has the same meaning as in Def. 5. The following equivalences then hold:

$$\begin{aligned} & \mathsf{SM}(\mathsf{cnv}(F)) \land (P \leftrightarrow Q) && \text{(iii)} \\ \equiv\ & F[P, Q] \land \neg\exists P'(F[P', Q] \land P' {<} P) \land (P \leftrightarrow Q) && \text{(iv)} \\ \equiv\ & \mathsf{CIRC}[F[P, Q]; P; \emptyset] \land (P \leftrightarrow Q). && \text{(v)} \end{aligned}$$

To get rid of the biconditionals $(P \leftrightarrow Q)$ in (iii), projection can be employed: From $\mathsf{SM}(\mathsf{cnv}(F)) \equiv \mathsf{project}(\mathsf{SM}(\mathsf{cnv}(F)) \land (P \leftrightarrow Q), \hat{P})$ it follows that

$$\mathsf{SM}(\mathsf{cnv}(F)) \equiv \mathsf{project}(\mathsf{CIRC}[F[P,Q];P;\emptyset] \land (P \leftrightarrow Q), \hat{P}). \qquad \text{(vi)}$$

Based on equivalence (vi), the correspondence of $\mathsf{ans}(F)$ to $\mathsf{SM}(\mathsf{cnv}(F))$ can be shown by proving that for two interpretations that are related in a certain way the one is a model of $\mathsf{SM}(\mathsf{cnv}(F))$ if and only if the other is a model of $\mathsf{ans}(F)$: Let I be a structure over P and Q as predicate symbols. Define I' as the structure obtained from I by replacing all atoms $p_i(A)$ with $\circ(p_i(A))$ and all atoms $q_i(A)$ with $\bullet(q_i(A))$. Define I'' as the structure that contains the same literals with predicate $\bullet$ as I' and contains $+\circ(A)$ $(-\circ(A))$ whenever it contains $+\bullet(A)$ $(-\bullet(A))$. Thus the literals with predicate $\circ$ are chosen in I'' such that it satisfies SYNC. The following statements are then equivalent:

$$\begin{aligned}
\langle I, \beta\rangle &\models \mathsf{SM}(\mathsf{cnv}(F)). && \text{(vii)}\\
\langle I, \beta\rangle &\models \mathsf{project}(\mathsf{CIRC}[F[P,Q];P;\emptyset] \land (P \leftrightarrow Q), \hat{P}). && \text{(viii)}\\
\langle I', \beta\rangle &\models \mathsf{project}(\mathsf{CIRC}[F[\circ,\bullet];\circ;\emptyset] \land \mathsf{SYNC}, \hat{\circ}). && \text{(ix)}\\
\langle I', \beta\rangle &\models \mathsf{project}(\mathsf{CIRC}[F;\circ;\emptyset] \land \mathsf{SYNC}, \hat{\circ}). && \text{(x)}\\
\langle I'', \beta\rangle &\models \mathsf{CIRC}[F;\circ;\emptyset] \land \mathsf{SYNC}. && \text{(xi)}\\
\langle I'', \beta\rangle &\models \mathsf{circ}(F, \mathsf{POS} \cup \hat{\bullet}) \land \mathsf{SYNC}. && \text{(xii)}\\
\langle I'', \beta\rangle &\models \mathsf{ans}(F). && \text{(xiii)}
\end{aligned}$$

8 Conclusion

We have introduced an operator raise which can be used to define circumscription in a compact way. The definition of that operator – in a semantic framework where structures are represented by sets of literals – is identical to that of literal projection, except that a set inclusion is replaced by a proper set inclusion.

An approach to an intuitive understanding of the raise operator is to consider minimization as passed through from the "object language level" (the extents of predicates is minimized) to the "meta level" of the semantic framework: Raise expresses that model agreement conditions are minimized. Accordingly, the predicate minimization conditions (commonly abbreviated by $P' < P$ in definitions of circumscription) have not to be made explicit with the raise operator, but are "built-in". In addition, the approach to "minimize model agreement conditions" effects that the raise operator straightforwardly covers certain generalizations of circumscription: Raise has – aside of a formula – just a set of literals as argument, such that, depending on the composition of this set, not only parallel circumscription with varied predicates can be expressed, but also predicate maximization conditions. Moreover, also minimization, maximization and agreement conditions can be expressed that apply only to a subset of the instances of a predicate.

The characterization of circumscription in terms of the raise operator is immediately useful to prove properties of circumscription in a streamlined way. The introduced semantic framework with the project and raise operators is a basis for future research, including the further elaboration of common and differing properties of both operators, the exploration of applications that involve combinations of circumscription and projection, and the investigation of possibilities for transferring and interleaving methods for non-monotonic reasoning, such as computation of stable models, with methods for second-order quantifier elimination and the closely related projection computation.

References

1. P. Doherty, W. Łukaszewicz, and A. Szałas. Computing circumscription revisited: A reduction algorithm. *J. Autom. Reason.*, 18(3):297–338, 1997.
2. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Spektrum Akademischer Verlag, Heidelberg, 4th edition, 1996.
3. P. Ferraris, J. Lee, and V. Lifschitz. A new perspective on stable models. In *IJCAI-07*, pages 372–379, 2007.
4. P. Ferraris, J. Lee, and V. Lifschitz. Stable models and circumscription. 2009. To appear; Draft retrieved on May 17th 2009 from https://www.cs.utexas.edu/users/otto/papers/smcirc.pdf.
5. D. M. Gabbay, R. A. Schmidt, and A. Szałas. *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. CollegePublications, 2008.
6. M. Gelfond, H. Przymusinska, and T. Przymusinski. The extended closed world assumption and its relationship to parallel circumscription. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 133–139, 1986.
7. J. Lang, P. Liberatore, and P. Marquis. Propositional independence – formula-variable independence and forgetting. *JAIR*, 18:391–443, 2003.
8. V. Lifschitz. Circumscription. In *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.
9. T. Przymusinski. An algorithm to compute circumscription. *Artificial Intelligence*, 83:59–73, 1989.
10. C. Wernhard. Literal projection for first-order logic. In *JELIA 08*, pages 389–402, 2008.
11. C. Wernhard. *Automated Deduction for Projection Elimination*. Number 324 in Dissertationen zur Künstlichen Intelligenz (DISKI). AKA/IOS Press, 2009.
12. C. Wernhard. Literal projection and circumscription – extended version. Technical report, Technische Universität Dresden, 2009. Available from http://cs.christophwernhard.com/papers/projection-circumscription.pdf.
13. C. Wernhard. Tableaux for projection computation and knowledge compilation. In *TABLEAUX 2009*, pages 325–340, 2009.

Inductive Reasoning for Shape Invariants

Lilia Georgieva[1] and Patrick Maier[2]

[1] School of Math. and Comp. Sciences, Heriot-Watt University, Edinburgh
`http://www.macs.hw.ac.uk/~lilia/`
[2] LFCS, School of Informatics, University of Edinburgh
`http://homepages.inf.ed.ac.uk/pmaier/`

Abstract. Automatic verification of imperative programs that destructively manipulate heap data structures is challenging. In this paper we propose an approach for verifying that such programs do not corrupt their data structures. We specify heap data structures such as lists, arrays of lists, and trees inductively as solutions of logic programs. We use off-the-shelf first-order theorem provers to reason about these specifications.

1 Introduction

In this paper we show how to reason effectively about pointer programs using automatic first-order theorem provers. Common approaches to such reasoning rely on transitive-closure to express reachability in linked data structures. However, first-order theorem provers cannot handle transitive closure accurately, because there is no finite first-order axiomatisation. Instead, various approximations have been proposed for proving (non-)reachability in linked data structures. Yet, there is no universal scheme for approximating transitive closure — the choice of approximation depends on the data structure and on the type of (non-)reachability problem at hand.

We propose a different approach to reasoning about pointer programs. Instead of reasoning about reachability, we reason about the *extension* of heap data structures, i. e., about sets of heap cells. This is sufficient to express many (non-)reachability problems, e. g., "x is reachable from the head of the list", or "the lists pointed to by x and y are separate".

We define common data structures, including acyclic lists, cyclic lists, sorted lists, and binary trees, as logic programs. More precisely, the logic programs define *shape types*, i. e., monadic predicates capturing the extension (set of heap cells) of the given data structure. These logic programs, if viewed as universal first-order theories, have many models, some of which will contain junk, i. e., unreachable heap cells; see Fig 3, showing a shape type for a singly linked list containing junk. The issue of junk models can be avoided if we confine ourselves to least models, i. e., to inductive reasoning. We approximate this inductive least model reasoning by first-order verification conditions. The main contributions of this paper are the following:

– We present logic programs defining a number of common shape types (Section 2). The programs are carefully chosen to harness the power of automatic resolution-based theorem provers (in our case study `SPASS` [29]) and SMT solvers with heuristic-driven quantifier instantiation (in our case study Yices [6]).

– We describe a methodology to verify that pointer programs maintain shape invariants (Section 3), which may express properties like "a data structure is a sorted doubly-linked list". The method relies on user-provided code annotations and on verification condition generation.

2 Modelling Data Structures

2.1 Logical Heap Model

We work in the framework of *many-sorted first-order logic with equality*, assuming familiarity with the basic syntactic and semantic concepts.

Notation. A *signature* Σ declares finite sets of sorts $\Sigma_{\mathcal{S}}$, function symbols $\Sigma_{\mathcal{F}}$ and relation symbols $\Sigma_{\mathcal{R}}$. Function symbols f and relation symbols R have associated arities, usually written as $R \subseteq T_1 \times \cdots \times T_m$ resp. $f : T_1 \times \cdots \times T_n \rightarrow T_0$ (or $f : T$ if f is a constant). Given signatures Σ and Δ, their union $\Sigma\Delta$ is a signature. We call Δ an *extension* of Σ if $\Delta = \Sigma\Delta$; we call the extension *relational* if additionally $\Delta_{\mathcal{S}} = \Sigma_{\mathcal{S}}$ and $\Delta_{\mathcal{F}} = \Sigma_{\mathcal{F}}$.

A Σ*-algebra* $\mathbf{A}$ interprets sorts $T \in \Sigma_{\mathcal{S}}$ as carriers $T^{\mathbf{A}}$, function symbols $f \in \Sigma_{\mathcal{F}}$ as functions $f^{\mathbf{A}}$, and relation symbols $R \in \Sigma_{\mathcal{R}}$ as relations $R^{\mathbf{A}}$. We call $\mathbf{B}$ a Δ-*extension* (or simply *extension*) of $\mathbf{A}$ if Δ is an extension of Σ and $\mathbf{B}$ is a Δ-algebra whose Σ-reduct $\mathbf{B}|_{\Sigma}$ is $\mathbf{A}$.

First-order formulas over Σ are constructed by the usual logical connectives (including the equality predicate $=$). We write $\mathbf{A}, \alpha \models \phi$ to denote that formula ϕ is true in Σ-algebra $\mathbf{A}$ under variable assignment α; we may drop α if ϕ is closed.

Logical model of program state. We consider programs in a subset of the programming language C. With regard to the heap, these programs may allocate and free records (`struct`s in C terminology) on the heap, and they may dereference and update pointers to these records. However, they may not perform address arithmetic, pointer type casts, or use variant records (`union`s in C terminology). Under these restrictions, any given program state (i. e., heap plus values of program variables) can be viewed a many-sorted Σ-algebra $\mathbf{A}$ in the following way. (i) The C types are viewed as sorts. There are two classes of sorts: *value* sorts corresponding the C base types (`int`, `float`, etc.) and *pointer* sorts corresponding to C record types. (ii) The elements of the carrier $T^{\mathbf{A}}$ of a value sort T are the values of the C type `T`. $\mathbf{A}$ interprets the standard functions (like addition) and relations (like order) on value sorts as intended. (iii) The elements of the carrier $T^{\mathbf{A}}$ of a pointer sort T are the addresses of records of the C type `T` in the given heap, plus the special address NULL_T, which represents `NULL` pointers of type `T`. (iv) A field `f` of type `T'` in a record of type `T` corresponds to a unary function symbol $f : T \rightarrow T'$. Its interpretation $f^{\mathbf{A}}$ maps addresses in $T^{\mathbf{A}}$ to elements of $T'^{\mathbf{A}}$ (i. e., to values or addresses, depending on whether T' is a value or pointer sort). (v) To capture the values of program variables, we extend the signature Σ with constants, one per program variable. For example, the program variable `x` of type `T` is represented by the logical constant x of sort T; the value of `x` is the interpretation $x^{\mathbf{A}}$ of x in $\mathbf{A}$.

```
typedef double D;          // data values

struct L_Node {            // list nodes
  struct L_Node* next;
  struct L_Node* prev;
  D data;
};
typedef struct L_Node* L; // lists

struct T_Node {            // tree nodes
  struct T_Node* left;
  struct T_Node* right;
  struct T_Node* parent;
  D val;
};
typedef struct T_Node* T; // binary trees
```

Value sorts:	D
Pointer sorts:	L, T
Constants:	$\mathit{NULL}_L : L$
	$\mathit{NULL}_T : T$
Functions:	$\mathit{next}, \mathit{prev} : L \to L$
	$\mathit{data} : L \to D$
	$\mathit{left}, \mathit{right}, \mathit{parent} : T \to T$
	$\mathit{val} : T \to D$
Relations:	$\leq\, \subseteq D \times D$

Fig. 1. Data type declaration in C (left) and corresponding signature Σ (right).

Figure 1 shows a sample C data type declaration and the corresponding signature Σ. As an aside, note that the unary functions in our heap model are total, unlike models of separation logic where the heap is represented by partial functions. A Σ-algebra $\mathbf{A}$ may thus contain junk, i. e., unreachable cells pointing to whatever they like. This does not matter as we will restrict our attention to the well-behaved clusters of the heap cells that are cut out by the shape types presented in the next section, and ignore all the rest.[3]

2.2 Shape Types as Logic Programs

Logic programs. Let Σ and Δ be signatures such that $\Sigma\Delta$ is a relational extension of Σ. A clause (over $\Sigma\Delta$) is called *Δ-Horn* (resp. *definite Δ-Horn*) if it contains at most one (resp. exactly one) positive Δ-literal. A *$\langle\Sigma, \Delta\rangle$-LP* is a finite set of Δ-Horn clauses over $\Sigma\Delta$.

Given a Σ-algebra $\mathbf{A}$, we call a $(\Sigma\Delta)$-extension $\mathbf{B}$ of $\mathbf{A}$ an $\mathbf{A}$-*model* of $\mathcal{P}$ if $\mathbf{B} \models \mathcal{P}$; we call $\mathcal{P}$ $\mathbf{A}$-*satisfiable* if it has an $\mathbf{A}$-model. Note that for some $\mathbf{A}$, $\mathcal{P}$ may not be $\mathbf{A}$-satisfiable (because $\mathcal{P}$ may contain non-definite Δ-Horn clauses). However, a standard argument (Proposition 1) shows that if $\mathcal{P}$ is $\mathbf{A}$-satisfiable then it has a least $\mathbf{A}$-model $\mathbf{B_0}$ (in the sense that $R^{\mathbf{B_0}} \subseteq R^{\mathbf{B}}$ for all $R \in \Delta_{\mathcal{R}}$ and all $\mathbf{A}$-models $\mathbf{B}$ of $\mathcal{P}$); we denote $\mathbf{B_0}$ by $\mathrm{lm}(\mathcal{P}, \mathbf{A})$. Thus, $\mathcal{P}$ may be viewed as a transformer taking a Σ-algebra and computing its least $(\Sigma\Delta)$-extension consistent with $\mathcal{P}$ (if it exists).

Proposition 1. *Let $\mathcal{P}$ be a $\langle\Sigma, \Delta\rangle$-LP and $\mathbf{A}$ a Σ-algebra. If $\mathcal{P}$ is $\mathbf{A}$-satisfiable then it has a least $\mathbf{A}$-model.*

Shape types. Informally, a *shape type* is a unary predicate on the heap, characterising the collection of heap cells that form a particular data structure (e. g., a sorted list). Its purpose is twofold: It serves to enforce integrity constraints (like sortedness) on the data structure, and it provides a handle to specify invariants (like separation of two lists).

[3] This is very much what a programmer does, who is also not concerned about the contents of unreachable memory locations.

In the remainder of this section, we will define a number of shape types by $\langle \Sigma, \Delta \rangle$-LPs $\mathcal{P}$, where Σ is the signature of the heap (cf. Section 2.1), which $\Sigma\Delta$ extends with a unary predicate S. Given a heap $\mathbf{A}$, the least model $\mathrm{lm}(\mathcal{P}, \mathbf{A})$ may be viewed as an *annotated heap*, tagging heap cells (as belonging to the interpretation of S) which form the data structure specified by $\mathcal{P}$. We note that the existence of least models will be guaranteed by Proposition 1 because the LPs will be evidently $\mathbf{A}$-satisfiable in all intended heaps $\mathbf{A}$ (i. e., in all $\mathbf{A}$ where the data structure in question is not corrupt).

Shape types of segments of linked lists. Figure 2 presents $\langle \Sigma, \Delta \rangle$-LPs defining the shape types of various list segments (singly- or doubly-linked, sorted or not). The programs are parameterised by input and output signatures Σ and Δ (the latter declaring only the single symbol S).

The simplest LP $\mathcal{P}_{\text{List}}$ defines the shape type S of unsorted singly-linked list segments. The input signature comprises the sort of list cells L, the *next*-pointer function, the pointer p to the head, and the pointer q to the tail of the list beyond the segment. The clauses express (in order of appearance) that (i) the first cell of the tail (pointed to by q) does not belong to S, (ii) the head of the segment belongs to S unless p points to the tail, (iii) no cell in S points to the head, (iv) S is closed under following *next*-pointers up to q, and (v) each cell in S is pointed to by at most one cell in S (i. e., no sharing in S). Figure 3 (top) shows two models of $\mathcal{P}_{\text{List}}$. The first one is the intended least model, where the interpretation of S really is the set of cells forming the list segment from p to q, whereas in the second model the interpretation of S contains some *junk*, i. e., cells that are unreachable from the head.

The LP $\mathcal{P}_{\text{DList}}$ defines the shape type S of doubly-linked list segments from head cell p to last cell r (where the cells ahead of p and behind r are s and q, respectively), see the picture in Figure 3 (middle). The program defines S as both, a *next*-linked list segment from p to q, and a *prev*-linked list segment from r to s. Moreover, it demands that p belongs to S iff r does, and that p and r are the only cells in S that may point to s and q, respectively. Finally, the last two clauses force the *next*-pointers inside S to be converses of the *prev*-pointers, and vice versa.

The LP $\mathcal{P}_{\text{SList}}$ defines the shape type S of sorted, singly-linked list segments. Its parameter list extends that of $\mathcal{P}_{\text{List}}$ by the data sort D, the total ordering $\leq$ on D, and the *data* field. It adds one more clause to $\mathcal{P}_{\text{List}}$, for comparing the data values of adjacent elements in the list segment.

Finally, the LP $\mathcal{P}_{\text{SDList}}$ combines the LPs $\mathcal{P}_{\text{DList}}$ and $\mathcal{P}_{\text{SList}}$, defining the shape type S of sorted, doubly-linked list segments.

Shape types of cyclic lists. The LP $\mathcal{P}_{\text{CList}}$ in Figure 2 defines the shape type S of cyclic singly-linked lists. Its input signature comprises the sort of list cells L, the *next*-pointer function, and the pointer p into the cyclic list. The clauses express (in order of appearance) that: (i) `NULL` does not belong to S, (ii) the cell pointed to by p belongs to S unless p points to `NULL`, (iii) S is closed under following *next*-pointers, and (iv) each cell in S is pointed to by at most one cell in S (i. e., no sharing in S).
The LP $\mathcal{P}_{\text{CDList}}$ defines the shape type S of doubly-linked cyclic lists by extending $\mathcal{P}_{\text{CList}}$ with a clause forcing *next* and *prev* inside S to be converses. We remark that the LPs for

LP for singly-linked list segments

$$\begin{aligned}
&\mathcal{P}_{\text{List}}[L; \mathit{next} : L \to L, p, q : L; S \subseteq L] \\
&\;= \{\neg\, S(q), \\
&\qquad S(p) \vee p = q, \\
&\qquad \forall x{:}L \;.\; S(p) \wedge S(x) \Rightarrow \mathit{next}(x) \neq p, \\
&\qquad \forall x{:}L \;.\; S(x) \wedge \mathit{next}(x) \neq q \Rightarrow S(\mathit{next}(x)), \\
&\qquad \forall x, y, z{:}L \;.\; S(x) \wedge S(y) \wedge S(z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y\}
\end{aligned}$$

LP for doubly-linked list segments

$$\begin{aligned}
&\mathcal{P}_{\text{DList}}[L; \mathit{next}, \mathit{prev} : L \to L, p, q, r, s : L; S \subseteq L] \\
&\;= \mathcal{P}_{\text{List}}[L; \mathit{next}, p, q; S] \cup \mathcal{P}_{\text{List}}[L; \mathit{prev}, r, s; S] \\
&\;\cup \{S(r) \Rightarrow S(p),\;\; S(p) \Rightarrow S(r), \\
&\qquad \forall x{:}L \;.\; S(x) \wedge \neg\, S(\mathit{prev}(x)) \Rightarrow x = p \wedge \mathit{prev}(x) = s, \\
&\qquad \forall x{:}L \;.\; S(x) \wedge \neg\, S(\mathit{next}(x)) \Rightarrow x = r \wedge \mathit{next}(x) = q, \\
&\qquad \forall x{:}L \;.\; S(x) \wedge S(\mathit{next}(x)) \Rightarrow \mathit{prev}(\mathit{next}(x)) = x, \\
&\qquad \forall y{:}L \;.\; S(y) \wedge S(\mathit{prev}(y)) \Rightarrow \mathit{next}(\mathit{prev}(y)) = y\}
\end{aligned}$$

LP for singly-linked sorted list segments

$$\begin{aligned}
&\mathcal{P}_{\text{SList}}[D, L; \mathit{next} : L \to L, \mathit{data} : L \to D, p, q : L, \leq\; \subseteq D \times D; S \subseteq L] \\
&\;= \mathcal{P}_{\text{List}}[L; \mathit{next}, p, q; S] \\
&\;\cup \{\forall x{:}L \;.\; S(x) \wedge S(\mathit{next}(x)) \Rightarrow \mathit{data}(x) \leq \mathit{data}(\mathit{next}(x))\}
\end{aligned}$$

LP for doubly-linked sorted list segments

$$\begin{aligned}
&\mathcal{P}_{\text{SDList}}[D, L; \mathit{next}, \mathit{prev} : L \to L, \mathit{data} : L \to D, p, q, r, s : L, \leq\; \subseteq D \times D; S \subseteq L] \\
&\;= \mathcal{P}_{\text{DList}}[L; \mathit{next}, \mathit{prev}, p, q, r, s; S] \cup \mathcal{P}_{\text{SList}}[D, L; \mathit{next}, \mathit{data}, p, q, \leq; S]
\end{aligned}$$

LP for singly-linked cyclic lists

$$\begin{aligned}
&\mathcal{P}_{\text{CList}}[L; \mathit{next} : L \to L, p, \mathit{NULL}_L : L; S \subseteq L] \\
&\;= \{\neg\, S(\mathit{NULL}_L), \\
&\qquad S(p) \vee p = \mathit{NULL}_L, \\
&\qquad \forall x{:}L \;.\; S(x) \Rightarrow S(\mathit{next}(x)), \\
&\qquad \forall x, y, z{:}L \;.\; S(x) \wedge S(y) \wedge S(z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y\}
\end{aligned}$$

LP for doubly-linked cyclic lists

$$\begin{aligned}
&\mathcal{P}_{\text{CDList}}[L; \mathit{next}, \mathit{prev} : L \to L, p, \mathit{NULL}_L : L; S \subseteq L] \\
&\;= \mathcal{P}_{\text{CList}}[L; \mathit{next}, p, \mathit{NULL}_L; S] \cup \{\forall x{:}L \;.\; S(x) \Rightarrow \mathit{prev}(\mathit{next}(x)) = x \wedge \mathit{next}(\mathit{prev}(x)) = x\}
\end{aligned}$$

LP for arrays of singly-linked NULL-terminated lists

$$\begin{aligned}
&\mathcal{P}_{\text{ListArray}}[I, L; \mathit{next} : L \to L, a : I \to L, \mathit{NULL}_L : L; S \subseteq I \times L] \\
&\;= \{\forall i{:}I \;.\; \neg\, S(i, \mathit{NULL}_L), \\
&\qquad \forall i{:}I \;.\; S(i, a(i)) \vee a(i) = \mathit{NULL}_L, \\
&\qquad \forall i{:}I\; \forall x{:}L \;.\; S(i, a(i)) \wedge S(i, x) \Rightarrow \mathit{next}(x) \neq a(i), \\
&\qquad \forall i{:}I\; \forall x{:}L \;.\; S(i, x) \wedge \mathit{next}(x) \neq \mathit{NULL}_L \Rightarrow S(i, \mathit{next}(x)), \\
&\qquad \forall i{:}I\; \forall x, y, z{:}L \;.\; S(i, x) \wedge S(i, y) \wedge S(i, z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y, \\
&\qquad \forall i, j{:}I\; \forall x{:}L \;.\; S(i, x) \wedge S(j, x) \Rightarrow i = j\}
\end{aligned}$$

LP for arrays of singly-linked cyclic lists

$$\begin{aligned}
&\mathcal{P}_{\text{CListArray}}[I, L; \mathit{next} : L \to L, a : I \to L, \mathit{NULL}_L : L; S \subseteq I \times L] \\
&\;= \{\forall i{:}I \;.\; \neg\, S(i, \mathit{NULL}_L), \\
&\qquad \forall i{:}I \;.\; S(i, a(i)) \vee a(i) = \mathit{NULL}_L, \\
&\qquad \forall i{:}I\; \forall x{:}L \;.\; S(i, x) \Rightarrow S(i, \mathit{next}(x)), \\
&\qquad \forall i{:}I\; \forall x, y, z{:}L \;.\; S(i, x) \wedge S(i, y) \wedge S(i, z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y, \\
&\qquad \forall i, j{:}I\; \forall x{:}L \;.\; S(i, x) \wedge S(j, x) \Rightarrow i = j\}
\end{aligned}$$

LP for binary trees

$$\begin{aligned}
&\mathcal{P}_{\text{Tree}}[T; \mathit{left}, \mathit{right} : T \to T, r, \mathit{NULL}_T : T; S \subseteq T] \\
&\;= \{\neg\, S(\mathit{NULL}_T), \\
&\qquad S(r) \vee r = \mathit{NULL}_T, \\
&\qquad \forall x{:}T \;.\; S(r) \wedge S(x) \Rightarrow (\mathit{left}(x) \neq r \wedge \mathit{right}(x) \neq r), \\
&\qquad \forall x{:}T \;.\; S(x) \wedge \mathit{left}(x) \neq \mathit{NULL}_T \Rightarrow S(\mathit{left}(x)), \\
&\qquad \forall x{:}T \;.\; S(x) \wedge \mathit{right}(x) \neq \mathit{NULL}_T \Rightarrow S(\mathit{right}(x)), \\
&\qquad \forall x, y, z{:}T \;.\; S(x) \wedge S(y) \wedge S(z) \wedge \mathit{left}(x) = z \wedge \mathit{left}(y) = z \Rightarrow x = y, \\
&\qquad \forall x, y, z{:}T \;.\; S(x) \wedge S(y) \wedge S(z) \wedge \mathit{right}(x) = z \wedge \mathit{right}(y) = z \Rightarrow x = y, \\
&\qquad \forall x, y, z{:}T \;.\; S(x) \wedge S(y) \wedge S(z) \wedge \mathit{left}(x) = z \wedge \mathit{right}(y) = z \Rightarrow x = y, \\
&\qquad \forall x, y, z{:}T \;.\; S(x) \wedge S(y) \wedge S(z) \wedge \mathit{left}(x) = y \wedge \mathit{right}(x) = z \Rightarrow y \neq z\}
\end{aligned}$$

LP for binary trees with parent pointers

$$\begin{aligned}
&\mathcal{P}_{\text{PTree}}[T; \mathit{left}, \mathit{right}, \mathit{parent} : T \to T, r, s, \mathit{NULL}_T : T; S \subseteq T] \\
&\;= \mathcal{P}_{\text{Tree}}[T; \mathit{left}, \mathit{right}, r, \mathit{NULL}_T; S] \\
&\;\cup \{s \neq \mathit{NULL}_T \Rightarrow S(r), \\
&\qquad S(r) \Rightarrow s = \mathit{parent}(r), \\
&\qquad \neg\, S(s), \\
&\qquad \forall x{:}T \;.\; S(x) \wedge S(\mathit{left}(x)) \Rightarrow \mathit{parent}(\mathit{left}(x)) = x, \\
&\qquad \forall x{:}T \;.\; S(x) \wedge S(\mathit{right}(x)) \Rightarrow \mathit{parent}(\mathit{right}(x)) = x, \\
&\qquad \forall y{:}T \;.\; S(y) \wedge S(\mathit{parent}(y)) \Rightarrow (\mathit{left}(\mathit{parent}(y)) = y \vee \mathit{right}(\mathit{parent}(y)) = y)\}
\end{aligned}$$

Fig. 2. LPs defining shape types of list segments, cyclic lists, arrays of lists, binary trees.

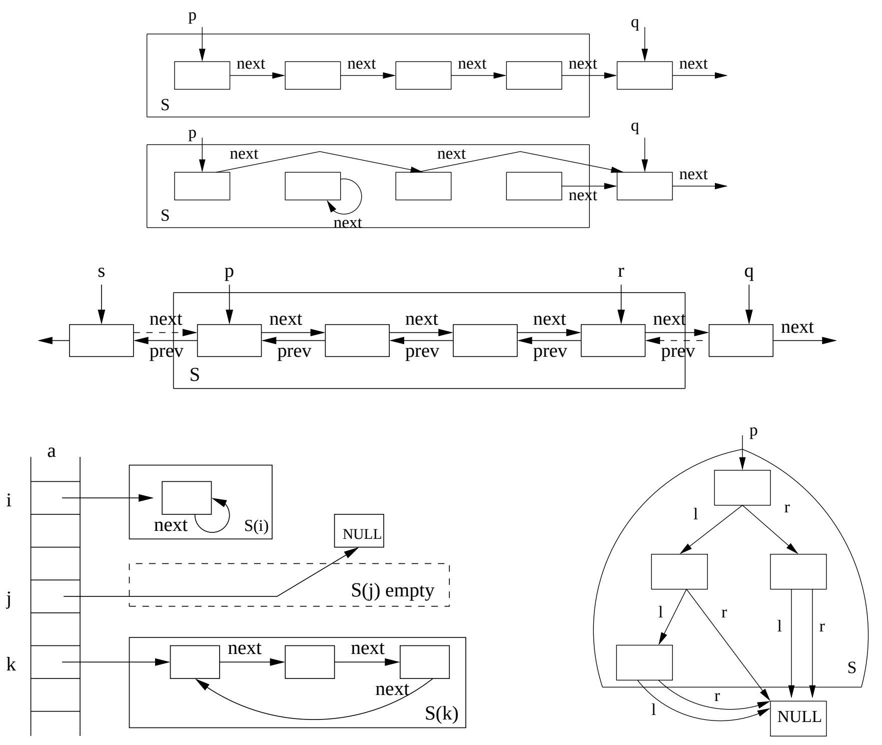


Fig. 3. From top to bottom: shape types S of singly-linked list segments (intended least model and model with unreachable junk), doubly-linked list segments, array of cyclic lists, and binary trees.

cyclic lists are more elegant than the corresponding LPs for singly- and doubly-linked list segments.

Shape types of arrays of lists. The LP $\mathcal{P}_{\text{CListArray}}$ in Figure 2 defines the shape type S of arrays of singly-linked cyclic lists, see Figure 3 (bottom left) for a graphical depiction. The input signature comprises the array index sort I, the list cell sort L, the *next*-pointer function, and the function a mapping array indices to pointers into the lists. The shape type S is a binary relation between array indices and list cells. Note that we model arrays as functions from index type to element type[4], ignoring array bounds. In the light of this, the shape type S may be viewed as an array of sets of list cells rather than as a binary relation.

The first four clauses of $\mathcal{P}_{\text{CListArray}}$ state that for each index i, the unary relation $S(i, \cdot)$ — S with fixed first argument i — is the shape type of a cyclic singly-linked list pointed to by $a(i)$; note how these four clauses correspond to the clauses of $\mathcal{P}_{\text{CList}}$.

[4] Our model assumes that arrays do not live in the heap.

The last clause states that the shape types $S(i,\cdot)$ and $S(j,\cdot)$ must be disjoint for distinct indices i and j.

The LP $\mathcal{P}_{\text{ListArray}}$ defines the shape type S of arrays of singly-linked `NULL`-terminated lists in a similar way. Its first five clauses force each shape type $S(i,\cdot)$ to be a singly-linked list segment from $a(i)$ to NULL_L, and the last clause forces disjointness of distinct shape types $S(i,\cdot)$ and $S(j,\cdot)$.

Shape types of binary trees. The LP $\mathcal{P}_{\text{Tree}}$ defines the shape type S of plain binary trees, see the picture in Figure 3 (bottom right). The input signature comprises the sort of tree nodes T, the *left*- and *right*-pointer functions and the pointer r to the root. The clauses borrow heavily from the LP $\mathcal{P}_{\text{List}}$ for singly-linked list segments (with q replaced by NULL_T) and express that: (1) `NULL` does not belong to S, (2) the root belongs to S unless r points to `NULL`, (3) no node in S points to the root, (4-5) S is closed under following *left*- and *right*-pointers up to `NULL`, and (6-9) there is no sharing in S because each node in S is pointed to by at most one node in S (clauses 6-8) and has distinct *left*- and *right*-successors (clause 9).

The LP $\mathcal{P}_{\text{PTree}}$ defines the shape type S of binary trees with parent pointers by extending $\mathcal{P}_{\text{Tree}}$. The additional clauses express that (1-2) s is the parent of the root r unless r is not in S, in which case s must be `NULL`, (3) s does not belong to S, and (4-6) the *parent*-pointers are converse to the union of the *left*- and *right*-pointers.

3 Verifying Pointer Programs

We aim to verify imperative programs that manipulate dynamic data structures on the heap. Given the code of a C function plus specifications of its input and output (and possibly of loop invariants), we want to verify that the program maintains certain *shape invariants*, e. g., that the sorted list being updated remains a list and sorted.

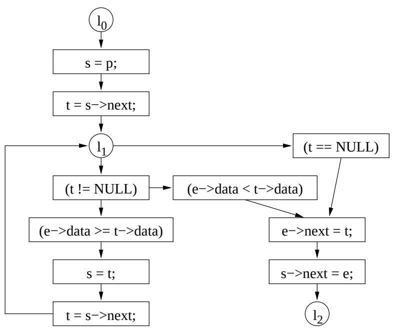
Notation. Given a signature Σ, we define the signature Σ' as a copy of Σ where all functions f (except the constants NULL_T) and relations R are replaced by *primed* functions f' resp. relations R'. Given a Σ-formula ϕ (resp. a $\langle\Sigma,\Delta\rangle$-LP $\mathcal{P}$), we write ϕ' (resp. $\mathcal{P}'$) for the Σ-formula (resp. $\langle\Sigma',\Delta'\rangle$-LP) that arises from ϕ (resp. $\mathcal{P}$) by replacing all functions f (except NULL_T) and relations R by f' and R', respectively.

3.1 Verification Problem

We verify C functions by checking verification conditions. To do this, we convert the code of a function into a control flow graph (CFG) and find a set of cut locations, to which we attach shape invariants. Each path between cut locations gives rise to a verification condition (VC), which claims that the path establishes the invariant at its end location, given that the invariant at the start location was assumed.

We will not elaborate on the well-known techniques for translating C code to CFGs and identifying cut locations; the reader may consult Figure 4 for an example. The figure shows the code for inserting an element `e` into a non-empty sorted list pointed to by `p`. The CFG has three cut locations l_0 (the entry location) to l_2 (the exit location), with four paths σ_1 to σ_4 between them.

```
void insert(L p, L e)
{
  L s = p;
  L t = s->next;
  while (t != NULL) {
    if (e->data >= t->data) {
      s = t;
      t = s->next;
    }
    else break;
  }
  e->next = t;
  s->next = e;
}
```



path	path formula π
$\sigma_1 : l_0 \to l_1 =$ `s = p;` `t = s->next;`	$s' = p \wedge$ $t' = \mathit{next}(s') \wedge$ $p' = p \wedge e' = e \wedge \mathit{next}' = \mathit{next} \wedge \mathit{data}' = \mathit{data}$

$\sigma_2 : l_1 \to l_1 =$ `(t != NULL)`
`(e->data >= t->data)`
`s = t;`
`t = s->next;`
	$t \neq \mathit{NULL}_L \wedge$
	$\mathit{data}(e) \geq \mathit{data}(t) \wedge$
	$s' = t \wedge$
	$t' = \mathit{next}(s') \wedge$
	$p' = p \wedge e' = e \wedge \mathit{next}' = \mathit{next} \wedge \mathit{data}' = \mathit{data}$

$\sigma_3 : l_1 \to l_2 =$ `(t != NULL)`
`(e->data < t->data)`
`e->next = t;`
`s->next = e;`
	$\exists \mathit{next}_1 {:} L \to L \;.$
	$t \neq \mathit{NULL}_L \wedge$
	$\mathit{data}(e) < \mathit{data}(t) \wedge$
	$\mathit{next}_1(e) = t \wedge (\forall x{:}L \;.\; x = e \vee \mathit{next}_1(x) = \mathit{next}(x)) \wedge$
	$\mathit{next}'(s) = e \wedge (\forall x{:}L \;.\; x = s \vee \mathit{next}'(x) = \mathit{next}_1(x)) \wedge$
	$p' = p \wedge e' = e \wedge s' = s \wedge t' = t \wedge \mathit{data}' = \mathit{data}$

$\sigma_4 : l_1 \to l_2 =$ `(t == NULL)`
`e->next = t;`
`s->next = e;`
	$\exists \mathit{next}_1 {:} L \to L \;.$
	$t = \mathit{NULL}_L \wedge$
	$\mathit{next}_1(e) = t \wedge (\forall x{:}L \;.\; x = e \vee \mathit{next}_1(x) = \mathit{next}(x)) \wedge$
	$\mathit{next}'(s) = e \wedge (\forall x{:}L \;.\; x = s \vee \mathit{next}'(x) = \mathit{next}_1(x)) \wedge$
	$p' = p \wedge e' = e \wedge s' = s \wedge t' = t \wedge \mathit{data}' = \mathit{data}$

loc.	formula ϕ of shape invariant $\langle \mathcal{P}, \phi \rangle$ — see Section 3.1 for LP $\mathcal{P}$
l_0	$S \cap E = \emptyset \wedge S(p) \wedge E(e) \wedge \mathit{next}(e) = \mathit{NULL}_L \wedge \mathit{data}(p) \leq \mathit{data}(e)$
l_1	$S = S_0 \wedge E = E_0 \wedge p = p_0 \wedge e = e_0 \wedge$ $S \cap E = \emptyset \wedge S(p) \wedge E(e) \wedge \mathit{next}(e) = \mathit{NULL}_L \wedge \mathit{data}(p) \leq \mathit{data}(e) \wedge$ $S(s) \wedge \mathit{data}(s) \leq \mathit{data}(e) \wedge \mathit{next}(s) = t \wedge (t = \mathit{NULL}_L \vee S(t))$
l_2	$S = S_0 \uplus E_0 \wedge p = p_0 \wedge e = e_0$

path	shape effect ε
σ_1	$S' = S \wedge E' = E$
σ_2	$S' = S \wedge E' = E$
σ_3	$S' = S \uplus E$
σ_4	$S' = S \uplus E$

Fig. 4. Insert an element into a non-empty list sorted in ascending order: C code, control flow graph, paths through the CFG, shape invariants, and shape effects.

State signature. Associated with a C function is a *state signature* Σ, the signature of the Σ-algebras serving as logical models of program state, see Section 2.1. In the following, Σ always refers to a fixed state signature.

In the example of Figure 4, Σ declares the value sort D and the pointer sort L, the constants $p, e, s, t, \mathit{NULL}_L : L$, the functions *data* $: L \to D$ and *next* $: L \to L$ and the order relation $\leq \subseteq D \times D$.

Path formulas. A path σ through the CFG is a sequence consisting of variable assignments `x = e;` array updates `a[i] = e;` heap updates `x->f = e;` and conditions `(c)` where `e` is an expression (an R-value in C terminology) and `c` is a conditional expression.[5] The translation of such paths to first-order logic is well-known and will not be detailed here; the reader is referred to Figure 4 for examples. The *path formula* π resulting from translation of a path σ is a $(\Sigma\Sigma')$-formula, where the signatures Σ and Σ' belong to the state at the start and end locations of σ, respectively. Two things to note. First, part of each path formula is an explicit "frame condition" stating which variables and pointer fields do not change; note the use of second-order equalities like $\mathit{next}' = \mathit{next}$ as short-hands for more complex first-order expressions. Second, path formulae, like the ones for paths σ_3 and σ_4, may start with a string of second-order $\exists$-quantifiers to project away intermediate state; these quantifiers will always be eliminable by Skolemisation.

Shape invariants. A *Δ-shape invariant* is a pair $\langle \mathcal{P}, \phi \rangle$, where $\mathcal{P}$ is a $\langle \Sigma, \Delta \rangle$-LP and ϕ is a $(\Sigma\Delta)$-formula. Its purpose is to constrain the program state by the relating shape types, which are defined by the LP $\mathcal{P}$, with each other or with program variables.

Shape invariants are associated with cut locations in the CFG. Figure 4 presents the shape invariants for the `insert` function. These shape invariants involve two shape types S and E defined by the LP $\mathcal{P} = \mathcal{P}_{\text{SList}}[D, L; \mathit{next}, \mathit{data}, p, \mathit{NULL}_L, \leq; S] \cup \mathcal{P}_{\text{List}}[L; \mathit{next}, e, \mathit{NULL}_L, \leq; E] \cup \{\forall x{:}L \,.\, \neg\, S(x) \vee \neg\, E(x)\}$. I. e., S is the shape type of `NULL`-terminated sorted lists pointed to by p, E of `NULL`-terminated lists pointed to by e, and both shape types are disjoint. Note that the LP $\mathcal{P}$ is common to all three invariants. The shape invariant at l_0, for instance, stipulates that the lists S and E are disjoint and non-empty (because they contain their heads p and e), E is of length 1 (because the `next`-pointer of its head e is `NULL`), and the data at p is less than or equal to the data at e. Note the use of set-relational expressions like $S \cap E = \emptyset$ as short-hand for more complex first-order expressions. The shape invariant at l_2 stipulates that the list S is the sum of the start lists S_0 and E_0[6] and that the program variables p and e retain their start values p_0 and e_0, respectively. This, together with sortedness of S, which is enforced by the LP defining S, is a statement of functional correctness of `insert`.

Verification condition. Given a path σ from ℓ to k, let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, and $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k. To prove correctness of σ, we must show that every execution establishes the *post* shape invariant $\langle \mathcal{Q}, \psi \rangle$ at the end, provided that the *pre* shape invariant $\langle \mathcal{P}, \phi \rangle$ held at the start. This translates to the following verification condition.

$$\forall (\Sigma\Sigma')\text{-algebra } \mathbf{A} : \mathrm{lm}(\mathcal{P}, \mathbf{A}) \models \pi \wedge \phi \implies \mathrm{lm}(\mathcal{Q}', \mathbf{A}) \models \psi' \qquad \text{(VC)}$$

Note that the antecedent of (VC) tacitly depends on the existence of $\mathrm{lm}(\mathcal{P}, \mathbf{A})$, and the succedent tacitly states that the existence of $\mathrm{lm}(\mathcal{Q}', \mathbf{A})$ follows from the antecedent.

[5] To keep the presentation simple, we ignore dynamic memory allocation and function calls. Both could be handled: memory allocation through tracking the set of allocated heap cells, function calls through extra cut locations before and after call sites.

[6] The use of subscript 0 indicates values of program variables or shape types at the initial location l_0. Strictly speaking, a shape invariant is not just constraining the program state at location ℓ, but the relation between the initial state and the state at location ℓ.

The trouble with (VC) is that it requires reasoning in least models, i. e., inductive reasoning. The next section presents our methodology to rephrase the inductive condition (VC) in first-order logic.

3.2 Approximating Inductive Reasoning

The obvious problem with using first-order provers for reasoning about shape types is their ignorance of least models. For instance, a VC on a path σ may be invalid in first-order logic because there is a counter model which picks the least interpretation for shape type S at the start of σ but the greatest interpretation for S at the end.

To deal with this problem, we weaken the VC by speculatively assuming a *shape effect* relating shape types at the start and end of σ. Often, the weakened VC becomes provable in first-order logic. However, we still have to justify the assumed shape effect. We do so by proving two further first-order VCs, which together imply that the shape effect is an inductive consequence of the LPs defining the shape types.

Shape effects. Given a path σ from ℓ to k, let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the $\varDelta$-shape invariant at ℓ, and $\langle \mathcal{Q}, \psi \rangle$ the $\varLambda$-shape invariant at k. A *shape effect* for σ is a $(\varSigma\varDelta\varSigma'\varLambda')$-formula ε which is *back-and-forth total*, that is,

– $\forall(\varSigma\varDelta\varSigma')$-algebra $\mathbf{A}$: $\mathbf{A} \models \mathcal{P} \cup \{\pi\} \implies \exists(\varSigma\varDelta\varSigma'\varLambda')$-extension $\mathbf{B}$: $\mathbf{B} \models \varepsilon$, and
– $\forall(\varSigma\varSigma'\varLambda')$-algebra $\mathbf{C}$: $\mathbf{C} \models \mathcal{Q}' \cup \{\pi\} \implies \exists(\varSigma\varDelta\varSigma'\varLambda')$-extension $\mathbf{D}$: $\mathbf{D} \models \varepsilon$.

The purpose of a shape effect ε is to relate the shape types at the start of path σ with those at the end. A convenient way to specify simple shape effects is to write set-relational expressions like $S' = S \uplus E$ (cf. Figure 4) as short-hands for more complex quantified expressions. This style also makes it easy to check the totality requirement. For example, back-and-forth totality of the shape effect $S' = S \uplus E$ for σ_3 holds because (1) every $(\varSigma\varDelta\varSigma')$-algebra which interprets S and E disjointly (which is enforced by the LP $\mathcal{P}$) has a $(\varSigma\varDelta\varSigma'\varLambda')$-extension which interprets S' as the sum of S and E, and (2) every $(\varSigma\varSigma'\varLambda')$-algebra (which interprets S') has a $(\varSigma\varDelta\varSigma'\varLambda')$-extension which interprets S and E disjointly such that their sum is S'. Note that in this particular case, totality is independent of the path formula π.

Notation. Given a $\varSigma$-algebra $\mathbf{A}$, one may need to compare one $\mathbf{A}$-model of the $\langle \varSigma, \varDelta \rangle$-LP $\mathcal{P}$ to another one. Logically, this can be done by fusing the two models, which requires duplicating all symbols that are not shared, i. e., all relations in $\varDelta$.

We define the signature $\hat{\varDelta}$ as a copy of $\varDelta$ where all relations R are replaced by *capped* relations $\hat{R}$. Given a $\varDelta$-shape invariant $\langle \mathcal{P}, \phi \rangle$, we write $\langle \hat{\mathcal{P}}, \hat{\phi} \rangle$ for the $\hat{\varDelta}$-shape invariant that arises from $\langle \mathcal{P}, \phi \rangle$ by replacing all relations R in $\varDelta$ with $\hat{R}$. Given a shape effect ε (as defined above) for a path σ, we write $\hat{\varepsilon}$ for the $(\varSigma\hat{\varDelta}\varSigma'\hat{\varLambda}')$-formula ε that arises from ε by replacing all relations R in $\varDelta\varLambda$ with R'; note that $\hat{\varepsilon}$ is also a shape effect for σ.

Verification conditions. Given a path σ from ℓ to k, let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k, and ε the shape effect for σ. To prove the inductive condition (VC) it suffices to prove the following three first-order conditions.

$$\mathcal{P} \cup \mathcal{Q}' \cup \{\pi, \varepsilon, \phi\} \models \psi' \tag{VC1}$$

$$\mathcal{P} \cup \{\pi, \varepsilon, \phi\} \models \mathcal{Q}' \tag{VC2}$$

$$\begin{gathered} \mathcal{Q}' \cup \hat{\mathcal{Q}}' \cup \{\textstyle\bigwedge_{S \in \Lambda_\mathcal{R}} \hat{S}' \subseteq S', \bigvee_{S \in \Lambda_\mathcal{R}} \hat{S}' \neq S', \pi, \varepsilon, \hat{\varepsilon}\} \models \\ \mathcal{P} \cup \hat{\mathcal{P}} \cup \{\textstyle\bigwedge_{S \in \Delta_\mathcal{R}} \hat{S} \subseteq S, \bigvee_{S \in \Delta_\mathcal{R}} \hat{S} \neq S\} \end{gathered} \tag{VC3}$$

(VC1) and (VC2) together state preservation of shape invariants, subject to the (yet unjustified) assumption of a shape effect ε. (VC2) can be read as a model transformation: Given any model of $\mathcal{P}$ that satisfies the path formula and the pre shape invariant, the shape effect ε will produce models of $\mathcal{Q}'$. Finally, (VC3) implies that ε preserves minimal models. It can be seen as a reverse model transformation which preserves order: Given any two models of $\mathcal{Q}'$ such that both satisfy the path formula and one is strictly contained in the other, the shape effects ε and $\hat{\varepsilon}$ will produce two models of $\mathcal{P}$ such that one is strictly contained in the other.

Soundness. The following theorem proves that the conditions (VC1) – (VC3) together imply that a shape effect is an inductive consequence (i. e., entailed in the least model) of the LPs defining the shape types. Soundness of the first-order verification conditions w. r. t. to the inductive condition (VC) is an easy corollary.

Theorem 2. *Let σ be a path from ℓ to k. Let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k, and ε the shape effect for σ. Assume that* (VC2) *and* (VC3) *hold. For all $(\Sigma\Sigma')$-algebras $\mathbf{A}$, if lm$(\mathcal{P}, \mathbf{A})$ exists and lm$(\mathcal{P}, \mathbf{A}) \models \pi \wedge \phi$ then lm$(\mathcal{P} \cup \mathcal{Q}', \mathbf{A})$ exists and lm$(\mathcal{P} \cup \mathcal{Q}', \mathbf{A}) \models \varepsilon$.*

Proof. Towards a contradiction assume there is a $(\Sigma\Sigma')$-algebra $\mathbf{A}$ such that $\mathrm{lm}(\mathcal{P}, \mathbf{A})$ and $\mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A})$ exist and $\mathrm{lm}(\mathcal{P}, \mathbf{A}) \models \pi \wedge \phi$, but $\mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A}) \not\models \varepsilon$. As ε is total, the $(\Sigma\Delta\Sigma')$-model $\mathrm{lm}(\mathcal{P}, \mathbf{A})$ of π extends to a $(\Sigma\Delta\Sigma'\Lambda')$-model $\mathbf{B}$ of ε. Thus, $\mathbf{B} \models \mathcal{P} \cup \{\pi, \varepsilon, \phi\}$, which by (VC2) implies $\mathbf{B} \models \mathcal{Q}'$. Note that $\mathbf{B}$ is not an extension of the $(\Sigma\Sigma'\Lambda')$-model $\mathrm{lm}(\mathcal{Q}', \mathbf{A})$, for if it were then $\mathbf{B} = \mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A})$ and hence $\mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A}) \models \varepsilon$, which would contradict our assumption. Thus $\mathbf{B}|_{\Sigma\Sigma'\Lambda'}$ is a non-least, hence non-minimal, $\mathbf{A}$-model of $\mathcal{Q}'$, which implies that $\mathbf{B}$ has a $(\Sigma\Delta\Sigma'\Lambda'\hat{\Lambda}')$-extension $\mathbf{C}$ such that $\mathbf{C} \models \mathcal{Q}' \cup \hat{\mathcal{Q}}' \cup \{\bigwedge_{S \in \Lambda_\mathcal{R}} \hat{S}' \subseteq S', \bigvee_{S \in \Lambda_\mathcal{R}} \hat{S}' \neq S'\}$. As the shape effect $\hat{\varepsilon}$ is total, the $(\Sigma\Sigma'\hat{\Lambda}')$-model $\mathbf{C}|_{\Sigma\Sigma'\hat{\Lambda}'}$ of π extends to a $(\Sigma\hat{\Delta}\Sigma'\hat{\Lambda}')$-model $\mathbf{D}$ of $\hat{\varepsilon}$. Hence, the $(\Sigma\Delta\hat{\Delta}\Sigma'\Lambda'\hat{\Lambda}')$-algebra $\mathbf{E}$, which extends both $\mathbf{C}$ and $\mathbf{D}$, is a model of $\mathcal{Q}' \cup \hat{\mathcal{Q}}' \cup \{\bigwedge_{S \in \Lambda_\mathcal{R}} \hat{S}' \subseteq S', \bigvee_{S \in \Lambda_\mathcal{R}} \hat{S}' \neq S', \pi, \varepsilon, \hat{\varepsilon}\}$. By (VC3), this implies $\mathbf{E} \models \mathcal{P} \cup \hat{\mathcal{P}} \cup \{\bigwedge_{S \in \Delta_\mathcal{R}} \hat{S} \subseteq S, \bigvee_{S \in \Delta_\mathcal{R}} \hat{S} \neq S\}$, i. e., $\mathbf{E}$ is an extension of a non-minimal $\mathbf{A}$-model of $\mathcal{P}$, contradictory to $\mathbf{E}|_{\Sigma\Delta\Sigma'} = \mathbf{C}|_{\Sigma\Delta\Sigma'} = \mathbf{B}|_{\Sigma\Delta\Sigma'} = \mathrm{lm}(\mathcal{P}, \mathbf{A})$. □

Corollary 3. *If* (VC1) – (VC3) *hold then* (VC) *holds.*

3.3 Experiments

We have used our methodology to successfully verify a number of simple heap-manipulating algorithms, including the sorted list insert function from Figure 4. Other examples include functions for inserting and deleting elements into binary trees, and functions for moving elements between ring buffers organised in an array. All functions were manually annotated with shape invariants and effects. Due to lack of space, we do not report on these experiments in detail.

We have run our experiments on the theorem provers SPASS and Yices. Both provers succeeded to prove all verification conditions. The typical run-time per VC was below 10 seconds for SPASS, below 1 second for Yices. More experiments are necessary to determine whether our methodology scales to more complex code.

We remark on the somewhat surprising fact that Yices succeeded on all VCs, despite its incomplete heuristics for quantifier instantiation (which we did not assist using the trigger mechanism). We suspect that a key reason for this is our choice of defining shape types by logic programs, which to a first-order theorem prover are just universally quantified clauses; avoiding existential quantifiers seems to suit Yices well. However, we did observe cases where Yices' instantiation heuristic was sensitive to the formulation of particular clauses (especially the no-sharing clauses for binary trees).

4 Related Work

Efficient theorem provers make first-order logic attractive framework for studying reachability in mutable linked data structures. However, transitive closure, essential for properties of pointer structures presents a challenge because first-order theorem provers cannot handle transitive closure.

Various approaches for program analysis that use first-order logic have been investigated. We next discuss the most prominent.

The logic of interpreted sets and bounded quantification is used for specifying properties of heap manipulating programs [18]. The logic uses first-order logic and is interpreted over a finite partially-ordered set of sorts. It provides a ternary reachability predicate and allows bounded universal quantification over two different kinds of (potentially unbounded) sets. Following this approach first-order SMT solvers, augmented with theories, are used for precise verification of heap-manipulating programs. An alternative framework uses ground logic enriched with ternary predicate [22].

The use of a decidable fragment of first-order logic augmented with arithmetic on scalar field to specify properties of data structures is studied in [20]. In contrast to ours, this approach does not use theories for recursive predicates like reachability, and relies on user provided ghost variables to express properties of data structures.

In [19] a first-order formula, in which transitive closure occurs is simulated by a first-order formula, where transitive closure is encoded by adding a new relation symbol for each binary predicate. This together with inductively defined first-order axioms assures that transitive closure is interpreted correctly. A set of axioms defines the properties of transitive closure inductively. The axioms are not complete over infinite models. If the axioms are such that every finite, acyclic model satisfying them must interpret the encoding of transitive closure as the reflexive, transitive closure of its interpretation of the transitive binary relation, then the axiom schema is complete [3]. Its incompleteness notwithstanding, the induction schema in [19] allows for automatically proving properties of simple programs using SPASS [29].

Alternative approach for symbolic shape analysis [30] uses the framework of (extensions of) decidable fragments of first-order logic e. g., guarded fixpoint logic [10]. The logic expresses reachability along paths and from a specific point, but not reachability between a pair of program variables [12].

Syntactically defined logics for shape analysis, such as local shape logic $\mathcal{LSL}$ [26] and role logic [15], are closely related to first-order logic. Our approach is applicable to their translation in first-order logic. The logic $\mathcal{LSL}$ [26] is strictly less expressive than the two variable fragment of first order logic with counting. Role logic [15] is variable free logic, which is equally expressive as first-order logic with transitive closure and consequently undecidable. A decidable fragment of role logic is as expressive as the two variable fragment of first-order logic with counting. Role logic is closely related to description logic which we have investigated for symbolic shape analysis [9].

Approaches based on three valued logic, which use over-approximation, have been studied in [13, 32]. The semantics of statements and the query of interest are expressed in three valued logic. Only restricted fragments of the logic are decidable [12].

Prominent verification approaches for analysis of data structures use parameterised abstract domains; these analyses include parametric shape analysis [2] as well as predicate abstraction [1, 11] and generalisations of predicate abstraction [16, 17]. Similarly to our approach, reasoning about reachability in program analysis and verification following parametric shape analysis or generalisations of predicate abstraction, are dependent on the invariants that the program maintains for the specific data structure that it manipulates. An algorithm for inferring loop invariants of programs that manipulate heap-allocated data structures, parameterised by the properties to be verified is implemented in Bohne [31]. Bohne infers universally quantified invariants using symbolic shape analysis based on Boolean heaps [24]. Abstraction predicates can be Boolean-valued state predicates (which are either true or false in a given state) or predicates denoting sets of heap objects in a given state (which are true of a given object in a given state).

An algebraic approach towards analysis of pointer programs in the framework of first-order logic is presented in [21]. The underlying pointer-structures and properties such as reachability and sharing are modelled by binary relations and the properties are calculated by a set of rewrite rules.

Separation logic [23] is distinguished by the use of a spatial form of conjunction $(P * Q)$, which allows the spatial orientation of a data structure to be captured without having to use auxiliary predicates. Least and greatest fixpoint operators can be added to separation logic, so that pre- and post-condition semantics for a while-language can be wholly expressed within the logic [27]. Formalisation of recursively defined properties on inductively (and co-inductively) defined data structures is then achievable in the language. The addition of the recursion operators in separation logic leads to alterations to the standard definition of syntactic substitution and the classic substitution; the reasons are related both to the semantics of stack storage and heap storage as well as to the inclusion of the recursion operators [27]. Inductive shape analysis based on separation logic using programmer supplied invariant checkers and numerical domain constraints is proposed in [4]. This approach is applicable to more complex data structures defined by counting, namely red-black trees.

In [25] pointer-based data structure of singly-linked lists and a theory of linked lists is defined as a class of structures of many-sorted first-order logic. The theory is expressive and allows for reasoning about cells, indexed collections of cells, and the reachability of a certain cell from another. The theory is developed for linked lists only.

Alternative languages for modelling and reasoning include modal μ-calculus [14, 28], expressive description logics [5], the propositional dynamic logic [8] and temporal logics [7], and rewriting approaches based on first-order logic [21].

5 Conclusion

In this paper we study imperative programs with destructive update of pointer fields. We model shape types, such as linked lists, cyclic lists, and binary trees as least models of logic programs. We approximate the inductive reasoning about least models by first-order reasoning. We demonstrate that the method is effective for simple programs.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein.

References

[1] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213. ACM, 2001.

[2] M. Benedikt, T. W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *ESOP'99*, LNCS 1576, pages 2–19. Springer, 1999.

[3] D. Calvanese. Finite model reasoning in description logics. In *KR'96*, pages 292–303. Morgan Kaufman, 1996.

[4] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260. ACM, 2008.

[5] G. De Giacomo and M. Lenzerini. Concept language with number restrictions and fixpoints, and its relationship with mu-calculus. In *ECAI'94*, pages 411–415. John Wiley and Sons, 1994.

[6] B. Dutertre and L. De Moura. The YICES SMT solver, 2006. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf.

[7] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B*, pages 995–1072. Elsevier and MIT, 1990.

[8] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[9] L. Georgieva and P. Maier. Description logics for shape analysis. In *SEFM'05*, pages 321–331. IEEE, 2005.

[10] E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *LICS'99*, pages 45–54. IEEE, 1999.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'02*, pages 58–70. ACM, 2002.

[12] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL'04*, LNCS 3210, pages 160–174. Springer, 2004.

[13] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV'04*, LNCS 3114, pages 281–294. Springer, 2004.

[14] D. Kozen. Results on the propositional μ-calculus. In *ICALP'82*, LNCS 140, pages 348–359. Springer, 1982.

[15] V. Kuncak and M. C. Rinard. On role logic. Technical Report 925, MIT CSAIL, 2003. Available at http://arxiv.org/abs/cs.PL/0408018.

[16] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV'04*, LNCS 3114, pages 135–147. Springer, 2004.

[17] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, pages 115–126. ACM, 2006.

[18] S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *POPL'08*, pages 171–182. ACM, 2008.

[19] T. Lev-Ami, N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE'05*, LNCS 3632, pages 99–115. Springer, 2005.

[20] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV'05*, LNCS 3576, pages 476–490. Springer, 2005.

[21] B. Möller. Linked lists calculated. Technical Report 1997-07, Department of Computer Science, University of Augsburg, 1997.

[22] G. Nelson. Verifying reachability invariants of linked structures. In *POPL'83*, pages 38–47. ACM, 1983.

[23] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, LNCS 2142, pages 1–19. Springer, 2001.

[24] A. Podelski and T. Wies. Boolean heaps. In *SAS'05*, LNCS 3672, pages 268–283. Springer, 2005.

[25] S. Ranise and C. G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM'06*, pages 206–215. IEEE, 2006.

[26] A. Rensink. Canonical graph shapes. In *ESOP'04*, LNCS 2986, pages 401–415. Springer, 2004.

[27] É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theor. Comput. Sci.*, 351(2):258–275, 2006.

[28] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Inf. Comput.*, 81(3):249–264, 1989.

[29] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In *CADE'02*, LNCS 2392, pages 275–279. Springer, 2002.

[30] T. Wies. Symbolic shape analysis. Master's thesis, Saarland University, Saarbrücken, 2004.

[31] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. C. Rinard. On verifying complex properties using symbolic shape analysis. Technical Report MPI-I-2006-2-1, Max-Planck Institute for Computer Science, 2006. Available at http://arxiv.org/abs/cs.PL/0609104.

[32] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS'04*, LNCS 2988, pages 530–545. Springer, 2004.

Analysis of Authorizations in SAP R/3*

Manuel Lamotte-Schubert and Christoph Weidenbach

Max Planck Institute for Informatics, Campus E1 4,
D-66123 Saarbrücken
{lamotte,weidenbach}@mpi-inf.mpg.de

Abstract. Today many companies use an ERP (Enterprise Resource Planning) system such as SAP R/3 to run their daily business ranging from financial issues down to the actual control of a production line. Already due to their sheer size, these systems are very complex. In particular, developing and maintaining the authorization setup is a challenge. The goal of our effort is to automatically analyze the authorization setup of an SAP R/3 system against business policies. To this end we formalize the processes, authorization setup as well as the business policies in first-order logic. Then, properties can be (dis)proven fully automatically with our theorem prover SPASS. We exemplify our approach on the purchase process, a typical constituent of any SAP R/3 installation.

1 Introduction

Enterprise Resource Planning (ERP) systems are built to integrate all facets of the business across a company including areas like finance, planning, manufacturing, sales, or marketing. The broader the functionality of such a system, the larger the number of users, the greater the dynamics of a company, the more complex is the administration of the authorizations. In particular, this applies to the SAP R/3 system offered by SAP [1]. In this paper we investigate the authorization setup of SAP R/3. Although SAP R/3 is not the newest release, the most recent release SAP ERP 6.0 actually shares the same authorization subsystem.

Our approach is depicted in Fig. 1. When a company decides to use an ERP system like SAP R/3, it first formulates its business as processes. For example, a typical purchase process starts with the creation of a purchase requisition out of a purchase request, followed by the release of such a requisition, and finally the transformation of the released requisition into the purchase that is eventually sent to a supplier. The processes directly induce an authorization concept. Very often each step of a process corresponds to a particular role of a company employee. For our example, the transformation of the released requisition into a purchase is a typical buyer activity. The development of processes and the authorization concept is guided by business policies. For our example, a business

* SAP, SAP R/3 and SAP ERP 6.0 are registered trademarks of SAP AG in Germany and in several other countries.

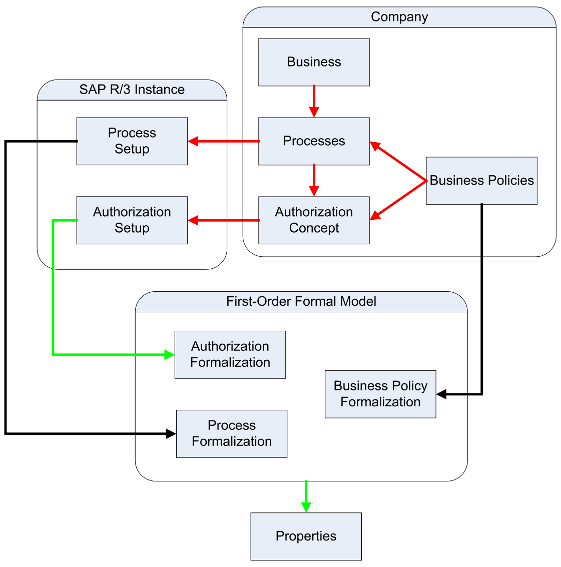


Fig. 1. Analysis of authorizations in SAP R/3

policy might require that the activity of creating a requisition and creating a purchase must always be separated, performed by different persons, and therefore must not be contained in one authorization role. This is a typical rule out of the *Segregation/Separation of Duties* (SoD) approach. Once the processes and authorization concept are defined, the configuration is implemented into an SAP R/3 instance leading to a corresponding process and authorization setup. Due to the sheer size of an SAP project, the number of processes, different employee roles and the highly dynamic development of such a system over time, it is practically impossible to guarantee the compliance of the business policies with the process and authorization setup. Furthermore, it is non-trivial to set up new authorization roles for employees following organizational changes in the business without destroying the overall compliance between the authorization setup and the business policies.

We suggest to solve this problem by first-order logic theorem proving. We model the process and authorization setup in first-order logic and automatically analyze it with respect to a first-order formulation of the business policies. Spass always terminates for provable (it ends with a proof) and non-provable cases (it ends with a saturated set of clauses). The termination of Spass enables the use of an abduction principle deriving missing facts. Then, defining new authorization roles can be solved by saturating abductive queries (Sect. 5). Formulating the processes and business policies has to be done by hand (Sect. 4). However, the authorization setup can be formulated automatically and we suggest a tool pipeline (Sect. 4.3). In practice, the changes to the authorization setup, e.g., caused by organizational changes in a company, cause the most headache to SAP authorization administrators. Business policies and processes are less likely to change and if they change this is not done on a daily basis but by additional smaller SAP change/introduction projects. Therefore, our approach offers a reasonable amount of automatization. As an example SAP R/3 instance for studying the SAP internal process and authorization setup we used the SAP R/3 system run by the Max Planck society.

There have been other efforts to address the verification of the authorization setup in SAP R/3. SAP itself offers a tool collection for Governance, Risk and Compliance. The main difference to our approach is that these tools are only able to check compliance with respect to the transactions performed during concrete runs of the system and are not able to prove the overall compliance of the authorization setup with the business policies. Furthermore, there is no tool support for the business policy compliant generation of new authorization roles available up to now. Other efforts include the general verification of role-based access control principle together with constraints like SoD [2] but they are neither connected to the SAP R/3 system nor they do incorporate business processes. To the best of our knowledge, there has been no attempt so far to analyze the authorizations in SAP R/3 together with the business processes and business policies.

The paper is organized as follows. After explaining the basic first-order notation (Sect. 2) the SAP R/3 internal mechanisms are studied with respect to processes and authorizations in Sect. 3, followed by the formalization in first-order logic (Sect. 4). Due to space limitations we only explain important aspects of the developed first-order theory, hiding details that are not needed to understand the main ideas. Nevertheless, the overall formalization can be obtained from the SPASS homepage (`spass-prover.org`) in the "prototype and experiments" section. Our results on experiments are contained in Sect. 5 and the paper ends with a small conclusion and ideas for future work (Sect. 6).

2 Background

The formalization of the process, authorization setup and business policies is accomplished using first-order logic without equality. The following syntax definition as well as the semantics of the used language is taken from [3].

A first-order language is constructed over a signature $\Sigma = (\mathcal{F}, \mathcal{R})$, where $\mathcal{F}$ and $\mathcal{R}$ are non-empty, disjoint, in general infinite sets of function and predicate symbols, respectively. Every function or predicate symbol has some fixed arity. In addition to these sets that are specific for a first-order language, we assume a further, infinite set $\mathcal{X}$ of variable symbols disjoint from the symbols in Σ. Then the set of all *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is defined as usual. A term not containing a variable is a *ground term.* If $t_1, \ldots, t_n$ are terms and $R \in \mathcal{R}$ is a predicate symbol with arity n, then $R(t_1, \ldots, t_n)$ is an *atom.* An atom or the negation of an atom is called *literal.* Disjunctions of literals are *clauses* where all variables are implicitly universally quantified. *Formulae* are recursively constructed over atoms and the operators $\supset$ (implication), $\equiv$ (equivalence), $\wedge$ (conjunction), $\vee$

(disjunction), $\neg$ (negation) and the quantifiers $\forall$ (universal), $\exists$ (existential) as usual. For convenience, we often write $\forall x_1, \ldots, x_n \,.\, \phi$ instead of $\forall x_1 \ldots . \forall x_n \,.\, \phi$ and analogously for the existential quantifier and assume the descending binding precedence $\neg$, $\wedge$, $\vee$, $\supset$, $\forall$, $\exists$.

The formal model uses predicate symbols whose first letter is always uppercase and the predicate itself is italic, e.g. the predicate *Access* is used to represent the authorization access relation for a user, the atom *Access*(`MUELLER`, `ME51N`) expresses that user `MUELLER` holds all rights to perform the transaction `ME51N`, the creation of a purchase requisition. Constants originating from SAP R/3 are always written in typewriter font, e.g. `MUELLER`. In general, function names start with a lowercase letter different from "*x*", e.g. the function *authObj* is used to represent an authorization (object). Variables are always prefixed with "*x*" and written lowercase, for example, *xu*, *xwrk*.

Although we do not explicitly define sorts, our formulae are actually many-sorted. For the explanation of our predicate and function usage we sometimes refer to these "implicit" sorts by putting them in square brackets. For example, the "declaration"

Access(< *user* >, *authObj*(< *auth object name* >, < *auth field* >, < *value* >))

explains that the first argument of an *Access* atom is a user term and the second argument a term representing an authorization (object).

3 SAP R/3 Setup and Business Policies in Detail

We use the SAP terminology throughout our work in order to describe the relevant aspects of the SAP R/3 system. The definition of terms adopted from SAP are written in italics.

Authorization Setup. The SAP R/3 authorization architecture is a complex structure and consists of several components interacting with each other. The key data structure is an *authorization*, an instance of an *authorization object*, that is eventually assigned via a *profile* to a user and typically grants the access to one particular action inside SAP R/3. In order to align authorizations with process steps, they are grouped in *roles*.

In detail, an authorization object is a named entity that holds one or more named authorization fields, similar to a class structure of a programming language. Together with appropriate field values, the authorization object constitutes the authorization. An authorization is therefore an instance of an authorization object, similar to the instance of a programming language class. The relation is shown in Fig. 2.

There are *single* and *composite roles* for the grouping of authorizations available. A single role groups authorizations whereas composite roles serve as containers for single roles. Single roles have a name and a list of authorizations. For example, Fig. 3 shows the structure of single role with name `ZBANF_WRK_INF_ED` by means of the concrete authorization `S_TCODE` with a field `TCD` and the value `ME51N`. The overall role `ZBANF_WRK_INF_ED` contains all authorizations required to create requisitions.

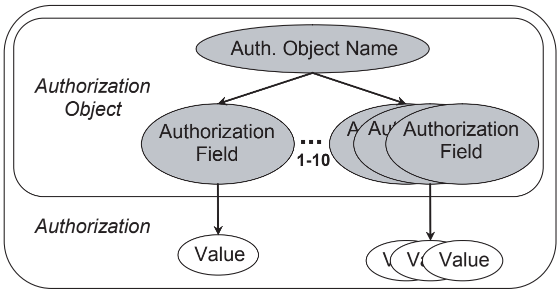


Fig. 2. Authorization object and authorization

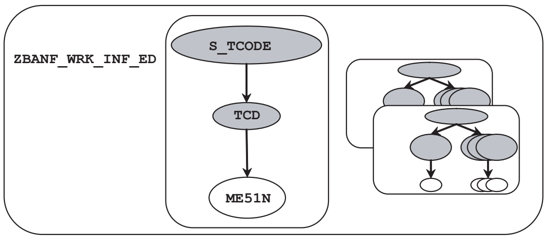


Fig. 3. Single role including authorizations

Single and composite roles generate profiles in SAP R/3 that are then eventually assigned to the user.

In our first-order model authorization objects become terms starting with function *authObj* and ground instances of those are authorizations. For the formulation of roles and profiles we use the respective predicates *SingleRole*, *CompositeRole*, and *UserProfile*. Mappings between objects are represented by functions and concrete values become constants.

Process Setup. A process is a consecutive flow of transactions in SAP R/3. Typically, the different actions are enabled by the creation or change of data inside the system due to preceding transactions. There is a unique identification code for any transaction, for example, `ME51N` stands for the transaction to create a requisition. If a user executes a particular transaction in the system then the effective authorizations from the users' authorization profile are checked. These checks are called *authorization checks.*

Each authorization check consists of two parts: (i) the presence of the required authorization object in the authorization profile associated with the user (for this purpose only the authorization object name is compared) is checked, and if this check succeeds, (ii) the required value(s) for the transaction are compared with the value(s) present in the value field(s) of the authorization assigned to the user. In particular, the second check succeeds if all value fields with the corresponding values of the object match to the required fields and values (AND-combination). A match can mean simple equality, e.g. the right to change data, or comparisons with respect to some ordering, e.g. the amount of money is below some threshold. If one check fails, then the overall check of the authorization fails.

The first authorization check in every transaction is the check for the transaction code which is triggered by the SAP R/3 system before the actual transaction starts. The name of the corresponding authorization object for this check is `S_TCODE`. This object has only one authorization field `TCD` which serves as a container for the required transaction code. All further authorization checks are implemented in the transaction.

Example: Purchase Process and its Authorization Checks. The purchase process introduced in Sect. 1 is a typical constituent of the SAP R/3 system and is used in this paper as a running example. The creation of the requisition as well as the creation of the order are mapped by exactly one transaction in SAP R/3 whereas the release transaction implicitly additionally calls the transaction to view the requisition. Furthermore, releases of requisitions require release strategy settings done once at the initial configuration of an SAP R/3 system.

An SAP R/3 purchase requisition document is created to request the purchase of goods or services by calling the transaction `ME51N` in the SAP R/3 system. This transaction code is subject to the first authorization check and must be present as an authorization in the users' authorization profile.

The creation of a requisition needs to fill different fields, for example, the plant field for which the item is destined for. Some of these fields are protected by authorization objects, for example, the plant field is protected by the authorization object `M_BANF_WRK` with the two authorization fields `ACTVT` (activity) and `WERKS` (plant). The field activity requires the concrete value `01` (for "create") and the value for the plant field depends on the data entered in the requisition. If the data for the plant has been entered, the users' authorization to perform the action "create" for the entered plant is checked. The other protected fields document type and purchase group are protected in a similar way.

Release procedures for requisitions are used in SAP R/3 to approve requisitions which exceed a certain budget limit before they can be converted to an order. The SAP R/3 system uses so-called release strategies to achieve such approvals. A release strategy is an object that contains conditions for its application as well as a small process definition. This process defines the required actions to eventually release the requisition.

The order is the request to the supplier or another plant of the company to deliver the requisitioned material or service under terms and conditions agreed before. A released requisition is the prerequisite to create an order that is connected to the requisition. The authorization checks in the create order transaction are performed analogously to the checks occurring during the create requisition transaction.

Business Policies. Business policies are constraints on the business. A lot of business constraints follow best-practice approaches, for example, Segregation/Separation of Duties (SoD). This approach is considered in our work and requires that there is no single individual having the control over two or more phases of a process, so that a deliberate fraud is more difficult to occur. In the purchase process, this means that the requisitioner must be distinct from the releasing person and the buyer.

On the SAP R/3 transactional level, it means that a single user is not allowed to have the authorizations to perform the appropriate transactions to create requisitions, release requisitions and orders for some plant, material group, purchasing group and organization[1].

4 SAP R/3 Formal Authorization Analysis

The SAP R/3 authorization system is formalized using first-order logic without equality. The formalization represents the process (we use the purchase process as example) and authorization setup as well as the formalized business policies. The prerequisite for the construction of the formalization is a snapshot of an SAP R/3 system, i.e. the formalization represents the state of the system at a given time.

For the goal of proving compliance (abduce changes) of the authorization setup with the business policies, we perform a number of abstractions, easing the size and depth of the formalization. We assume that we always have only one item per purchase requisition/order. We do not deeply model numbers, for example, amounts of money. Numbers are formalized as constants, intervals of numbers are also described as constants, for example *GREATER_1000_LESS_10000_EUR*, and the corresponding ordering relations between these constants are established. Within the authorization check procedure and the release strategy appliance checks, the SAP R/3 system uses a comprehensive pattern matching mechanism. For simplification, we formalized only exact matching and the asterisk symbol matching every required value. Composed values of an asterisk and a string are currently not supported by our formalization.

We formalized most of the SAP R/3 system parts in form of monadic predicates because Spass offers particular reduction support for these predicates via soft typing [4]. The large set of authorization components like roles and profiles is modeled by the monadic predicates, while the assignment of these components to the users is represented by an implication. The set of process states in the SAP R/3 system is modelled by a set of predicates; and the abstraction of its dynamic behavior, which is relevant for authorization, is captured implications. The premise of such an implication represents the conditions for the process step while the conclusion stands for the effects after the execution of the corresponding transactions in this step. The form of business policies is individual and therefore the formalization of the policies depends on the type of the policy.

4.1 Authorization and Process Setup

The authorization setup layer consists of several predicates representing the way where authorizations are arranged and eventually assigned to a user. A single

[1] These are the properties protected by authorization objects.

role is modeled by the unary predicate *SingleRole*. The function *authObj* with arity 3 therein maps the authorization value to the authorization field of the authorization object. The authorization object together with the value represents the authorization that is then assigned to the single role by the binary function *singleRoleEntry*. Each authorization that is contained in some SAP R/3 single role results in a *SingleRole* atom in the formalization.

$$\begin{aligned}&\mathit{SingleRole}(\mathit{singleRoleEntry}(<\mathit{single\ role\ name}>,\\&\qquad \mathit{authObj}(<\mathit{auth\ object\ name}>,<\mathit{auth\ field}>,<\mathit{value}>)))\end{aligned}$$

A composite role is modeled by the unary predicate *CompositeRole*. The function *compositeRoleEntry* therein associates the single role given by its name with the composite role.

$$\begin{aligned}&\mathit{CompositeRole}(\mathit{compositeRoleEntry}(<\mathit{composite\ role\ name}>,\\&\qquad <\mathit{single\ role\ name}>))\end{aligned}$$

The effective authorizations associated with a user are stored in the authorization profile. This profile is modeled by the unary predicate *UserProfile*. The function *userProfileEntry* maps the authorizations given by the function *authObj* to the user; any authorization check looks for the required authorization only in the users authorization profile.

$$\begin{aligned}&\mathit{UserProfile}(\mathit{userProfileEntry}(<\mathit{user}>,\\&\qquad \mathit{authObj}(<\mathit{auth\ object\ name}>,<\mathit{auth\ field}>,<\mathit{value}>)))\end{aligned}$$

The following formula exactly models the mechanism of the SAP R/3 user authorization profile creation. The assignment of a role to a user implies the assignment of the appropriate generated authorization profile. Whenever a single role or composite role is going to be assigned to a user via the predicate *Holds*, the authorization part is extracted and the corresponding authorization profile entry (representing the effective authorization) for the user is created:

$$\begin{aligned}&\forall\ \mathit{xu, xpn, xsrn, xcrn, xaon, xaof, xav}\ .\\&\quad(\mathit{SingleRole}(\mathit{singleRoleEntry}(\mathit{xsrn}, \mathit{authObj}(\mathit{xaon, xaof, xav}))) \wedge\\&\quad \mathit{Holds}(\mathit{xu, xsrn})) \vee\\&\quad(\mathit{CompositeRole}(\mathit{compositeRoleEntry}(\mathit{xcrn, xsrn})) \wedge\\&\quad \mathit{SingleRole}(\mathit{singleRoleEntry}(\mathit{xsrn}, \mathit{authObj}(\mathit{xaon, xaof, xav}))) \wedge\\&\quad \mathit{Holds}(\mathit{xu, xcrn}))\\&\supset\\&\quad \mathit{UserProfile}(\mathit{userProfileEntry}(\mathit{xu}, \mathit{authObj}(\mathit{xaon, xaof, xav})))\end{aligned}$$

The authorization check result − access or decline − is represented in our first-order formalization by the binary predicate *Access*. If the atom *Access* is valid, the access to the function or data protected by the authorization object is granted. Otherwise, it is not. In other words, a valid instance of the following *Access* atom expresses that the user <*user*> has successfully passed the authorization check of the appropriate authorization which is denoted by the authorization object with its field and value.

$$\mathit{Access}(<\mathit{user}>, \mathit{authObj}(<\mathit{auth\ object\ name}>, <\mathit{auth\ field}>, <\mathit{value}>))$$

In the authorization check procedure, the required authorization object information together with the appropriate authorization value are compared to the authorization present in the users authorization profile. All properties, namely the authorization object name, the field and the value must be equal in the check in order to succeed. This is modeled by the following implication. If the required authorization is present in the user authorization profile, then the access to this authorization is granted.

$$\begin{array}{l}\forall\ \mathit{xu, xaon, xaof, xav}\ .\\ \quad \mathit{UserProfile}(\mathit{userProfileEntry}(\mathit{xu}, \mathit{authObj}(\mathit{xaon, xaof, xav})))\\ \supset\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathit{xaon, xaof, xav}))\end{array}$$

Example: Purchase Process. The formalization of the transaction layer with its authorization checks is done manually in our example. We have introduced an additional layer by overloading the predicate symbol *Access*. The following transition shows the abstraction for the transaction “create a requisition”. It groups all authorization checks occurring during the execution of the transaction.

$$\begin{array}{l}\forall\ \mathit{xu, xwrk, xbsa, xekg}\ .\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{S\_TCODE}, \mathtt{TCD}, \mathtt{ME51N}))\ \wedge\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{M\_BANF\_WRK}, \mathtt{ACTVT}, \mathtt{01}))\ \wedge\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{M\_BANF\_WRK}, \mathtt{WERKS}, \mathit{xwrk}))\ \wedge\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{M\_BANF\_BSA}, \mathtt{ACTVT}, \mathtt{01}))\ \wedge\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{M\_BANF\_BSA}, \mathtt{BSART}, \mathit{xbsa}))\ \wedge\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{M\_BANF\_EKG}, \mathtt{ACTVT}, \mathtt{01}))\ \wedge\\ \quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\mathtt{M\_BANF\_EKG}, \mathtt{EKGRP}, \mathit{xekg}))\\ \supset\\ \quad \mathit{Access}(\mathit{xu}, \mathtt{ME51N})\end{array}$$

The first check represents the check of the transaction code (`ME51N`) carried out by the SAP R/3 system at the start of every transaction. The authorization objects `M_BANF_WRK`, `M_BANF_BSA`, and `M_BANF_EKG` check the plant, document type and purchase group, respectively. They have constants in the first field (`ACTVT`) checking the type of action (`01` stands for “create”, `02` stands for “change”, `03` stands for “view”) and variables in their second field standing for the values of the corresponding fields: plant, document type and purchase group. If the atom $\mathit{Access}(\mathit{xu}, \mathtt{ME51N})$ holds, then the user is allowed to execute the transaction in at least one instance, for example for one plant (variable *xwrk*), one document type (variable *xbsa*) and one purchase group (variable *xekg*). Later in the context of the requisition creation, when the exact values are known from the requisition, the values of the variables have to be evaluated and checked again.

The previously mentioned additional transactional layer for authorization checks makes it more comfortable to model the formalization of the purchase process steps. The purchase process starts with the existence of a purchase request whose data is then entered into the SAP R/3 system by the purchase requisitioner. After the data has been entered, the request has become an SAP R/3 requisition object that is represented by the atom *RequisitionCreated*. An instance of this atom contains all needed details.

RequisitionCreated(<user>, <document type>, <position>,
<material>, <plant>, <purchasing group>,
<purchasing organization>, <material group>, <price>, <id>)

The following implication represents the creation of the corresponding SAP R/3 requisition object. The predicate *Requisition* is an arbitrary purchase request for an item and the predicate *RequisitionCreated* is represents this item in the SAP R/3 system, created by the user denoted by the variable *xu*. This user *xu* needs access to the create transaction (`ME51N`). As mentioned, the values of the variables *xwrk*, *xbsa*, *xekg* are again subject to authorization checks because at this point the values of the variables are known (namely the values from the requisition that is going to be created). This results in the following formula.

$$\begin{aligned}
&\forall\ \mathit{xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid}\ .\\
&\quad \mathit{Requisition}(\mathit{xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid}) \wedge\\
&\quad \mathit{Access}(\mathit{xu}, \texttt{ME51N}) \wedge\\
&\quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\texttt{M\_BANF\_WRK}, \texttt{WERKS}, \mathit{xwrk})) \wedge\\
&\quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\texttt{M\_BANF\_BSA}, \texttt{BSART}, \mathit{xbsa})) \wedge\\
&\quad \mathit{Access}(\mathit{xu}, \mathit{authObj}(\texttt{M\_BANF\_EKG}, \texttt{EKGRP}, \mathit{xekg}))\\
&\supset\\
&\quad \mathit{RequisitionCreated}(\mathit{xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl,}\\
&\qquad\quad \mathit{xgswrt, xid})
\end{aligned}$$

A more complex and interesting step in the purchase process is the release of a requisition where a release strategy has applied. The function *property* relates a value to a property name and represents a condition property for the application of some release strategy. The class construction is eventually used to group several properties belonging to a release strategy.

$$\begin{aligned}
&\mathit{ReleaseStrategy}(<\mathit{release\ strategy\ name}>,\\
&\qquad \mathit{class}(<\mathit{characteristics\ class\ name}>,\\
&\qquad\qquad \mathit{property}(<\mathit{property\ name}>, <\mathit{value}>)))
\end{aligned}$$

Release strategies consist of one or more single release steps which are declared by the atom *ReleaseRequirement*. This atom groups the strategy name and the required code for each step.

$$\mathit{ReleaseRequirement}(<\mathit{release\ strategy\ name}>, <\mathit{release\ code}>)$$

Figure 4 shows the formalization of one release step for an existing requisition object. The existence of the requisition is checked by the first atom *Requisition-Created* in the premise. Subsequent atoms address the application checks of the release strategy *xfrgstrat* for which the characteristics denoted by the variables *xekg* (purchasing group), *xwrk* (plant), and *xgswrt* (total amount of money of the requisition) are used. The predicate *ReleaseRequirement* retrieves the release code for the release step in the release strategy and is then subject to an authorization check. In order to proceed with the release step, the user, denoted by the variable *xu2*, needs authorizations for the release (with the code *xfrgco*) as well as for the transaction (`ME54N`) in order to perform the release step. Please note that the user performing the release step (*xu2*) is different from the user who has created the requisition (*xu1*) which is enforced by the business policies (see Sect. 4.2). The conclusion expresses the fact that the user *xu2* has performed the release step with the code *xfrgco* in the overall release of the requisition.

$$
\begin{array}{l}
\forall\ \mathit{xu1},\ \mathit{xu2},\ \mathit{xbsa},\ \mathit{xpos},\ \mathit{xmat},\ \mathit{xwrk},\ \mathit{xekg},\ \mathit{xekorg},\ \mathit{xmatkl},\ \mathit{xgswrt},\\
\qquad\quad \mathit{xfrgstrat},\ \mathit{xfrgco},\ \mathit{xcl},\ \mathit{xid}\ .\\
\quad \mathit{RequisitionCreated}(\mathit{xu1},\ \mathit{xbsa},\ \mathit{xpos},\ \mathit{xmat},\ \mathit{xwrk},\ \mathit{xekg},\ \mathit{xekorg},\ \mathit{xmatkl},\\
\qquad\quad \mathit{xgswrt},\ \mathit{xid})\ \wedge\\
\quad \mathit{ReleaseStrategy}(\mathit{xfrgstrat}, \mathit{class}(\mathit{xcl}, \mathit{property}(\texttt{FRG\_CEBAN\_EKGRP}, \mathit{xekg})))\ \wedge\\
\quad \mathit{ReleaseStrategy}(\mathit{xfrgstrat}, \mathit{class}(\mathit{xcl}, \mathit{property}(\texttt{FRG\_CEBAN\_WERKS}, \mathit{xwrk})))\ \wedge\\
\quad \mathit{ReleaseStrategy}(\mathit{xfrgstrat}, \mathit{class}(\mathit{xcl}, \mathit{property}(\texttt{FRG\_CEBAN\_GSWRT}, \mathit{xgswrt})))\ \wedge\\
\quad \mathit{ReleaseRequirement}(\mathit{xfrgstrat},\ \mathit{xfrgco})\ \wedge\\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\texttt{M\_EINK\_FRG}, \texttt{FRGCO}, \mathit{xfrgco}))\ \wedge\\
\quad \mathit{Access}(\mathit{xu2}, \texttt{ME54N})\ \wedge\\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\texttt{M\_BANF\_WRK}, \texttt{WERKS}, \mathit{xwrk}))\ \wedge\\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\texttt{M\_BANF\_BSA}, \texttt{BSART}, \mathit{xbsa}))\ \wedge\\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\texttt{M\_BANF\_EKG}, \texttt{EKGRP}, \mathit{xekg}))\\
\supset\\
\quad \mathit{RequisitionReleasedStep}(\mathit{xu2},\ \mathit{xfrggr},\ \mathit{xfrgstrat},\ \mathit{xfrgco},\ \mathit{xbsa},\ \mathit{xpos},\ \mathit{xmat},\\
\qquad\quad \mathit{xwrk},\ \mathit{xekg},\ \mathit{xekorg},\ \mathit{xmatkl},\ \mathit{xgswrt},\ \mathit{xid})
\end{array}
$$

Fig. 4. Single release step in the SAP R/3 purchase process

Concerning the overall release of a requisition, there are further formulae which define the required single release steps for a complete release of the requisition.

The released requisition is eventually the precondition to create an order object in SAP R/3 that is connected to the requisition. The formalization of this step is analogous to the creation of a requisition.

4.2 Business Policies

The SoD business policy for the purchase process expresses that there should be no user having the control over two or more phases of a process. Very often in smaller companies, this is relaxed into a less strict requirement stating that there should be no user who is allowed to perform the complete purchase process in one instance. The relaxed version of SoD is formalized by the following formula. Starting from the purchase request, there are no values for which the user *xu* can perform the three steps of the purchase process.

$$\begin{aligned}
&\neg\exists\; xu, xbsa, xwrk, xekg, xekorg, xmatkl, xgswrt, xpos, xmat, xid\;.\\
&\quad \mathit{RequisitionCreated}(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl,\\
&\qquad xgswrt, xid) \wedge\\
&\quad \mathit{RequisitionReleased}(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl,\\
&\qquad xgswrt, xid) \wedge\\
&\quad \mathit{OrderCreated}(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt,\\
&\qquad xid)
\end{aligned}$$

4.3 Automatic Authorization Extraction

The authorization checks occurring in the SAP R/3 system can be extracted by looking directly into the source code and by exploring the connection between the transaction and the associated program code. The authorization check at the beginning of a transaction can be read from an internal data dictionary. This is done by the transaction `SE93` which is used to manage the association between the transaction code, the program code and the authorization check. All further checks are implemented into the program code using the *AUTHORITY-CHECK* statement. For convenience, we used the SAP R/3 System Trace tool which monitors, among other things, the authorization checks taking place during the execution of a transaction.

The extraction of the SAP R/3 system users and its authorizations is achieved using the User Information System[2]. It is able to report information about users, their roles and profiles as well as information about authorizations, authorization objects or transactions. The result of a query to this information system can be easily stored in text-format (see Fig. 5) and lists, for example, all authorizations present in a profile.

5 Results

We used the theorem prover SPASS, Version *3.0* [5] for our experiments. The theory containing the SAP R/3 general authorization structure and its instantiation to the purchase process consists of 156 formulae with a size of 41 KByte resulting in 177 clauses. The experiments ran on a Dell PowerEdge 1950 server

[2] Transaction `SUIM`

```
Profile
  |
  ---   Z:EK1_INFO   <PRO>
      |
      |--   M_BEST_WRK <OBJ> Plant in Purchase Order
      |   |
      |   ---   M_BEST_WRKAL <AUT> Plant in Purchase Order
      |       |
      |       |--   ACTVT      <FLD> Activity
      |       |   |
      |       |   ------*
      |       |
      |       ---   WERKS      <FLD> Plant
      |           |
      |           ------INFO
      ...
```

Fig. 5. Export of authorizations from SAP R/3

running at 3.14 GHz equipped with 16 GB RAM, 64-bit Debian Linux, Kernel 2.6.24.2.1.amd64-smp.

One of the key results is that the overall formalization can be finitely saturated. This is mainly due to the fact that there is no recursion over the business processes, and consequently, ordering mechanisms are sufficient for saturation. Our experiments also include the user authorization data. SPASS always terminated within the experiments, ending with either a saturated set of clauses or a proof. A finite saturation means that the given conjecture could not be proven and therefore doesn't hold. If no conjecture was given, it states there is no contradiction in the input formulae and consequently the authorization setup and the business policies are compatible. The fact that SPASS always terminates is also an important prerequisite for the actual development of the theory as it enables inspection of models and detection of accidental inconsistencies.

Every experiment run took less than 20 seconds. The saturation of the input theory, including the user data and business policy, took 13 seconds with SPASS run with default settings. It can be tweaked by predefining a particular selection strategy to less than 1 second.

The ability to run a variety of different queries in addition to the general inspection of the setup was also one of the original motivations to do this work. Having SPASS terminating on queries further enables the use of an abduction principle [6, 7]. We give SPASS the query to be proven and then the saturated clauses out of the query represent a set of abductive answers. This is complete for the propositional case as stated in [6]. Completeness is open for the full first-order case. For example, it is interesting whether a particular user, e.g., MUELLER in our running example, is able to perform the step to create a requisition, maybe for the given plant INF. Such a conjecture is formalized and fed to SPASS as the conjecture:

$$
\begin{aligned}
&\exists\ \mathit{xbsa}, \mathit{xekg}, \mathit{xekorg}, \mathit{xmatkl}, \mathit{xgswrt}, \mathit{xpos}, \mathit{xmat}, \mathit{xid}\ .\\
&\quad \mathit{Requisition}(\mathit{xbsa}, \mathit{xpos}, \mathit{xmat}, \mathtt{INF}, \mathit{xekg}, \mathit{xekorg}, \mathit{xmatkl}, \mathit{xgswrt}, \mathit{xid})\\
&\supset\\
&\quad \mathit{RequisitionCreated}(\mathtt{MUELLER}, \mathit{xbsa}, \mathit{xpos}, \mathit{xmat}, \mathtt{INF}, \mathit{xekg}, \mathit{xekorg},\\
&\qquad\quad \mathit{xmatkl}, \mathit{xgswrt}, \mathit{xid})
\end{aligned}
$$

In our example setup the conjecture holds and can be proved in less than 1 second.

Removing MUELLER's access rights to the corresponding transaction ME51N from the theory and rerunning the above conjecture results in a saturation without proof in 8 seconds. Now the purely negative clauses resulting from the query can be interpreted as abductive answers to the query. For example, the generated clause

$$\neg \mathit{Access}(\mathtt{MUELLER}, \mathtt{ME51N})$$

expresses that the right to execute transaction ME51N is missing in order to successfully create a requisition.

6 Conclusion and Future Work

This paper has presented an effort to the automatic analysis of an SAP R/3 process and authorization set up with respect to given business policies using the purchase process as a case study.

To accomplish automatic verification, the SAP R/3 process setup, the authorization setup and the business policies have been formalized in first-order logic. The formalization decisions were taken from a detailed analysis of the SAP R/3 system instance run by the Max Planck society. We could show that the developed formalization can be automatically analyzed by SPASS. Any proof attempt with SPASS we have done in this context terminated. We can automatically check compliance of business policies, properties with respect to specific user authorization configurations as well as automatically abduce compliant changes to the authorization set up.

There are a number of open questions left for future work. Our model of numbers by first-order constants could be overcome by using SPASS(LA) [8], our currently experimental prover for the hierarchic extension of linear arithmetic by first-order logic. For the first-order formula class presented in this paper as well as for such an extended first-order formula class over linear arithmetic decidability is open.

Eventually, it is an open question how our model scales with respect to a more integrated formalization of an SAP R/3 instance. In our example, we analyzed only the purchase process and up to ten users while a typical instance has about 50–200 processes and up to several thousand users. We are optimistic that this is not out of range to first-order theorem proving because processes as well as users can be analyzed almost independently.

The concrete formalization of authorizations differs for individual (non-SAP) software systems. However, theoretic aspects of our approach like termination, scalability or completeness in verification tasks remain similarly and can be transferred to other systems.

References

1. SAP Press Release: SAP Holds Top Rankings in Worldwide Market Share (July 2008) http://www.sap.com/about/newsroom/news-releases/press.epx?pressid=9913.
2. Yuan, C., He, Y., He, J., Zhou, Z.: A verifiable formal specification for rbac model with constraints of separation of duty. In: Information Security and Cryptology. Volume 4318 of LNCS., Springer (2006) 196–210
3. Nonnengart, A., Weidenbach, C.: 6. In: Computing small clause normal forms. Volume 1. Elsevier, Amsterdam, the Netherlands (January 2001) 335–367
4. Ganzinger, H., Meyer, C., Weidenbach, C.: Soft typing for ordered resolution. In McCune, W., ed.: Proceedings of the 14th International Conference on Automated Deduction (CADE-14). Volume 1249 of Lecture Notes in Computer Science., Townsville, Australia, Springer (1997) 321–335
5. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System description: Spass version 3.0. In Pfenning, F., ed.: CADE-21 : 21st International Conference on Automated Deduction. Volume 4603 of LNCS., Springer (2007) 514–520
6. Dimova, D.: Propositional abduction. Bachelor's thesis, Universität des Saarlandes (September 2007)
7. Eiter, T., Makino, K.: On computing all abductive explanations. In: Eighteenth national conference on Artificial intelligence, Menlo Park, CA, USA, American Association for Artificial Intelligence (2002) 62–67
8. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic: SUP(LA). In Ghilardi, S., Sebastiani, R., eds.: Frontiers of Combining Systems. 7th International Symposium FroCos 2009, Proceedings. LNCS, Springer (2009) Accepted for publication.

Towards Verification of Security-Aware E-services

Silvio Ranise

Dipartimento di Informatica
Università di Verona, Italy

Abstract. We study an extension of the relational transducers introduced by Abiteboul, Vianu, Fordham, and Yesha, which are capable of specifying transaction protocols and their interplay with security constraints. We investigate the decidability of relevant verification problems such as goal reachability and log validation.

1 Introduction

Web-services providing e-commerce capabilities to support business transactions between several parties over the Internet are more and more widespread. The development of such services involves several issues involving security, authentication, and the design of business models to name a few.

Relational transducers have been proposed [17] to model the transaction behavior of e-services for e-commerce and to allow for the analysis of the underlying transaction protocols. Roughly, a relational transducer is a machine capable of translating an input sequence of relations into an output sequence of relations; its state being a relational database. A particular class of relational transducers, called semi-positive cumulative (spocus) transducers, has been identified and studied because its specification is declarative and simple to understand, and some interesting verification problems turn out to be decidable (see again [17]). The input-output mapping in spocus transducers is implicitly defined by a set of non-recursive (variant of) Datalog rules. This is based on the observation that any set of Datalog rules defines a query which is a (partial) function from relational databases (the input relations of the transducers) to answer relations (the output relations of the transducer), see, e.g., [16].

Although spocus transducers support declarative specifications and some interesting verification problems are decidable, one of their major shortcomings is the lack of support for the specification and verification of security requirements. In particular, the class of security requirements pertaining to authorization play a crucial role in several business transactions. For example, failure to meet authorization constraints may cause economic losses and may even have legal implications. In this paper, we propose an extension of the spocus transducers that overcome this problem. In particular, following [12], we extend the rules for input-output in spocus transducers with constraint Datalog rules to formalize access policy statements. This allows us to develop our ideas in the framework of first-order logic and to adapt and reuse specification techniques belonging to the line of research which uses extensions of Datalog to express policy statements (see, e.g., [7, 2, 11]). Technically, the situation is more complex than with spocus transducers as certain patterns in authorization policies (typically, delegation) require recursion (see, e.g., [3]). The (variant of) Datalog rules used in [17] for spocus transducers were assumed to be non-recursive.

Plan of the paper. Section 2 introduces some background notions on first-order logic, relational databases, and Datalog rules. Section 3 extends spocus transducers with (constraint) Datalog rules for access policy specifications and introduces two verification problems. In Section 3.1, a simple (yet representative) business process specified by means of a policy-aware transducer illustrates the need to have recursive Datalog rules and a first-order theory to be able to correctly specify delegation and trust relationships, respectively, which are relevant to capture important business rules. As the verification problems can be reduced to satisfiability problems for a certain class of formulae, Section 4 defines such a class and proves the decidability of the satisfiability problem under suitable assumptions (for lack of space, proofs are in the extended version [13] of this paper) and Section 5 shows how to reduce verification to satisfiability. In particular, Section 5.1 discusses the hypotheses under which the fix-point computation of the query induced by a set of constraint Datalog rules terminates. Section 6 concludes.

2 Preliminaries

We assume familiarity with the basic syntactic and semantic notions of first-order logic (see, e.g., [8]). We work in first-order logic with equality and consider the equality symbol $=$ as a logical constant. A *constraint* is a (finite) conjunction of literals. An *expression* is a term, an atom, a literal, or a formula. A $\Sigma(\underline{x})$-*expression* is an expression built out of the symbols in a signature Σ where at most the variables in the sequence $\underline{x}$ may occur free, and we write $E(\underline{x})$ to emphasize that E is a $\Sigma(\underline{x})$-expression. (Below, by abuse of notation, we consider sequences also as finite sets and use the standard set-theoretic operations to combine them.)

Let Σ be a signature. A *Σ-theory T* is a set of Σ-sentences. In this paper, we assume that all theories are consistent (i.e. they admit at least one model). A Σ-formula $\varphi(\underline{x})$ is T-satisfiable iff there exists a model $\mathcal{M}$ of T (i.e. $\mathcal{M} \models T$) such that $\mathcal{M} \models \exists \underline{x}.\varphi(\underline{x})$. The *satisfiability modulo theory* T (in symbols, SMT(T)) *problem* consists of establishing the T-satisfiability of any quantifier-free Σ-formula. A formula $\varphi(\underline{x})$ is *T-valid* if for every model $\mathcal{M}$ of T, we have $\mathcal{M} \models \forall \underline{x}.\varphi(\underline{x})$. Two formulae φ and ψ are *T-equivalent* iff the formula $\varphi \leftrightarrow \psi$ is T-valid. A theory T admits *quantifier elimination* if for an arbitrary formula $\varphi(\underline{x})$ (possibly containing quantifiers), it is possible to compute a T-equivalent quantifier-free formula $\varphi'(\underline{x})$. A Σ-theory T is *locally finite* if Σ is finite and, for every set of constants $\underline{a}$, there are finitely many ground terms $t_1, ..., t_{k_{\underline{a}}}$, called *representatives*, such that for every ground $(\Sigma \cup \underline{a})$-term u, we have $T \models u = t_i$

for some i. If the representatives are effectively computable from $\underline{a}$ and t_i is computable from u, then T is *effectively* locally finite. For simplicity, we will often say "locally finite" to mean "effectively locally finite."

We consider the *Bernays-Schönfinkel-Ramsey* (BSR) class, also called Effectively Propositional Logic, whose satisfiability problem is well-known to be decidable. A formula of the BSR class is of the form $\exists \underline{x}.\forall \underline{y}.\varphi(\underline{x}, \underline{y})$, where $\underline{x}, \underline{y}$ are (disjoint) tuples of variables and φ is a quantifier-free formula built out of a signature containing only relation and constant symbols (i.e. no function symbol occurs in φ).

A *finite instance of (a n-ary relation) r* is a formula of the form

$$\forall \underline{x}.r(\underline{x}) \leftrightarrow \bigvee_{j=1}^{m} \underline{x} = \underline{c}^j,$$

where $\underline{x} = x_1, ..., x_n$ is a sequence of variables, $\underline{c}^j = c^j_1, ..., c^j_n$ is a sequence of constants (for $j = 1, ..., m$), and $\underline{x} = \underline{c}^j$ abbreviates $(x_1 = c^j_1 \wedge \cdots \wedge x_n = c^j_n)$. Let $\underline{R}$ be a finite set of predicate symbols, a *database over* $\underline{R}$ is a conjunction of finite instances of $r \in \underline{R}$; we sometimes call $\underline{R}$ a *database schema.*

A *Datalog* formula is a BSR formula of the form

$$\forall \underline{x}, \underline{y}. \bigwedge_{i=1}^{n} A_i(\underline{x}, \underline{y}) \rightarrow A_0(\underline{x}) \text{ also written as a } \textit{rule } A_0(\underline{x}) \;\leftarrow\; \bigwedge_{i=1}^{n} A_i(\underline{x}, \underline{y}),$$

where A_i is an atom (for $i = 0, 1, ..., n$) and $\underline{x}, \underline{y}$ are disjoint tuples of variables. Furthermore, A_0 is said the *head*, $\bigwedge_{i=1}^{n} A_i(\underline{x}, \underline{y})$ the *body* of the rule, and when $n = 0$, the Datalog rule is also called a *fact*. A set of Datalog rules is *semi-positive* if whenever a negative literal appears in the body of a rule, then the relation symbol of this literal is not the relation symbol of any literal which is the head of a non-trivial rule in the set. Another generalization of Datalog rules is that of *constraint Datalog rule* where, for some Σ-theory T, one quantifier-free $\Sigma(\underline{x}, \underline{y})$-formula may appear in its body. A set of Datalog (semi-positive or constraint Datalog) rules is *recursive* if some predicate symbol occur both in the heads and the bodies of the rules; otherwise, we say it to be *non-recursive*.

3 Security-Aware Transducers

For the rest of the paper, we fix a Σ-theory T and a relational signature $\underline{R} := In \cup Out \cup DB \cup Policy$ such that $X \cap Y = \emptyset$ and $X \cap \Sigma = \emptyset$ for $X, Y \in \{In, Out, DB, Policy\}$.

Definition 1. *A* (cumulative and policy-aware) transducer (over $(\Sigma, \underline{R})$) *is a tuple* $(\underline{past}, \tau, \omega, \pi)$ *where*

- $\underline{past} = past_{i_1}, ..., past_{i_n}$ *for* $In = \{r_{i_1}, ..., r_{i_n}\}$ *is a sequence of fresh predicate symbols, i.e.* $\underline{past} \cap (\Sigma \cup \underline{R}) = \emptyset$*,*

– *given a finite database over the database schema $I \subseteq In$ of the form*

$$\bigwedge_{r_i \in I} (\forall \underline{x}.r_i(\underline{x}) \leftrightarrow \bigvee_{j=1}^{n_i} \underline{x} = \underline{c}_i^j),$$

where c_i^j are constant symbols, $\tau(I)$ is a finite set of (cumulative) transition*s of the form*

$$\forall \underline{x}.past'_{r_i}(\underline{x}) \leftrightarrow \bigvee_{j=1}^{n_i} \underline{x} = \underline{c}_i^j \vee past_{r_i}(\underline{x}),$$

for each $r_i \in I$ and

$$\forall \underline{x}.past'_{r_i}(\underline{x}) \leftrightarrow past_{r_i}(\underline{x}),$$

for each $r_i \in In \setminus I$. Let τ be the set of cumulative transitions for every finite instance over every possible sub-set of In;

– *ω is a a finite set of semi-positive non-recursive rules of the form $A_0 \leftarrow \bigwedge_{i=1}^n A_i$ such that (i) A_0 is an Out-atom, (ii) A_i is a $(In \cup \underline{past} \cup DB)$-literal (for $i = 1, ..., n$), and (iii) every variable appearing in a rule must occur in at least one positive literal;*

– *π is a finite set of (possibly recursive) constraint Datalog rules of the form $A_0 \leftarrow \bigwedge_{i=1}^n A_i \wedge \psi$ such that (i) A_0 is a Policy-atom, (ii) A_i is a $(DB \cup Policy \cup \underline{past})$-atom, for $i = 1, ..., n$, (iii) ψ is a quantifier-free Σ-formula, and (iv) the set of variables occurring in A_0 are a sub-set of the variables occurring in $\bigwedge_{i=1}^n A_i$.*

To understand why what we have just defined is a transducer, recall that (constraint) Datalog rules define queries on databases; which, in turn, can be seen as mappings associating a so-called answer relation to each database (see, e.g., [6] for details). In this perspective, we can consider the union of (non-recursive) semi-positive rules in ω with the constraint (Datalog) rules in π as defining a mapping between databases over $In \cup \underline{past} \cup DB$ to databases over $Out \cup \underline{past}$. By taking $Policy = \emptyset$ and T to be the empty theory, it is easy to see that our notion of transducer reduces to that of relational Spocus transducer in [17]. Notice that the set π may contain recursive (constraint) Datalog rules. Recursion is crucial to express important mechanisms in policy management such as delegation. So, although, recursion complicates the reduction of verification problems for cumulative and policy-aware transducers to logical satisfiability problems (see Section 5 below), it is of paramount importance for naturally expressing several patterns of policy management.

Below, given a cumulative and policy-aware transducer $(\underline{past}, \tau, \omega, \pi)$ over $(\Sigma, \underline{R})$ and a $(\Sigma \cup \underline{R} \cup \underline{past})$-formula φ, the formula obtained by replacing each symbol $s \in In \cup Out \cup \underline{past}$ occurring in φ with a fresh symbol s^j and each symbol $p' \in \underline{past}'$ with fresh symbols p^{j+1} will be denoted by $ren(\varphi, j)$ for $j \geq 0$.

Definition 2. *Let* $(\underline{past}, \tau, \omega, \pi)$ *be a cumulative and policy-aware transducer over* $(\Sigma, \underline{R})$ *and db be a finite instance over DB. The sequence* $\mathcal{I}_1, \mathcal{O}_1; ...; \mathcal{I}_n, O_n$ *is a* (finite) run *(with length* n*) of* $(\underline{past}, \tau, \omega, \pi)$ *iff the formula*

$$\begin{array}{l} db \wedge \bigwedge_{p \in \underline{past}} \forall \underline{x}. p^0(\underline{x}) \leftrightarrow false \wedge \\ \bigwedge_{j=1}^{n} ren(\mathcal{I}_j, j) \wedge ren(\tau(\mathcal{I}_j), j) \wedge ren(\omega, j) \wedge ren(\pi, j) \wedge ren(\mathcal{O}_j, j) \end{array} \qquad (1)$$

is T*-satisfiable, where* $\mathcal{I}_1, ..., \mathcal{I}_n$ *is a finite sequence where each* $\mathcal{I}_j$ *is an instance over In, for* $j = 1, ..., n$*, called the* input sequence*, and* $\mathcal{O}_1, ..., \mathcal{O}_n$ *is a finite sequence where each* $\mathcal{O}_j$ *is an instance over Out, for* $j = 1, ..., n$*, called the* output sequence*.*

Given a cumulative and policy-aware transducer $(\underline{past}, \tau, \omega, \pi)$ over $(\Sigma, \underline{R})$ and a $(\Sigma \cup \underline{R} \cup \underline{past})$-formula φ, $\mathcal{R}_1; ...; \mathcal{R}_n$ be a run of the transducer, and $Log \subseteq (In \cup Out)$; then, $prj(Log, \mathcal{R}_1; ...; \mathcal{R}_n)$ is a finite sequence of formulae obtained from $\mathcal{R}_1; ...; \mathcal{R}_n$ by forgetting all those finite instances over $(In \cup Out) \setminus Log$. A *goal* is a BSR formula of the form

$$\exists \underline{x}. \bigwedge_{j=1}^{n} A_j(\underline{x})$$

where A_j is an *Out*-literal, for $j = 1, ..., n$.

Definition 3. *Let* $(\underline{past}, \tau, \omega, \pi)$ *be a cumulative and policy-aware transducer over* $(\Sigma, \underline{R})$ *and* $Log \subseteq In \cup Out$*. We define the following two verification problems for* $(\underline{past}, \tau, \omega, \pi)$*:*

Goal reachability: *does a given goal* γ *hold in the last output of some run of* $(\underline{past}, \tau, \omega, \pi)$*?*

Log validity: *given a finite sequence* $\mathcal{L}_1, ..., \mathcal{L}_n$ *where each* $\mathcal{L}_j$ *is a finite instance over Log for* $j = 1, ..., n$*, does there exist a finite run* $\mathcal{S}_1; ...; \mathcal{S}_n$ *of* $(\underline{past}, \tau, \omega, \pi)$ *such that* $\mathcal{L}_1, ..., \mathcal{L}_n = prj(Log, \mathcal{S}_1; ...; \mathcal{S}_n)$*?*

Goal reachability consists of checking whether a set of goals can be reached by some run of the transducer. This verification problem is a first sanity check on the design of the business model underlying the transducer as the latter is usually conceived to reach a certain goal, e.g., delivering a product provided that certain conditions are met. Log validation consists of checking whether a given log sequence can be generated by some input sequence. This problem arises, for example, when the transducer of a supplier is allowed to run on a customer's site for efficiency or convenience. The trace provided by the log allows the supplier to validate the transaction carried out by the customer.

3.1 Example

We consider a business model where a customer wants to buy a book, is billed for it, pays, and then takes delivery and a discount voucher for his/her next purchase if he/she is a preferred customer and is an employee of an accredited company. One kind of preferred customers are those affiliated to a organization EOrg, which issues credentials to certify that a person is one of its members. The business process has a database for accredited companies which are trusted by the business model to pass to other companies the fact of being accredited as well as the possibility of declaring other companies being accredited. The identification of an employee of a company is done by issuing a certificate signed by the company. Companies are organized according to a certain hierarchy which, for the purpose of this paper, can be assumed to be a partial order. The customer, willing to get a discount voucher should provide suitable valid credentials that he/she is a preferred customer (e.g., a member of EOrg) and an employee of an accredited company.

We need the theory of partial orders T_{po} to specify the trust relationship between companies. The signature Σ_{po} of T_{po} consists of the binary relation $\mathsf{is\_trusted\_by}$ (written infix) and its axioms are the following three sentences:

$$\forall x, y, z.(x \ \mathsf{is\_trusted\_by}\ y \wedge y \ \mathsf{is\_trusted\_by}\ z \rightarrow x \ \mathsf{is\_trusted\_by}\ z),$$
$$\forall x.x \ \mathsf{is\_trusted\_by}\ x \text{, and } \forall x, y.(x \ \mathsf{is\_trusted\_by}\ y \wedge y \ \mathsf{is\_trusted\_by}\ x \rightarrow x = y)\,.$$

Intuitively, x $\mathsf{is\_trusted\_by}$ y means that company y trusts company x with respect to the fact of being accredited and the possibility of delegating this capability.

The business model can be formalized by a policy-aware transducer (whose policies use the theory T_{po}) as follows. We fix the following relational signature $\underline{R} := \mathit{In} \cup \mathit{Out} \cup \mathit{DB} \cup \mathit{Policy}$, where $\mathit{In} := \{\mathit{order}, \mathit{pay}, \mathit{eorg}, \mathit{emplcert}\}$, $\mathit{Out} := \{\mathit{sendbill}, \mathit{deliver}, \mathit{sendvoucher}\}$, $\mathit{DB} := \{\mathit{price}, \mathit{available}, \mathit{accredited}\}$, $\mathit{Policy} := \{\mathit{preferred}, \mathit{employee}, \mathit{employeeof}\}$.

The business process has three databases *price*, *available*, and *accredited* storing the prices of books, their availability, and the set of (root) accredited companies, respectively. A customer interacts with the business system by inserting tuples in four input relations, *order*, *pay*, *eorg*, and *emplcert*. The system responds by producing output relations *sendbill*, *deliver*, and *sendvoucher* and it keeps track of the history of the business transaction using the state relations: past_{order}, past_{pay}, past_{eorg}, and $\mathit{past}_{emplcert}$. The state of the transducer cumulatively adds the tuples inserted into the input relations. The output rules ω of the transducer are the following:

$$\begin{aligned}
\mathit{sendbill}(X, Y, Z) &\leftarrow \mathit{order}(X, Z) \wedge \mathit{price}(X, Y) \wedge \neg \mathit{past}_{pay}(X, Y, Z)\\
\mathit{deliver}(X, Z) &\leftarrow \mathit{past}_{order}(X, Z) \wedge \mathit{price}(X, Y) \wedge\\
&\qquad \mathit{pay}(X, Y, Z) \wedge \neg \mathit{past}_{pay}(X, Y, Z)\\
\mathit{sendvoucher}(Z) &\leftarrow \mathit{past}_{deliver}(X, Z) \wedge \mathit{preferred}(Z) \wedge \mathit{employee}(Z)\,.
\end{aligned}$$

The first rule governs the sending of a bill whose amount is Y about a book X to a customer Z when an order is placed on X by Z, the price of X is Y, and Z has not already been paid for X. The second rule enables the delivery of a book X to Z if X has been ordered by Z and it is being paid the correct price by Z. Finally, the last rule is about sending the discount voucher to Z if a book has

input sequence	$\mathit{order}(\mathsf{Book}_1, \mathsf{Alice})$ $\mathit{order}(\mathsf{Book}_2, \mathsf{Bob})$ $\mathit{eorg}(\mathsf{Alice})$	$\mathit{pay}(\mathsf{Book}_1, 8, \mathsf{Alice})$ $\mathit{pay}(\mathsf{Book}_2, 15, \mathsf{Bob})$ $\mathit{emplcert}(\mathsf{Alice}, \mathsf{Comp}_3)$	$\mathit{eorg}(\mathsf{Bob})$
output sequence	$\mathit{sendbill}(\mathsf{Book}_1, 8, \mathsf{Alice})$ $\mathit{sendbill}(\mathsf{Book}_2, 15, \mathsf{Bob})$	$\mathit{deliver}(\mathsf{Book}_1, \mathsf{Alice})$ $\mathit{deliver}(\mathsf{Book}_2, \mathsf{Bob})$	$\mathit{sendvoucher}(\mathsf{Alice})$

Table 1. A run of a policy-aware transducer

been delivered to Z and this last is both a preferred customer and an employee of an accredited company. Finally, the policy rules π of the transducer are the following:

$$\begin{aligned}
\mathit{preferred}(Z) &\leftarrow \mathit{past}_{eorg}(Z)\\
\mathit{employee}(Z) &\leftarrow \mathit{past}_{emplcert}(Z, U) \wedge \mathit{employeeof}(Z, U)\\
\mathit{employeeof}(Z, F) &\leftarrow \mathit{accredited}(F)\\
\mathit{employeeof}(Z, U) &\leftarrow \mathit{employeeof}(Z, F) \wedge U \ \mathsf{is\_trusted\_by}\ F
\end{aligned}$$

The first rule simply establishes whether Z is a preferred customer by checking if the credential saying that Z is an EOrg member has been presented. (There can be other rules of this kind once the business model establishes that customers having a certain affiliation become preferred customers.) The last three rules check if Z has presented a certificate saying that he is an employee of a company which is in the suitable trust relationship with some accredited company. Notice the use of recursion to express delegation of the capability of certifying the fact of being an employee of an accredited company to some of its units. Since *employeeof* is a recursive predicate, $\omega \cup \pi$ is not a set of non-recursive semi-positive rules and, hence, the associated transducer is not spocus; all the techniques developed in [17] cannot be used to investigate verification problems for this business process. Also the results in [14] cannot be used here as the rules considered in that work (which allow for non-cumulative transducers) are assumed to be non-recursive. An interesting line of future work is to allow for non-cumulative rules also for the policy-aware transducers considered in this paper (i.e. in presence of recursive rules for policy specification).

Table 1 shows an example of a run for the transducer specified above. We have assumed that $\mathit{price}(\mathsf{Book}_1, 8)$, $\mathit{price}(\mathsf{Book}_2, 15)$, $\mathit{accredited}(\mathsf{Comp}_1)$, Comp_3 $\mathsf{is\_trusted\_by}$ Comp_2, and Comp_2 $\mathsf{is\_trusted\_by}$ Comp_1 . Now, if we take $Log = Out$, then the instance of the log validity problem for the run in Figure 1 reduces to the checking T_{po}-satisfiability of the following formula (only an excerpt is shown for the sake of conciseness):

$$\begin{pmatrix}
\forall x, y.\mathit{price}(x, y) \leftrightarrow ((x = \mathsf{Book}_1 \wedge y = 8) \vee (x = \mathsf{Book}_2 \wedge y = 15)) \wedge \\
\forall x.\mathit{accredited}(x) \leftrightarrow x = \mathsf{Comp}_1 \hfill \wedge \\
(\mathsf{Comp}_3\ \mathsf{is\_trusted\_by}\ \mathsf{Comp}_2) \wedge (\mathsf{Comp}_2\ \mathsf{is\_trusted\_by}\ \mathsf{Comp}_1)
\end{pmatrix} \wedge$$

$$\begin{pmatrix}
\forall x, y.\mathit{past}^0_{order}(x, y) \leftrightarrow \mathit{false} \wedge \forall x, y, z.\mathit{past}^0_{pay}(x, y, z) \leftrightarrow \mathit{false} \wedge \\
\forall x.\mathit{past}^0_{eorg}(x) \leftrightarrow \mathit{false} \wedge \forall x, y.\mathit{past}^0_{emplcert}(x, y) \leftrightarrow \mathit{false}
\end{pmatrix} \wedge$$

$$\forall x, y.(order^0(x,y) \leftrightarrow (x = \mathsf{Book_1} \wedge y = \mathsf{Alice}) \vee (x = \mathsf{Book_2} \wedge y = \mathsf{Bob})) \wedge$$
$$\forall x, y.(eorg^0(x) \leftrightarrow x = \mathsf{Alice} \wedge$$
$$\forall x, y, z.(order^0(x,z) \wedge price^0(x,y) \wedge \neg past^0_{pay}(x,y,z) \rightarrow sendbill^0(x,y,z)) \wedge ...$$

From the piece formula above, it is not difficult to see that the output sequence (at time instant 0) of the transducer is exactly the one depicted in the first column, second line of Table 1.

4 An extension of the BSR class

Since the goal reachability and the log validity problems will be reduced to the satisfiability of a certain class of first-order formulae (see Section 5), we define such a class of formulae and study their decidability.

Let T be Σ-theory, we are interested in studying the T-satisfiability of formulae of the form

$$\exists \underline{x}.\forall \underline{y}.\varphi(\underline{x}, \underline{y})$$

where φ is a quantifier-free $\Sigma^{\underline{R}}$-formula and $\Sigma^{\underline{R}} := \Sigma \cup \underline{R}$ such that $\Sigma \cup \underline{R} = \emptyset$. Sentences of this form are called $BSR(\Sigma^{\underline{R}})$-sentences. If T is the empty theory, then $BSR(\Sigma^{\underline{R}})$-sentences are BSR formulae.

We show the decidability of $BSR(\Sigma^{\underline{R}})$-sentences under suitable hypotheses on the theory T in two steps. First, we eliminate universal quantifiers by identifying finitely many instances which are sufficient for (un-)satisfiability checking (under suitable hypotheses). Second, we show the decidability of the resulting quantifier-free formulae.

Lemma 1 (Instantiation). *Let T be a locally finite Σ-theory and the class of models of T be closed under sub-structures. Furthermore, let $\underline{R}$ be a finite set of predicate symbols such that $\Sigma \cap \underline{R} = \emptyset$. The $BSR(\Sigma^{\underline{R}})$-sentence*

$$\exists \underline{x}.\forall \underline{y}.\varphi(\underline{x}, \underline{y})$$

is satisfiable iff it is satisfiable in a finite model iff the quantifier-free formula

$$\exists \underline{x}. \bigwedge_{\sigma} \varphi(\underline{x}, \underline{y}\sigma)$$

is satisfiable, where σ ranges over the substitutions mapping the variables $\underline{y}$ into the set of representative $\Sigma(\underline{x})$-terms and $\underline{y}\sigma$ denotes the simultaneous application of σ to the variables of the tuple $\underline{y}$.

We are left with the problem of showing the decidability of the satisfiability of quantifier-free formulae over $\Sigma^{\underline{R}}$.

Theorem 1 (Decidability). *Let T be a locally finite Σ-theory whose class of models is closed under sub-structures and the SMT(T) problem be decidable; $\underline{R}$ be a finite set of predicate symbols such that $\Sigma \cap \underline{R} = \emptyset$. Then, the satisfiability of $BSR(\Sigma^{\underline{R}})$-sentences is decidable.*

The assumption of the background theory T to be locally finite, at first, seems to be quite restrictive. However, the constraint domains considered as relevant for trust management in [12] have quite simple algebraic structures so as to allow for efficient algorithm to answer policy access queries. We believe that locally finite theories can be put to productive use for the verification of policy-aware transducers as suitable abstractions of this class of constraint domains.

5 Decidability of goal reachability and log validity

Before describing the reduction of the goal reachability and the log validity problems to the satisfiability of the extension of the BSR class considered in the previous section, we re-consider how semi-positive (non-recursive) rules and (constraint) Datalog rules define queries on databases, or, equivalently, mapping of databases to answer relations. As already observed in Section 3, this is crucial to characterize the input-output behavior of the class of relational transducers considered in this paper.

5.1 Constraint Datalog queries

A *database query* is a mapping associating to each database an (answer) relation. A set of (constraint) Datalog rules can be seen as a way to implicitly define queries as follows. Let T be a Σ-theory admitting quantifier-elimination and $\underline{R}$ be a database schema such that $\Sigma \cap \underline{R} = \emptyset$. Let $\Pi = \omega \cup \pi$ be a finite set of rules such that $\underline{R} = \underline{B} \cup \underline{E}$ for some cumulative and policy-aware transducer $(\underline{past}, \tau, \omega, \pi)$, where $\underline{B}$ is the set of predicate symbols occurring only in the body of the rules in Π and not in their heads, and $\underline{E}$ is the set of predicate symbols occurring both in the body and head of the rules in Π. Consider a rule $\rho \in \Pi$ of the form $e(\underline{x}) \leftarrow \bigwedge_{i=1}^{n} b_i(\underline{x}, \underline{y}) \wedge \psi(\underline{x}, \underline{y})$ where $e \in \underline{E}$, the predicate symbol of b_i is in $\underline{R}$ (for $i = 1, ..., n$), and ψ is a quantifier-free $\Sigma(\underline{x}, \underline{y})$-formula, the *constraint query associated to* ρ is the formula

$$\exists \underline{y}. \bigwedge_{i=1}^{n} b_i(\underline{x}, \underline{y}) \wedge \psi(\underline{x}, \underline{y}).$$

If b_i contains only symbols in $\underline{B}$ (as it is the case of the semi-positive non-recursive rules in ω), then it is possible to replace each occurrence of such symbols with disjunctions of conjunctions of equalities (recall the definition of finite instance in Section 2). In this way, the resulting (existentially quantified) formula turns out to be a $\Sigma(\underline{x}, \underline{y})$-formula and the decidability result above (i.e. Theorem 1) can be used. This is the key insight used in reducing both log validity and goal reachability to the satisfiability of BSR formulae in [17] which in our framework corresponds to the case there $\pi = \emptyset$. However, this is not enough for the class of transducers considered in this paper because of the presence of the constraint Datalog rules in Π which may be recursive (we have already argued that this complication is necessary to be able to express certain patterns

```
function constraintFixPoint($F$ : constraint facts,  $R$ : constraint Datalog rules)
1  $results \longleftarrow F$; $Changed \longleftarrow true$;
2  while $Changed$ do
3      $Changed \longleftarrow false$
4      foreach $rule \in R$ do
5          foreach $tuple$ of constraint facts constructed from $results$ do
6              $newres \longleftarrow$ constraint facts obtained by
                              constraint rule application between $rule$ and $tuple$
7              foreach $fact \in newres$ do
8              if ($results \not\models fact$) then $results \longleftarrow results \cup \{fact\}$;
9              $Changed \longleftarrow true$;
10             end
11         end
12     end
13 end
14 return $results$
```

Fig. 1. Least fix-point computation of constraint Datalog rules

of policy management such as delegation). As a consequence, we need a fix-point characterization for the queries associated to a set Π of possible recursive rules. To this end, we proceed as follows. First, we regard each finite instance of $r \in In \cup Out \cup DB$ as a constraint fact, i.e. $\forall r(\underline{x}) \leftrightarrow \bigvee_{j=1}^{m} \underline{x} = \underline{c}^j$ is transformed into the Datalog rule $r(\underline{x}) \leftarrow \bigvee_{j=1}^{m} \underline{x} = \underline{c}^j$, where $\underline{x}$ is a tuple of variables and $\underline{c}^j$ is a tuple of constants, for $j = 1, ..., m$. Let $r_0(\underline{x}) \leftarrow \bigwedge_{i=1}^{n} r_i(\underline{x}) \wedge \psi_0(\underline{x})$ be a constraint Datalog rule in Π for $r_i \in \underline{R}$ and $i = 0, 1, ..., n$ and $r_i(\underline{x}_{k_i}) \leftarrow \psi_i(\underline{x}_{k_i})$ be a constraint fact for ψ_i be a $\Sigma(\underline{x}_i)$-quantifier-free formula, k_i the arity of r_i, and $i = 1, ..., n$. A *constraint rule application* produces $m \geq 0$ facts of the form $r_0(\underline{x}) \leftarrow \psi'_j(\underline{x})$ where ψ'_j is a quantifier-free $\Sigma(\underline{x})$-formula for $j = 1, ..., m$ ($m \geq 0$) and $\bigvee_{j=1}^{m} \psi'_j$ is equivalent (by the elimination of quantifiers in T) to the formula

$$\exists \underline{y}.(\bigwedge_{i=1}^{n} \psi_i(\underline{x}_{k_i}) \wedge \psi_0(\underline{x})),$$

where $\underline{y}$ is the tuple of variables occurring in the body of the rule but not in the head. The least fix-point of a set of constraint Datalog rules can be computed by the algorithm in Figure 1. The function constraintFixPoint terminates when all derivable new facts are implied by previously derived facts.[1] The requirement

[1] The test at line 8 can be reduced (by eliminating quantifiers) to a T-satisfiability test on a conjunction of two quantifier-free Σ-formulae. To understand how, consider two facts $r_1(\underline{x}) \leftarrow \psi_1(\underline{x})$ and $r_2(\underline{y}) \leftarrow \psi_2(\underline{y})$. To check that latter is a logical consequence of the former (modulo T), it is sufficient to check the T-validity of $\forall \underline{x}.\psi_1(\underline{x}) \rightarrow \forall \underline{y}.\psi_2(\underline{y})$. This, reasoning by refutation, is equivalent to check the T-unsatisfiability of the negation of the previous formula, which, by eliminating quantifiers can be reduced to a quantifier-free formula.

that T admits elimination of quantifiers is not sufficient to guarantee termination of the algorithm in Figure 1. We need the following notion which is adapted from [16].

Definition 4. *Let T be a Σ-theory and $\underline{x}$ be a finite set of variables. If for every (possibly infinite) T-satisfiable set S of $\Sigma(\underline{x})$-constraint, there exists a finite subset $S_{fin} \subseteq S$ such that for every ψ in S, there exists a ψ' in S_{fin} for which $\forall \underline{x}.\psi(\underline{x}) \to \psi'(\underline{x})$ is T-valid, then T is said to be* constraint compact*. If S_{fin} is effectively computable from S, then T is said to be* effectively constraint compact *(in the following, when we say 'constraint compact', we in fact always mean 'effectively constraint compact').*

Constraint compactness is a sufficient condition for the termination of the algorithm in Figure 1.

Theorem 2 ([16]). *Let T be a constraint compact theory. Then, the algorithm in Figure 1 terminates for every query.*

The sketch of the proof is as follows. By contradiction, assume that the algorithm does not terminate. Thus, every iteration of the loop produces at least one fact which is not a consequence of those which were already derived. Since there are only finitely many different predicate symbols, there must be at least one such symbol for which the algorithm derives an infinite set S of facts. But, T is assumed to be constraint compact and so we can replace S with a finite subset S_{fin} such that the facts in $S \setminus S_{\mathit{fin}}$ are consequence of those in S_{fin}. As the algorithm checks for the logical consequence of newly derived facts before adding it to the resulting set of derived facts, all the facts in $S \setminus S_{\mathit{fin}}$ should not be considered: a contradiction.

Interestingly, locally finite theories are also constraint compact.

Proposition 1. *If T is an (effectively) locally finite theory, then it is also (effectively) constraint compact.*

Proof. Let S be a set of constraints over a finite set $\underline{x}$ of variables. Then, since the theory T is locally finite, there exists a finite set *RTerms* of $\Sigma(\underline{x})$-terms such that for every term u in S, there exists $t \in \mathit{RTerms}$, $T \models u = t$. By using the terms in *RTerms* and the finite signature Σ, it is possible to build only finitely many distinct $\Sigma(\underline{x})$-atoms $\psi_1(\underline{x}), ..., \psi_n(\underline{x})$ (also called representative atoms) such that for any further $\Sigma(\underline{x})$-atom $\psi(\underline{x})$, there exists $i \in \{1, ..., n\}$, $T \models \forall \underline{x}.\psi_i(\underline{x}) \leftrightarrow \psi(\underline{x})$. If we consider a $\Sigma(\underline{x})$-constraint $\phi(\underline{x})$, it is possible to collect all the atoms occurring in it, say $\{\alpha_1, ..., \alpha_m\}$; for each α_j (for $j = 1, ..., m$), find the equivalent representative atom ψ_{k_j}, and then substitute each α_j with the corresponding ψ_{k_j} in ϕ and then eliminating duplicates. The result will be a constraint taken from the finitely many distinct conjunctions of literals built out of the finitely many representative atoms $\psi_1(\underline{x}), ..., \psi_n(\underline{x})$. This can be generalized to sets of constraints in the obvious way. Clearly, this implies the constraint compactness of T. □

As an immediate consequence of the last two facts we obtain the following result.

Corollary 1. *Let T be an (effectively) locally finite theory. Then, the algorithm in Figure 1 terminates for every query.*

This will be a crucial ingredient in the reduction of goal reachability and log validity to the satisfiability of the BSR($\Sigma^{\underline{R}}$) class.

5.2 Reduction to satisfiability

We first consider the log validity problem as goal reachability can be reduced to this problem.

Lemma 2. *Let T be a locally finite Σ-theory admitting elimination of quantifiers. Furthermore, let $(\underline{past}, \tau, \omega, \pi)$ be a cumulative and policy-aware transducer over $(\Sigma, \underline{R})$, db be a database over DB, and Log $\subseteq$ In $\cup$ Out. The log validity problem for $(\underline{past}, \tau, \omega, \pi)$ reduces to the T-satisfiability problem of an effectively computable BSR($\Sigma^{\underline{R}}$)-formula.*

Proof. The proof is along the lines of Theorem 3.1 in [17] for Spocus transducers. A log $L_1, ..., L_n$ is valid if there exists an input sequence $I_1, ..., I_n$ generating it. We view $I_1, ..., I_n$ as database over the set of predicate symbols obtained by making n copies of each relation $r \in \mathit{In}$ yielding $r_1, ..., r_n$. To state that the input sequence $I_1, ..., I_n$ yields the log $L_1, ..., L_n$, we require that the relations in $I_1, ..., I_n$ recorded by the log have the values specified by $L_1, ..., L_n$ together with the relations in *Out* determined by $I_1, ..., I_n$. For input relations, the situation is easy (we analyze concrete cases as the generalization are straightforward). For example, suppose that L_j requires that a tuple $\underline{c}$ belongs to the relation r_j in the input sequence; this can be stated by the following BSR($\underline{R}$) formula:

$$\exists \underline{x}.(r_j(\underline{x}) \wedge \underline{x} = \underline{c}). \tag{2}$$

For example, if the log specifies that r_j contains the tuples $\underline{c}$ and $\underline{d}$, the inclusion of r_j in the log is state by the following BSR($\underline{R}$) formula:

$$\forall \underline{x}.(r_j(\underline{x}) \rightarrow (\underline{x} = \underline{c} \vee \underline{x} = \underline{d})). \tag{3}$$

For relation in the output sequence, we need to resort to Corollary 1 above: the fix-point algorithm in Figure 1 terminates on all queries and generates a (finite) set of constraint facts. Then, for each fact of the form $r_j(\underline{x}) \leftarrow \psi_j^i(\underline{x})$ in the output set, we take the disjunction of each ψ_j^i for $i \geq 0$. Let $\varphi_j(\underline{x})$ be the resulting quantifier-free $\Sigma(\underline{x})$-formula. The remainder is similar to the case of input relations. Requiring that a tuple belongs to the relation r_j in the output sequence can be expressed by a formula similar to (2) except that r_j is replaced by φ_j obtained by the fix-point computation. Similarly, if the log specifies that r_j contains the two tuples $\underline{c}$ and $\underline{d}$, the inclusion of r_j in the log is stated by a formula which is similar to (3) except that r_j is replaced by φ_j obtained by the fix-point computation. It is easy to see that the resulting formula is a BSR($\underline{R}$) formula and its satisfiability is equivalent to the existence of an input sequence yielding the desired log. □

Before stating our main decidability results for log validity and goal reachability of cumulative and policy-aware transducers, we observe the following. An immediate consequence of Lemma 2 and Theorem 1 is the decidability of both verification problems for cumulative and policy-aware transducers whose Σ-theory is locally finite and admits elimination of quantifiers. Unfortunately, local finiteness and elimination of quantifiers are difficult to reconcile as the former is obtained for theories with simple "algebraic structure" while the latter is satisfied for those with rich "algebraic structure". To illustrate, we consider three theories. The theory whose signature contains only constants $c_1, ..., c_n$ and axiomatized by the following sentences:

$$\begin{aligned} c_i \neq c_j \text{ for } i, j = 1, ..., n \text{ and } i \neq j \\ \forall x. x = c_1 \vee \cdots x = c_n, \end{aligned}$$

is both locally finite and admits elimination of quantifier. The theory T_{lo} of linear orders whose signature contains the binary relation $\preceq$ and axiomatized by the following sentences:

$$\begin{aligned} \forall x, y, z.(x \preceq y \wedge y \preceq z \rightarrow x \preceq z) \qquad \forall x, y.(x \preceq y \wedge y \preceq x \rightarrow x = y) \\ \forall x. x \preceq x \qquad \forall x, y.(x \preceq y \vee y \preceq x) \end{aligned}$$

is locally finite, the SMT(T_{lo}) problem is decidable, but it does not admit elimination of quantifiers. The theory T_{dlo} of dense linear orders is the theory over the same signature of T_{lo} and obtained by adding the following two sentences to the set of axioms of T_{lo}:

$$\forall x, y.(x \prec y \rightarrow \exists z.(x \prec z \wedge z \preceq y)) \;\text{ and }\; \exists x, y. x \neq y\,,$$

where $t_1 \prec t_2$ abbreviates $t_1 \preceq t_2 \wedge t_1 \neq t_2$ for t_1, t_2 variables. The SMT(T_{dlo}) problem is decidable, T_{dlo} admits elimination of quantifiers, but it is not locally finite.

Fortunately, it is sometimes possible to reconcile local finiteness and quantifier elimination. For the theories T_{lo} and T_{dlo}, it is possible to show that a quantifier-free formula is T_{lo}-satisfiable iff it is T_{dlo}-satisfiable. So, to check the T_{lo}-satisfiability of BSR($\Sigma^{\underline{R}}$) sentences, we can use its being locally finite to apply the instantiation procedure underlying the proof of Lemma 2 while we can use the fact that T_{dlo} admits elimination of quantifiers to obtain the termination of the fix-point algorithm in Figure 1 (as we know that the T_{lo}-satisfiability of quantifier-free formulae is preserved). This phenomenon is not an accident but an application of the notion of model completeness of a theory.

A Σ-theory T is *model complete* iff, for all models $\mathcal{M}, \mathcal{N}$ of T, if $\mathcal{M}$ is a sub-structure of $\mathcal{N}$, then $\mathcal{M}$ is also an *elementary* sub-structure of $\mathcal{N}$, i.e. for every Σ-formula $\varphi(\underline{x})$ and all elements $\underline{a}$, we have that $\mathcal{M} \models \varphi(\underline{a})$ iff $\mathcal{N} \models \varphi(\underline{a})$. Intuitively, the elementary sub-model $\mathcal{M}$ of a model $\mathcal{N}$ preserves the set of satisfiable first-order formulae (and hence of quantifier-free formulae in particular). It is possible to show that T_{dlo} is model complete (see, e.g., [4]). The key property (see Remark 3.5.6 in [4]) is that $\mathcal{M}$ is a sub-model of the set of universal sentences which are logical consequences of T_{dlo} (if T is a theory, the set of universal sentences which are T-valid is denoted with $T^{\forall}$) iff $\mathcal{M}$ is a sub-model of some model of $T_{dlo}^{\forall}$. (The validity of universal sentences is the dual of the satisfiability of quantifier-free formulae.) Now, since $T_{dlo}^{\forall} = T_{lo}^{\forall}$ (see [10]), we have formally justified the use of T_{lo} in Lemma 2 and that of T_{dlo} to eliminate quantifiers in the algorithm of Figure 1. Notice that the class of models of T_{lo} is closed under sub-structures since its axioms are universal sentences (see, e.g., [4]). Furthermore, these observations constitute the sketch of the proof of our first (main) decidability result about the verification of cumulative and policy-aware transducers.

Theorem 3. *Let $(\underline{past}, \tau, \omega, \pi)$ be a cumulative and policy-aware transducer over $(\Sigma, \underline{R})$, db be a database over DB, and Log $\subseteq$ In $\cup$ Out. If (a) T is a locally finite theory such that the SMT(T) problem is decidable and its class of models is closed under sub-structures, (b) $T_* \supseteq T$ is a model complete theory admitting elimination of quantifiers, and (c) $T^{\forall} = T_*^{\forall}$, then the log validity problem for $(\underline{past}, \tau, \omega, \pi)$ is decidable.*

We are now ready to state and prove our second (main) decidability result.

Theorem 4. *Let $(\underline{past}, \tau, \omega, \pi)$ be a cumulative and policy-aware transducer over $(\Sigma, \underline{R})$, db be a database over DB, and Log $\subseteq$ In $\cup$ Out. If (a) T is a locally finite theory such that the SMT(T) problem is decidable and its class of models is closed under sub-structures, (b) $T_* \supseteq T$ is a model complete theory admitting elimination of quantifiers, and (c) $T^{\forall} = T_*^{\forall}$, then the goal reachability problem for $(\underline{past}, \tau, \omega, \pi)$ is decidable.*

Proof. Again the proof is along the lines of Theorem 3.2 in [17] for Spocus transducers. First of all, observe that only runs of length two should be considered. To understand why this is so, consider an input sequence $I_1, ..., I_n$ with $n > 2$. Since outputs depend only on the current input, the database, and the state relations (storing the union of all previous inputs), the last output in the run of the transducer on $I_1, ..., I_n$ is the same as the last output on the run of the same transducer on the following input sequence of length 2: $(I_1 \cup \cdots I_{n-1}), I_n$. At this point, the problem is reduced to that of the satisfiability of a BSR($\underline{R}$) with the same technique described in the proof of Lemma 2. □

6 Conclusion

We have introduced a class of policy-aware relational transducers. We have studied the hypotheses under which log validity and goal reachability are decidable for this class of transducers by a reduction to the satisfiability problem of an extension of the BSR class of formulae.

There are two main lines of future work. First, we intend to study the decidability of those verification problems involving two or more transducers such as containment and equivalence [17]. Second, it would be interesting to see how to implement the algorithm for checking the satisfiability of the extension of the BSR class on top of state-of-the-art theorem provers or Satisfiability Modulo Theories solvers. The latter, in particular, with the recent interest in developing decision procedures for the BSR class (see, e.g., [5]) and techniques for quantifier instantiation (see, e.g., [9]) seem to be ideal candidates.

Acknowledgments. The work presented in this paper was partially supported by the FP7-ICT-2007-1 Project no. 216471, "AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures." We thank Silvio Ghilardi and the members of the AVANTSSAR project for many stimulating discussions.

References

1. Franz Baader and Silvio Ghilardi. Connecting many-sorted theories. *Journal of Symbolic Logic*, 72:535–583, 2007.
2. M. Y. Becker, C. Fournet, and A. D. Gordon. Security Policy Assertion Language (SecPAL). http://research.microsoft.com/en-us/projects/SecPAL/.
3. M. Y. Becker and S. Nanz. The Role of Abduction in Declarative Authorization Policies. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2008.
4. C.-C. Chang and J. H. Keisler. *Model Theory.* North-Holland, Amsterdam-London, third edition, 1990.
5. L. de Moura and N. Bjørner. Deciding effectively propositional logic using dpll and substitution set. In *Int. Joint Conference on Automated Reasoning*, 2008.
6. J. Van den Bussche. *Constraint Databases*, chapter Constraint databases, queries, and query languages. Springer, 2000.
7. J. DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
8. H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York-London, 1972.
9. Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 2009. To appear.
10. S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3-4), 2004.
11. Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of CSF 2008*, pages 149–162. IEEE Computer Society, 2008.
12. N. Li and J. C. Mitchell. Datalog with constraints: a foundation for trust management langauges. In *PADL'03*, pages 58–73, 2003.
13. S. Ranise. Towards Verification of Security-Aware E-Services. Extended version of the paper with the same title in FTP'09.
14. M. Spielmann. Verification of relational transducers for electronic commerce. In *19th ACM Symp. on Princ. of DB Systems (PODS).* ACM Press, 2000.
15. C. Tinelli and C. G. Zarba. Combining non-stably infinite theories. *Journal of Automated Reasoning*, 34(3), 2005.
16. D. Toman. Computing the well-founded semantics for constraint extensions of datalog. In *2nd Int. WS on Constraint Database Systems*, volume 1191 of *LNCS*, pages 64–79, 1997.
17. S. Abitebouland V. Vianu, B. Fordham, and Y. Yesha. Relational Transducers for Electronic Commerce. *J. of Comp. and Sys. Sciences*, 61:236–269, 2000.

A Fixed Point Representation of References

Susumu Yamasaki

Department of Computer Science, Okayama University, Okayama, Japan
yamasaki@momo.cs.okayama-u.ac.jp

Abstract. This position paper is concerned with the *reference* in computer science. We have a formal representation of lazy references in contrast to eager and failure ones. The representation problem is motivated by static analysis in Web accessibility. A fixed point theory is adopted for such an analysis.

1 Introduction

To make analyses in Web usability or accessibility, we aim at capturing the link situation on the Web sites and referential relations among Web site pages. For an apperception of the link structure, this position paper deals with static analysis of relations of *references* which are concretized as Web site pages. The total reference structure is described by a fixed point of an associated mapping for the structure. As regards static analysis, several frameworks have been well established. Hybrid logic, which involves both state-dependent and modal operators, is a formal system with logical meanings of states and worlds ([1, 2]). Relations between the events are discussed through predicates in classical and modal logic ([3, 9]). The event as the cause-and-effect relationship is made clear from the view of rule-based system ([21]). Correlation between action and knowledge has also been studied ([14]). A mathematical behaviour of action is formulated in [13], while action may be captured by modal logic ([6]). The agent technology style is current as in [15], where algebraic approach to process originates from [8, 12] such that a logical viewpoint is given in the paper ([10]). A multi-agent is well designed in terms of modal logic ([7]).

In this position paper, based on the first-order logic (or the propositional logic) analysis approach ([5]), we see a mathematical aspect of reference structures relevant to Web site pages with fixed point theory. A Web site page recursively includes page references, where the page is itself a reference from other site pages. So far we see that there is a simple structure for some page A as a primary one: A primary reference A (recursively) includes references $B_1, \ldots, B_n$, where A may be referred to by others, and some of $B_1, \ldots, B_n$ may not be available without any correct link. As regards how to make use of the references, we can think that:

– To visit the (page) reference is regarded as eager.
– Not to visit (but to see only the name of) the (page) reference is regarded as lazy.

– Non-available reference for visit is regarded as a failure.

Whether or not a (page) reference is visited is supposedly determined by the user (visitor). The primary reference (which the user now pays attention to and which includes other references in) is thus interpreted as:

(i) eager if all the included references are eager.
(ii) lazy if it never occurs as a primary one such that it is designated as lazy, or if it is a primary one where at least one included reference is not eager and other included references are eager or lazy.
(iii) a failure if it is not available as a primary one, or if a primary reference with at least one included reference is a failure.

Note that the primary reference is interpreted as eager if it contains no reference. The classification of eager and lazy references for this case looks like the standard evaluation about call-by-value (eager) and call-by-name (lazy) modes of [17]. We then have a problem to see what set of lazy references is. The set of all considerable references is still finite, but it must be large enough to want to have a treatment to cover the case that the set may be countably infinite. A fixed point theory for the complete lattice is a technique as in [11, 18], to be incorporated into analysis and classification of eager and lazy reference sets, where the references are organized likely by recursive rule structures of the form: the reference including reference sequences.

2 Representation of References

In this paper, we consider recursive structures of references, which are given as a set of finite or countably infinite rules of the form $A \rhd A_1 \dots A_l$ $(l \geq 0)$, where A, A_i are *references*. A is the head, while $A_1 \dots A_l$ is the successor (sequence). We suppose in a set of rules that each head is followed by a unique successor.

Syntactically, we assume:

(i) a set P of rules of the form $A \rhd A_1 \dots A_l$ $(l \geq 0)$ where any two rules with the same head, $A \rhd B_1 \dots B_m$ and $A \rhd C_1 \dots C_n$, have the same successor, and
(ii) a set B_P of all references occurring in the set P.

The interpretation of references is defined as *eager*, *lazy* and a *failure*: Assume a set P of rules. Given a set L, we have inferences to inductively define the predicates *eager* and $lazy_L$ which are mutually exclusive:

$$\text{(ir1)} \quad \frac{A\rhd \text{ is in } P}{eager(A)}$$

$$\text{(ir2)} \quad \frac{\begin{array}{l} A \rhd A_1 \dots A_m \text{ is in } P \quad (m > 0) \\ \text{for all } A_i \ (1 \leq i \leq m),\ eager(A_i) \end{array}}{eager(A)}$$

$$\text{(ir3)}\quad \frac{\begin{array}{l}A \text{ does not occur in the head of any rule}\\ A \text{ is in } L\end{array}}{\mathit{lazy}_L(A)}$$

$$\text{(ir4)}\quad \frac{\begin{array}{l}A \rhd A_1 \dots A_m \text{ is in } P \quad (m > 0)\\ \text{for all } A_j \quad (1 \leq j \leq m),\ \mathit{eager}(A_j) \text{ or } \mathit{lazy}_L(A_j)\\ \text{for some } A_k\ (1 \leq k \leq m),\ \mathit{not\ eager}(A_k)\end{array}}{\mathit{lazy}_L(A)}$$

Semantically, we say that:

(i) If $\mathit{eager}(A)$, the reference A is eager.
(ii) If $\mathit{lazy}_L(A)$, the reference A is lazy.
(iii) If neither $\mathit{eager}(A)$ nor $\mathit{lazy}_L(A)$, the reference A is a failure.

Example 1. Assume a set P containing: (i) $A \rhd B$, and (ii) $B \rhd A\ C$. Note that neither a rule $A\rhd$ nor a rule $A \rhd B, C$ can be included into the set P, as long as the rule $A \rhd B$ (with the head A) is in P. What set of references may be lazy? To see it, we have exhaustive cases:

(1) A, B, C are failures, unless there is some lazy reference.
(2) C may be lazy, whether or not both of A and B are lazy. Neither A nor B can be lazy, if C still remains to be a failure.
(3) For A and C to be lazy, all the A, B, C are lazy. Similarly for B and C to be lazy, all are lazy.

A mapping $T_P : 2^{B_P} \to 2^{B_P}$ is defined to be

$$T_P(I) = \{A \mid \exists A \rhd A_1 \dots A_l \in P.\ A_1, \dots, A_l \in I\}.$$

Note that the mapping T_P is similar to the mapping associated with a logic program ([11]), such that it collects eager references based on the set I of eager references. Such a mapping is often adopted. As easily seen, if $I \subseteq J$, then $T_P(I) \subseteq T_P(J)$, that is, T_P is monotone. In what follows, we have the notation:

$$T_P^n(I) = \begin{cases} I & (n = 0) \\ T_P(T_P^{n-1}(I)) & (n > 0) \end{cases}$$

for a subset $I \subseteq B_P$. The mapping T_P is continuous: For any ω-chain $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots$,

$$\cup_{k\in\omega}\ T_P(I_k) = T_P(\cup_{k\in\omega}\ I_k).$$

Thus T_P has the least fixed point, $\cup_{n\in\omega}\ T_P^n(\emptyset)$, which is denoted by $lfp(T_P)$.

The following mapping looks like the one for logic programs with negation (as in [16, 19, 20]), but the present usage is not relevant to the treatment of negations in 3-valued logic. To capture the set of lazy references, we make use of the following mapping S_P. With respect to a subset $K \subseteq B_P$,

$$\begin{aligned} P[K] = \{A \rhd A_1 \dots A_m \mid \\ \exists A \rhd A_1 \dots A_m B_1 \dots B_n \in P\ (m \geq 0, n \geq 0).\ B_1, \dots, B_n \in K\}. \end{aligned}$$

Note that $P[\emptyset] = P$. A mapping $S_P : 2^{B_P} \to 2^{B_P}$ is defined to be

$$S_P(K) = \cup_{j\in\omega}\ T^j_{P[K]}(\emptyset) = lfp(T_{P[K]}).$$

The set $S_P(K)$ denotes the collection of eager and lazy references based on the set K of lazy references. It follows that $S_P(\emptyset) = lfp(T_{P[\emptyset]}) = lfp(T_P)$. When $J \subseteq K$, $A \in T^i_{P[J]}(\emptyset) \Rightarrow A \in T^i_{P[K]}(\emptyset)$. It is because:

(i) (basis) In case that $i = 0$, it trivially holds.

(ii) (induction step) In case that $i > 0$:

$$\begin{aligned}
&A \in T^i_{P[J]}(\emptyset)\\
&\Rightarrow \exists A \rhd A_1 \dots A_m \in P[J].\ A_1, \dots, A_m \in T^{i-1}_{P[J]}(\emptyset)\\
&\Rightarrow \exists A \rhd B_1 \dots B_n \in P[K] \text{ such that } \{B_1, \dots, B_n\} \subseteq \{A_1, \dots, A_m\}
\end{aligned}$$

It follows that $B_1, \dots, B_n \in T^{i-1}_{P[J]}(\emptyset)$. By induction hypothesis, we can assume that $B_1, \dots, B_n \in T^{i-1}_{P[J]}(\emptyset) \Rightarrow B_1, \dots, B_n \in T^{i-1}_{P[K]}(\emptyset)$. Therefore $A \in T^i_{P[K]}(\emptyset)$.

This concludes that $S_P(J) = \cup_{i\in\omega}\ T^i_{P[J]}(\emptyset) \subseteq \cup_{i\in\omega}\ T^i_{P[K]}(\emptyset) = S_P(K)$. That is, the mapping S_P is monotone. By monotonicity of S_P, $S_P(J_i) \subseteq S_P(\cup_{i\in\omega}\ J_i)$ for any ω-chain $J_0 \subseteq J_1 \subseteq J_2 \subseteq \dots$. Thus $\cup_{i\in\omega}\ S_P(J_i) \subseteq S_P(\cup_{i\in\omega}\ J_i)$. On the other hand, to show the opposite subset relation, we firstly assume that $A \in S_P(\cup_{i\in\omega}\ J_i)$. Then:

$$\begin{aligned}
&A \in S_P(\cup_{i\in\omega}\ J_i)\\
&\Rightarrow \exists j \in \omega.\ A \in T^j_{P[\cup_{i\in\omega}\ J_i]}(\emptyset)\\
&\Rightarrow \exists k \in \omega.\ A \in T^j_{P[J_k]}(\emptyset)\\
&\Rightarrow A \in \cup_{j\in\omega}\ T^j_{P[J_k]}(\emptyset) = S_P(J_k)
\end{aligned}$$

Therefore $S_P(\cup_{i\in\omega}\ J_i) \subseteq S_P(J_k)$ for some $k \in \omega$ such that $S_P(\cup_{i\in\omega}\ J_i) \subseteq \cup_{k\in\omega}\ S_P(J_k)$. That is, S_P is continuous. By means of the definition of $S_P(K)$ with respect to the mapping $T_{P[K]}$, $S_P(K)$ is the least fixed point of $T_{P[K]}$ such that we can see the following lemma.

Lemma 1. (1) For any $A \rhd A_1 \dots A_m \in P[K]$ $(m \geq 0)$,

$$A_1, \dots, A_m \in S_P(K) \text{ iff } A \in S_P(K).$$

(2) For any $A \rhd A_1 \dots A_m \in P$ $(m \geq 0)$,

$$A_1, \dots, A_m \in S_P(K) \cup K \text{ iff } A \in S_P(K).$$

(3) For any $A \rhd A_1 \dots A_m \in P$ $(m > 0)$,

$$\begin{gathered}
A_1, \dots, A_m \in S_P(K) \cup K \text{ and there is at least one } A_i \notin S_P(\emptyset)\\
\text{iff } A \in S_P(K) - S_P(\emptyset).
\end{gathered}$$

Proof. (1) For the rule $A \rhd A_1 \dots A_m \in P[K]$ $(m \geq 0)$:

$$\begin{aligned} & A \in S_P(K) \\ & \Leftrightarrow A \in \cup_{i \in \omega} \; T^i_{P[K]}(\emptyset) \\ & \Leftrightarrow A_1, \dots, A_m \in \cup_{i \in \omega} \; T^i_{P[K]}(\emptyset) \\ & \Leftrightarrow A_1, \dots, A_m \in S_P(K) \end{aligned}$$

(2) For the rule $A \rhd A_1 \dots A_m \in P$ $(m \geq 0)$, we can derive a rule $A \rhd B_1 \dots B_n \in P[K]$ such that $\{B_1 \dots B_n\} \subseteq \{A_1 \dots A_m\}$. The set $\{B_1 \dots B_n\}$ is obtained by removing each A_i of $\{A_1 \dots A_m\}$ for $A_i \in K$. By means of (1), $B_1 \dots B_n \in S_P(K)$ iff $A \in S_P(K)$. Thus

$$A_1, \dots, A_m \in S_P(K) \cup K \text{ iff } A \in S_P(K).$$

(3) By means of (2), $A_1, \dots, A_m \in S_P(K) \cup K$ $(m \geq 0)$ iff $A \in S_P(K)$. There is some $A_i \notin S(\emptyset)$ iff $A \notin S_P(\emptyset)$, by (2) for the case that $K = \emptyset$. It follows that

$$A_1, \dots, A_m \in S_P(K) \cup K \ (m > 0) \text{ and there is at least one } A_i \notin S_P(\emptyset) \\ \text{iff } A \in S_P(K) - S_P(\emptyset).$$

3 Lazy Reference Set Related to Fixed Point

In this section, we examine the set of lazy references.

Lemma 2. Assume the set P of rules. A reference A is in $S_P(\emptyset)$ iff it is eager.

Proof. (1) Assume $eager(A)$.
 (i) If $eager(A)$ by means of (ir1), then $A \rhd$ is in P such that $A \in S_P(\emptyset)$ (by Lemma 1 (2)).
 (ii) If $eager(A)$ by means of (ir2), then a rule $A \rhd A_1 \dots A_m$ is in P and for all A_i $(1 \leq i \leq m)$, the predicates $eager(A_i)$ are supposed. By induction hypothesis for $eager(A_i)$ $(1 \leq i \leq n)$, $A_i \in S_P(\emptyset)$, such that by Lemma 1 (2), $A \in S_P(\emptyset)$. This completes the induction.
(2) Assume that $A \in S_P(\emptyset)$. We prove it by induction on m for the rule $A \rhd A_1 \dots A_m$ $(m \geq 0)$, with respect to $A \in S_P(\emptyset)$.
 (i) If $m = 0$, that is, $A \rhd$ is in P, then $eager(A)$ (by the inference (ir1)).
 (ii) If $m > 0$ such that $A \rhd A_1 \dots A_m$ is in P, by induction hypothesis of $eager(A_i)$ $(1 \leq i \leq m)$ for $A_i \in S_P(\emptyset)$, we have $eager(A)$ with the inference (ir2). This completes the induction.

Lemma 3. Assume the set P of rules. A reference $A \in B_P$ does not occur in the head of any rule iff $A \in \overline{S_P(B_P)}$.

Proof. (i) Assume that the reference A occurs in the head of some rule such that there is a rule $A \rhd A_1 \dots A_m$ in P $(m \geq 0)$. It follows that $A \rhd$ is in $P[B_P]$. Thus $A \in T_{P[B_P]}(\emptyset) \subseteq S_P(B_P)$.
(ii) On the other hand, assume that $A \in S_P(B_P)$. Then $A \in S_P(B_P) = \cup_{i \in \omega} \; T^i_{P[B_P]}(\emptyset)$, which demonstrates that A occurs in the head of some rule.

For the lazy reference, we need the superset relation $L \supseteq S_P(L) - S_P(\emptyset)$ for a subset $L \subseteq B_P$. By Lemma 2, a set of lazy references has no common reference with the set $S_P(\emptyset)$ (the set of eager references). Assume a set $M \subseteq \overline{S_P(B_P)} \subseteq \overline{S_P(\emptyset)}$ such that M may be a set of references not occurring in the heads and be designated as lazy. We next investigate a fixed point of the equation $L = (S_P(L) - S_P(\emptyset)) \cup M$ for some $M \subseteq \overline{S_P(B_P)}$ by the following two theorems.

Theorem 1. *The set P of rules is supposedly given, where $L \subseteq \overline{S_P(\emptyset)}$. If $L = \{A \mid lazy_L(A)\}$,*

$$L = (S_P(L) - S_P(\emptyset)) \cup M \textit{ for some set } M \subseteq \overline{S_P(B_P)}.$$

Proof. If $L = \emptyset$, then the theorem trivially holds. Assume that $L = \{A \mid lazy_L(A)\} \neq \emptyset$. Suppose $lazy_L(A)$ ($A \in L$ by the assumption). We prove inductively that:

- $A \in L$ occurs in the head of some rule iff $A \in S_P(L) - S_P(\emptyset)$.
- $A \in L$ does not occur in the head of any rule iff $A \in M$ for some $M \subseteq \overline{S_P(B_P)}$.

We see that:

$$\begin{aligned}
&A \text{ occurs in the head of some rule}\\
&\Leftrightarrow \text{there is a rule } A \rhd A_1 \dots A_m \in P\ (m > 0) \text{ such that}\\
&\qquad\qquad \exists A_i.\ (A_i \text{ is not eager}),\ \text{and}\\
&\qquad\qquad \forall A_j.\ (A_j \text{ is eager or lazy})\\
&\Leftrightarrow \text{there is a rule } A \rhd A_1 \dots A_m \in P\ (m > 0) \text{ such that:}\\
&\qquad\qquad \exists A_i.(A_i \notin S_P(\emptyset)) \text{ and } \forall A_j.(A_j \in S_P(L) \cup L)\\
&\Leftrightarrow A \in S_P(L) - S_P(\emptyset)\\
&\qquad \text{(by Lemma 1 (3))}
\end{aligned}$$

$A \in L$ does not occur in the head of any rule iff $A \in \overline{S_P(B_P)}$ (Lemma 3) such that $A \in M$ for some $M \subseteq \overline{S_P(B_P)}$. This completes the proof.

Lemma 4. Assume a fixed point L of the equation $L = (S_P(L) - S_P(\emptyset)) \cup M$ for some set $M \subseteq \overline{S_P(B_P)}$. Then

(i) $L \subseteq \overline{S_P(\emptyset)}$.
(ii) $S_P(L) - S_P(\emptyset) \subseteq S_P(\overline{S_P(\emptyset)})$.
(iii) $M \subseteq \overline{S_P(\overline{S_P(\emptyset)})}$.

Proof. (i) $S_P(L) - S_P(\emptyset) \subseteq \overline{S_P(\emptyset)}$. $M \subseteq \overline{S_P(B_P)} \subseteq \overline{S_P(\emptyset)}$. It follows that $L \subseteq \overline{S_P(\emptyset)}$.
(ii) By (i), applying the monotone mapping S_P, $S_P(L) \subseteq S_P(\overline{S_P(\emptyset)})$. Then $S_P(L) - S_P(\emptyset) \subseteq S_P(\overline{S_P(\emptyset)})$.
(iii) Since $S_P(\overline{S_P(\emptyset)}) \subseteq S_P(B_P)$ by monotonicity of the mapping of S_P, $\overline{S_P(B_P)} \subseteq \overline{S_P(\overline{S_P(\emptyset)})}$. On the assumption that $M \subseteq \overline{S_P(B_P)}$, $M \subseteq \overline{S_P(\overline{S_P(\emptyset)})}$.

In Lemma 4, we suppose that a set M is designated as lazy.

Theorem 2. *Assume that a set P of rules is given, such that $L = (S_P(L) - S_P(\emptyset)) \cup M$ where $M \subseteq \overline{S_P(B_P)}$. Then $L = \{A \mid lazy_L(A)\}$.*

Proof. If $L = \emptyset$, the theorem trivially holds. We now suppose that $L \neq \emptyset$.
(1) Take any reference $A \in L$. We prove inductively with the following cases (i) and (ii) that $lazy_L(A)$. (It follows that $L \subseteq \{A \mid lazy_L(A)\}$.)

(i) Assume that $A \in S_P(L) - S_P(\emptyset) \neq \emptyset$.

$A \in S_P(L) - S_P(\emptyset)$
$\Rightarrow$ there is a rule $A \rhd A_1 \ldots A_m$ $(m > 0)$ in P such that:
$A_1, \ldots, A_m \in S_P(L) \cup L$ and at least one A_i is not in $S_P(\emptyset)$
(by Lemma 1 (3))
$\Rightarrow$ there is a rule $A \rhd A_1 \ldots A_m$ $(m > 0)$ in P such that:
$A_1, \ldots, A_m$ are eager or lazy, and at least one A_i is not eager
(by induction hypothesis) : $A_j \in S_P(L) - S_P(\emptyset) \Rightarrow A_j$ is lazy;
$A_j \in S_P(\emptyset) \Rightarrow A_j$ is eager; $A_j \in L - (S_P(L) - S_P(\emptyset)) \Rightarrow A_j$ is lazy
$\Rightarrow A$ is lazy, that is, $lazy_L(A)$

(ii) Assume that $A \in M \subseteq \overline{S_P(B_P)}$. By Lemma 3, A does not occur in the head of any rule. If $A \in L$, then $lazy_L(A)$.
By (i) and (ii), we conclude that $L \subseteq \{A \mid lazy_L(A)\}$.

(2) We next prove that if $lazy_L(A)$ then $A \in L$.

(i) If A occurs in the head of some rule, then there is a rule $A \rhd A_1 \ldots A_m$ such that each A_j is eager or lazy ($A_j \in S_P(\emptyset) \cup L$), and at least one A_i is not eager ($A_i \notin S_P(\emptyset)$). It follows that $A \in S_P(L) - S_P(\emptyset) \subseteq L$.
(ii) If A does not occur in the head of any rule, $A \in L$ because of $lazy_L(A)$.

As the conclusion of (2), $L \supseteq \{A \mid lazy_L(A)\}$, by which we conclude that $L = \{A \mid lazy_L(A)\}$, as well as the proof (1). This completes the proof.

By Theorems 1 and 2, we see that L is a fixed point of the equation:

$$L = (S_P(L) - S_P(\emptyset)) \cup M \text{ for some set } M \subseteq \overline{S_P(B_P)}$$

iff $L = \{A \mid lazy_L(A)\}$. As is seen, there is a least fixed point of the equation. Note that $M \subseteq \overline{S_P(B_P)}$ is not uniquely determined for the equation $L = (S_P(L) - S_P(\emptyset)) \cup M$. In the next section, instead of the equation $L = (S_P(L) - S_P(\emptyset)) \cup M$, we take a superset relation $L \supseteq S_P(L) - S_P(\emptyset)$ without such a set M.

4 Soundness and Completeness of Reference Laziness

We firstly have a soundness theorem of the predicate $lazy_L(A)$ (which states that A is lazy with the set $L \subseteq \overline{S_P(\emptyset)}$), with respect to membership of A in L or in $S_P(L) - S_P(\emptyset)$ with some set L', where

$$S_P(L) - S_P(\emptyset) \subseteq S_P(L') - S_P(\emptyset) \subseteq L' \subseteq \overline{S_P(\emptyset)}.$$

Theorem 3. *Given a set P of rules, assume $lazy_L(A)$, where $L \subseteq \overline{S_P(\emptyset)}$. Then $A \in L$, or there is L' such that $A \in S_P(L) - S_P(\emptyset) \subseteq S_P(L') - S_P(\emptyset) \subseteq L' \subseteq \overline{S_P(\emptyset)}$.*

Proof. Assume that $lazy_L(A)$. (1) We prove inductively that $A \in L$, or $A \in S_P(L) - S_P(\emptyset)$ as follows:

(i) If A doe not occur in the head of any rule, A must be in L because of the predicate $lazy_L(A)$.
(ii) If A occurs in the head of some rule, then:

$$\begin{array}{l} \text{there is a rule } A \rhd A_1 \ldots A_m \in P\ (m > 0) \text{ such that:} \\ \qquad\qquad \exists A_i.\ (A_i \text{ is not eager}),\ \text{and} \\ \qquad\qquad \forall A_j.\ (A_j \text{ is eager or lazy}) \\ \Rightarrow \text{there is a rule } A \rhd A_1 \ldots A_m \in P\ (m > 0) \text{ such that:} \\ \qquad\qquad \exists A_i.(A_i \notin S_P(\emptyset)) \text{ and } \forall A_j.(A_j \in S_P(L) \cup L) \\ \Rightarrow A \in S_P(L) - S_P(\emptyset) \end{array}$$

(2) Now we assume the case that $lazy_L(A)$ such that $A \in S_P(L) - S_P(\emptyset)$. With $L_0 = L$ and $L_1 = S_P(L) - S_P(\emptyset)$, we have an ω-chain $L_1 \subseteq L_2 \subseteq \ldots$, owing to monotonicity of S_P,

$$\begin{array}{l} S_P(L_0) - S_P(\emptyset) = L_1 \\ S_P(L_0 \cup L_1) - S_P(\emptyset) = L_2 \\ \cdots\ \cdots \\ \cdots\ \cdots \\ S_P(\cup_{i \in \omega}\ L_i) - S_P(\emptyset) = \cup_{i \geq 1}\ L_i \end{array}$$

where $S_P(\cup_{i \in \omega}\ L_i) = \cup_{i \in \omega}\ S_P(L_i)$ by continuity of S_P. Take $L' = \cup_{i \in \omega}\ L_i \supseteq \cup_{i \geq 1}\ L_i$. Then

$$A \in L_1 \subseteq \cup_{i \geq 1}\ L_i = S_P(\cup_{i \in \omega}\ L_i) - S_P(\emptyset) = S_P(L') - S_P(\emptyset) \subseteq L'.$$

Because $L_i \subseteq \overline{S_P(\emptyset)}$ $(i \in \omega)$ by the construction of L_i, $L' = \cup_{i \in \omega}\ L_i \subseteq \overline{S_P(\emptyset)}$. This completes the proof.

We next have a completeness theorem of the predicate $lazy_L(A)$ (which states that A is lazy with the set $L \subseteq \overline{S_P(\emptyset)}$), with respect to membership of A in $S_P(L) - S_P(\emptyset)$, where

$$S_P(L) - S_P(\emptyset) \subseteq L \subseteq \overline{S_P(\emptyset)}.$$

Theorem 4. *Assume a set P of rules such that $\emptyset \neq S_P(L) - S_P(\emptyset) \subseteq L$ for a set $L \subseteq \overline{S_P(\emptyset)}$. If $A \in S_P(L) - S_P(\emptyset)$, then $lazy_L(A)$.*

Proof. Assume that $A \in S_P(L) - S_P(\emptyset)$. By Lemma 1 (3), there is a rule

$$A \rhd A_1 \dots A_m \ (m > 0)$$

such that $A_1, \dots, A_m \in S_P(L) \cup L$ and at least one A_i in in $\overline{S_P(\emptyset)}$. Because $A_1, \dots, A_m$ are all in $S_P(L) \cup L$ and at least one A_i is in $\overline{S_P(\emptyset)}$, we see the cases for each A_j:

(i) $A_j \in L$
(ii) $A_j \in S_P(\emptyset) \subseteq S_P(L) \Rightarrow eager(A_j)$ (by Lemma 2)
(iii) $A_j \in S_P(L) - S_P(\emptyset) \subseteq S_P(L) \Rightarrow lazy_L(A_j)$ (by induction hypothesis for A_j)

If A_i is in $\overline{S_P(\emptyset)}$, then A_i is in L or $lazy_L(A_i)$ excluding the case (ii). By the inferences (ir3) and (ir4), we can conclude that $lazy_L(A)$.

5 Concluding Remarks

We have dealt with a finite or countably infinite set of rules, where the set of lazy references is represented by means of fixed point approach. Practically only a finite set is needed, where the theoretical considerations are available from static analysis views as in this paper. Given a set of P of rules with a set L of designated lazy references, we have soundness and completeness of reference laziness in the following sense:

(1) (soundness) The predicate $lazy_L(A)$ (which states that the reference A is lazy with the set $L \subseteq \overline{S_P(\emptyset)}$) is sound with respect to membership of A in L or in $S_P(L) - S_P(\emptyset)$, with some set L' such that

$$S_P(L) - S_P(\emptyset) \subseteq S_P(L') - S_P(\emptyset) \subseteq L' \subseteq \overline{S_P(\emptyset)}.$$

(2) (completeness) The predicate $lazy_L(A)$ (which states that A is lazy with the set $L \subseteq \overline{S_P(\emptyset)}$) is complete with respect to membership of A in $S_P(L) - S_P(\emptyset)$, where

$$S_P(L) - S_P(\emptyset) \subseteq L \subseteq \overline{S_P(\emptyset)}.$$

In addition to the soundness, the designation of lazy references may step by step construct some set L' which is relative to the soundness of the predicate $lazy_L(A)$ with respect to membership of A in $S_P(L) - S_P(\emptyset)$.

The set of finite-failure references (the finite-failure set) may be defined. This is similar to finite failure of logic programming ([11]), however, a unique successor (which may be the empty) for each head may be allowable in this case.

We can define the finite-failure set FF_P to be $FF_P = \cup_{d \in \omega} FF_P^d$, where:

$$FF_P^0 = \{A \in B_P \mid A \text{ does not occur in the head of any rule}\} - L,$$
$$FF_P^d = \{A \in B_P \mid \exists A \rhd A_1 \dots A_m \in P,\ \exists A_i.\ A_i \in FF_P^{d-1}\} - L \ (d > 0).$$

When $L = \emptyset$, regarding the reference as a proposition with the propositional Horn logic, we have

$FF_P = \overline{\cap_{i \in \omega} T_P^i(B_p)}$ (where T_P^i stands for i-times applications to the set B_P).

If we allow the case that there are more than two rules with a head including different successors, which is prohibited in the set of rules of this paper, the rule set conceives the interpretation that the reference A is both eager and lazy. Even if such a case is involved, the properties as in the propositional Horn logic may be of use for the treatments of references. It may be a problem to see a relation between the lazy reference set and the set $\cap_{i \in \omega} T_P^i(B_P)$. How we temporarily have a set L may affect some reasonable considerations about the relation.

References

1. Areces,C. and Blackburn,P., Repairing the interpolation in quantified logic, Annals of Pure and Applied Logic, 123, 287–299, 2003.
2. Brauner,T., Natural deduction for hybrid logics, J. of Logic and Computation, 14, 329–353, 2004.
3. Cervesato,I., Chittaro,L. and Montanari,A., A general modal framework for the event calculus and its skeptical and credulous variants, Proc. of 12th European Conference on Artificial Intelligence, pp.12–16, 1996.
4. Dean,T. and Boddy,M., Reasoning about partially ordered events, Artificial Intelligence, 36, pp.375–399, 1988.
5. Genesereth,M.R. and Nilsson,N.J., Logical Foundations of Artificial Intelligence, Morgan Kaufmann, 1988.
6. Giordano,L., Martelli,A. and Schwind,C., Ramification and causality in a modal action logic, J. of Logic and Computation, 10, pp.625–662, 2000.
7. Harpern,J.Y. and Lakemeyer,G., Multi-agent only knowing, J. of Logic and Computation, 11, pp.41–70, 2001.
8. Hoare,C.A.R., Communicating Sequential Processes, Prentice-Hall, 1985.
9. Kowalski,R.A., Database updates in the event calculus, J. of Logic Programming, 12, 121–146, 1992.
10. Kucera,A. and Esparza,J., A logical viewpoint on process-algebraic quotients, J. of Logic and Computation, 13, pp.863–880, 2003.
11. Lloyd,J.W., Foundations of Logic Programming, 2nd, Extended Edition, Springer-Verlag, 1993.
12. Milner,R., Communication and Concurrency, Prentice-Hall, 1989.
13. Mosses,P.M., Action Semantics, Cambridge University, 1992.
14. Reiter,R., Knowledge in Action, The MIT Press, 2001.
15. Russell,S. and Norvig,P., Artificial Intelligence–A Modern Approach–, Prentice-Hall, 1995.
16. Shepherdson,J.C., Negation in logic programming, In Minker,J. (ed.), Foundations of Deductive Databases and Logic Programming, 19–88, 1987.
17. Winskel,G., The Formal Semantics of Programming Languages, MIT Press, 1993.
18. Yamasaki,S., A denotational semantics and dataflow construction for logic programs, Theoretical Computer Science, 124, pp.71-91, 1994.
19. Yamasaki,S. and Kurose,Y., A sound and complete proof procedure for a general logic program in no-floundering derivations with respect to the 3-valued stable model semantics, Theoretical Computer Science, 266, pp.489–512, 2001.
20. Yamasaki,S., Logic programming with default, weak and strict negations, Theory and Practice of Logic Programming, 6, pp.737-749, 2006.
21. Yamasaki,S. and Sasakura,M., A calculus effectively performing event formation with visualization, Lecture Notes in Computer Science, 4759, pp.287-294, 2008.

Static Types As Search Heuristics

Hao Xu

xuh@cs.unc.edu
Department of Computer Science,
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA

Abstract. Static analysis techniques have been studied for many years in the functional programming community, most prominently the use of inferred types to eliminate run-time checks. In analogy, if we regard checking whether an instance is useful for a proof as run-time checks and we can find types that eliminate irrelevant instances, we may also be able to prevent proof searches from checking those irrelevant instances, thereby improving the performance. This paper introduces a method that employs types as heuristics in proof search.

1 Introduction

Instance-based theorem proving techniques have been implemented in many theorem provers: [1], [2], [3], [4], and [5], among others. Some of these provers have been shown to prove problems in some categories faster than or comparable to resolution-based theorem provers[6]. On the other hand, resolution-based theorem provers continue to lead on other problem categories. While instance-based theorem provers are believed to have a number of advantages, such as being capable of generating models for satisfiable problems and making use of the highly efficient SAT solvers, their run-time performance is suboptimal without guidance from instance generation heuristics such as resolution, semantics, etc..

Static analysis techniques in compilers, such as using types to eliminate run-time checks, have been studied for many years. In analogy, if we regard checking whether an instance is useful for a proof as run-time checks, then we may find types that prevent proof searches from checking some irrelevant instances, thereby improving the performance. In general, static analysis techniques generate heuristics from the input, in contrast to various static transformation techniques which preprocess the input. Following is a (incomplete) list of the techniques used in static analysis:

Strategy Selection chooses a strategy that is believed to best suit the problem. Implementation: E[7], Vampire[8], and DCTP[5], among others.

Restriction finds instances that may be pruned from the search space. Implementation: FM-Darwin[9], Paradox[10], and OSHL-S, among others.

Ordering determines what kind of ordering can lead to contradiction faster. Implementation: E(literal ordering), among others.

This paper introduces a method that employs inferred types from resolution as heuristics in first-order logic proof search. The general idea is deriving a type inference algorithm from a complete calculus, and using types to filter out a subset of instances that are not generated by the calculus by searching only well-typed instances. Properly designed types should bring some of the advantages from one method to another.

2 Notations

A vector is written as $[a_1, \ldots, a_n]$, or vertically as in (5). The ith component of a vector $\mathbf{v}$ is v_i. We write $\mathbf{v}^n$ to explicitly mark that $\mathbf{v}$ has n components. A vector is implicitly converted to a set. For example, in $\mathbf{v} \subset \mathbf{x}$. Given a signature Σ and a set of variables Var, Term is the set of terms over $\Sigma \cup \mathsf{Var}$; GrTerm the set of terms over Σ. An expression is a term or a literal. Expr is the set of expressions over $\Sigma \cup \mathsf{Var}$; GrExpr the set of expressions over Σ. By default, $\Sigma = FS(S)$ and $\mathsf{Var} = VS(S)$, given a clause set S. Constants are nullary function symbols. The empty clause is denoted by $\emptyset$. The function fv maps an expression to a set of free variables occurring in that expression. $\mathbf{fv}(x)$ is the vector constructed from $fv(x)$ by sorting the element lexicographically (other total orderings should also work). A substitution is a partial function $\theta : \mathsf{Var} \to \mathsf{Term}$. $fv(\theta) = \bigcup_{v \in dom(\theta)} fv(\theta(v))$. A substitution of variables $\mathbf{v}$ with terms $\mathbf{t}$ is denoted by $[\mathbf{t}/\mathbf{v}]$. If $dom(\theta) \subset D$, the domain extension $\theta|_D$ of θ to D is defined by $\theta|_D(v) = \theta(v)$, if $v \in dom(\theta)$ and $\theta|_D(v) = v$, if $v \in D \backslash dom(\theta)$. If $fv(e) \not\subset dom(\theta)$, $e\theta = e\theta|_{dom(\theta) \cup fv(e)}$. The well-formedness requirement for a substitution θ is that $\theta(v) = v$, for all $v \in fv(\theta) \cap dom(\theta)$, namely, the substitution is idempotent. The well-formedness requirement for an mgu σ of literals L, N is that $fv(\sigma) \subset dom(\sigma) = fv(L) \cup fv(N)$. The complement of a literal L is denoted by $\overline{L}$.

3 A Theory

3.1 An Example

Example 1. $S = \{\{P(X), Q(f(X)\}, \{\neg P(f(Y)), \neg Q(Y)\}, \{Q(a)\}, \{\neg Q(f(f(a))\}\}$.

Since S is unsatisfiable, there exists a minimal (but not necessarily minimum) set S_1 of ground instances of S that is unsatisfiable. By minimality of the instances in S_1, for all ground literals $N \in \bigcup S_1$, $\overline{N} \in \bigcup S_1$. For example, $S_1 = \{\{P(f(a)), Q(f(f(a))\}, \{\neg P(f(a)), \neg Q(a)\}, \{Q(a)\}, \{\neg Q(f(f(a)))\}\}$. If we use a brute force method to find S_1, then both X and Y are initiated to terms in $x = \{a, f(a), f(f(a)), \ldots\}$. Our goal is to restrict the instantiation of variables to proper subsets of x given by a instantiation function $I : \mathsf{Term} \to \mathcal{P}(\mathsf{GrTerm})$. We establish a similar minimality requirement for I: for any literal $N \in S$ and any ground instance N_1 in $I(N)$, there exists L s.t. $\overline{N_1} \in I(L)$. Rewriting this requirement as an inequality of sets, we obtain:

$$\bigcup_{L \in \bigcup S} I(L) \supset \bigcup_{L \in \bigcup S} \overline{I(L)} \tag{1}$$

Informally speaking, all instances of literals in S should be resolvable. In the example, the only literal in S that is unifiable with $\overline{P(X)}$ is $\neg P(f(Y))$. Therefore, any instance of $P(X)$ must have a complement, which must be an instance of $\neg P(f(Y))$. Hence $I(\neg P(f(Y))) \supset \overline{I(P(X))}$. If $I(e) = \{e[\mathbf{t}/\mathbf{v}] | t_i \in I(v_i), \mathbf{v} = \mathbf{fv}(e)\}$, then $\{f(t)|t \in I(Y)\} \supset I(X)$. We restrict the proof search by instantiating any variable v to elements of $I(v)$, which we refer to as the semantics of the type of the variable v.

3.2 Algebra $\mathcal{V}$

In this section, we introduce an algebra $\mathcal{V}$. The motivation of $\mathcal{V}$ is to enable simultaneously manipulating functions and sets, as in linear algebra. There are several groups of notations used in this paper. Algebra $\mathcal{V}$: ρ, τ type; v variable; $\mathbf{X}$ term of the form $[v_i]_{i=1}^n$; $A, \mathbf{A}, \mathbf{B}, F, G, H, T, \mathbf{T}$ term. Mathematics: D domain; x, y set; f, g function; $\mathbf{a}, \mathbf{b}, \mathbf{v}$ vector; l, m, n, o, p natural number. Logic: u, v, X, Y variable; t term; e expression; L, N literal; C, D clause; S clause set; a, c, f function symbol.

Definition 1. *The types are: set of vectors of length n $\{n\}$; function $\tau \to \tau'$.*

Types are mainly used to define what terms are meaningful in $\mathcal{V}$. To avoid the complexity of recursive domain equations, we define families of constructors indexed by natural numbers. The indices do not affect the mapping of the function that a term denotes, but the domain and range of that function.

Definition 2. *$T : \tau$ denotes that T is a term of type τ. $\tau_{m,n} = \{m\} \to \{n\}$.*

$$\begin{array}{llll}
\emptyset_n : \{n\} & \mathit{SumUnit} & \lambda \mathbf{v}^n.\emptyset_m : \{n\} \to \{m\} & \mathit{FSumUnit} \\
\cup_n : \{n\} \to \{n\} \to \{n\} & \mathit{Sum} & \sqcup_{m,n} : \tau_{m,n} \to \tau_{m,n} \to \tau_{m,n} & \mathit{FSum} \\
\cap_n : \{n\} \to \{n\} \to \{n\} & \mathit{Inter} & \sqcap_{m,n} : \tau_{m,n} \to \tau_{m,n} \to \tau_{m,n} & \mathit{FInter} \\
[] : \{0\} & \mathit{ProdUnit} & \lambda \mathbf{v}^n.[] : \{n\} \to \{0\} & \mathit{FProdUnit} \\
\dfrac{p = m + n}{\bullet_{m,n} : \{m\} \to \{n\} \to \{p\}} & \mathit{Prod} & \dfrac{p = m + n}{\blacksquare_{l,m,n} : \tau_{l,m} \to \tau_{l,n} \to \tau_{l,p}} & \mathit{FProd} \\
p_{i/n} : \{n\} \to \{1\} & \mathit{Proj} & \circ_{m,n,p} : \tau_{m,n} \to \tau_{n,p} \to \tau_{m,p} & \mathit{Comp} \\
\dfrac{\mathbf{v} \supset fv(e)}{\mathit{\Lambda} \mathbf{v}^n.e : \{n\} \to \{1\}} & \mathit{TFunc} & v : \{1\} & \mathit{Var} \\
\dfrac{\mathbf{v} \supset fv(e)}{\mathit{\Lambda}^{-1} \mathbf{v}^n.e : \{1\} \to \{n\}} & \mathit{RTFunc} & \dfrac{T : \rho \to \tau \quad T' : \rho}{TT' : \tau} & \mathit{App}
\end{array}$$

We use parentheses to delimit terms when ambiguous. We omit subscripts and superscripts indicating vector length and type indices when not ambiguous. There are two kinds of terms used in the paper: a term in the set Term and a term in $\mathcal{V}$. We refer to the latter as a term of $\mathcal{V}$ to disambiguate only when necessary. $TFunc$ is called a term function; $RTFunc$ a reverse term function.

Let us now switch context to mathematical objects. As in domain theory, we say that a function f is continuous if $x_0 \subset x_1 \subset \ldots$ implies $\bigcup_{i=0}^{\infty} f(x_i) = f(\bigcup_{i=0}^{\infty} x_i)$. Define $\mathbf{a}^m \mathbf{b}^n = [a_1, \ldots, a_m, b_1, \ldots, b_n]$, $x \times y = \{\mathbf{a}^m \mathbf{b}^n | \mathbf{a}^m \in x, \mathbf{b}^n \in$

$y\}$, $\prod_{i=1}^{n} x_i = x_1 \times (\ldots \times x_n \times \{[]\})$, $\bigcup_{i=1}^{n} x_i = x_1 \cup \ldots \cup x_n$, $\bigcap_{i=1}^{n} x_i = x_1 \cap \ldots \cap x_n$, and $[x_i]_{i=1}^{n} = [x_1, \ldots, x_n]$. For example, $[a][f(a)] = [a, f(a)]$, $\{[a], [a']\} \times \{[f(a)]\} = \{[a, f(a)], [a', f(a)]\}$. If we define $D_n = \{[x_i]_{i=1}^{n} | x_i \in \mathsf{GrExpr}\}$, then the domains D_τ are defined as follows, where $[D_\rho \to D_\tau]$ is the set of functions from D_ρ to D_τ.

$$D_{\{n\}} = \mathcal{P}(D_n) \quad D_{\rho\to\tau} = [D_\rho \to D_\tau]$$

Definition 3. *An interpretation is a partial function $I : \mathsf{Var} \to \mathcal{P}(\mathsf{GrTerm})$. An interpretation is trivial if $I(v) = \emptyset$ for some variable v.*

Given an interpretation I, the semantic function $()^*_I$ maps a term $T : \tau$ of $\mathcal{V}$ to an element of D_τ.

Definition 4. *The semantic function $()^*_I$ is defined as follows. The semantics of function terms are given by partial evaluation.*

$$\begin{array}{rr}
(\emptyset_n)^*_I = \emptyset & (\lambda\mathbf{v}.\emptyset_n)^*_I(x) = \emptyset \\
(\cup_n)^*_I(x)(y) = x \cup y & (\sqcup_{m,n})^*_I(f)(g)(x) = f(x) \cup g(x) \\
(\cap_n)^*_I(x)(y) = x \cap y & (\sqcap_{m,n})^*_I(f)(g)(x) = f(x) \cap g(x) \\
([])^*_I = \{[]\} & (\lambda\mathbf{v}.[])^*_I(x) = \{[]\} \\
(\bullet_{m,n})^*_I(x)(y) = x \times y & (\blacksquare_{l,m,n})^*_I(f)(g)(x) = f(x) \times g(x) \\
(p_{i/n})^*_I(\mathbf{x}^n) = x_i & (\circ_{m,n,p})^*_I(f)(g)(x) = f(g(x)) \\
(\Lambda\mathbf{v}.e)^*_I(x) = \{e[\mathbf{t}/\mathbf{v}] | \mathbf{t} \in x\} & (v)^*_I = I(v) \\
(\Lambda^{-1}\mathbf{v}.e)^*_I(x) = \{\mathbf{t} | e[\mathbf{t}/\mathbf{v}] \in x\} & (FT)^*_I = (F)^*_I((T)^*_I)
\end{array}$$

Notations such as $\emptyset, \cup, \cap$ are reused as both set-theoretical notations and terms of $\mathcal{V}$. A term is variable-free if there is no subterm that is a variable. Since the denotations of *variable-free* terms are independent of I, we usually make I implicit when the term is variable-free. For example, $(\Lambda[v_1].f(v_1))^*\{[a]\} = \{[f(a)]\}$, $(\Lambda^{-1}[v_1].f(v_1))^*\{[f(a)]\} = \{[a]\}$, $(p_{1/2})^*\{[a, f(a)]\} = \{[f(a)]\}$.

$I \models T = T'$ if and only if $(T)^*_I = (T')^*_I$. $T = T'$ if and only if it holds for all I. A subset order is defined on set terms: $I \models T \sqsubset T'$ if and only if $(T)^*_I \subset (T')^*_I$. The order is extended to functions $I \models F \sqsubset G$ if and only if for all terms T, $I \models FT \sqsubset GT$. $T \sqsubset T'$ if and only if it holds for all I. The reverse is denoted by $\sqsupset$. F is continuous if and only if $(F)^*_I$ is continuous for all I.

We write $[T_i]_{i=1}^{n} = [T_1, \ldots, T_n] = T_1 \bullet \ldots \bullet T_n \bullet []$, $[F_i]_{i=1}^{n} = [F_1, \ldots, F_n] = T_1 \blacksquare \ldots \blacksquare T_n \blacksquare \lambda\mathbf{v}.[]$. We usually want to convert a vector to a term of the form $[v_i]_{i=1}^{n}$ and conversely, which we call algebraify and dealgebraify. If $\mathbf{v}$ is a vector of variables, then $\mathcal{A}\mathbf{v}$ is a term $v_1 \bullet \ldots \bullet v_n \bullet []$; if $\mathbf{X}$ is a term of the form $[v_i]_{i=1}^{n}$, then $\mathcal{A}^{-1}\mathbf{X} = \mathbf{v}$. As a convention, we have $\mathbf{X} = \mathcal{A}\mathbf{v}$ and $\mathbf{v} = \mathcal{A}^{-1}\mathbf{X}$.

We make the following auxiliary definitions, where $\mathbf{v}_e = \mathbf{fv}(e)$, f is a function symbol of arity o, $\mathbf{v}_f = [v_1, \ldots, v_o]$, and $F_n : \{n\} \to \{n\}$ for some natural number n. $F_n^0 \triangleq \cup_n \emptyset_n$. $F_n^{k+1} \triangleq F_n F_n^k$. $f^{-1} \triangleq (\Lambda^{-1}\mathbf{v}_f.f(v_1, \ldots, v_o))$. $e \triangleq (\Lambda\mathbf{v}_e.e)\mathbf{X}_e$. $I(e) \triangleq (\Lambda\mathbf{v}_e.e)^* I(\mathbf{v}_e)$. $I(\mathbf{v}^n) \triangleq \prod_{i=1}^{n} I(v_i)$. For example, if f is binary, then $f(X, Y) = (\Lambda[X, Y].f(X, Y))[X, Y]$, $f^{-1} = \Lambda^{-1}[X, Y].f(X, Y)$. We write binary functions as infix, for example, $\sqcup FG$ is written as $F \sqcup G$ except for composition where we write $\circ FG$ as FG.

It can be proved that (a) if $F \sqsupset G$ and $F \sqsupset H$, then $F \sqsupset G \sqcup H$; (b) $F \sqcup G \sqsupset F$ and $F \sqcup G \sqsupset G$; (c) $(F \sqcup G)H = FH \sqcup GH$; (d) $F(G \sqcup H) = FG \sqcup FH$; (e) $[F_i]_{i=1}^n T = [F_i T]_{i=1}^n$ (f) $(\lambda \mathbf{v}.\emptyset)F = \lambda \mathbf{v}.\emptyset$; (g) $p_i[T_k]_{k=1}^n = T_i$, where $i \in \{1, \ldots, n\}$.

3.3 Constraints

Definition 5. *(a) An* instantiation set *of an expression e is a set of ground substitutions Θ s.t. for every $\theta \in \Theta$,$dom(\theta) \supset fv(e)$. The* instance set *of e w.r.t. Θ is $\{x\theta | \theta \in \Theta\}$. (b) An instantiation set of $\mathbf{v}$ induced by an interpretation I s.t. $dom(I) \supset \mathbf{v}$ is the set $\{[\mathbf{t}/\mathbf{v}] | \mathbf{t} \in I(\mathbf{v})\}$. (c) An instantiation set is complete for a clause set S if the instances obtained from it are inconsistent. An interpretation is complete if the induced instantiation set is complete.*

In this section we formalize the ideas discussed in Section 3.1 in terms of $\mathcal{V}$. In the following discussion, S is a set of clauses s.t. $\emptyset \notin S$ and for any $C, D \in S$, $fv(C) \cap fv(D) = \emptyset$.

Suppose that $L, N \in \bigcup S$ and that σ is a well-formed mgu of $L, \overline{N}$. We require that for every $v \in fv(L), \sigma(v) \neq v, t = \sigma(v), \mathbf{v} = \mathbf{fv}(t)$

$$I \models v \sqsupset (\Lambda \mathbf{v}.t)\mathbf{X} \tag{2}$$

and for every $u \in fv(L), \sigma(u) \neq u, t = \sigma(u), \mathbf{v} = \mathbf{fv}(t), v = v_i$,

$$I \models v \sqsupset p_i(\Lambda^{-1}\mathbf{v}.t)u \tag{3}$$

The apparent minimum solution to (1) is the constant function $I(v) = \emptyset$ for any variable v. Therefore, we add the following constraints to ensure that the solution is complete for S. Given an arbitrary partial function $g : \mathsf{Var} \to \mathsf{GrTerm}$ s.t. $dom(g) \supset fv(S)$, for all $v \in fv(S)$, $c = g(v)$,

$$I \models v \sqsupset c \tag{4}$$

In effect, (4) pins down a specific class of inconsistent sets of instances which are generated by instantiating a resolution proof to a ground resolution proof.

The algorithm for constraints generation (2), (3), and (4) is shown as follows.

Algorithm 1. *(Constraints Generation)*

1. *Given a set of nonempty clauses S, rename so that any two clauses have no common variables.*
2. *For every clause C, every literal L in C, every clause D in S, and every literal N in D such that $\overline{N}$ and L are unifiable by mgu σ, generate constraint (2) and (3).*
3. *For all variable X generate constraint (4).*

Denote the constraints generated by a clause set S by $cons(S)$. If I makes a constraint true, then we say that I is a solution to the constraint. If any solution to constraints K is also a solution to constraints K' and vice versa, then we say that $K \equiv K'$.

Next, we look at an example.

Example 2. We choose $g(X) = f(a), g(Y) \underline{= a \text{ and}}$ generate constraints for Example 1. Because $P(X)$ is unifiable with $\overline{\neg P(f(Y))}$ with mgu $[f(Y)/X]$, the algorithm generates the following constraints $I \models X \sqsupset f(Y), I \models Y \sqsupset p_1 f^{-1} X$. Other constraints are generated similarly. We usually write the constraints in an equivalent but more compact form. $I \models X \sqsupset f(Y) \cup p_1 f^{-1}(Y) \cup f(a), I \models Y \sqsupset f(X) \cup p_1 f^{-1}(X) \cup a$. We also make I implicit when not ambiguous.

Sometimes it is more compact to present a theory if we write inequality constraints in a vector normal form. For example, the constraints in Example 2 can be rewritten in the vector form:

$$I \models \mathbf{X} \sqsupset \begin{bmatrix} \Lambda\mathbf{v}.f(Y) \sqcup p_1 \Lambda^{-1}\mathbf{v}.f(X)p_2 \sqcup \Lambda\mathbf{v}.f(a) \\ \Lambda\mathbf{v}.f(X) \sqcup p_2 \Lambda^{-1}\mathbf{v}.f(Y)p_1 \sqcup \Lambda\mathbf{v}.a \end{bmatrix} \mathbf{X}, \text{ where } \mathbf{X} = \begin{bmatrix} X \\ Y \end{bmatrix} \quad (5)$$

Definition 6. *A normal form of constraints is $I \models \mathbf{X} \sqsupset \mathbf{AX}$, where $\mathbf{X}$ is of the form $[v_i]_{i=1}^n$ and $\mathbf{A}$ is of the form $[A_i]_{i=1}^n$, where A_i is produced by the following grammar $\mathsf{A} \to \lambda\mathbf{v}.\emptyset | \left(\Lambda\mathbf{v}.t | p(\Lambda^{-1}\mathbf{v}.t)p'\right)^{*,\sqcup}$ where $t \in \mathsf{Term}$, $\mathbf{v} \subset \mathsf{Var}$, p and p' are Proj, there is no repeated terms in A, and the terms in A are ordered in a certain total order.*

3.4 Theorems[1]

In this section, we look at a few theorems. Theorem 1 shows that $\mathbf{X} \sqsupset \mathbf{AX}$ is solvable. Theorem 2 shows that a solution to constraints *cons*(S) is complete for S, which is the main result of this section. Lemma 1 shows that resolution does not affect the solution of the constraints. If we assume that the semantics of S is given by resolution, then Lemma 1 is analogous to preservation (or subject reduction) of a type system.

Theorem 1. *The minimum solution I to an inequality $\mathbf{X} \sqsupset \mathbf{AX}$ is given by $I(\mathbf{v}) = \bigcup_{i=0}^{\infty}(\mathbf{A}^i)^*\emptyset$. The rhs is a fixpoint of the function $f(x) = (\mathbf{A})^*(x)$.*

Lemma 1. *Given a set of clauses S s.t. for any clauses $C, D \in S$, $fv(C) \cap fv(D) = \emptyset$, the binary resolvent D of two clauses C_1 and C_2 in S by some well-formed mgu, cons(S) $\equiv$cons$(S \cup \{D\})$.*

Theorem 2. *Given a set of clauses S s.t. for any clauses $C, D \in S$, $fv(C) \cap fv(D) = \emptyset$, any solution I to cons(S) is complete for S.*

4 Context Free Type

Generally speaking, I may have complicated structures. We need to find a finite representation for these sets that facilitates enumeration of terms, as one of the purposes of types is generating terms. By Theorem 1, $\mathbf{A}$ is a finite representation of the minimum solution of the constraint, but it is not very efficient as it includes

[1] A long version of this paper can be found at http://cs.unc.edu/~xuh/oshls.

*RTFunc*s which require pattern matching. Besides, it is also hard to generate terms in certain order, say, the size-lexicographic order. The approach introduced in this section converts the constraints into a grammar that produces a solution without *RTFunc*s.

Given a *FProd* $\mathbf{A} = [A_i]_{i=1}^n$ in the normal form, for any component A of $\mathbf{A}$, we denote the *FSum* of *TFunc* or *FSumUnit* by $A^{\rightarrow}$, and the *FSum* of other terms (which contain *RTFunc*s) by $A^{\leftarrow}$, for example, $A^{\rightarrow} = \bigcup_{i=1}^n (\Lambda\mathbf{v}.t_i)$ and $A^{\leftarrow} = \bigcup_{i=1}^n p_i(\Lambda^{-1}\mathbf{v}.t_i)p_i$. Using this notation, an inequality can be written in the following form.

$$\mathbf{X} \sqsupseteq (\mathbf{A}^{\rightarrow} \sqcup \mathbf{A}^{\leftarrow})\mathbf{X} \tag{6}$$

Example 3. Suppose (5). $\mathbf{A}^{\rightarrow} = \begin{bmatrix} \Lambda\mathbf{v}.f(Y) \sqcup \Lambda\mathbf{v}.f(a) \\ \Lambda\mathbf{v}.f(X) \sqcup \Lambda\mathbf{v}.a \end{bmatrix}$. $\mathbf{A}^{\leftarrow} = \begin{bmatrix} p_1\Lambda^{-1}\mathbf{v}.f(X)p_2 \\ p_2\Lambda^{-1}\mathbf{v}.f(Y)p_1 \end{bmatrix}$. $\mathbf{A}^{\leftarrow}\mathbf{A}^{\rightarrow} = \begin{bmatrix} p_1\Lambda^{-1}\mathbf{v}.f(X)\Lambda\mathbf{v}.f(X) \sqcup p_1\Lambda^{-1}\mathbf{v}.f(X)\Lambda\mathbf{v}.a \\ p_2\Lambda^{-1}\mathbf{v}.f(Y)\Lambda\mathbf{v}.f(Y) \sqcup p_2\Lambda^{-1}\mathbf{v}.f(Y)\Lambda\mathbf{v}.f(a) \end{bmatrix} = \begin{bmatrix} \Lambda\mathbf{v}.X \\ \Lambda\mathbf{v}.Y \sqcup \Lambda\mathbf{v}.a \end{bmatrix}$, by equations $\begin{array}{ll} p_i(\Lambda^{-1}\mathbf{v}.f(v_i))\Lambda\mathbf{v}.f(t) &= \Lambda\mathbf{v}.t \\ p_i\Lambda^{-1}\mathbf{v}.f(v_i)\Lambda\mathbf{v}.a &= \lambda\mathbf{v}.\emptyset \end{array}$.

We generalize the idea illustrated in Example 3. $\mathbf{A}^{\leftarrow}\mathbf{A}^{\rightarrow}$ can be expanded to an *FProd* of *FSum* of terms of the form $p_k(\Lambda^{-1}\mathbf{v}.t)(\Lambda\mathbf{v}.t')$ or $\lambda\mathbf{v}.\emptyset$. In general, not all terms of the form $p_k(\Lambda^{-1}\mathbf{v}.t)(\Lambda\mathbf{v}.t')$ can be converted to a simpler, equal term. The inequalities shown as follows, where $ll' \in \mathsf{Pos}, l \in \mathsf{Pos}$, can be used to simplify them to simpler terms.

$$p_k(\Lambda^{-1}\mathbf{v}.t)(\Lambda\mathbf{v}.t') \sqsubseteq \begin{cases} p_k(\Lambda^{-1}\mathbf{v}.t'')p_{k'} & \sigma = mgu(t', t), \sigma(v_k) = v_k, \\ & v_k \in fv(t''), t'' = \sigma(v_{k'}) \notin \mathsf{Var} \\ \Lambda\mathbf{v}.t'' & \sigma = mgu(t', t), \sigma(v_k) = t'' \\ \lambda\mathbf{v}.\emptyset & t', t \text{ not unifiable} \end{cases}$$

For example, if $\mathbf{v} = [X, Y]$, then $p_1(\Lambda^{-1}\mathbf{v}.f(f(X)))(\Lambda\mathbf{v}.f(Y)) \sqsubseteq p_1(\Lambda^{-1}\mathbf{v}.f(X))p_2$, $p_1(\Lambda^{-1}\mathbf{v}.f(X))(\Lambda\mathbf{v}.f(f(Y))) \sqsubseteq \Lambda\mathbf{v}.f(Y)$, $p_1(\Lambda^{-1}\mathbf{v}.f(f(X)))(\Lambda\mathbf{v}.a) \sqsubseteq \lambda\mathbf{v}.\emptyset$. If two simplifications are applicable simultaneously, then we choose one arbitrarily. The reduced term can be rearranged into an equal normal form. If the original term is $\mathbf{T}$, then we denote the normal form of the reduced term by $s(\mathbf{T})$. $s(\mathbf{T}) \sqsupseteq \mathbf{T}$ for any $\mathbf{T}$.

Algorithm 2. *(CFT) Given an inequality $\mathbf{X} \sqsupseteq \mathbf{A}\mathbf{X}$, compute $\mathbf{B}_k$ as follows.*

$$\mathbf{B}_0 = \mathbf{A}$$
$$\mathbf{B}_{k+1} = \mathbf{B}_k \sqcup s(\mathbf{B}_k^{\leftarrow}\mathbf{B}_k^{\rightarrow})$$

Theorem 3. *(Termination) For any constraint $\mathbf{X} \sqsupseteq \mathbf{A}\mathbf{X}$, there is an integer N s.t. $\mathbf{B}_N = \mathbf{B}_{N+1}$.*

Lemma 2. $\bigcup_{i=0}^{\infty}(\mathbf{B}_N^i)^*\emptyset = \bigcup_{i=0}^{\infty}((\mathbf{B}_N^{\rightarrow})^i)^*\emptyset$.

Theorem 4. *(Soundness) If $I(\mathbf{v}) = \bigcup_{i=0}^{\infty}((\mathbf{B}_N^{\rightarrow})^i)^*\emptyset$, then $I \models \mathbf{X} \sqsupseteq \mathbf{A}\mathbf{X}$.*

Unlike $\mathbf{A}$, $\mathbf{B}_N^{\rightarrow}$ does not contain *RTFunc*s. Constraint $\mathbf{X} \sqsupset \mathbf{B}_N^{\rightarrow}\mathbf{X}$ can be straightforwardly converted to BNF productions by syntactically converting "$\cup$" to "$|$" and "$\sqsupset$" to "$\rightarrow$". The resulting grammar generates the solution I.

Example 4. Suppose (5). $\mathbf{B}_0 = \mathbf{A}$. $s(\mathbf{A}^{\leftarrow}\mathbf{A}^{\rightarrow})$ is shown in (b). $\mathbf{B}_1$ is shown in (a). $s(\mathbf{B}_1^{\leftarrow}\mathbf{B}_1^{\rightarrow}) \sqsubset \mathbf{B}_1$. Therefore, we may choose $N = 1$. $\mathbf{X} \sqsupset \mathbf{B}_1^{\rightarrow}\mathbf{X}$ is shown in (c). A grammar is generated as shown in (d). The solution generated by this grammar is $I(\mathbf{v}) = \{[f(a)], [f(f(f(a)))], \ldots\} \times \{[a], [f(f(a))], \ldots\}$.

$$\begin{bmatrix} p_1\Lambda^{-1}\mathbf{v}.f(X)p_2 \sqcup \Lambda\mathbf{v}.f(Y) \sqcup \Lambda\mathbf{v}.f(a) \sqcup \Lambda\mathbf{v}.X \\ p_2\Lambda^{-1}\mathbf{v}.f(Y)p_1 \sqcup \Lambda\mathbf{v}.f(X) \sqcup \Lambda\mathbf{v}.a \sqcup \Lambda\mathbf{v}.Y \end{bmatrix}$$

(a)

$$\underset{\text{(b)}}{\begin{bmatrix} \Lambda\mathbf{v}.X \\ \Lambda\mathbf{v}.Y \sqcup \Lambda\mathbf{v}.a \end{bmatrix}} \quad \underset{\text{(c)}}{\begin{bmatrix} X \\ Y \end{bmatrix} \sqsupset \begin{bmatrix} f(Y) \cup f(a) \cup X \\ f(X) \cup a \cup Y \end{bmatrix}} \quad \underset{\text{(d)}}{\begin{matrix} X \rightarrow f(Y)|f(a)|X \\ Y \rightarrow f(X)|a|Y \end{matrix}}$$

5 Related Work

The (Inst-Gen) rule introduced in [11] uses unification as guide for instance generation. The algorithms presented in [12] also make use of unification to find blockages. [13] shows that using a proper semantics for OSHL is implicitly performing unification. One of the differences between our approach and others is that we infer types before proof search, which divides a theorem proving algorithm into the static analysis stage and the run-time stage. About clause linking, we have the following observation(probably others also have the same observation): in order for any instance $C\sigma$ of C to be useful, each literal $L\sigma$ in $C\sigma$ should be resolvable with some other literals. Given a set of clauses S, a clause C, and a literal L in C, we may *decompose* C on L w.r.t. S to a set of instances of C, $\{C\sigma_i | i \in \{1, 2, \ldots, n\}\}$, s.t. for any literal N in S, σ_i is an mgu of L and $\overline{N}$ for some $i \in \{1, 2, \ldots, n\}$ without affecting the completeness. Our approach generates criteria inspired by this observation. For a survey of instance-base methods, refer to [14].

Sort inference algorithms are implemented in some finite model finders such as Paradox[10] and FM-Darwin[9]. Sorted unification[15] uses sets of unary predicates as sorts. A key difference between our approach and multi-sorted logic is that the types are inferred from S, instead of being part of the logic itself. Therefore, we need to guarantee that the types inferred are backward compatible with the untyped version S. Theorems in this paper is essential for our approach but irrelevant to a multi-sorted logic without inferred sort. Our approach can be viewed as a type system[16] that is equipped with a type inference algorithm. The difference between our type system and those that are usually found in programming languages is that our type system is used to restrict instance generation while type systems in programming languages are usually used to eliminate expressions whose reduction leads to errors. In advanced compilers, some type systems are also used to guide optimization, in which sense it is similar to our type system. The presentation of this paper makes (very primitive) use of domains[17] and is loosely related to a type theory[18] in the sense that a type system is related to a type theory.

A closely related concept to grammar is tree grammar, where the rhs is composed of trees instead of sequences. The choice of grammar reflects the fact that terms can be internally represented by sequences instead of trees.

6 Discussion

It is possible to refine the constraints (2) and (3) by transforming them into $FInter$s. We denote the $FSum$ of rhs of all constraints (2) and (3) generated by L by $A^L\mathbf{X}$. With this notation, we can combine multiple constraints (2) and (3) generated by L into one constraint: for every $L \in C, C \in S, v \in fv(L)$, $I \models v_L \sqsupset \sqcap_{L\in C} A^L\mathbf{X}$.

The granularity of types is the key to the efficacy of these types. CFT is useful for some problems as shown in Table 1.[2] Finer granularity may be achieved by using enhanced grammars. These grammars are useful when a variable occurs twice in a literal, for example, $P(f(X,X))$. If we have a literal $\neg P(Y)$, then productions produced by CFT include $Y \to f(X,X)$ which does not reflect the correlation between the two occurrences of X. If we make use of an enhanced grammar to mark that X should always be instantiated to the same term, then we have better approximation to the minimum solution. A possible solution is adding correlation markers to correlated variables.

Table 1. Comparison of Performance with and without CFT in OSHL-S 0.5.0

Problem Set	Total	CFT on	CFT off	Difference
SYN	902	487	454	7.3%
PUZ	70	54	49	10.2%
SWV	479	136	123	10.5%

The constraints generated by CFT may have redundancy which affects the time and space complexity of the term generator. We implemented the following algorithms to reduce redundancy. Suppose that $\mathbf{X} \sqsupset \mathbf{T}$, where $\mathbf{X} = [v_k]_{k=1}^n, \mathbf{T} = [T_k]_{k=1}^n, T_i = \bigcup_{s=1}^m T_{i,s}, i \in \{1,2,\ldots,n\}$. In each iteration of Algorithm 2, if $T_{i,o} = v_j$ and $T_{j,p} = v_i$, then put v_i, v_j into an equivalence class; choose a representative for each equivalence class; for every representative v_i and $v_j \neq v_i$ in the equivalence class, set T_i to $T_i \cup T_j$ and T_j to v_i and replace v_j by v_i in $\mathbf{T}$; and for every T_i s.t. $T_{i,o} = v_i$ for some o, remove v_i from T_i. A variable is *nontrivial* if the rhs of its production contains a term containing only variables that are nontrivial. We slightly modify constraint (4): for some $\mathbf{T} = [T_i]_{i=1}^n$,

[2] Problem sets: TPTP 3.2.0, unsatisfiable or theorem only. Time limit: 30s. Running environment: P4 2.4GHz, 512M, Windows XP, Sun JDK 6.0, OSHL-S 0.5.0, Prover-Tools. Website for the prover and tools: http://cs.unc.edu/~xuh/oshls/.

where $T_i = c$ or $\emptyset$, for $i \in \{1, 2, \ldots, n\}$ so that there is no trivial variable, $\mathbf{X} \sqsupset \mathbf{T}$. To find a minimal $\mathbf{T}$, first mark all trivial variables; then, break a chain of trivial variables by adding a constant to one of them; repeat the process until there is no trivial variable. Other optimizations include inlining productions that contain no variable and removing redundant terms.

References

1. Plaisted, D.A., Zhu, Y.: Ordered semantic hyper linking. In: J. Autom. Reason. (2000)
2. Baumgartner, P., Fuchs, A., Tinelli, C.: Darwin: A theorem prover for the model evolution calculus. In Schulz, S., Sutcliffe, G., Tammet, T., eds.: IJCAR ESFOR (aka S4). Electronic Notes in Theoretical Computer Science (2004)
3. Claessen, K.: Equinox, a new theorem prover for full first-order logic with equality. Presentation at Dagstuhl Seminar 05431 on Deduction and Applications (October 2005)
4. Korovin, K.: iProver — an instantiation-based theorem prover for first-order logic (system description). In: Proc. IJCAR '08, Berlin, Heidelberg, Springer-Verlag (2008)
5. Letz, R., Stenz, G.: DCTP: A disconnection calculus theorem prover. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proc. IJCAR-2001, Siena, Italy. Volume 2083 of LNAI., Springer, Berlin (June 2001)
6. Sutcliffe, G.: CASC-J4 the 4th IJCAR ATP system competition. In: Proc. IJCAR '08, Berlin, Heidelberg, Springer-Verlag (2008)
7. Schulz, S.: System abstract: E 0.81. In: Proc. 2nd IJCAR. Volume 3097 of LNAI., Springer (2004)
8. Riazanov, A., Voronkov, A.: The design and implementation of vampire. AI Commun. **15**(2-3) (2002)
9. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic **7** (2009)
10. Claessen, K.: New techniques that improve mace-style finite model finding. In: Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications. (2003)
11. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. LICS (2003)
12. Hooker, J.N., Rago, G., Chandru, V., Shrivastava, A.: Partial instantiation methods for inference in first-order logic. J. Autom. Reason. **28**(4) (2002)
13. Plaisted, D.A., Miller, S.: The relative power of semantics and unification. In Andreas Podelski, A.V., Wilhelm, R., eds.: Workshop on Programming Logics in memory of Harald Ganzinger, Saarbruecken, Germany (2005)
14. Jacobs, S., Waldmann, U.: Comparing instance generation methods for automated reasoning. J. Autom. Reason. **38**(1-3) (2007)
15. Weidenbach, C.: First-order tableaux with sorts. Logic Journal of the IGPL **3**(6) (1995) 887–906
16. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
17. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science, Clarendon Press (1994) 1–168
18. Coquand, T.: Type theory. In Zalta, E.N., ed.: The Stanford Encyclopedia of Philosophy. (Spring 2009)

Author Index