ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/47861702

IJCAR'06 Workshop : Disproving'06: Non-Theorems, Non-Validity, Non-Provability

Article

Source: OAI

CITATION
1

READS
25

3 authors:

Wolfgang Ahrendt
Chalmers University of Technology
53 PUBLICATIONS 699 CITATIONS

SEE PROFILE

Peter Baumgartner
National ICT Australia Ltd
148 PUBLICATIONS 1,980 CITATIONS

SEE PROFILE

Hans de Nivelle
University of Wroclaw
60 PUBLICATIONS 823 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

StaRVOOrS View project

Automating/Formalizing Reasoning about Maintainability of UML Software Designs View project

All content following this page was uploaded by Peter Baumgartner on 22 May 2014.

The user has requested enhancement of the downloaded file.

IJCAR 2004

Second International Joint Conference on Automated Reasoning

University College Cork, Cork, Ireland

Workshop Programme

Workshop on Disproving: Non-Theorems, Non-Validity, Non-Provability

Wolfgang Ahrendt, Peter Baumgartner,
Hans de Nivelle (Chairs)

WS 1 – July 5

Preface

Our field is called *automated theorem proving* because traditionally it has been concerned with the art of finding proofs automatically. In the beginning, researchers were motivated by the wish to build computer systems that can automatically solve hard, mathematical problems. When searching for a hard proof, it is acceptable for a system to eat up all resources and not to recognise false theorems.

However in the last years, one has become aware of the fact that for applications, one also needs to be able to efficiently identify non-theorems. For example, automated theorem proving systems are now being used as assistants which must automatically solve easy subtasks in large, interactive projects. For such problems, the expectations to the automated theorem prover are different: The input problems are not terribly hard, usually contain additional irrelevant information, and often they are not provable. In case the subgoal is incorrect, it is not acceptable to simply remain silent and consume all resources in an interactive system.

Apart from the applications, the field of disproving has triggered many interesting theoretical research questions, which are interesting on their own. For example, one of the contributed papers addresses the problem of how to repair (modify) a non-theorem in such a way that it becomes a theorem.

We also have two contributions about finding counter-models in non-standard logics. One of the contributions addresses this problem for resource logics, the other for Gödel-Dummett logic.

The workshop consists of seven contributed talks and one invited talk by Alan Bundy with title *Finding and Using Counter Examples*. In addition, we share an invited speaker, Toby Walsh, with the Workshop on Pragmatics of Decision Procedures in Automated Reasoning. We are grateful to the organisers, Silvio Ranise and Cesare Tinelli, to make this possible. We thank our PC for their reviewing efforts:

Christian Fermüller	Ulrich Furbach
Bernhard Gramlich	Deepak Kapur
Bill McCune	Renate Schmidt
Carsten Schürmann	Graham Steel
Cesare Tinelli	Andrei Voronkov

We are indebted to Andrei Voronkov for providing an installation of his PC-expert system, which greatly simplified organising the reviewing work.

June 2004,

Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle

Contents

Unsound Theorem Proving
Christopher Lynch **1**

The TM System for Repairing Non-Theorems
Simon Colton and Alison Pease **13**

Bounded Model Generation for Isabelle/HOL
Tjark Weber **27**

Reducing Symmetries to Generate Easier SAT Instances
Jian Zhang **37**

Finding and Using Counter-Examples (invited talk)
Alan Bundy **45**

The Use of Proof Planning Critics to Diagnose Errors in the Base Cases of Recursive Programs
Louise A. Dennis **47**

Resource Graphs and Countermodels in Resource Logics
Didier Galmiche and Daniel Méry **59**

Gödel-Dummett counter-models through matrix computation
Dominique Larchey-Wendling **77**

Unsound Theorem Proving

Christopher Lynch *

June 16, 2004

Abstract

We discuss the benefits of complete unsound inference procedures for efficient methods of disproof. We give a framework for converting a sound and complete saturation-based inference procedure into successive unsound and complete procedures, that serve as successive approximations to the theory. The idea is to successively add new statements in such a way that the inference procedure will halt. Then the satisfiability is evaluated over a stronger theory.

We illustrate this framework with Knuth-Bendix Completion, and show that in some theories these successive approximations become weaker and weaker, and sometimes become a decision problem. Then we illustrate the framework with a new method for the (nonground) word problem, based on Congruence Closure. We show a class where this becomes a decision procedure. Also, we show that this new inference system is interesting in its own right. Given a particular goal, in many cases we can halt the procedure at some point and say that all the equations for solving the goal have been generated already. This is generally not possible in Knuth-Bendix Completion.

1 Introduction

The major problem in automated theorem proving, that of deciding the unsatisfiability of a set of statements, is undecidable in general. This is true in first order logic and equational logic, for example. There exist sound and complete theorem provers. If a theorem prover is sound, then that means that when it gives a proof of a theorem, you are guaranteed that it is correct. If a theorem prover is complete, then when a conjecture is true, a proof is guaranteed to be found.

Automated theorem provers have been used to prove difficult theorems. But the search space is so large that in practice, more efficient incomplete sound theorem provers are often used. Then proofs can be trusted, but disproofs cannot.

Our focus in automated theorem proving is not in finding proofs for difficult mathematical theorems. Instead, we are interested in using theorem provers to solve verification problems. When used in that context, many of the conjectures given to a theorem prover will be false. So we would like to be able to trust a result that a conjecture is false. In that case, incomplete theorem provers are useless. Also, complete and sound theorem provers are generally not so efficient.

*Department of Math and Computer Science, Clarkson University, Potsdam, New York, clynch@clarkson.edu (on sabbatical at Naval Research Laboratory)

Therefore, we create a framework for unsound complete theorem provers for disproving conjectures. This technique is useful to weed out theorems which are obviously not true. There are simple examples where theorem provers run forever trying to solve conjectures that are trivially false to a human. In addition, we could use an unsound and complete theorem prover in combination with a sound and incomplete theorem prover to approximate a conjecture from both sides.

The framework given in this paper is for saturation theorem provers, which operate by continually inferring new statements implied by previous statements. Our framework consists of a modification to a saturation theorem prover. We modify it by adding statements that are not necessarily implied by previous statements. These potentially false statements are chosen in such a way as to force the theorem prover to halt. The effect of this is to evaluate a stronger conjecture. The procedure is complete, so that if the stronger conjecture is false, then the given conjecture is false. The theorem prover has approximated the theory with a stronger theory: an unsound approximation.

We iterate this process. We run the theorem prover again, but this time try to approximate the theory with a weaker theory than before In this way, we continually approximate the given theory.

In our framework, we have also shown that it is possible to create a weak approximation, and gradually attempt to make this approximation stronger. We iterate the construction of the two approximations, one strong and one weak. Anything found true in the weak one is true, and things found false in the strong one are false. In some cases this becomes a decision procedure.

For examples, we instantiate this framework with two concrete inference systems. First is Knuth-Bendix Completion [3], where we show a class of theories where this becomes a decision procedure. However, the direct purpose of this paper is not in finding new decision procedures; that is only an example of the kinds of things that can be done within this framework.

The second inference system which we use to instantiate this framework is a new inference procedure, as far as we are aware. It is based on Abstract Congruence Closure, for ground equational theories [4, 6] which we extend to nonground theories. For this inference system, we show that it is sometimes possible to examine the set of equations during the derivation and deduce that the conjecture can never be proved from this point. This is generally impossible to do in theorem proving, because even if the equations become large, it is always possible that an equation can be simplified to a smaller one.

In Section 2, we introduce theorem proving derivations, and give the framework for unsound theorem proving. In Section 3, we instantiate the framework with Knuth-Bendix Completion. In Section 4, we introduce Nonground Congruence Closure and instantiate the framework with that. We conclude the paper with a discussion of how to apply this method to Resolution and Paramodulation inference systems, and a comparison with related work. This paper does not contain any of the proofs. All of the proofs and all of the technical details can be found at `www.clarkson.edu/~clynch/papers/uf.ps/`

2 Framework

Basic definitions of Theorem Proving Derivations are from [2, 9]. A *saturation* inference system is an inference system that starts with some set of statements, and uses transformation rules to create new statements and delete old ones. Transformation rules are of the form $\Gamma \longrightarrow \Delta$, where Γ and Δ are both sets of statements. The meaning of a transformation rule is that the statements in Γ should be replaced bhy the statements of Δ. There are two kinds of transformation rules: inference rules and deletion rules. *Inference rules* are of the form $\{C_1, \cdots, C_n\} \longrightarrow \{C_1, \cdots, C_n, C\}$. It indicates that in the presence of $C_1, \cdots, C_n$, C should be added. We will write that inference rule in the following notation:

$$\frac{C_1 \cdots C_n}{C}$$

Deletion rules will be of the form $\{C_1, \cdots, C_n\} \longrightarrow \{C_2, \cdots, C_n, D_1, \cdots, D_m\}$. This means that if the statements $C_1, \cdots, C_n$ exist in the current set of statements, then C_1 should be deleted and $D_1, \cdots, D_m$ added. Inference rules represent rules that **must** be performed in an inference procedure, and deletion rules **may** be performed if desired.

Given a set of inference rules I and deletion rules D, an (I,D) *theorem proving derivation* is a (possibly infinite) sequence $S_1, S_2, \cdots$ of sets of statements such that each S_{i+1} is obtained by applying an inference rule from I or a deletion rule from D to clauses of S_i. We define $S_\infty = \bigcup_{i>=1} \bigcap_{j>=i} S_j$ The clauses in S_∞ represent the set of *persistent* statements, i.e., the statements that are never deleted. Given a set of inference and deletion rules, since we assume they are applied according to some strategy, then we can assume there is one theorem proving derivation for each set of statements.

We assume a well-founded ordering $<$ on the statements. Based on that ordering, there is a notion of redundancy. A statement C is *redundant* in S if there are statements $C_1, \cdots, C_n \in S$ such that each $C_i < C$ for all i, and $C_1, \cdots, C_n \models C$. We will construct the deletion rules so that they cannot be performed unless C_1 is redundant in $\{C_2, \cdots, C_n, D_1 \cdots D_m\}$. A set of statements S is said to be *saturated* if the conclusion of every inference rule from S is either in S or is redundant in S. A theorem proving derivation $S_1, S_2, \cdots$ is *fair* if for every inference from S_∞ with conclusion C, there exists an i such that $C \in S_i$ or C is redundant in S_i. If $S_1, S_2, \cdots$ is fair, then S_∞ is saturated.

The inference rules are *sound* if $C_1, \cdots, C_n \models C$. The deletion rules are *sound* if $C_1, \cdots, C_n \models D_i$ for all i. Inference rules are designed so that each saturated set has certain properties. The most common is the refutational property. In that case, we distinguish a new atom called $\bot$, usually called the empty clause, which indicates that a set of statements is unsatisfiable. We have the important definitions of *soundness* and *completeness* of an inference system.

Definition 1 *A set of inference rules I and deletion rules D is* sound *if for every fair theorem proving derivation $S_1, S_2, \cdots$, if $\bot \in S_\infty$ then S_1 is unsatisfiable. I is* complete *if $\bot \in S$ for every saturated and unsatisfiable S.*

It is obvious that a set of inference and deletion rules is sound if each individual inference and deletion rule is sound.

Ideally, a theorem prover should be sound and complete. Then we are guaranteed that the existence or non-existence of $\perp$ determines whether a set of statements is satisfiable or not. Of course, the problem of theorem proving is, in general, undecidable for first order logic. So, in practice, theorem provers that have proved important results are not always complete. For example, the Robbins Algebra problem was proved with an incomplete theorem prover [8].

Throughout the history of automated theorem proving, until very recently, much of the emphasis has been on solving very hard theorems. A theorem proving contest is run every year at the CADE conference, with the main emphasis on proving unsatisfiability. In the past, theorem prover developers have come up with methods which destroy the completeness while retaining soundness, because this usually helped to find theorems faster. A simple example of a sound and incomplete strategy is that strategy which discards every clause with more than a given number of symbols.

In this paper, we are interested in satisfiability. Therefore, we will develop strategies that destroy soundness but do not destroy completeness. Soundness of an inference system is implied by soundness of the inference and deletion rules. We will relax that requirement. In particular, we will allow unsound deletion rules, while still requiring the inference rules to be sound. We will keep the requirement that deletion rules only remove redundant statements. Therefore, the inference systems we consider will still be complete, but not sound.

In the rest of this section, five different ideas will be discussed. First is the idea of *Unsound Theorem Proving*. That is the idea of modifying a sound and complete theorem proving procedure so that it may be unsound, but that it remains complete and it terminates, so that it can decide satisfiability in some cases. The second well-known ideas is *Incomplete Theorem Proving* which modifies a sound complete theorem proving procedure so that it may become incomplete, but it remains sound and it terminates. This procedure can show unsatisfiability but not satisfiability. The third idea is *Iterative Unsound Theorem Proving*. This iterates Unsound Theorem Proving, with the goal of becoming more and more sound each time, thereby proving the satisfiability of more statements. The third idea is *Iterative Incomplete Theorem Proving* which iterates Incomplete Theorem Proving, with the goal of becoming more complete each time. The final idea is *Iterative Unsound and Incomplete Theorem Proving*, which simultaneously iterates Unsound and Incomplete Theorem Proving.

The idea of Unsound Theorem Proving is presented now. After each inference rule is performed, we look at the conclusion. In some cases, we will keep the conclusion. In other cases, we will perform a deletion rule where the statements $D_1, \cdots, D_m$ might not follow from previous statements. Therefore, the inference rules remain sound, but the deletion rules do not. This gives us an unsound but complete finite theorem proving derivation. We will assume that all the D_i come from a predetermined finite set F, so it will prove whether or not a larger set of statements is unsatisfiable. If this larger set is satisfiable, the original set is satisfiable. Since F is finite, this procedure must halt.

For Incomplete Theorem Proving, we also look at the conclusion of each inference. If the conclusion is not in F, then we do not add it. Therefore, we lose completeness. But since F is finite, the procedure terminates.

For Iterative Unsound Theorem Proving, we run the Unsound Theorem Proving Procedure. If this returns "unsatisfiable" to us, then we cannot be sure that the answer

is correct, because of unsoundness. So we repeat the procedure with a larger set F. And this process is iterated. Iterative Incomplete Theorem Proving is similar, except that in this case we cannot trust a result of "satisfiable", so in that case we iterative Incomplete Theorem Proving for a larger value of F.

Iterative Unsound and Incomplete Theorem Proving is a combination of the two processes. We choose an F, then run the Unsound Theorem Proving procedure for that F. If it returns "unsatisfiable", we run the Incomplete Theorem Proving procedure for the same F. If that returns "satisfiable" we choose a larger value of F and iterate.

Define an F-replacement deletion rule as follows:

Definition 2 *Let F be a set of statements. A deletion rule*

$$\frac{C}{D_1 \cdots D_m}$$

is an F-replacement *if $C \notin F$ and $D_1, \cdots, D_m \in F$.*

Non-F replacement rules will always be designed so that every clause not in F is the premise of a non-F replacement rule.

If I is a set of inference rules, and D is a set of deletion rules containing an F-replacement rule, and F is a finite set, then every (I, D) derivation is finite, because only statements from F are saved. The key idea of this paper is that a complete set of inference and deletion rules can be augmented with an F-replacement rule, so that any derivation from the augmented set of rules will halt, and if $\perp$ is not generated from S then S is satisfiable.

We describe the Iterated Unsound and Incomplete Theorem Proving Process. Let S be the set of statements for which we want to decide satisfiability. Let I and D be a sound and complete set of inference and deletion rules. Let $\hat{F} = F_1, F_2, \cdots$ be a monotonic sequence of finite sets of statements, i.e., $F_k \subseteq F_{k+1}$ for all k. We define $F_\infty = \bigcup_{k \geq 1} F_k$. For each k let I_k be a modification of I such that all inferences with a conclusion in F_k are not performed. For each k, let D_k be D augmented with an F_k replacement rule. Then the $(I, D, \hat{F})$ *derivation from* S is the following:

1. Let $k = 1$
2. Let $S_{i,1}, S_{i,2}, \cdots$ be an (I, D_k) derivation with $S_{i,1} = S$.
3. Let $S'_{i,1}, S'_{i,2}, \cdots$ be an (I_k, D) derivation with $S'_{i,1} = S$.
4. If $\perp \notin S_\infty$, halt and say SATISFIABLE.
5. If $\perp \in S'_\infty$, halt and say UNSATISFIABLE.
6. Let $k = k + 1$
7. Go to 2

We will show that this process is sound. In fact we extend the definition of soundness to say that if $\perp$ is not produced or if the function returns SATISFIABLE, then S is satisfiable. The process is complete if F_∞ contains all statements.

Theorem 1 *Let I and D be a sound and complete set of inference and deletion rules. Let $\hat{F}$ be a monotonic sequence of finite sets of statements. Suppose that every (I, D) derivation from $S \subseteq F_\infty$ only produces statements in F_∞*[1]*. Let $S \subseteq F_\infty$. Then if*

[1]This is true, for example, if F_∞ contains all statements.

the $(I, D, \hat{F})$ *derivation returns SATISFIABLE (resp. UNSATISFIABLE) then* S *is satisfiable (resp. unsatisfiable). Also, if* S *is unsatisfiable, then the* $(I, D, \hat{F})$ *derivation returns UNSATISFIABLE.*

The proof is due to the fact that each (I, D_k) derivation is finite and complete, and each (I_k, D) derivation is finite and sound.

We often consider F_∞ to be the set of all statements, then it is trivially the case that all derivations from $S \subseteq F_\infty$ only contain statements in F_∞.

We point out some of the benefits of this procedure over the (I, D) procedure. First of all, if a set of statements is satisfiable, then this procedure is more likely to give an answer. There are simple cases where sound and complete derivations will not halt. We give some examples later in the paper. Also, suppose that we have a set of satisfiable statements S, for which the (I, D) derivation is finite. It still might be better to use an $(I, D, \hat{F})$ derivation, because the proof of satisfiability might be simpler. A stronger theory may have a smaller saturated set, and therefore a smaller proof.

The $(I, D, \hat{F})$ derivation might actually become a decision procedure. We give some examples later in the paper. An example of this is when for every satisfiable set of statements S, there is a k such that $\bot$ is not in the (I, D_k) derivation.

Theorem 2 *Let* (I, D) *be sound and complete. Let* $\hat{F}$ *be a monotonic sequence of finite sets of statements such that all derivations from* $S \subseteq F_\infty$ *only contain statements in* F_∞*. Let* $\hat{G}$ *be a sequence such that all* $G_k \subseteq F_k$*. Suppose that for every satisfiable* $S \subseteq G_\infty$*, there is a* k *such that* $\bot$ *is not in the* (I, D_k) *derivation from* S*. Then the* $(I, D, \hat{F})$ *procedure is a decision procedure for all* $S \in G_\infty$*.*

It is not necessary to know that the $(I, D, \hat{F})$ procedure is a decision procedure in order for it to be one, whereas it is necessary to know in advance if the (I_k, D) derivation is a decision procedure in order for it to be one. We will discuss that issue further in the next section.

These ideas can also be applied to existential problems, i.e., unification problems. In that case, the (I, D_k) and the (I_k, D) derivation both produce a complete set of unifiers. One is an over-approximation, and one is an under-approximation. This could be a useful way to approximate unification.

3 Knuth-Bendix Completion

We have presented an abstract framework for using unsound theorem proving to determine satisfiability, and develop decision procedures. But that framework is not useful unless some interesting examples fit into the framework. In particular, what are the F_k in the sequence $F_1, F_2, \cdots$, and even more important, what are the values of the D_k used in the F_k replacement deletion rules.

Next we extend this framework to Knuth-Bendix Completion [3], which is an inference system over equations $s \approx t$ and disequations $s \not\approx t$. Completion consists of Inference rules **Critical Pair**, **Narrowing** and **Equation Resolution**, plus Deletion rule **Simplification**. Now we will apply our framework to Knuth-Bendix Completion. First we define the sequence $F_1, F_2, \cdots$.

Definition 3 *Given a term t, let $|t|$ be the number of non-variable symbols in t. Then F_k is the set of equations $s \approx t$ and disequations $s \not\approx t$ such that $|s| \leq k$ and $|t| \leq k$.*

Clearly F_∞ is the set of all equations and disequations. Another possibility for F_k is to let k be a limit on the depth of the terms in F_k. The F_k replacement deletion rule is to add an equation that subsumes one not in F_k:

Definition 4 *The* Unsound Subsumption *deletion rule is the rule $\{A\} \longrightarrow \{A'\}$ where $A'\sigma = A$ for some σ, $A \notin F_k$ and $A' \in F_k$.*

This replaces an equation or disequation A, with A' where A' is a new equation or disequation that strictly subsumes A, and $A' \in F_k$. It is easy to find such an A'. It is just necessary to replace subterms in A with variables. The best idea is to replace as few subterms as possible, so that A' is in F_k but not in F_{k-1}. We will assume that we will always select the subterm to replace from the side of the equation with the most symbols, if one side has more symbols.

Next we show how the $(I, D, \hat{F})$ derivation becomes a decision procedure for some theories (sets of equations). For example, consider the theories E we will call *size preserving linear theories*:

Definition 5 *Let E be a set of equations. Then E is* size preserving linear *if and only if for every $s \approx t \in E$, $|s| = |t|$ and each variable that occurs in $s \approx t$ occurs exactly once in s and once in t.*

Definition 6 *For all k, define G_k to be the set of size preserving linear equations in F_k, and all ground disequations in F_k.*

The following theorem is implied by the fact that once an equation e outside of G_k is created, then any rule with e as one of its premises will have a conclusion that is not in G_k.

Theorem 3 *Let (I, D) be the Inference and Deletion Rules of KB Completion. Then (I_k, D) is a decision procedure for all members of G_k.*

There is a similar theorem for unsound derivations.

Theorem 4 *Let I and D be the Inference and Deletion Rules of KB Completion. Then $(I, D, \hat{F})$ with Unsound Subsumption is a decision procedure for all S in G_∞.*

As we did in the framework, we once again point out the distinction between those two theorems. For Theorem 3, it is necessary to know in advance that (I_k, D) will be a decision procedure. But for Theorem 4, $(I, D, \hat{F})$ is a decision procedure, and it is not necessary to know that in order for it to be one. For example, $(I, D, \hat{F})$ is a decision procedure for the theory $\{f(f(x)) \approx g(f(x)), h(a) \approx b\}$, whereas (I_k, D) would require a new theorem in order to turn it into a decision procedure. It is worth pointing out that Knuth-Bendix Completion will normally not halt for many size preserving linear theories, such as $\{f(f(x)) \approx g(f(x))\}$.

4 Nonground Congruence Closure

Now we extend the Abstract Congruence Closure algorithm of [6, 4] to equations with variables. That algorithm works by creating new constants representing equivalence classes of terms. In our approach, we create new function symbols in addition to new constants. The function symbols when applied to terms represent equivalence classes. So the function symbol itself represents a parametrized equivalence class. We apply the Knuth-Bendix procedure to the flattened equations. The result might not be flat. Therefore, we flatten the conclusion of the inference, and create a new function symbol. This process can go on forever, but it is still complete, if we order the new (possibly infinitely many) symbols in a well-founded way.

There are some advantages of this approach over Knuth-Bendix Completion. Equations are kept small, and inferences are easy to perform. The ordering used is trivial to calculate on flat terms. Also, rewrite chains are polynomial in the number of equations.

Unfortunately, this procedure may not halt on sets of equations where Knuth-Bendix Completion halts. However, when we apply unsound theorem proving to this method, it appears to have advantages over Knuth-Bendix Completion. In many instances of traditional theorem proving, it is possible to tell that if there was a proof we would have found it already. This corresponds to Knuth-Bendix Completion being able to determine that all future equations will be larger than a given size. As far as we know, there is no way to do that in Knuth-Bendix Completion, aside from coming up with some meta-theorem, as in the previous section, or using unsound theorem proving. In the Congruence Closure method, unsound theorem proving is not even necessary. Although we present it to strengthen this approach, and to handle additional classes of equations.

We will now define Nonground Congruence Closure. First, define the height $Ht(t)$ of a term t such that $Ht(x) = 0$ for all variables x, and $Ht(f(t_1, \cdots, t_n)) = 1 + max\{Ht(t_1), \cdots, Ht(t_n)\}$. The *depth* of a subterm s of t will be the maximum depth of s in the tree representation of t. Let $Vars(t)$ be the set of variables in t. Let $root(t)$ be the top symbol of t. We define *flat equations*. All equations can be flatttened.

Definition 7 *An equation $s = t$ is* flat *if (i) $Ht(s) \leq 2$ and $Ht(t) \leq 1$, (ii) $Vars(t) \subseteq Vars(s)$ and t is linear*[2]*, and (iii) if $depth(x, s) = 2$ then x only occurs once in s.*

We are going to consider a signature Σ, and an infinite set of new function symbols $C = \{c_1, c_2, \cdots\}$. Let $\Sigma_C = \Sigma \cup C$ be an extended signature. We assume a total precedence $<_p$ on the symbols, with the requirement that $arity(f) < arity(g)$ implies that $f <_p g$. Furthermore if $i < j$ and $arity(c_i) = arity(c_j)$ then $c_i <_p c_j$. This last fact guarantees that the precedence order is well-founded. From the precedence ordering, we can define an ordering $<_f$ on ground terms.

Definition 8 *Let $s = f(s_1, \cdots, s_n)$ and $t = g(t_1, \cdots, t_m)$. Then $s >_f t$ if*

1. *$|s| > |t|$, or*
2. *$|s| = |t|$ and $f >_p g$, or*

[2] Each variable occurs at most once in t.

3. $|s| = |t|$ *and* $f = g$ *and* $\{s_1, \cdots, s_n\} >_f \{t_1, \cdots, t_n\}$[3].

In the long version we show that $<_F$ is really a well-founded monotonic ordering. We also show that it is simple to compute on flat equations.

The inference and deletion rules for Congruence Closure are the same as the inference and deletion rules for Knuth-Bendix Completion, with the addition of one flattening deletion rule that will be performed once after a non-flat equation is created by an inference or deletion rule.

Flattening:

$$\frac{u \approx v}{u \approx c(x_1, \cdots, x_n) \quad v \approx c(x_1, \cdots, x_n)}$$

where $u \approx v$ is not flat[4], $\{x_1, \cdots, x_n\} = Vars(u) \cap Vars(v)$, and c is a new function symbol from C.

Note that the result of a Critical Pair or Simplification Rule will be an equation $u \approx v$ such that $Ht(u) \leq 2$ and $Ht(v) \leq 2$. Therefore, the conclusion of Flattening will always be a flat equation, since $c(x_1, \cdots, x_n)$ is linear, and all of its variables also appear in u and v.

Compare equations by defining $s \approx t <_f u \approx v$ if $\{s, t\} <_f \{u, v\}$, where $<_f$ is its own multiset extension. Since the two new equations imply the replaced equation, and because the new equations are smaller, this is an instance of removing a redundant equation. Therefore, the Congruence Closure inference system is sound and complete.

Now that the Nonground Congruence Closure inference procedure is defined, we fit it into our framework. We have defined the inference rules so that all equations are flat, but the disequations are not necessarily flat. It would also be possible to flatten the disequations, but we chose not to approach it that way.

For unsound theorem proving, we need to define a sequence $F_1, F_2, \cdots$.

Definition 9 *Let* F_k *be the set of equations and disequations such that for all* $s \not\approx t$ *in* F_k*,* $|s| \leq k$ *and* $|t| \leq k$*, and the only function symbols that can appear in* F_k *are the function symbols of* $\Sigma \cup \{c_1, \cdots, c_k\}$*.*

Then the F_k replacement rule is the *Combine Equivalence Class* deletion rule.

Definition 10 Combine Equivalence Classes *is the deletion rule* $\{u = c_j(x_1, \cdots, x_n)\} \longrightarrow u = c_i(y_1, \cdots, y_m)\}$*, where* $j > k$*,* $i \leq k$*,* $arity(c_i) \leq arity(c_j)$[5]*, and* $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_n\}$*.*

Notice that each F_k is finite, and that the Combine Equivalence Classes rule will replace a term not in F_k with a term in F_k, assuming that A is not a disequation with too many symbols on one side. But we will use this rule in inferences where such rules are never created. The Combine Equivalence Classes rule has the effect of preventing the

[3]Here we mean the multiset extension of $>_f$

[4]For instance, because $Ht(u) = Ht(v) = 2$.

[5]We can assume some initial constant c_i, or set of constants with small arity, if necessary so that this is always posssible.

inference procedure from creating new function symbols at some point, which creates an unsound, complete inference procedure.

Given an equation $s \approx t \in E$, we sometimes write it as $s \rightarrow t$ if $s >_f t$. Then $\rightarrow$ represents the rewrite relation, and $\rightarrow^*$ represents its reflexive and transitive closure. We will define a size function called *minsize* on all symbols and all terms, with respect to a set of equations. The intention will be that $minsize(t, E) = min\{|s| \mid s \in T_\Sigma$ and $s \rightarrow^* t\}$, where $minsize(t) = |t|$ for all $t \in T_\sigma$.

Definition 11 *Let E be a set of equations. If x is a variable, then $minsize(x, E) = 0$. Define $minsize(f(t_1, \cdots, t_n), E) = 1 + \Sigma_{1 \leq i \leq n} minsize(t_i, E)$. Define $minsize(c, E) = min\{minsize(t, E) \mid t \rightarrow c(x_1, \cdots, x_n) \in E\}$.*

In the long version, we show that for every term t, *minsize* has some value, and $minsize(t, E)$ is the size of the smallest term in T_Σ which rewrites to t.

For any term t, define $maxsym(t, E) = max\{minsize(c, E) \mid c$ is a symbol in $t\}$, and define $maxsym(s \approx t, E) = min\{maxsym(s, E), maxsym(t, E)\}$. If $u \geq_f v$, then define an equation $u \approx v \in E$ to be *expanding* if $maxsym(u, E) \leq maxsym(v, E)$ and every variable of u occurs in v.

Lemma 1 *Let n be a number and $u \not\approx v$ be a ground disequation in T_Σ such that $|u| \leq n$ and $|v| \leq n$. Let S be a set of equations appearing in a Theorem Proving derivation from some subset of T_Σ. Let $S_n = \{s \approx t \in S \mid maxsym(s) \leq n$ and $maxsym(t) \leq n\}$. Suppose that S_n is saturated under the Nonground Congruence Closure rules, and all equations in S_n are expanding. Then $S_n \cup \{u \not\approx v\}$ is unsatisfiable if and only if $S \cup \{u \not\approx v\}$ is unsatisfiable.*

The lemma follows from the fact that for an expanding set of equations, once an equation $s \approx t$ appears with $maxsym(s) > n$ or $maxsym(t) > n$, then any descendent of that equation will also have that property.

Suppose that we are trying to prove the unsatisfiability of a set or equations and disequations. And supppose that at some point of the theorem proving derivation, we have saturated all equations of the form $s \approx t$ with $maxsym(s) \leq n$ and $maxsym(t) \leq n$. If all such equations are expanding, then (I_n, D) is a decision procedure for the word problem for any equation $u \approx v$ with $|u| \leq n$ and $|v| \leq n$. Furthermore, the $(I, D_n, \hat{F})$ procedure will be a decision procedure, even though we may not know that it is.

If we can show that some set of equations S will only create expanding equations in the saturation, then the $(I, D, \hat{F})$ procedure is a decision procedure for S. For example, we can show that it forms a decision procedure for size preserving linear theories.

Theorem 5 *The $(I, D, \hat{F})$ procedure is a decsision procedure for size preserving linear theories.*

Finally, we consider another interesting theory, where Knuth-Bendix Completion does not halt, but it is not size preserving. The theory is $\{f(g(f(x))) \approx g(f(x))\}$. If we flatten this theory, we get equations $g(f(x)) = c_1(x)$ and $f(c_1(x)) = c_1(x)$ (assuming a simplification). There is one inference that can be done on these two equations. Its result adds the two equations $g(c_1(x)) = c_2(x)$ and $c_1(c_1(x)) = c_2(x)$. If we could

continue this process infinitely, then for all i and j, we get $f(c_i(x)) = c_i(x)$, $g(c_i(x)) = c_{i+1}(x)$ and $c_i(c_j(x)) = c_{i+j}(x)$. Notice that $minsize(g, E) = minsize(f, E) = 1$, and $minsize(c_i, E) = i+1$ for all i. All of the rules in the infinite saturated set are expanding, so both the unsound and traditional method will give us a decision procedure.

5 Conclusion

We have discussed the benefits of unsound theorem proving, for disproving conjectures. It can often find disproofs when traditional methods do not.

We gave a framework for unsound and complete theorem proving, which amounts to proof in a stronger theory, which can be decided. We discussed how to iterate the process to attempt to find weaker and weaker approximations, and we showed how this can be combined with a sound and incomplete theorem prover, and how to iterate them both to continually attempt to refine approximation from both sides.

We instantiated our framework with Knuth-Bendix Completion and a nonground Congruence Closure method, based on ground Congruence Closure methods [6, 4]. Our Nonground Congruence Closure is new, as far as we know. However, it is in the same spirit as what is done in [10], which also uses the Knuth-Bendix inference rules followed by eaqer splitting of equations introducing new constant symbols, and it also has an arity-compatible precedence. The inference system of [10] was shown to terminate for standard theories. A difference is that we allow depth-2 linear variables to appear at depth one on the right hand side of rules. This means that we can capture all equational theories, but of course it makes theorem proving undecidable. We gave some evidence to indicate that our Nonground Congruence Closure be especially powerful in combination with unsound theorem proving.

We did not discuss how to instantiate the framework with clausal theorem proving methods like Resolution and Paramodulation. However, we can quickly suggest a method for unsound deletion. In the paper, we have shown how to prevent terms from becoming too large. For clauses, we must also prevent them from becoming too long. A simple method to do that is to delete some literals when a clause gets too long. There may be other more sophisticated and interesting methods.

Our work can be compared with other approximation methods. For example, [1] shows how to disprove false conjectures by translating them into second-order monadic logic. This is an unsound approximation in the same sense as our paper. In [7], an efficient approximation of E-unification is given by modifying a goal-directed inference method. Those two papers give a single approximation, using a completely different method than ours. Many goal directed theorem proving procedures and constraint solving methods could be thought of as successive unsound approximations. The paper of [5] is close in spirit to our paper. It discusses how to get successive approximations by converting first order clauses into ground clauses, and then applying a satisfiability test. When a ground solution is found, it must be verified for soundness. It also discusses other approximations besides ground clauses. We are not aware of other work besides ours which successively modifies a saturation procedure to produce strong models.

References

[1] Serge Autexier and Carsten Schurmann. Disproving false conjectures. In *LPAR*, volume 2850 of *Lecture Notes in Computer Science*, pages 33–48. Springer, September 2003.

[2] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[3] Leo Bachmair, Nachum Dershowitz, and David Plaisted. *Completion without Failure*, volume II. Academic Press, 1989.

[4] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David McAllester, editor, *Automated Deduction — CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78, Pittsburgh, PA, jun 2000. Springer-Verlag.

[5] Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *IEEE Symposium on Logic in Computer Science*, pages 55–64, Ottawa, Ont., jun 2003. IEEE.

[6] Deepak Kapur. Shostaks congruence closure as completion. In *International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 23–37, Bacelona Spain, jun 1997. Springer-Verlag.

[7] Christopher Lynch and Barbara Morawska. Approximating e-unification. In *15th Annual Workshop on Unification Theory*, Siena, Italy, 2001.

[8] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[9] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[10] Robert Nieuwenhuis. Complexity analysis by basic paramodulation. *Information and Computation*, 147:1–21, 1998.

The TM System for Repairing Non-Theorems

Simon Colton

Department of Computing
Imperial College, London
United Kingdom
sgc@doc.ic.ac.uk

Alison Pease

School of Informatics
University of Edinburgh
United Kingdom
alisonp@dai.ed.ac.uk

Abstract

We describe a flexible approach to automated reasoning, where non-theorems can be automatically altered to produce proved results which are related to the original. This is achieved in the TM system through an interaction of the HR machine learning program, the Otter theorem prover and the Mace model generator. Given a non-theorem, Mace is used to generate examples which support the non-theorem, and examples which falsify it. HR then invents concepts which categorise these examples and TM uses these concepts to modify the original non-theorem into specialised theorems which Otter can prove. The methods employed by TM are inspired by the piecemeal exclusion, strategic withdrawal and counterexample barring methods described in Lakatos's philosophy of mathematics. In addition, TM can also determine which modified theorems are likely to be interesting and which are not. We demonstrate the effectiveness of this approach by modifying non-theorems taken from the TPTP library of first order theorems. We show that, for 98 non-theorems, TM produced meaningful modifications for 81 of them. This work forms part of two larger projects. Firstly, we are working towards a full implementation both of the reasoning and the social interaction notions described by Lakatos. Secondly, we are aiming to show that the combination of reasoning systems such as those used in TM will lead to a new generation of more powerful AI systems.

Key words: Automated theorem modification, automated reasoning, model generation, machine learning, automated theory formation, philosophy of mathematics.

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

1 Introduction

Mathematics has developed in a much more organic way than its rigid textbook presentation of definition-theorem-proof would suggest. Automated theorem proving systems more closely reflect the textbook notion of mathematics than a developmental approach. In particular, most deduction systems are designed either to prove results if they are true, or find counterexamples if they are false, but not both. System designers also assume that the concepts mentioned in the conjecture are correctly defined and actually relate to the mathematical notions the user is interested in. Clearly, the adoption of these assumptions is not conducive to the kind of exploration more common in research mathematics, in which concept definitions change and become more sophisticated, and flawed conjectures and proofs are gradually refined. Hence, it is time to increase the flexibility of reasoning systems to better handle ill-specified problems.

We describe here the development of the Theorem Modifier (TM) system. This takes a set of axioms and a conjecture in first order logic and tries to prove it. If this fails, TM attempts to modify the conjecture into a set of theorems which it can prove. To achieve this flexibility, TM combines the power of three automated reasoning systems, namely the HR machine learning program [1], the Otter theorem prover [11] and the Mace model generator [12]. As described in §3, TM uses these systems in ways prescribed in the philosophy of mathematics developed by Lakatos [9]. In particular, TM performs counterexample-barring, piecemeal exclusion and strategic withdrawal. These techniques are further explained in §2. As a simple example of TM working, given the non-theorem that all groups are Abelian, it states that it cannot prove the original result, but it has discovered that self-inverse groups are Abelian. To evaluate this approach, in §4, we describe how TM successfully found meaningful modifications to 81 of 98 non-theorems derived from the TPTP library of first order theorems [16].

The development of the TM system forms part of two larger projects. Firstly, we are working towards a full implementation both of the reasoning and the social interaction notions described by Lakatos in [9]. Secondly, we are aiming to show that the combination of reasoning systems such as those used in TM will lead to a new generation of AI systems which are able to solve problems which individual techniques cannot.

2 Background

The way in which TM forms modified theorems is inspired by the notions expressed in the philosophy of mathematics presented by Imre Lakatos [9], as described in §2.1 below. The implementation of these ideas is heavily dependent on third party software, in particular the Otter, Mace and HR programs. Of these, HR is the least well known, so we describe this in §2.2.

2.1 *Lakatos's Philosophy of Mathematics*

Our inspiration for TM comes from Lakatos, who presented a fallibilist approach to mathematics, in which proofs, conjectures and concepts are fluid and open to negotiation [9]. Lakatos strongly criticised the deductivist approach in mathematics, which presents definitions, axioms and theorem statements as immutable ideas which come from nowhere into a mathematician's empty mind. Rather than a mysterious and ever-increasing set of truths, Lakatos saw mathematics as an adventure in which – via patterns of analysis which he categorised into various methods – conjectures and proofs are gradually refined but never certain. He rejected the view that discovery in mathematics is essentially irrational and should be left to the psychological arena, as championed by, for instance, Popper [15]. Instead, he outlined a heuristic approach which holds that mathematics progresses by a series of primitive conjectures, proofs, counterexamples, proof-generated concepts, modified conjectures and modified proofs. Lakatos demonstrated his argument using case studies including the development of Euler's conjecture that for any polyhedron, the number of vertices (V) minus the number of edges (E) plus the number of faces (F) equals two.

Lakatos's treatment of exceptions is noteworthy for two reasons. Firstly, he highlights their existence in mathematics – traditionally thought of as an exact subject. Secondly, he shows how exceptions, rather than simply being annoying problem cases which would force a mathematician to abandon a conjecture, can be used to further knowledge. He does this via two methods; *piecemeal exclusion* and *strategic withdrawal*. Piecemeal exclusion works by generalising from a counterexample to a class of counterexamples and then excluding this class from the faulty conjecture. For instance, Lakatos showed how, by examining the hollow cube which is a counterexample to Euler's conjecture, mathematicians modified the conjecture to 'for any polyhedron without cavities, $V - E + F = 2$' [9]. Put formally, suppose that we have the conjecture $\forall\, x\ (A(x) \Rightarrow B(x))$, a set of counterexamples N such that $\forall\, x \in N, A(x) \wedge \neg B(x)$, and a set of positive examples P such that $\forall\, x \in P, A(x) \wedge B(x)$. To perform piecemeal exclusion, find a concept $\mathcal{C}$ such that $\forall\, x \in N,\ C(x)$, and $\forall\, x \in P,\ \neg C(x)$, then modify the conjecture to: $\forall\, x\ (\neg C(x) \wedge A(x)) \Rightarrow B(x)$. When there is only one counterexample and no simply expressed concept which covers it, piecemeal exclusion extends to *counterexample-barring*, in which the counterexample is explicitly forbidden in a modified conjecture, i.e., given a single counterexample $x_1 \in N$, one modifies the conjecture to: $\forall\, x \neq x_1\ (A(x) \Rightarrow B(x))$.

Strategic withdrawal works by considering the examples supporting a conjecture, finding a concept which covers a subset of these, and limiting the domain of the conjecture to that of the concept. For instance, by examining the supporting examples of Euler's conjecture, such as the cube, tetrahedron and octahedron, mathematicians retreated to the 'safe' domain of convex polyhedra (*i.e.* polyhedra whose surface is topologically equivalent to the surface of a

sphere). Put formally, given the above conjecture, set of supporting examples P and counterexamples N, first find a concept $\mathcal{C}$ such that $\forall\ x \in P,\ C(x)$, and $\forall\ x \in N,\ \neg C(x)$, then modify the conjecture to: $\forall\ x\ (C(x) \wedge A(x)) \Rightarrow B(x)$.

Clearly, an implementation of a theorem modification system along the lines suggested by Lakatos requires three core functionalities. Firstly, an ability to prove theorems is required. We achieved this by incorporating the Otter program [11] into the system. Otter is a powerful first order resolution theorem prover which has been used for many discovery tasks in algebraic domains, e.g., [13]. Secondly, an ability to generate counterexamples to non-theorems is required. We achieved this by incorporating the Mace program [12] into the system. Mace is a powerful model generator which employs the Davis-Putnam method for generating models to first order sentences. Thirdly, an ability to suggest modifications to non-theorems in the light of counterexamples is required. We achieved this by incorporating the HR program into the system. HR is described below.

2.2 The HR System

HR is named after the mathematicians Hardy and Ramanujan, and the core functionality of this system is described in [1]. HR performs descriptive induction to form a theory about a set of objects of interest which are described by a set of background concepts, as detailed further in [3]. This is in contrast to predictive learning systems which are used to solve the particular problem of finding a definition for a target concept. The theories HR produces contain concepts which relate the objects of interest; conjectures which relate the concepts; and proofs which explain the conjectures. Theories are constructed via theory formation steps which attempt to construct a new concept. HR builds new concepts from old ones using a set of 15 generic production rules [4] which include:

- The *exists* rule: this adds existential quantification to the new concept's definition
- The *negate* rule: this negates predicates in the new definition
- The *match* rule: this unifies variables in the new definition
- The *compose* rule: this takes two old concepts and combines predicates from their definitions in the new concept's definition

For a more formal description of these production rules, and the others that HR uses, see [3] or [4].

For each concept, HR calculates the set of examples which have the property described by the concept definition. Using these examples, the definition, and information about how the concept was constructed and how it compares to other concepts, HR estimates how interesting the concept is [6], and this drives a heuristic search. As it constructs concepts, it looks for empirical

relationships between them, and formulates conjectures whenever such a relationship is found. In particular, HR forms equivalence conjectures whenever it finds two concepts with exactly the same examples, implication conjectures whenever it finds a concept with a proper subset of the examples of another, and non-existence conjectures whenever a new concept has an empty set of examples. HR is also able to make near-conjectures whenever the relationship has only a few counterexamples. To attempt to determine the truth of each conjecture, they are passed to a third party theorem prover and a third party counterexample finder (usually Otter and Mace, but there are interfaces to other reasoning systems [17]). HR also works hard to break the conjectures into lemmas which are easier to prove, and it will also extract prime implicates which may be more interesting to the user [2].

HR has been used for a variety of discovery projects in mathematics. It has been particularly successful in number theory [5] and algebraic domains [14]. Moreover, we have used HR to improve the abilities of Artificial Intelligence systems, most notably constraint solvers [7], and we are currently extending HR to perform discovery tasks in other scientific domains, in particular bioinformatics. While we have used HR to generate first order conjectures [8], the application described in this paper is the first one in which we have applied HR to the problem of *proving*, rather than generating, theorems.

3 Automated Theorem Modification

Users supply TM with a conjecture of the form: $A \Rightarrow C$ where A is a conjoined set of axioms which describe the domain they are working in, and C is the statement of the conjecture they wish to prove/modify/disprove. The theorem is supplied in Otter first-order syntax, which means that C must be negated, as Otter will derive a contradiction using resolution. TM assumes that C is placed in the last line of input, preceded by a line per axiom. We hope to relax such restrictions in future versions of the program. For the purposes of this paper, we also assume that we are working in an algebraic domain, where algebraic objects comprise a set of elements and a set of operators relating those elements which are constrained as prescribed by the axioms. An example algebra is group theory, where there is a single operator which satisfies the associativity, identity and inverse axioms.

3.1 Forming Modified Theorems

How the TM program operates can be characterised by how and when it calls the Otter, Mace and HR programs, and how it implements the piecemeal exclusion, strategic withdrawal and counterexample-barring methods described in §2. To begin with, TM checks whether the conjecture is true, i.e., $A \Rightarrow C$. It does this by invoking Otter for a period of time specified by the user, and if Otter is successful, this is reported to the user and TM stops. If the theorem

cannot be proved by Otter in the time given, TM then uses Otter to attempt to prove that the negation of C follows from A. For reasons we shall see later, if the negation of the theorem is true, then TM will not be able to modify this conjecture using its current techniques.

Next, TM checks whether the conjecture is true if and only if the objects in the domain (which are all algebraic objects) are trivial – in the sense that they have only one element – and whether the conjecture is true if and only if the objects are non-trivial. To do this, Otter is asked to prove: $A \Rightarrow ((\forall\, a, b\ (a = b)) \Leftrightarrow C)$ and $A \Rightarrow ((\exists\, a, b\ (a \neq b)) \Leftrightarrow C)$ respectively. If either can be proved, then TM returns the modified theorem that the conjecture is true for trivial/non-trivial algebras only. These are special cases, and checking for them is in line with Lakatos's counterexample-barring. If TM were to follow Lakatos's advice directly, then it would first find counterexamples to the theorem and try to prove that if they are excluded from the conjecture, it is true. However, in all but a few cases, we have found that Otter is not good at proving such results, as describing the models to be excluded leads to a great number of first order sentences being added to the input file for Otter. The exception, of course, is when the algebra to be excluded is trivial, as we have seen above that this can be simply stated. However, in algebraic domains, theorems which are true for all but the trivial algebra are quite rare. In fact, the opposite is often true: the theorem is true *only for* the trivial algebra. For these reasons, we decided that having TM check initially for these two simple modifications was a better idea than implementing full counter-example barring techniques.

If none of these preliminary checks have been successful, then the conjecture is either a non-theorem, or is too difficult for Otter to prove in the time available. In either case, this presents an opportunity to modify the theorem in order to enable Otter to prove it. We have so far concentrated on modifying a conjecture by specialising it, i.e., adding in extra conditions which enable Otter to prove the modified theorem. To do this, TM first finds some example algebras which support the conjecture, by using Mace to generate models which satisfy A and for which C holds. Mace is then used to generate some examples which contradict the conjecture, i.e., models which satisfy A but which break the conjecture C. Mace is given a limit for both time and size. Normally, we ask Mace to find an example of size 1, an example of size 2, etc., up to size 8, and that it can spend 10 seconds on each search. For instance, when we give TM the false conjecture that all groups are Abelian, it uses Mace to find an example Abelian group for each size 1 to 8, which support the conjecture. However, it also finds a non-Abelian group of size 6 and a non-Abelian group of size 8, which falsify the conjecture.

The supporting and falsifying examples generated by Mace are given as the objects of interest to a session using HR. HR is also supplied with the file containing the statement of the conjecture in Otter format. From this, it extracts the background concepts in the domain, e.g., in group theory, HR

would extract the concept of groups, elements, multiplication, identity and inverse. These form the basis of the theory HR forms, i.e., all concepts it produces will be derived from these. TM then uses HR to form a theory for a user-set number of theory formation steps, usually taken to be between 1000 and 5000. In this time, HR generates many concepts which can be interpreted as specialisations of the algebra, such as Abelian groups, self-inverse groups, etc. For instance, in group theory, given the groups up to size 8 as input, in 5000 steps, HR generates 37 specialisations of the concept of group.

¿From the theory produced by HR, TM identifies all the specialisation concepts and extracts those which describe only algebras that support the conjecture. For example, in the session associated with the non-theorem that all groups are Abelian, amongst others, HR invents the concept of groups which are self inverse, i.e., $\forall\, a\ (a = a^{-1})$. It turns out that these form a subset of the examples which supported the conjecture, and hence TM extracts this from HR's theory. For each extracted specialisation, M, TM forms the modified conjecture: $(A \wedge M) \Rightarrow C$ by adding M to the axioms. Otter is invoked to see which of these modifications can be proved, and any which are proved are presented to the user. Note that, in addition to the specialisations that HR produces, TM also extracts any concepts which have been conjectured to be logically equivalent to a specialisation – these concepts are not normally allowed into the theory as distinct items, but HR records the conjectured equivalence of the definitions. This functionality is turned on by default, but the user can set a flag to stop it happening, which will produce faster results (as fewer calls to Otter will be made), but has the potential to miss interesting modifying specialisations.

3.2 Identifying Uninteresting Modifications

Unfortunately, there are a number of reasons why the modifications generated by this process can be uninteresting for the user. TM takes care to discard any it can prove to be uninteresting, and highlights any which have a greater chance than normal of being uninteresting. In particular, some specialisations that HR produces are true only of the trivial algebra. As most conjectures are also true of the trivial algebra, the modifications usually hold, but are uninteresting. For instance, the modified conjecture: "all groups which are the trivial group are Abelian" holds very little interest. Hence, whenever a modification has been proved, and the examples satisfying the definition of the specialisation M in the modification amount to just the trivial algebra, TM invokes Otter to check whether: $A \Rightarrow (M \Leftrightarrow (\forall\, a, b\ (a = b)))$ It is unlikely, but not impossible that such re-definitions of the trivial algebra will be interesting to the user (for instance, the re-definition might contain an unusual combination of background concepts). Hence, in TM's output, the modification is set-aside from the others, but it is not discarded.

Another problem arises when HR derives concepts which are re-definitions of the conjecture statement. Obviously, adding this to the axioms would make

the conjecture trivially true, e.g., all Abelian groups are Abelian. Hence, for every specialisation, M, where *every* supporting example has the property prescribed by M, TM uses Otter to try to prove: (i) $M \Leftrightarrow C$ (ii) $A \Rightarrow (M \Leftrightarrow C)$ and (iii) $M \Rightarrow C$. Often M is just a simple restatement of C, and of no interest, but it sometimes happens that the equivalence of C and M is quite surprising and non-trivial to prove, hence the modification is valid. Hence, if TM proves any of the three results above, it presents the modification to the user separately, and provides the result as a possible indication of why the modification is true and a caution that it may be uninteresting because the specialisation trivially proves the conjecture.

This process of modifying conjectures by specialising them is an implementation of Lakatos's strategic withdrawal method, whereby a concept which excludes all of the counterexamples is discovered and the conjecture is specialised to only apply to examples satisfying that concept. Note also that when HR uses the negate rule, which TM instructs it to, for every specialisation M, the negation $\neg M$ will also be produced. Hence, if the examples of M contained all the *falsifying* examples for the conjecture, then $\neg M$ would describe a subset of the supporting examples, and hence would be used in a modification attempt. Recalling that the piecemeal exclusion strategy involves finding a concept which covers all the counterexamples (and possibly more), then excluding the concept from the conjecture, we see that TM is also using piecemeal exclusion to form the modifications.

3.3 Summary of Theorem Modification

To summarise, in our running example that all groups are Abelian, TM undertakes the following process. Firstly, it tries and fails to prove that the conjecture is true already, and similarly fails to prove that the negation of the conjecture follows from the axioms (i.e., it fails to prove that all groups are non-Abelian). If the latter were true, then no amount of specialisation would improve matters. TM also fails to prove that a group is Abelian if and only if it is trivial, and that a group is Abelian if and only if it is non-trivial. It then employs Mace to generate some Abelian groups which support the conjecture and some non-Abelian groups which falsify the conjecture. Both sets of examples are given, along with the conjecture statement as input to HR, which forms a theory of groups containing many specialisations of the notion of group. From this theory, TM extracts all those specialisations which describe only groups which support the conjecture. When using one of these, namely self-inverse groups, in a modified conjecture, Otter proves the theorem and TM reports that it can prove that self-inverse groups are Abelian, even though the original conjecture is false. In contrast to the usual proof-or-fail output from a theorem prover, TM outputs 5 different types of result:

- The original conjecture is true: $A \Rightarrow C$
- The negation of the conjecture is true: $A \Rightarrow \neg C$

- The conjecture is true only for trivial algebras $(A \Rightarrow C) \Leftrightarrow Triv$
- It is true only for non-trivial algebras $(A \Rightarrow C) \Leftrightarrow \neg Triv$
- The original conjecture is false, but various modifications of it are true, $(A \wedge M) \Rightarrow C$

In the latter case, when appropriate, TM can also warn the user that the modification may be trivially true because either M is only true of the trivial algebra, or because one of the following lemmas holds: $M \Leftrightarrow C$, $A \Rightarrow (M \Leftrightarrow C)$ or $M \Rightarrow C$.

4 Testing and Evaluation

We used the TPTP library [16] to supply a set of non-theorems for experiments designed to test the hypothesis that TM can find meaningful modifications to non-theorems. We looked at four categories within TPTP, namely GRP (groups), FLD (fields), RNG (rings) and COL (combinatory logic). Unfortunately, we found only 9 non-theorems which were suitable, because (a) there aren't many non-theorems (b) many were actually just statements of axioms for which models can be found and (c) many were not in the form of axioms followed by conjecture, e.g., there are many non-theorems stating that one set of axioms is not equivalent to another set.

In order to provide a more substantial test set, we took theorems from the above TPTP categories and altered them to become non-theorems. The alterations included (i) removing axioms (ii) changing/removing quantifiers (iii) altering variables and constants and (iv) altering bracketing. In this fashion, we produced 158 altered theorems, which we used alongside the 9 proper non-theorems. We found that 30 of our altered TPTP theorems were still theorems (TM told us this). Mace produced the same examples to both support and falsify the conjecture, for 39 of the remaining 137 non-theorems. This was due to constants such as an identity element being used in the conjecture statement without reference in the axioms, or to variables being instantiated differently. We removed these non-theorems from the test set, leaving us with a core of 98 non-theorems.

In addition to testing the effectiveness of TM, we also wanted to determine whether any alterations to the setup would improve the performance. It was clear from an early stage that giving Mace extra time and range did not improve matters, as it only found a few more examples which did not affect the specialising concepts that HR found. Also, we experimented by giving Otter more time, but we have not seen any evidence that this improves performance of TM – it is often the case that if a prover is going to solve a problem, it will do so quickly, and giving a little extra time will not help for more difficult problems. Hence, we concentrated on altering the way in which we ran HR. We ran three sessions using TM to attempt to modify each of the 98 non-theorems. Otter and Mace were given 10 seconds, with Mace looking for examples up to size 8, and HR was allowed 1000 theory formation steps in

Session	1	2	3
Equivalent to trivial algebra	24	24	24
No valid modifications	11	10	9
Only redefinition modifications	8	8	8
Valid modifications with caution	18	18	18
Valid modifications no caution	37	38	39
Total valid modifications	79	80	81
Average number of modifications per non-theorem	0.8	1.3	3.1
Average time to generate modifications (s)	73	120	253

Table 1
Results from modification attempts on 98 non-theorems

the first two sessions, and 3000 steps in the third session. In the first session, however, the ability to use equivalence conjectures to harvest specialisations was turned off. The results are presented in table 1.

In the sessions, we found that in many cases, the only modifications came with a caution that the specialisation may trivially make the theorem true. However, in around two thirds of these cases, upon looking at the modification, it was found to be valid, i.e., not obviously just a restatement of the conjecture. Taking these into account, in addition to the modifications stating that the conjecture is true if and only if the algebra is trivial (a valid modification), TM produced proper modifications for 79, 80 and 81 of the 98 of the non-theorems respectively, i.e., 81%, 82% and 83%. We believe that such a success rate is very encouraging. These figures don't appear to provide much evidence of improvement by running HR for longer and allowing it to use information from equivalence conjectures. However, if we look at the average number of modifications produced in the three sessions, we see that using the setup as in the first session, on average TM will find 0.8 proved modifications, but using the setup as in the third session, TM will find 3.1 modifications per theorem. However, the time taken to produce these modifications triples.

To illustrate why TM highlights theorems for which $M \Rightarrow C$, we can look at the non-theorem we generated from TPTP theorem GRP001. This states that, if all elements in the group square to give the identity, then the group must be Abelian, i.e., $(\forall\ a\ (a * a = id)) \Rightarrow (\forall\ a, b\ (a * b = b * a))$. We removed the inverse and associativity axioms to make this into a non-theorem. TM found only two specialisations to perform the modification, both of which were cautioned. The first was: $\nexists\ b, c, d\ (b * c = d \wedge c * b \neq d)$. This is obviously the specialisation into Abelian groups, hence, including this specialisation into a modified theorem produced: "in Abelian groups, if all elements square to give the identity, then the group is Abelian", which is trivially true. Hence

TM was right in this case to caution us about this theorem.

In contrast, however, when we gave TM the non-theorem which we generated from TPTP theorem GRP011-4, it produced specialisations which were not at all obvious, and hence made interesting modifications. GRP011-4 is the left cancellation law, i.e., $\forall\ a, b, c\ ((a * b = a * c) \Rightarrow b = c))$. We took out the identity and inverse axioms to generate a non-theorem, and one of the five (cautioned) specialisations was: $\nexists\ b, c, d\ (b * c = d \wedge b * d \neq c)$ Hence the modified theorem states that, in algebras for which $\forall\ x, y\ (x * (x * y) = y))$, the left cancellation law holds (with no mention of associativity).

TM managed to find valid modifications for 7 out of the 9 non-theorems that we took directly from the TPTP library, and these provide interesting illustrative examples of TM working as it should do. Firstly, TM successfully modified 3 out of 5 non-theorems in combinatory logic – a new domain for HR. For example, non-theorem COL073-1 states that, given certain axioms: $\forall\ y\ ((apply(y, f(y))) = (apply(f(y), apply(y, f(y)))))$. TM found a single specialisation from the 7 supporting examples Mace provided:
$\nexists\ b, c, d\ (apply(b, c) = d \wedge apply(b, d) \neq c)$. However, this was only true of the trivial algebra, and while Otter couldn't prove an equivalence between this specialisation and the trivial algebra, we cannot rule it out.

In group theory, the first non-theorem in the library is GRP024-4, which states that, given the definition of the commutator operator on two elements x and y being $comm(x, y) = x * y * x^{-1} * y^{-1}$, then this operator is associative if and only if the product of the commutator is always in the centre of the group (defined to be the set of elements which commute with all others). Hence this theorem states that: $\forall\ x, y, z\ (comm(comm(x, y)), z) = comm(x, comm(y, x)) \Leftrightarrow \forall\ u, v, w\ (comm(u, v) * w = w * comm(u, v)))$. Mace could not find any counterexamples to this, but it did find four groups for which the conjecture is true. As strategic withdrawal doesn't need any counterexamples, TM could continue. It found that, with the extra axiom that the groups are self inverse (i.e., $\forall\ x\ (x = x^{-1})$), the conjecture actually holds.

The first of two ring theory non-theorems taken directly from the TPTP library was RNG007-5, which states that, given a ring for which $\forall\ x\ (x*x = x)$, then $\forall\ x\ (x * x = id)$. Given the first property as an axiom, TM proved that the second property is equivalent to being the trivial algebra, which gives good justification for implementing this functionality. The second ring theory non-theorem was RNG031-6, which states that the following property, P, holds for all rings: $\forall\ w, x\ ((((w * w) * x) * (w * w)) = id)$ where id is the additive identity element. Mace found 7 supporting examples for this, and 6 falsifying examples. HR produced a single specialisation concept which was true of 3 supporting examples: $\nexists\ b, c\ (b * b = c \wedge b + b \neq c)$. Otter then proved that P holds in rings for which HR's invented property holds. Hence, while TM couldn't prove the original theorem, it did prove that, in rings for which $\forall\ x\ (x*x = x+x)$, property P holds. The specialisation here has an appealing symmetry. A proof of the modified theorem is given in the appendix.

5 Conclusions and Further Work

We have described and demonstrated the effectiveness of the TM automated theorem modification system. This is based on an implementation of methods prescribed in Lakatos's philosophy of mathematics, and relies on the interaction of the HR, Otter and Mace programs. In tests, TM modified 7 out of 9 non-theorems from the TPTP library into interesting, proved alternatives, and on an artificial set of 98 non-theorems, it produced meaningful modifications 80% of the time, which we believe is highly encouraging given that this is only the first version of the software. We intend to improve the implementation in at least the following ways:

• enabling it to strengthen modifications after it has weakened the original conjecture. For instance, if it has proved $A \Rightarrow (P \Leftrightarrow Q)$, try $A \Rightarrow (P \wedge Q)$. We expect this to result in more interesting theorems;

• extending the domains on which it works, to security protocols, chemistry and bioinformatics as well as other mathematical domains;

• automatically evaluating the modifications further, to enable TM to recognise interesting modifications from all those produced. For instance it might consider aspects of the proof, such as its length;

• using a failed proof attempt to suggest modifications to a conjecture C. This is Lakatos's method of *lemma incorporation*: given a counterexample to a conjecture, find which step of the proof it violates and then modify the conjecture by making that step a condition. The modified conjecture therefore becomes: $\forall\ x$ which satisfy proof step i, C holds. We are currently implementing this method;

• exploring the possibilities of using TM to suggest case splits for difficult but true theorems. For instance, given a theorem: $\forall\ x\ P(x) \Rightarrow Q(x)$, and a concept $C(x)$ which covers all the supporting examples and no counterexamples, then TM would form and attempt to prove (i) $\forall\ x\ (C(x) \wedge P(x)) \Rightarrow Q(x)$ and (ii) $\forall\ x\ (\neg C(x) \wedge P(x)) \Rightarrow Q(x)$. This suggests ways of automatically rephrasing a conjecture statement into one which can be proved.

TM is part of a larger project, in which we are implementing all of the methods prescribed by Lakatos in [9]. The aim of this project is to (a) provide a computational model for the use of Lakatos's ideas and (b) enhance the model and implementation of automated theory formation (ATF) as described in [1]. Our model of Lakatos-enhanced theory formation has developed along two axes: the sophistication of the conjecture-correcting methods which Lakatos proposed, and the social nature of the discourse he described.

There are various reasons to automate theories of scientific discovery, including developing new techniques which aid scientists in their work [10]. We have demonstrated a new technique, namely automated theorem modification,

which has the potential to aid mathematicians, by adding more robustness and flexibility to automated theorem proving. We believe that such robustness – in the case of TM, gained by the integration of deductive, inductive and model based techniques – will play an important part in the next generation of automated theorem provers.

Acknowledgements

We would like to thank Alan Smaill and John Lee for their continued input to this project. Special thanks to Bill McCune and Geoff Sutcliffe for supplying data and for their input to this work, and to Roy McCasland for providing the proof found in the appendix. We are grateful to the anonymous referees for their useful comments on an earlier draft of this document, and to the organisers of the IJCAR workshop on disproving. This work has been supported by EPSRC platform grant GR/S01771/01.

References

[1] S Colton. *Automated Theory Formation in Pure Mathematics.* Springer-Verlag, 2002.

[2] S Colton. The HR program for theorem generation. In *Proceedings of CADE*, 2002.

[3] S Colton and S Muggleton. ILP for mathematical discovery. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, 2003.

[4] S Colton, A Bundy, and T Walsh. Automatic identification of mathematical concepts. In *Machine Learning: Proceedings of the 17th International Conference*, 2000.

[5] S Colton, A Bundy, and T Walsh. Automatic invention of integer sequences. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, 2000.

[6] S Colton, A Bundy, and T Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human Computer Studies*, 53(3):351–375, 2000.

[7] S Colton and I Miguel. Constraint generation via automated theory formation. In *Proceedings of CP-01*, 2001.

[8] S Colton and G Sutcliffe. Automatic generation of benchmark problems for automated theorem proving systems. In *Proceedings of the Seventh AI and Maths Symposium*, 2002.

[9] I Lakatos. *Proofs and Refutations: The logic of mathematical discovery.* Cambridge University Press, 1976.

[10] P. Langley. Lessons for the computational discovery of scientific knowledge. In *Proceedings of First International Workshop on Data Mining Lessons Learned*, 2002.

[11] W McCune. The OTTER user's guide. Technical Report ANL/90/9, Argonne National Labs, 1990.

[12] W McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratories, 1994.

[13] W McCune and R Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves, LNAI 1095*. Springer-Verlag, 1996.

[14] A Meier, V Sorge, and S. Colton. Employing theory formation to guide proof planning. In *Proceedings of the Tenth Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, LNAI 2385*. Springer, 2002.

[15] K Popper. *The Logic of Scientific Discovery.* Basic Books, 1959.

[16] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[17] J Zimmer, A Franke, S Colton, and G Sutcliffe. Integrating HR and tptp2x into MathWeb to compare automated theorem provers. In *Proceedings of the CADE'02 Workshop on Problems and Problem sets*, 2002.

Appendix

Theorem:
Let R be a ring such that $\forall\, x \in R, x + x = x * x$. Then $\forall\, x, y \in R, x^2yx^2 = id$.

Proof:
Let r be an arbitrary element in R. Then

$$-(r^2) = -(r * r) = -(r + r) = (-r) + (-r) = (-r) * (-r) = (-r)^2 = r^2$$

Hence $-(r^2) = r^2$, so $r^2 + r^2 = id$.

Now let x and y be arbitrary elements in R. Then:

$$\begin{aligned} x^2yx^2 &= (x^2y)(x^2) = ((x+x)y)(x^2) \\ &= (xy+xy)(x+x) = (xyx+xyx) + (xyx+xyx) \\ &= (xyx * xyx) + (xyx * xyx) = (xyx)^2 + (xyx)^2 \\ &= id \text{ [by above result]} \end{aligned}$$

QED.

Bounded Model Generation for Isabelle/HOL*

Tjark Weber

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
webertj@in.tum.de

Abstract

A translation from higher-order logic (on top of the simply typed λ-calculus) to propositional logic is presented, such that the resulting propositional formula is satisfiable iff the HOL formula has a model of a given finite size. A standard SAT solver can then be used to search for a satisfying assigment, and such an assignment can be transformed back into a model for the HOL formula. The algorithm has been implemented in the interactive theorem prover Isabelle/HOL, where it is used to automatically generate countermodels for non-theorems.

1 Introduction

Interactive theorem provers have been enhanced with numerous automatic proof procedures for different application domains. However, when an automatic proof attempt fails, the user usually gets little information about the reasons. It may be that an additional lemma needs to be proved, that an induction hypothesis needs to be generalized, or that the formula one is trying to prove is not valid. In such cases an automatic tool that can refute non-theorems would be useful.

This paper presents a translation from higher-order logic to propositional logic (quantifier-free Boolean formulas) such that the propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size, i.e. involving no more than a given number of elements. A standard SAT solver can then be used to search for a satisfying assignment, and if such an assignment is found, it can easily be transformed back into a model for the HOL formula.

An algorithm that uses this translation to generate (counter-)models for HOL formulas has been implemented in the interactive theorem prover Isabelle/HOL [14]. This algorithm is not a (semi-)decision procedure: if a formula does not have a model of a given size, it may still have larger or infinite models. The algorithm's applicability is also limited by its complexity, which is non-elementary for higher-order logic. Nevertheless, formulas that occur in practice often have small models, and the usefulness of an approach similar to the one described in this paper has been proved in [11].

*This work was supported by the PhD program Logic in Computer Science of the German Research Foundation.

Section 2 introduces the logic, a version of higher-order logic on top of the simply typed λ-calculus. The model generation algorithm, and in particular the translation into propositional logic are described in Section 3. We conclude with some final remarks in Section 4.

2 The HOL Logic

Our translation can handle a large fragment of the logic that is underlying the HOL [9] and Isabelle/HOL theorem provers. The logic is originally based on Church's simple theory of types [3]. In this section we present the syntax and set-theoretic semantics of the relevant fragment. A complete account of the HOL logic, including a proof system, can be found in [8].

We distinguish types and terms, intended to denote certain sets and elements of sets respectively. Types σ are given by the following grammar, where α ranges over a countably infinite set TV of type variables:

$$\sigma ::= \mathbb{B} \,|\, \alpha \,|\, \sigma \to \sigma.$$

Type variables stand for arbitrary non-empty sets. $\mathbb{B}$ (sometimes called o in the literature) denotes a distinguished two-element set $\{\top, \bot\}$. If σ_1 and σ_2 are types, then $\sigma_1 \to \sigma_2$ is the function type with domain σ_1 and range σ_2. It denotes the set of all[1] total functions from the set denoted by its domain to the set denoted by its range. As usual, $\to$ associates to the right, i.e. $\sigma_1 \to \sigma_2 \to \sigma_3$ is short for $\sigma_1 \to (\sigma_2 \to \sigma_3)$.

We assume a countably infinite set V of variables. A term t_σ of type σ is either an (explicitly typed) variable, logical constant, application, or λ-abstraction. Terms are given by the following grammar:

$$t_\sigma ::= x_\sigma \,|\, c_\sigma \,|\, (t_{\sigma' \to \sigma}\, t_{\sigma'})_\sigma \,|\, (\lambda x_{\sigma_1} .\, t_{\sigma_2})_{\sigma_1 \to \sigma_2},$$

where x ranges over variables, and c_σ is either $\Longrightarrow_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$ (implication) or $=_{\sigma'\to\sigma'\to\mathbb{B}}$ (equality on σ'), usually written in infix notation. Other logical constants, including $\vee_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$, $\wedge_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$, $\neg_{\mathbb{B}\to\mathbb{B}}$, and quantifiers of arbitrary order, can be defined as λ-terms [1]. Terms of type $\mathbb{B}$ are called formulas.

We now define the semantics of terms. Let t_σ be a term of type σ. Let $tv(t_\sigma) \subseteq TV$ be the set of all type variables that occur in t_σ. tv can be defined inductively with the help of an auxiliary function tv' that collects the type variables occuring in a type:

$$\begin{aligned} tv'(\mathbb{B}) &= \emptyset, \\ tv'(\alpha) &= \{\alpha\}, \\ tv'(\sigma_1 \to \sigma_2) &= tv'(\sigma_1) \cup tv'(\sigma_2), \end{aligned}$$

[1] The difference between standard and Henkin's general models [10], where function types may denote a subset of all total functions, is not relevant in the context of this paper: we will only consider finite models.

and

$$
\begin{aligned}
tv(x_\sigma) &= tv'(\sigma), \\
tv(c_\sigma) &= tv'(\sigma), \\
tv((t_{\sigma' \to \sigma}\, t'_{\sigma'})_\sigma) &= tv(t_{\sigma' \to \sigma}) \cup tv(t'_{\sigma'}), \\
tv((\lambda x_{\sigma_1} .\, t_{\sigma_2})_{\sigma_1 \to \sigma_2}) &= tv'(\sigma_1) \cup tv(t_{\sigma_2}).
\end{aligned}
$$

Note that $tv(t_\sigma)$ is not necessarily contained in $tv'(\sigma)$. Types σ with $tv'(\sigma) \neq \emptyset$ and terms t_σ with $tv(t_\sigma) \neq \emptyset$ are called polymorphic. Furthermore, let $fv(t_\sigma) \subseteq V$ be the set of all free variables that occur in t_σ, defined as usual:

$$
\begin{aligned}
fv(x_\sigma) &= \{x_\sigma\}, \\
fv(c_\sigma) &= \emptyset, \\
fv((t_{\sigma' \to \sigma}\, t'_{\sigma'})_\sigma) &= fv(t_{\sigma' \to \sigma}) \cup fv(t'_{\sigma'}), \\
fv((\lambda x_{\sigma_1} .\, t_{\sigma_2})_{\sigma_1 \to \sigma_2}) &= fv(t_{\sigma_2}) \setminus \{x_{\sigma_1}\}.
\end{aligned}
$$

It is obvious that $tv(t_\sigma)$ and $fv(t_\sigma)$ are finite.

An environment D for t_σ is a function that assigns to each type variable $\alpha \in tv(t_\sigma)$ a non-empty set D_α. A variable assignment A for t_σ w.r.t. an environment D maps each variable $x_{\sigma'} \in fv(t_\sigma)$ to an element $A(x_{\sigma'})$ of the set denoted by the type σ'. (For σ' a type variable, this set is given by $D_{\sigma'}$.) Given a variable assignment A, a variable $x_{\sigma'} \in fv(t_\sigma)$, and an element d of the set denoted by σ', let $A[x_{\sigma'} \mapsto d]$ be the assignment that maps $x_{\sigma'}$ to d, and $v \neq x_{\sigma'}$ to $A(v)$. Now

$$
\begin{aligned}
[\![x_\sigma]\!]_D^A &= A(x_\sigma), \\
[\![\Longrightarrow_{\mathbb{B} \to \mathbb{B} \to \mathbb{B}}]\!]_D^A &\text{ is the function that sends } \left\{ \begin{array}{lll} \top, \top & \text{to} & \top \\ \top, \bot & \text{to} & \bot \\ \bot, \top & \text{to} & \top \\ \bot, \bot & \text{to} & \top \end{array} \right. , \\
[\![=_{\sigma' \to \sigma' \to \mathbb{B}}]\!]_D^A &\text{ is the function that sends } x, y \in D(\sigma') \text{ to } \left\{ \begin{array}{ll} \top & \text{if } x = y \\ \bot & \text{otherwise} \end{array} \right. , \\
[\![(t_{\sigma' \to \sigma}\, t'_{\sigma'})_\sigma]\!]_D^A &= [\![t_{\sigma' \to \sigma}]\!]_D^A ([\![t'_{\sigma'}]\!]_D^A) \text{ (function application)}, \\
[\![(\lambda x_{\sigma_1} .\, t_{\sigma_2})_{\sigma_1 \to \sigma_2}]\!]_D^A &\text{ is the function that sends each } d \in D(\sigma_1) \text{ to } [\![t_{\sigma_2}]\!]_D^{A[x_{\sigma_1} \mapsto d]}.
\end{aligned}
$$

Hence the semantics of a term t_σ is an element of the set denoted by the type σ, i.e. $[\![t_\sigma]\!]_D^A \in D(\sigma)$.

3 Bounded Model Generation

The model generation for a HOL formula $\phi = t_{\mathbb{B}}$ proceeds in several steps. We first fix the size of the model by choosing an environment D for ϕ that contains only finite sets. (Note that environments are determined uniquely up to isomorphism by the size of the sets that they assign to type variables; the names of elements are irrelevant.) With a fixed finite size for every set denoted by a type variable, *every* set denoted by a type then has a finite size: clearly $|D(\mathbb{B})| = 2$, and $|D(\sigma_1 \to \sigma_2)| = |D(\sigma_2)|^{|D(\sigma_1)|}$. Our task now

is to find a variable assignment A with $[\![\phi]\!]_D^A = \top$. (To generate a countermodel, we can either consider $\neg\phi$, or – equivalently – search for a variable assignment A with $[\![\phi]\!]_D^A = \bot$.) At this point we can already view bounded model generation as a generalization of satisfiability checking, where the search tree is not necessarily binary, but still finite.

3.1 Translation into Propositional Logic

The input formula ϕ is translated into a propositional formula that is satisfiable if and only if such a variable assignment exists. Propositional formulas are given by the following grammar:

$$\varphi ::= \text{True} \,|\, \text{False} \,|\, p \,|\, \neg\varphi \,|\, \varphi \vee \varphi \,|\, \varphi \wedge \varphi,$$

where p ranges over a countably infinite set of Boolean variables. The translation is by induction over terms and types. As an intermediate data structure, trees of propositional formulas are used. A tree of height 1 and width m corresponds to a term whose type is a type variable (denoting a set of size m) or $\mathbb{B}$ (for $m = 2$), while an n-ary function or predicate is given by a tree of height $n + 1$. Application and λ-abstraction can be "lifted" from the term level to this intermediate data structure.

To define the translation more precisely, several auxiliary functions are needed. The translation $\mathcal{T}_D$ from terms to trees of propositional formulas is given by the following rules.

$$\begin{aligned}
\text{create}(\mathbb{B}) &= [(v), (v)], \\
\text{create}(\alpha) &= [(v), \ldots, (v)] \text{ of length } |D_\alpha|, \\
\text{create}(\sigma_1 \to \sigma_2) &= [\text{create}(\sigma_2), \ldots, \text{create}(\sigma_2)] \text{ of length } |D(\sigma_1)|,
\end{aligned}$$

$$\begin{aligned}
\text{TT} &= [\text{True}, \text{False}], \\
\text{FF} &= [\text{False}, \text{True}], \\
\delta_k^n &= \begin{cases} \text{True} & \text{if } n = k \\ \text{False} & \text{otherwise} \end{cases}, \\
\text{uv}_k^n &= [\delta_1^k, \ldots, \delta_n^k], \\
\text{consts}(\mathbb{B}) &= [\text{TT}, \text{FF}], \\
\text{consts}(\alpha) &= [\text{uv}_1^{|D_\alpha|}, \ldots, \text{uv}_{|D_\alpha|}^{|D_\alpha|}], \\
\text{consts}(\sigma_1 \to \sigma_2) &= \text{pick}(\underbrace{[\text{consts}(\sigma_2), \ldots, \text{consts}(\sigma_2)]}_{|D(\sigma_1)|}), \\
\Delta_k^n &= \begin{cases} \text{TT} & \text{if } n = k \\ \text{FF} & \text{otherwise} \end{cases}, \\
\text{UV}_k^n &= [\Delta_1^k, \ldots, \Delta_n^k],
\end{aligned}$$

$$\begin{aligned}
\text{apply}([t], [\varphi]) &= \text{treemap}((\lambda\varphi'.\, \varphi' \wedge \varphi), t), \\
\text{apply}([t_1, t_2, \ldots, t_n], [\varphi_1, \varphi_2, \ldots, \varphi_n]) &= \text{merge}(\vee, \text{apply}([t_1], [\varphi_1]), \\
&\qquad \text{apply}([t_2, \ldots, t_n], [\varphi_2, \ldots, \varphi_n])),
\end{aligned}$$

$$
\begin{aligned}
\text{all}([\varphi_1, \ldots, \varphi_n]) &= \varphi_1 \wedge \ldots \wedge \varphi_n, \\
\text{enum}([\varphi_1, \ldots, \varphi_n]) &= [\varphi_1, \ldots, \varphi_n], \\
\text{enum}([t_1, \ldots, t_n]) &= \text{map}(\text{all}, \text{pick}([\text{enum}(t_1), \ldots, \text{enum}(t_n)])),
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_D^B(x_\sigma) &= \begin{cases} B(x_\sigma) & \text{if } x_\sigma \in \text{dom}\, B \\ \text{create}(\sigma) & \text{otherwise} \end{cases}, \\
\mathcal{T}_D^B(\Longrightarrow_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}) &= [[\text{TT}, \text{FF}], [\text{TT}, \text{TT}]], \\
\mathcal{T}_D^B(=_{\sigma'\to\sigma'\to\mathbb{B}}) &= [\text{UV}_1^{|D(\sigma')|}, \ldots, \text{UV}_{|D(\sigma')|}^{|D(\sigma')|}], \\
\mathcal{T}_D^B((t_{\sigma'\to\sigma}\, t'_{\sigma'})_\sigma) &= \text{apply}(\mathcal{T}_D^B(t_{\sigma'\to\sigma}), \text{enum}(\mathcal{T}_D^B(t'_{\sigma'}))), \\
\mathcal{T}_D^B((\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1\to\sigma_2}) &= [\mathcal{T}_D^{B[x_{\sigma_1}\mapsto d_1]}(t_{\sigma_2}), \ldots, \mathcal{T}_D^{B[x_{\sigma_1}\mapsto d_{|D(\sigma_1)|}]}(t_{\sigma_2})], \\
&\qquad \text{where } [d_1, \ldots, d_{|D(\sigma_1)|}] = \text{consts}(\sigma_1).
\end{aligned}
$$

Some explanations are in order. (v) is a placeholder for a fresh Boolean variable, i.e. different occurrences of (v) are replaced by different variables. We use Boolean variables in a unary rather than in a binary fashion. This means that we need n variables to represent an element of a set of size n, rather than $\lceil \log_2 n \rceil$ variables. However, exactly one of these variables must later be set to True (which keeps the search space for the SAT solver small), and this encoding allows for a relatively simple translation of application. To ensure that exactly one of the Boolean variables $p_1, \ldots, p_n$ is set to True, a propositional formula

$$
\text{wf}_{[p_1,\ldots,p_n]} = \left(\bigvee_{i=1}^{n} p_i \right) \wedge \bigwedge_{\substack{i,j=1 \\ i\neq j}}^{n} (\neg p_i \vee \neg p_j)
$$

is constructed and later conjoined with the result of the translation.

TT and FF are trees corresponding to $\top$ and $\bot$, respectively. $\text{consts}(\sigma)$ returns a list with one tree for each element in $D(\sigma)$. $\text{pick}([x_1, \ldots, x_n])$ – where each x_i is again a list – is an auxiliary function that returns a list containing all possible choices of one element from each list x_i. For the special case $x_1 = \ldots = x_n$, this corresponds to all functions from an n-element set to elements of x_1.

The translation is parameterized by a partial assignment B of trees to bound variables. Initially this partial assigment is equal to $\emptyset$, and it is extended whenever the translation descends into the body of a λ-abstraction.

$\text{treemap}(f, t)$ applies the function f to every propositional formula in the tree t, thereby returning a new tree. $\text{merge}(f, t_1, t_2)$ merges two trees t_1 and t_2 by applying f to corresponding propositional formulas in t_1 and t_2. Here f is a function that takes 2 propositional formulas as arguments and returns one propositional formula. Hence the result is again a single new tree. t_1 and t_2 must have the same "structure", i.e. differ at most in the formulas that they contain (but not in their height or width). $\text{enum}(t)$, for t a tree representing an element of $D(\sigma)$, computes propositional formulas $\varphi_1, \ldots, \varphi_n$ expressing that t represents the first, …, n-th element of $D(\sigma)$. $\text{map}(f, l)$ simply applies f to every element in a list l.

The rule for $\mathcal{T}_D^B((t_{\sigma'\to\sigma}\, t'_{\sigma'})_\sigma)$ suggests that the translations of $t_{\sigma'\to\sigma}$ and $t'_{\sigma'}$ are completely independent of each other. This is not quite true. We require variables that

occur free in both subterms to be replaced by the same tree each time. To this end a mapping from free variables to trees of Boolean variables is built during the translation. For the sake of simplicity this is not shown in the above rules. The same mapping is later used to convert a Boolean assignment, returned by the SAT solver, to a variable assignment for ϕ.

Since ϕ is a term of type $\mathbb{B}$, the result $\mathcal{T}_D^{\emptyset}(\phi)$ of the translation must be a tree of the form $[\mathcal{T}_D^{\emptyset}(\phi)^{\top}, \mathcal{T}_D^{\emptyset}(\phi)^{\perp}]$ for some propositional formulas $\mathcal{T}_D^{\emptyset}(\phi)^{\top}, \mathcal{T}_D^{\emptyset}(\phi)^{\perp}$.

Proposition 3.1 *Soundness, Completeness. Let* $* \in \{\top, \perp\}$*, and let* WF *be the conjunction of all* wf*-formulas constructed during the translation. Then* $[\![\phi]\!]_D^A = *$ *for some variable assignment* A *if and only if* $\mathrm{WF} \wedge \mathcal{T}_D^{\emptyset}(\phi)^*$ *is satisfiable.*

The theorem can be proved by generalization from formulas to terms of arbitrary types, followed by structural induction over the term. We omit the details.

3.2 Finding a Satisfying Assignment

Satisfiability can be tested with an off-the-shelf SAT solver. To this end translations into DIMACS SAT and DIMACS CNF format [6] have been implemented. The translation into SAT format is trivial, whereas CNF format (supported by zChaff [13], BerkMin [7] and other state-of-the-art solvers) requires the Boolean formula to be in conjunctive normal form. To avoid an exponential blowup at this stage, we translate into definitional CNF, introducing auxiliary Boolean variables where necessary.

Isabelle/HOL runs on a number of different platforms, and installation should be as simple as possible. Therefore we have also implemented a naive DPLL-based [5, 19] SAT solver in Isabelle. This solver is not meant to replace the external solver for serious applications, but it has proved to be efficient enough for small examples, and hence allows users to experiment with the countermodel generation without them having to worry about the installation of an additional tool.

If the SAT solver cannot find a satisfying assigment, the translation is repeated for a larger environment. The user can specify several termination conditions: a maximal size for sets in the environment, a limit on the number of Boolean variables to be used, a runtime limit. The stepwise extension of the environment guarantees that if the SAT solver is complete, a model will be found that is minimal w.r.t. its size. Of course this is not necessarily true for incomplete (e.g. stochastic) SAT solvers.

3.3 Example Translation

Consider the formula $\phi = ((\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha} =_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to\mathbb{B}} y_{\alpha\to\alpha})_{\mathbb{B}}$. Its only type variable is α, and its only free variable is $y_{\alpha\to\alpha}$. In an environment D with $|D_\alpha| = 2$ (and hence $|D(\alpha \to \alpha)| = 2^2 = 4$), the subterms of ϕ are translated into the following trees:

$$\begin{aligned}
\mathcal{T}_D^{\emptyset}((\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha}) &= [[\mathrm{True}, \mathrm{False}], [\mathrm{False}, \mathrm{True}]], \\
\mathcal{T}_D^{\emptyset}(=_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to\mathbb{B}}) &= [\mathrm{UV}_1^4, \mathrm{UV}_2^4, \mathrm{UV}_3^4, \mathrm{UV}_4^4], \\
\mathcal{T}_D^{\emptyset}(y_{\alpha\to\alpha}) &= [[y_0, y_1], [y_2, y_3]]
\end{aligned}$$

with four Boolean variables y_0, y_1, y_2, y_3. Using the translation rule for application, we then obtain (Boolean formulas equivalent to)

$$\mathcal{T}_D^{\emptyset}(\phi)^{\top} = y_0 \wedge y_3$$

and

$$\mathcal{T}_D^{\emptyset}(\phi)^{\perp} = (y_0 \wedge y_2) \vee (y_1 \wedge y_2) \vee (y_1 \wedge y_3).$$

Additionally two wf-formulas are constructed, namely

$$\mathrm{wf}_{[y_0,y_1]} = (y_0 \vee y_1) \wedge (\neg y_0 \vee \neg y_1)$$

and

$$\mathrm{wf}_{[y_2,y_3]} = (y_2 \vee y_3) \wedge (\neg y_2 \vee \neg y_3).$$

3.4 Some Extensions: Sets, ϵ, and Datatypes

Several extensions to the logic described in Section 2 can straightforwardly be integrated into our framework. The type σ set of sets with elements from σ is isomorphic to $\sigma \to \mathbb{B}$. Set membership $x \in P$ becomes predicate application $P\,x$, and set comprehension $\{x.\,P\}$ can be translated simply as P.

Hilbert's choice operator, ϵ, is a polymorphic constant of type $(\sigma \to \mathbb{B}) \to \sigma$, satisfying the axiom

$$\phi_\epsilon : \quad \frac{\exists x.\, P\,x}{P(\epsilon P)}.$$

Similarly, *The*, also a constant of type $(\sigma \to \mathbb{B}) \to \sigma$, satisfies

$$\phi_{\mathit{The}} : \quad (\mathit{The}\,x.\,x = a) = a,$$

and *arbitrary* is a completely unspecified polymorphic constant. For the purpose of our translation $\mathcal{T}_D$, we can treat these logical constants just like free variables, and introduce Boolean variables that determine their interpretation. For ϵ and *The*, we then translate the conjunction of the original formula ϕ with the relevant axiom (i.e. $\phi_\epsilon \wedge \phi$ or $\phi_{\mathit{The}} \wedge \phi$, respectively, or $\phi_\epsilon \wedge \phi_{\mathit{The}} \wedge \phi$ if both ϵ and *The* occur in ϕ). Type variables in ϕ_ϵ (or ϕ_{The}) are instantiated to match the type of ϵ (or *The*) in ϕ.

Isabelle/HOL allows the definition of inductive datatypes [2]. In general, inductive datatypes with free constructors require an infinite model. We are currently working on their integration into this framework – e.g. by considering only finite fragments of the datatype. However, many important datatypes are non-recursive, and for these, the situation is simpler. Examples are the type σ option, which augments a given type σ by a new element, product types $\sigma_1 \times \sigma_2$, and sum types $\sigma_1 + \sigma_2$. The general syntax of a non-recursive datatype definition is given by

$$(\alpha_1, \ldots, \alpha_n)\sigma ::= C_1\, \sigma_1^1 \ldots \sigma_{m_1}^1 \mid \ldots \mid C_k\, \sigma_1^k \ldots \sigma_{m_k}^k,$$

where the C_i are the datatype's constructors, the σ_j^i specify their argument types, and all σ_j^i only refer to previously defined types and type variables from $\alpha_1, \ldots, \alpha_n$. Such a datatype can be interpreted in a finite model; its size is equal to $S := \sum_{i=1}^{k} \prod_{j=1}^{m_i} |D(\sigma_j^i)|$. Hence an element of this datatype can be represented by a tree of height 1 and width S, and a datatype constructor C_i is a function of type $\sigma_1^i \to \ldots \to \sigma_{m_i}^i \to (\alpha_1, \ldots, \alpha_n)\sigma$, representable by a tree of height $m_i + 1$.

3.5 Some Optimizations

We briefly describe some optimizations in the implementation of the translation $\mathcal{T}_D$. None of them affect soundness or completeness of the algorithm.

Variables of a type with size 1 can be represented by [True], using no Boolean variable at all (instead of one Boolean variable x together with a $\mathrm{wf}_{[x]}$-formula x). Similarly variables of a type with size 2, including variables of type $\mathbb{B}$, can be represented by a tree of the form $[x, \neg x]$, rather than by a tree $[x_0, x_1]$ and a $\mathrm{wf}_{[x_0,x_1]}$-formula $(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$.

The Boolean formulas that are constructed during the translation process are simplified as much as possible, using basic laws of $\neg$, $\vee$, $\wedge$, True and False. Closed HOL formulas simply become True or False. The SAT solver is used only to search for an interpretation of *free* variables.

More importantly, we avoid unfolding the definition of logical constants (i.e. $\mathrm{True}_{\mathbb{B}}$, $\mathrm{False}_{\mathbb{B}}$, $\neg_{\mathbb{B}\to\mathbb{B}}$, $\wedge_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$, $\vee_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$, $\forall_{(\sigma\to\mathbb{B})\to\mathbb{B}}$, $\exists_{(\sigma\to\mathbb{B})\to\mathbb{B}}$) as λ-terms as far as possible. Instead these constants are replaced directly by their counterparts in propositional logic. Since every type is finite, quantifiers of arbitrary order can be replaced by a finite conjunction or disjunction.

The latter suggests a more general optimization technique, applicable also to other functions and predicates (including e.g. equality): namely specialization of the rule for function application to particular functions. While any given function can be represented by a tree, it is often more efficient to implement a particular function's action on its arguments, assuming these arguments are given as trees already, than to use the general translation rule and apply it to the tree representing the function. For $=_{\sigma\to\sigma\to\mathbb{B}}$ this avoids creating a tree whose size is proportional to $|D(\sigma)|^2$, and instead uses a function that operates on trees representing elements of $D(\sigma)$, their size possibly proportional to $\log |D(\sigma)|$ only.

3.6 Examples

Table 1 shows some examples of formulas for which our algorithm can automatically find a countermodel. Type annotations are suppressed, and functions in the countermodel are given by their graphs. The main purpose of these examples is to illustrate the expressive power of the underlying logic. The countermodels are rather small, and were all found within a few milliseconds.

4 Conclusions and Future Work

We have presented a translation from higher-order logic to propositional formulas, such that the resulting propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size. A working implementation of this translation, consisting of roughly 2,800 lines of code written in Standard ML [12], is available in the Isabelle/HOL theorem prover. A standard SAT solver can be used to find a satisfying assignment for the propositional formula, and if such an assignment is found, it can be transformed into a model for the HOL formula. This allows for the automatic generation of finite countermodels for non-theorems in Isabelle/HOL. A similar translation

Property/Formula	Countermodel
"Every function that is onto is invertible." $(\forall y.\, \exists x.\, f\,x = y) \implies (\exists g.\, \forall x.\, g\,(f\,x) = x)$	$D_\alpha = \{a_0, a_1\}, D_\beta = \{b_0\}$ $f = \{(a_0, b_0), (a_1, b_0)\}$
"There exists a unique choice function." $(\forall x.\, \exists y.\, P\,x\,y) \implies (\exists! f.\, \forall x.\, P\,x\,(f\,x))$	$D_\alpha = \{a_0\}, D_\beta = \{b_0, b_1\}$ $P = \{(a_0, \{(b_0, \mathrm{True}), (b_1, \mathrm{True})\})\}$
"The transitive closure of $A \cap B$ is equal to the intersection of the transitive closures of A and B."	$D_\alpha = \{a_0, a_1\}$ $A = \{(a_0, a_1), (a_1, a_0), (a_1, a_1)\}$ $B = \{(a_0, a_0), (a_1, a_0), (a_1, a_1)\}$

Table 1: Examples

has been discussed before [11]; the main contributions of this paper are its extension to higher-order logic and the seamless integration with a popular interactive theorem prover.

So far we have applied the technique only to relatively small examples. The applicability of the algorithm is limited by its non-elementary complexity. We believe that the algorithm can still be useful for practical purposes, since many formulas have small models. To substantiate this claim, and to further evaluate the performance of our approach, we plan to carry out some larger case studies, possibly from the area of cryptographic protocol verification [15, 16].

We also plan to incorporate further optimizations [4, 17], and to extend the translation to other Isabelle/HOL constructs: most notably the full language of HOL, including type operators [8], but also inductive datatypes, axiomatic type classes [18], inductively defined sets, and recursive functions.

References

[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, second edition, July 2002.

[2] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1999.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[6] DIMACS satisfiability suggested format, 1993. Available from ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc.

[7] E. Goldberg and Y. Novikov. BerkMin: A fast and robust sat solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[9] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems Series*, pages 49–70. Elsevier, 1994.

[10] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[11] Daniel Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, pages 130–139, San Diego, November 2000.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised.* MIT Press, May 1997.

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the 38th Design Automation Conference*, Las Vegas, June 2001.

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[15] Larry C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[16] Graham Steel, Alan Bundy, and Ewen Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2), 2002.

[17] Tanel Tammet. Finite model building: improvements and comparisons. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[18] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.

[19] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Proceedings of the 8th International Conference on Computer Aided Deduction (CADE 2002)*, volume 2392 of *Lecture Notes in Computer Science.* Springer, 2002.

Reducing Symmetries to Generate Easier SAT Instances

Jian Zhang*
Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100080, China
Email: zj@ios.ac.cn

June 2004

Abstract

Finding countermodels is an effective way of disproving false conjectures. In first-order predicate logic, model finding is an undecidable problem. But if a finite model exists, it can be found by exhaustive search. The finite model generation problem in the first-order logic can also be translated to the satisfiability problem in the propositional logic. But a direct translation may not be very efficient. This paper discusses how to take the symmetries into account so as to make the resulting problem easier, and reports some experimental results.

1 Introduction

Compared with theorem proving, the subject of disproving false conjectures has been less studied. But it is actually very important, since for open questions, you do not know whether the conjecture holds or not. If you give a false conjecture to a typical resolution-based theorem prover, the prover either runs forever or terminates without producing any useful information. When this happens, you do not know whether it is because the conjecture is false or the inference rules are not enough or the prover is not so efficient.

An effective way of disproving such conjectures is to find a suitable countermodel, namely, a model of the axioms in which all the premises hold but the conjecture does not. However, for first-order predicate logic, determining the existence of models is an undecidable problem in general. Fortunately, in many cases, finite models exist for satisfiable formulas, and we can first try to find a finite model. If we succeed, the conjecture is disproved by the countermodel; but if we fail, we can not say that the conjecture is false or true.

Currently, there are roughly two main approaches to finite model generation in the first-order logic. The first approach translates the problem into a satisfiability (SAT) problem in the propositional logic, and uses a SAT algorithm (e.g., the DPLL algorithm) to solve it. See for example, [5, 7, 4]. The second approach treats the problem as a

*Supported in part by K.C. Wong Education Foundation (Hong Kong) and NSFC under grant no. 60125207.

constraint satisfaction problem, and uses backtracking search to find the interpretations of the functions/predicates directly. Tools like FINDER [10], FALCON [12], SEM [14] and Mace4 [8] are based on this approach.

Each of the above two approaches has some benefits and weaknesses. For example, the translation approach may generate too many propositional formulas, and the constraint solving (or direct search) approach may not be so efficient on some problems. But using the first-order clauses directly leads to larger reasoning steps and also gives us opportunity to eliminate symmetrical subspaces.

We have been studying how to combine the two approaches. One way is to improve the direct search procedure by incorporating successful techniques developed in the SAT community [3]. Alternatively, we can also improve the translation approach by combining it with first-order model searchers [13]. This paper compares different ways of exploiting symmetries in the problem specification, so that the resulting SAT problem instances are easier. Some examples and experimental results will be given. The experiments were carried out on a Dell desktop computer (Optiplex GX270, Pentium 4, 2.8 GHz, 2G memory).

2 Finite Model Searching

The finite model generation problem can be stated as follows. Given a set of first order formulas and a non-empty finite domain, find an interpretation of all the function symbols and predicate symbols appearing in the formulas such that every formula is true under this interpretation. Such an interpretation is called a *model*. Usually we also assume that the formulas are all clauses, and every variable in a clause is (implicitly) universally quantified.

We do not consider many-sorted formulas in this paper. Without loss of generality, an n-element domain is assumed to be $D_n = \{ 0, 1, \ldots, n-1 \}$. The Boolean domain is { FALSE, TRUE }. If the arity of each function/predicate symbol is at most 2, a finite model can be conveniently represented by a set of *multiplication tables*, one for each function/predicate. For example, a 3-element model of the clause $f(x, x) = x$ is like the following:

f	0	1	2
0	0	1	0
1	1	1	0
2	0	1	2

Here f is a binary function symbol and its interpretation is given by the above 2-dimensional matrix. Each entry in the matrix is called a *cell*.

A finite model generation problem may be translated to a propositional satisfiability problem. A model can be represented by a set of assignments to propositional variables. Suppose there are m cells ($c_0, c_1, \ldots, c_{m-1}$) in the multiplication tables of the functions. We can introduce mn propositional variables: p_{ij} ($0 \leq i < m$, $0 \leq j < n$), where p_{ij} is true if and only if the i'th cell c_i has the value j. In addition, we also need one propositional variable for each cell in the predicates' multiplication tables. The first-order clauses can be translated into propositional clauses accordingly. For more details, see for example, [5, 7].

Alternatively, we can also search for the values of the cells directly. A finite model generation problem may also be regarded as a constraint satisfaction problem (CSP), which has been studied by many researchers in Artificial Intelligence. The variables of the CSP are the *cell terms* (i.e., ground terms like $f(0,0)$, $f(0,1)$, etc.). The domain of each variable is D_n (except for predicates, whose domain is the Boolean domain). The constraints are the set of ground instances of the input clauses. The goal is to find a set of assignments to the cells (e.g., $f(0,1) = 2$) such that all the ground clauses hold.

Typically backtracking search is used to solve the above problem. The basic idea of the search procedure is roughly like the following: Repeatedly extend a partial model (denoted by *Pmod*) until it becomes a complete model (in which every cell gets a value). Initially *Pmod* is empty. *Pmod* is extended by selecting an unassigned cell and trying to find a value for it. Of course, when no value is appropriate for the cell, backtracking is needed and *Pmod* becomes smaller. Such a procedure may be depicted as a search tree. Each edge of the tree corresponds to choosing a value for some cell.

As mentioned in the Introduction, each of the translation approach and the direct search approach has benefits and weaknesses.

The propositional satisfiability (SAT) problem has been studied for more than 40 years. Many theoretical results have been obtained, and many efficient algorithms have been designed. In recent years, more and more highly efficient SAT solvers are being developed, such as zChaff [9] and BerkMin [2].

On the other hand, the direct search approach works on first-order clauses, and may employ some structural information to speed up the search process. One technique that has been proved to be very useful is the so-called *Least Number Heuristic* (LNH in short) [12, 14]. It is based on the observation that in typical benchmark problems, most of the domain elements are "equivalent" when search begins. So we need only choose a few representative values to assign to the cells, and many branches of the search tree can be skipped. The LNH is more effective at the first few levels of the search tree. On many problems, it can reduce the search space significantly, and yet the overhead is negligible. In contrast, few good methods are known to discover and use symmetries in propositional clauses.

It is certainly desirable to combine the benefits of the two approaches, so that more problems can be easily solved.

3 A Motivating Example

Let us look at an example, i.e., finding ortholattices [6]. The axioms (and lemmas) are as follows:

```
m(x,y) = m(y,x).    j(x,y) = j(y,x).
j(j(x,y),z) = j(x,j(y,z)).
c(c(x)) = x.        j(x,m(x,y)) = x.
m(x,y) = c(j(c(x),c(y))).
m(x,x) = x.         j(x,x) = x.
j(c(x),x) = 1.      m(c(x),x) = 0.
j(1,x) = 1.  j(x,1) = 1.  m(1,x) = x.  m(x,1) = x.
m(0,x) = 0.  m(x,0) = 0.  j(0,x) = x.  j(x,0) = x.
```

When asked to find a 13-element model of the above formulas, MACE 2.2 [7] takes 9.34 seconds to conclude that such a model does not exist. Most of the time is spent on SAT solving rather than obtaining the propositional clauses (DPLL time: 9.09 seconds). If we add the following two clauses to the input:

```
c(0) = 1.    c(2) = 3.
```

the execution time will be 1.14 seconds (DPLL time: 0.89 seconds). The reduction is significant.

These two clauses represent the initial two steps taken by SEM [14]. Note that in the first step, there is only one branch, i.e., SEM decides that only the value `1` can be assigned to `c(0)`. Similarly, in the second step, there is also one choice. So adding the two clauses does not change the satisfiability of the original problem.

4 Adding Formulas to Eliminate Symmetrical Subspaces

When solving the quasigroup problems, Fujita *et al.* [1, 11] add a few clauses which eliminate quite many symmetrical subspaces. This greatly reduces the search time. But the additional constraints are domain-specific, namely, they can only be applied to quasigroup problems and other similar problems.

MACE [7] has an option ('`-c`') which allows the user to impose the constraint that the constants are different from each other. It is quite helpful when finding counterexamples, because the negation of the conjecture usually contains Skolem constants. It is also useful when, for instance, finding non-commutative groups. But that option may miss some solutions, in which two constants are assigned the same domain element. For example, when this option is used, MACE fails to find a 10-element countermodel which shows that some equation (i.e., the equation E1 in [6]) does not hold for ortholattices. MACE has another option ('`-z`') which adds isomorphism constraints to the generated propositional formula. But it applies only to constants.

As a more general method, we can simulate the LNH by adding certain constraints. For simplicity, we assume that no domain elements appear in the input and that there is only one binary function symbol `f`. SAGE [4] adds the following constraints:

```
f(0,0)=0  |  f(0,0)=1.
f(0,1)=0  |  f(0,1)=1  |  f(0,1)=2.
f(1,0)=0  |  f(1,0)=1  |  f(1,0)=2  |  f(1,0)=3.
...
```

Of course, these are only an approximation to the LNH. Some combinations actually need not be considered. For example, when `f(0,0) = 0` and `f(0,1) = 1`, we should not consider the case `f(1,0) = 3`. But it still prunes the search tree greatly, since we now need to examine only 2 (instead of `n`) values for `f(0,0)`, only 3 (instead of `n`) values for `f(0,1)`, and so on.

To get an understanding of its effectiveness, let us look at the QG5 problem. It has only one binary function symbol '`f`' whose multiplication table should be a quasigroup. In addition to this property, it has the following axioms:

```
f(x,x) = x.
f(f(f(y,x),y),y) = x.
f(y,f(f(x,y),y)) = x.
f(f(y,f(x,y)),y) = x.
```

Suppose we try to find all of its models. If we do not use any method for eliminating isomorphism, there are 120 models of size 7, and 720 models of size 8. If we add the above three formulas to the input, there are 24 models of size 7, and 24 models of size 8. But when we use the LNH, there is only one model of size 7, and one model of size 8. We see that the method is helpful, but it is not good enough.

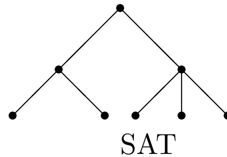
Here there is some tradeoff between the number of additional clauses and the effect of pruning. In general, to be more effective, we need to add more clauses. We are currently experimenting with different kinds of additional clauses.

5 Generating Multiple SAT Problems Dynamically

As mentioned in Section 2, a backtracking search method works by extending partial models. It is also mentioned that the LNH is more effective at the first few levels of the search tree. Below certain levels, the domain elements are no longer "equivalenet" and the heuristic is not effective. On the other hand, for many problems, propositional reasoning is more efficient at most nodes of the search tree.

Naturally one may think of combining first-order model searching with SAT solving, as demonstrated by the example in Section 3.

In [13], we propose a scheme for combining the two approaches, and report some experiences with SEM and MACE. The scheme looks like the following:



At the first few levels of the search, we use SEM with the LNH. Below certain levels (e.g., when every domain element is distinct), the search is tranfered to a SAT solver. The borderline can be decided by the user.

Let us look at the QG5 problem again. Its axioms are given at the end of the previous section. For this problem, the first 3 steps of SEM's search tree is like the following:

```
f(0,1) = 2;    f(2,0) = 3;     f(2,1) = 4.
```

At each step, SEM concludes – using the LNH and through various kinds of reasoning – that there is only one value that can be assigned to the corresponding cell. If we add these three equations to the input, and ask SEM to find all the models, without using the LNH, SEM will find that there are 2 models of size 7, 6 models of size 8. This is not too far away from the optimal numbers (1 model of size 7 and 1 model of size 8).

Table 1: Number of Partial Models for Various Problems

size	7	8	9	10	11	12	13
QG5	1	1	5	12	26	70	217
OL	23	54	849	6501	>10000	>10000	>10000
NCG	7	16	31	57	79	223	210

Thus adding the above constraints is quite helpful for eliminating isomorphic subspaces. Of course, we can ask SEM to go beyond the 3 steps and more subspaces can be eliminated. In general, more than one SAT instances are generated using this approach.

Is there any overhead? Yes. The main overhead will be the translation time (the time for obtaining propositional clauses from first-order ones), and perhaps reading/writing files. That will be significant if many SAT instances are generated. On the other hand, if the problem is difficult, the nodes of SEM's search tree are not so many, and the majority of work will be done by a SAT solver. The translation time is not so much, if compared with the searching time. In that case, the combination will be very helpful.

We have slightly modified SEM so that it backtracks when the LNH is no longer effective (i.e., when no domain elements are "equivalent"). We ask the modified program to count how many partial models it generates. Table 1 gives the number of partial models generated by SEM, on several problems. In addition to QG5 and ortholattice (OL) mentioned earlier, we have tested the program on the non-commutative group (NCG) problem. The first line of the table gives the size of the model, while the other lines give the corresponding numbers of partial models. We can see that, for NCG and QG5, there are not too many partial models. But OL has many partial models. This is probably because that the problem is too easy or has too many solutions.

We have also tried several other problems. Some of them are too easy, and some are too hard. For example, the combinatory logic problem `cl_BM` is already quite difficult when the size of the model is 6. So the results are not included in the above table. When the size is 6, 1599 partial models are generated; and when the size is 7, 49438 partial models are generated.

One way to reduce the number of partial models is to ask SEM to backtrack earlier (e.g., when there are still some "equivalent" domain elements). But then the symmetries are not exploited fully. Another way is to ask SEM to solve the easier subproblems, in which many cells are assigned values.

6 Concluding Remarks

As many highly efficient SAT solvers are being developed in recent years, it becomes more interesting to use them to find finite models and counter-examples in the first-order logic. If we take symmetries into account when generating propositional clauses, easier SAT instances may be obtained. In this paper, we have discussed two different approaches. One is static, which adds some formulas to the input and then gets a set of propositional clauses in the conventional way. The other is dynamic, which uses a first-order model searcher to derive some partial models, and then gets a number

of SAT instances (each corresponding to a partial model). Some issues are discussed, and experimental results are reported, using existing tools (or variations of them). We believe that the combination of first-order model searching and SAT solving is very promising for finding large models and counterexamples.

References

[1] M. Fujita, J. Slaney and F. Bennett, Automatic generation of some results in finite algebra, *Proc. 13th Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, 52–57, 1993.

[2] E. Goldberg and Y. Novikov, BerkMin: A fast and robust SAT solver, *Design, Automation, and Test in Europe (DATE'02)*, 142–149, 2002.

[3] Z. Huang, H. Zhang and J. Zhang, Improving first-order model searching by propositional reasoning and lemma learning, *Proc. 7th Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2004.

[4] Z. Huang and J. Zhang, Generating SAT instances from first-order formulas, *J. of Software*, to appear.

[5] S. Kim and H. Zhang, ModGen: Theorem proving by model generation, *Proc. 12th AAAI*, 162–167, 1994.

[6] W. McCune, Automatic proofs and counterexamples for some ortholattice identities, *Information Processing Letters*, 65(6): 285–291, 1998.

[7] W. McCune, MACE 2.0 reference manual and guide, Technical Memorandum ANL/MCS-TM-249, Argonne National Laboratory, Argonne, IL, USA, May 2001.

[8] W. McCune, Mace4 reference manual and guide, Technical Memorandum No. 264, Argonne National Laboratory, Argonne, IL, USA, Aug. 2003.

[9] M. Moskewicz *et al*, Chaff: Engineering an efficient SAT solver, *Proc. 38th Design Automation Conference*, 530–535, 2001.

[10] J. Slaney, FINDER: Finite domain enumerator – system description, *Proc. CADE-12*, 798–801, 1994.

[11] J. Slaney, M. Fujita and M. Stickel, Automated reasoning and exhaustive search: Quasigroup existence problems, *Computers and Mathematics with Applications* 29(2): 115–132, 1995.

[12] J. Zhang, Constructing finite algebras with FALCON, *J. Automated Reasoning* 17(1): 1–22, 1996.

[13] J. Zhang, Automatic symmetry breaking method combined with SAT, *Proc. ACM Symp. on Applied Computing*, 17–21, 2001.

[14] J. Zhang and H. Zhang, SEM: a system for enumerating models, *Proc. 14th IJCAI*, 298–303, 1995.

Finding and Using Counter-Examples

Alan Bundy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh, Scotland
Email: a.bundy@ed.ac.uk

June 2004

Abstract

Disproving conjectures, either by finding counterexamples or by refutation, is a neglected part of automated reasoning compared to proving theorems. Despite this neglect, it is an extremely important part of reasoning. For instance, in formal methods, initial implementations are notoriously error prone, and seldom meet their specifications. Moreover, industrial users of formal verification tools are much more likely to be impressed that a previously unsuspected, but important, bug was discovered than they are with a proof that a system is bug-free, especially when bugs are subsequently revealed during unanticipated uses of the system or when it is coupled to other components. It is time to reverse this neglect and give disproof the attention it deserves.

In this talk I will survey recent work in my research group on the detection and use of counter-examples to conjectures. This will include the automatic detection of counter-examples by the analysis of failed proofs, the use of refutation complete provers and testing against standard models. We will describe how these counterexamples can be used, for instance, to automatically generate attacks on security protocols, including previously unknown attacks on group protocols. Counterexamples can also be used to prevent theorem *proving* systems from wasting time proving false subgoals. Finally, we will discuss how counterexamples can be used automatically to correct faulty conjectures. The work reported is joint with Raul Monroy, Louise Dennis, Simon Colton, Alison Pease and Graham Steel.

The Use of Proof Planning Critics to Diagnose Errors in the Base Cases of Recursive Programs*

Louise A. Dennis
School of Computer Science and Information Technology, University of Nottingham,
lad@cs.nott.ac.uk

June 16, 2004

Abstract

This paper reports the use of proof planning to diagnose errors in program code. In particular it looks at the errors that arise in the base cases of recursive programs produced by undergraduates. It describes two classes of error that arise in this situation. The use of test cases would catch these errors but would fail to distinguish between them. The system adapts proof critics, commonly used to patch faulty proofs, to diagnose such errors and distinguish between the two classes. It has been implemented in λ*Clam*, a proof planning system, and applied successfully to a small set of examples.

The use of mathematical proof to show that a computer program meets its specification has a long history in Computer Science (e.g. [14, 13]). Considerable time and effort has been invested in creating computer-based tools to support the process of proving programs correct (e.g. [15, 8]). However the technique and tools are only used in very specialised situations in industry where programmers generally rely on testing and bug reports from users to assess the extent to which a program meets its specification.

There are several reasons why there is such poor uptake of the use of proof in industry. One is that the final proof will tell you if the program is correct, but failing to find a proof does not, on immediate inspection, help in locating errors. This problem can be particularly severe when using automated proof techniques which generally produce no proof trace in the case of failure. Many cases have been reported where the process of attempting a proof by hand has highlighted an error, for instance Paulson's discovery of new attacks against security protocols [16]. Anecdotal evidence suggests that errors are located by examining and reflecting on the process of the failed proof attempt.

It is worth noting the comparative success of model checking techniques (e.g. [11]). Model checkers are automated (though they require an expert user to convert the problem into an appropriate form) and return counterexamples when they fail. This confirms the analysis that automated support for error discovery is valuable and might aid a more widespread uptake of theorem proving technology.

This paper reports preliminary work using the proof planning paradigm (in particular the concept of proof critics) to diagnose the errors in program code. I focus on two classes

*This research was funded by EPSRC grant GR/S01771/01

of error that can arise in the base case of a recursive program and show how a proof critic can be written to distinguish between these two situations.

1 Proof Planning

Proof planning [1] is an Artificial Intelligence based technique for the automation of proof. One aspect of proof planning is the inspection of failed proof attempts by means of *proof critics* [10] which attempt to patch the proof.

Proof planners use AI-style planning techniques to generate proof plans. A proof plan is a proof of a theorem at some level of abstraction. The main planning operators used by a proof planner are called *proof methods*.

The first proof planner, *Clam* [3], focused on proof by mathematical induction using the rippling heuristic (a form of rewriting constrained to be terminating by meta-logical annotations) [2]. λ*Clam* [19, 5], which I used for this work, is a higher-order descendant of *Clam*which incorporates both a hierarchical method structure and proof critics.

λ*Clam* works by using depth-first planning with proof methods. Each node in the search tree is a subgoal under consideration at that point. The planner checks the preconditions for the available proof methods at each node and applies those whose preconditions succeed to create the child nodes. The plan produced is then a record of the sequence of method applications that lead to a trivial subgoal. λ*Clam*'s proof methods are believed to be sound although they are not currently reducible to sequences of inference rule applications. This means that while λ*Clam* outputs something that can be considered a proof in a similar way to a pen-and-paper correctness proof it does not produce a fully-formal proof.

1.1 Proof Methods

Proof method application is governed by preconditions (which may be either legal or heuristic in nature) and by a *proof strategy* (or compound method) which restricts the methods available depending on the progress through the proof. For instance we may wish to simplify a goal as much as possible by applying a rewriting method exhaustively before considering other procedures such as checking for tautologies.

In λ*Clam* a proof method can be *atomic* or *compound*. If it is compound then it is a sub-strategy built up from other methods and *methodicals*[1] [18]. Methodicals exist for repeats, sequencing methods, creating OR choices etc. and so complex proof strategies for controlling the search for a proof can be created.

1.1.1 The Proof Strategy for Induction

The proof strategy for induction can be seen in figure 1[2]. The diagram shows a top level repeat which attempts a disjunction of methods (in λ*Clam* these are attempted

[1] Analogous to a tactical in an LCF style theorem prover.

[2] There is no clear semantics for the use of diagrams to represent proof strategies. In this case boxes are used to indicate methods (both atomic and compound) and arrows to indicate method sequencing. Methods within methods indicate the method hierarchy, arrows that branch show OR choices and methods with more than one exit arrow indicate that they produce several goals which are treated differently.

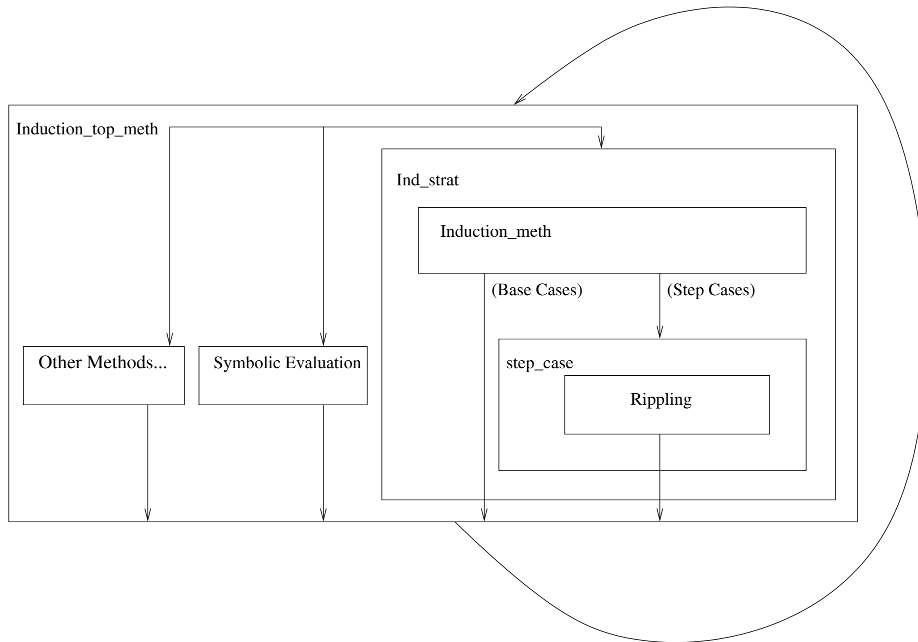


Figure 1: The Proof Strategy for Induction

from right to left, the planner backtracking out of failed choices). These include basic tautology checking, generalisation of common subterms and also symbolic evaluation and the induction strategy (`ind_strat`). Within the induction strategy, the induction method chooses an induction scheme and produces subgoals for base and step cases.

The top level strategy is reapplied to the base cases. The step cases are handled using rippling. The details of rippling are not important to the work described here and so are omitted from discussion. The results are then passed out to the top level strategy again. The process terminates when all subgoals have been reduced to *true*.

This proof strategy is used as the basis of the system for diagnosing errors in recursive programs discussed in this paper.

1.2 Proof Critics

A proof strategy provides a guide to which proof methods should be chosen at any given stage of the proof. Knowing which method is expected to apply gives additional information should the system generating the plan fail to apply it. Proof critics can be employed to analyse the reasons for failure and propose alternative choices. Critics are expressed in terms of preconditions and patches. The preconditions examine the reasons why the method has failed to apply. The proposed patch suggests some change to the proof plan or strategy. It may choose to propagate this change back through the plan and then continue from the current point, jump back to a previous point in the proof plan, or modify the current strategy being used by the planner, for instance by introducing new methods for consideration at that point.

In λ*Clam*v4, used for this work, critics can be built up into strategies using *criti-*

cals [9] in the same way that method strategies can be developed.

1.3 Related Work

Monroy [12] has already used proof planning to examine faulty conjectures. He follows work by Franova and Kodratoff [7] and Protzen [17] and attempts to synthesize a *corrective predicate* in the course of proof. The idea is that the corrective predicate will represent the theorem that the user intended to prove. This predicate is represented by a meta-variable, P, such that $P \rightarrow G$ where G is the original (non)theorem. P is instantiated during the course of a proof planning attempt. This approach assumes that, in some sense, the error arose because the original conjecture was too general.

The work reported here does not attempt to generate a corrective predicate. It seeks simply to diagnose the point in the code which is causing proof failure and allow a user to determine the appropriate modification. This allows for a more general class of errors to be identified beyond over-generalisations. Clearly there are advantages and disadvantages to both approaches and ideally they might at some point be combined into a system which both diagnosed the error and suggested a modification.

2 Novice Programs

In order to exploit the proof planning paradigm it is necessary to identify common patterns of proof in order to structure the proof strategy. In the case of faulty conjectures this would include identifying common patterns of proof failure which in turn requires the collection of a large body of data containing errors in order to observe the patterns of failure involved.

I have opted to study the programs produced by novice programmers, specifically undergraduates working on functional programming modules. It is relatively easy to acquire a large number of such programs and they are also likely to be well suited to proof by mathematical induction for which a mature proof strategy already exists (described above). On the downside such programs may not have a clear specification. Also, such programs may not contain the sort of bugs which we would expect to be generated by real programmers.

2.1 Errors in the Base Cases of Recursive Programs

I analysed a corpus of ML programs produced by students at the University of Edinburgh. This was a large set of approximately 150 scripts (attempting up to 4 problems) of which half were examined (the remainder being kept aside for later testing). These programs were all recursive in nature and a number of errors were identified and classified. This paper focuses specifically on errors occurring in the base cases of recursions.

An obvious approach to such problems is to filter the programs through test cases (in fact this is the approach adopted by many practitioners). However my analysis revealed two different ways in which errors may appear – these two different circumstances would not be distinguished by a counter-example alone.

The student programs contained some technical challenges for rippling which prevented the production of the proof plans showing that the step cases were correct. In

the rest of this paper I use a manufactured example in which the errors are duplicated but in which the basic problem is altered.

Consider the reverse function on lists[3] commonly defined as:

$$\begin{aligned} reverse(nil) &= nil, \\ reverse(X :: XS) &= reverse(XS) <> (X :: nil). \end{aligned}$$

The two errors that appeared to arise in student code were when either the base case of the recursion was incorrect in some way or it was omitted. A typical incorrect base case would be:

$$reverse(nil) = X :: XS.$$

The object of the research reported here was to distinguish between these two problems based on the failed proof attempt.

2.2 The Proof Strategy

The first challenge was to develop some sort of specification for the program. In this case I assume that the tutor has provided a "correct" version of the program and that the system is attempting to prove their equivalence[4]. In this case the initial proof goal is

$$\forall l.\ student_reverse(l) = tutor_reverse(l).$$

Hand proofs of these equivalences suggest that the proof stalls in the base case of the induction at either

$$X :: XS = nil$$

when the base case is incorrect or

$$student_reverse(nil) = nil$$

when the base case is missing.

In the case of the incorrect base case both the goal has been reduced to a falsehood (assuming a free constructor specification).

In the second case (missing base case) we have two inequal terms, one of which has been reduced to variables and type constructors while the other still contains a defined term.

2.3 Critics for missing and incorrect base cases

This analysis suggested that there should be a critic on the symbolic evaluation method. Symbolic Evaluation is, in fact, a compound method consisting of repeated applications of a rewrite method and is shown in figure 2. I implemented a modification to this method so that a critic strategy is called if the `rewrite` method fails. This is shown in figure 3[5]. The critic strategy calls an atomic critic, `check_equalities`. This is

[3]In what follows we use *nil* to indicate the empty list, :: to indicate the cons function that joins an element to the front of a list and $<>$ to indicate a built-in append function which joins two lists together.

[4]Obviously this scenario is unlikely in an industrial setting but was sufficient for the problem at hand.

[5]A dashed line is used here to indicate that the critic is invoked if the method fails.

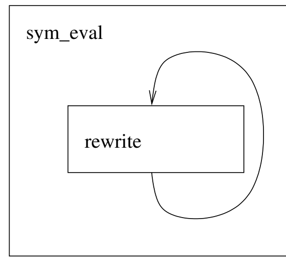


Figure 2: The Symbolic Evaluation Method

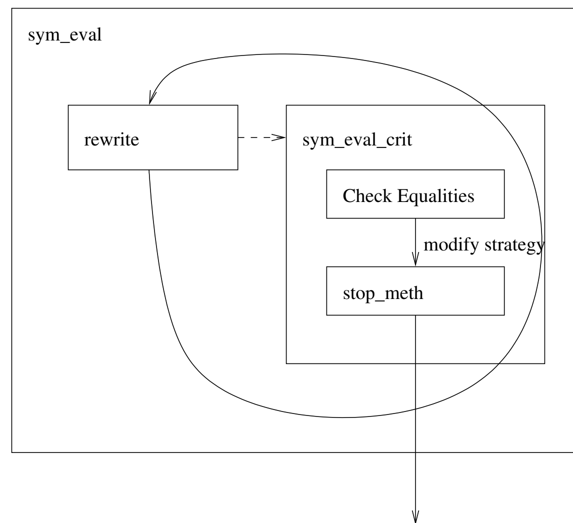


Figure 3: The Symbolic Evaluation Method and Base Case Diagnosis Critic

shown in figure 4 however I have chosen to present this in terms of preconditions and effects rather than patches since the critic does not patch the proof (or the theorem). The "known to be false" precondition checks a small internal list of non-theorems for a match – these assume a free constructor specification (ie. they contain $\neg(0 = s(N))$ where N is a variable) there are clearly some issues with this assumption and an obvious area for improving the critic is in making the implementation of this precondition more rigorous for instance by using I-Axiomatizations [4]. If its preconditions succeed then `check_equalities` processes its effects which in this case prints out an appropriate diagnosis message. If the critic succeeds the current strategy is changed so instead of proceeding as normal a method called `stop_meth` is invoked which closes the current proof branch immediately. Other proof branches are left open to be explored, potentially finding additional errors in the program (e.g. in the case where two base cases are incorrect or missing both are diagnosed – this occurs in some examples using the definition of *even*). If `check_equalities` fails then the system returns to the normal proof plan for induction.

Preconditions

Case 1 The current goal is known to be false.

OR

Case 2 The current goal is an equality and exactly one side of the equality is a simple term.

Effects

Case 1 Diagnose an incorrect base case.

Case 2 Diagnose a missing base case.

Figure 4: The Base Case Diagnosis Critic

3 Results

There are two sorts of results for the implementation of the base case diagnosis system. Firstly the system should correctly classify errors and secondly it should not diagnose actual theorems as faulty[6].

A handful of student programs were converted into λ*Clam*'s input format (modifying the programs in some cases because of the problems they posed to the step case proofs though preserving the errors) all of these were correctly diagnosed by the system. The student and tutor programs used are listed as an appendix to this paper. It should be noted that where a student has chosen and alternative base case, say 1 rather than 0, the program still diagnoses a missing base case since the two programs are not equivalent for 0 as input. It could be argued that this is instead an instance of an incorrect base case or even that it has arisen from an insufficiently well-defined specification on the part of the tutor. At the moment the system ignores these distinctions though it might be possible to extend it to identify rewrite rules defined from the student program that were not used in the proof.

The system was also run on λ*Clam*'s benchmark sets of theorems on lists and natural numbers – the new critic did not cause any of these to be incorrectly classified as faulty. The system can also prove the equivalence of the student program to the tutor one if if they have chosen a more complex form of recursion (eg. in the case of *reverse* having two base cases, one for the empty and one for the one element list and then a recursive case which removes two elements from the head of the list at a time).

[6]Within reason, I make no claims that this is a decision procedure which can never conclude that a true theorem has a flaw, however I want to be assured that the critic heuristic is *usually* correct.

4 Further Work

There is a risk that a proof has failed to go through because of some missing lemma, or inference rule. For instance suppose that constructors are not free but the system is unable to simplify $s(p(X))$ to X. In this case a falsehood could be detected where none exists. There are some obvious improvements which can be implemented to the existing falsehood detection system (already discussed above) but it would also be useful to link a counter-example generator into the system since the existence of an example where each program gave a different answer would guarantee that the student program was incorrect while the diagnosis system could provide additional information and guidance about the nature of the error.

Similarly in the detection of missing base cases it is possible that the student has supplied a base case which has been rewritten but to some term about which the system can not reason further (most obviously they may have used some built-in function which is not represented in the proof system). Tight integration between the student programming environment and the proof system would help overcome this but it would also be useful to detect whether the student side of the equation had been rewritten at all (in which case they have supplied some sort of base case in the program) or whether it was simply irreducible from the moment the goal was set up. It is also possible that they have implicitly made use of some equality between built-in functions which is not represented in the rewrite rules of the system. Once again the ability to generate a counter-example would provide a useful sanity check here.

My immediate intention is to port the existing proof plan and critic for base case diagnosis to IsaPlanner [6]. This is a newly developed proof planner containing much of the existing work on induction but also containing a wider knowledge base of rewrite rules, theorems and non-theorems and providing a more robust implementation base than λ*Clam*. Within this framework I hope to implement the more sophisticated ideas detecting falsehood and missing base cases discussed above. I also hope to investigate a wider set of examples and to use the actual programs produced by the students as the basis for theorems rather than porting their errors to functions more amenable to rippling.

4.1 Incorrect Step Cases

An obvious extension to this work is to look at errors occurring in the recursive case of functional programs. Several examples of these are also present in the data set. When the proof fails in these cases it takes place in the "tidying up" phase that follows the use of the `step case` method, once again failing during symbolic evaluation.

This will raise more serious issues about false positives since in a number of existing proof plans symbolic evaluation fails at this point, the method is backtracked out of and a subsidiary induction attempted[7].

[7]Lucas Dixon (personal communication) has suggested that a lemma speculation critic (already implemented in IsaPlanner) could solve this problem and would be called before the failure of symbolic evaluation.

5 Conclusion

This paper reported preliminary work to investigate the use of proof critics to diagnose program errors. It shows that, in principle at least, proof critics can be used to diagnose such errors and that they can be used to distinguish between different classes of error that would be picked up by the same counter-example. Potentially proof planning provides more information to a user about the nature of program error than a counter-example generator alone could.

References

[1] A. Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[2] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1. Elsevier, 2001.

[3] A. Bundy, F. van Harmelan, C. Horn, and A. Smaill. The oyster-clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648. Springer, 1990.

[4] H. Comon and R. Nieuwenhuis. Induction = I-axiomatization + first-order consistency. *Information and Computation*, 159(1–2), 2000.

[5] L. A. Dennis and J. Brotherston. *λclam v4: User/Developer's Manual.* Mathematical Reasoning Group, Division of Informatics, University of Edinburgh, 2002.

[6] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2003.

[7] M. Franova and Y. Kodratoff. Predicate synthesis from formal specification. In B. Neumann, editor, *10th European Conference on Artificial Intelligence*, pages 97–91. John Wiley and Sons, 1992.

[8] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[9] J. Gow. *The Dynamic Creation of Induction Rules Using Proof Planning.* PhD thesis, Centre for Intelligent Systems and their Applications, School of Informatics, University of Edinburgh, 2004.

[10] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[11] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, 1993.

[12] R. Monroy. Predicate synthesis for correcting faulty conjectures: The proof planning paradigm. *Automated Software Engineering*, 10(3):247–269, 2003.

[13] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.

[14] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[15] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer, 1994.

[16] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[17] M. Protzen. Patching faulty conjectures. In M. A. McRobbie and J. K. Slaney, editors, *13th Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1996.

[18] J. D. C. Richardson and A. Smaill. Continuations of proof strategies. In M. P. Bonancina and B. Gramlich, editors, *4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)*, Sienna, Italy, June 2001. Available from `http://www.logic.at/strategies/strategies01/`.

[19] J. D. C. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with lambda-clam. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 129–133. Springer, 1998.

6 Appendix

6.1 Tutor Programs

Insert Everywhere	$inserteverywhere(N, nil) = (N :: nil) :: nil$ $inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(X :: l), inserteverywhere(N, XS))$
Reverse	$reverse(nil) = nil$ $reverse(X :: XS) = reverse(XS) <> (X :: nil)$
Even	$even(0) = T$ $even(s(0)) = F$ $even(s(s(N)) = even(N)$

6.2 Student Programs

Insert Everywhere	
Version 1	$inserteverywhere(N, nil) = nil :: nil$ $inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(N :: l), inserteverywhere(N, XS))$
Version 2	$inserteverywhere(N, nil) = nil$ $inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(N :: l), inserteverywhere(N, XS))$
Version 3	$inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(N :: l), inserteverywhere(N, XS))$
Reverse	
Version 1	$reverse(nil) = X :: XS$ $reverse(X :: XS) = reverse(XS) <> (X :: nil)$
Version 2	$reverse(X :: XS) = reverse(XS) <> (X :: nil)$
Even	
Version 1	$even(0) = F$ $even(s(0)) = F$ $even(s(s(N)) = even(N)$
Version 2	$even(s(0)) = F$ $even(s(s(N))) = even(N)$
Version 3	$even(0) = T$ $even(s(s(N))) = even(N)$
Version 4	$even(s(s(N))) = even(N)$

Resource Graphs and Countermodels in Resource Logics

Didier Galmiche and Daniel Méry [1]

LORIA UMR 7503 - Université Henri Poincaré
Campus Scientifique, BP 239
54506 Vandœuvre-les-Nancy, France

Abstract

In this abstract we emphasize the role of a semantic structure called resource graph in order to study the provability in some resource-sensitive logics, like the Bunched Implications Logic (BI) or the Non-commutative Logic (NL). Such a semantic structure is appropriate for capturing the particular interactions between different kinds of connectives (additives and multiplicatives in BI, commutatives and non-commutatives in NL) that occur during proof-search and is also well-suited for providing countermodels in case of non-provability. We illustrate the key points with a tableau method for BI and present tools, namely BILL and CheckBI, which are respectively dedicated to countermodel generation and verification in this logic.

Key words: resources, proof-search, semantics, labels, countermodels.

1 Introduction

Over the past few years there has been an increasing amount of interest for resource-sensitive logical systems. The notion of *resource* is a basic one in many fields, including in computer science. The location, ownership, access to and, indeed, consumption of, resources are central concerns in the design of systems, such as networks, and in the design of programs, which access memory and manipulate data structures like pointers. Among so-called resource logics, we can mention Linear Logic [11] (LL) with its resource consumption interpretation, and Bunched Implications logic (BI) [15,16] with its resource sharing interpretation but also order-aware (non-commutative) logic (NL) [1]). As specification logics, they can represent features as interaction, resource distribution and mobility, non-determinism, sequentiality or coordination of entities. For instance, BI has been recently used as an assertion language for mutable data structures [12] and in this context it is important to verify pre- or post-conditions expressed in this logic but mainly to discover

[1] Email: galmiche@loria.fr, dmery@loria.fr

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

non-theorems and if possible to provide explanation about this non-provability by generating readable and usable countermodels.

For the above mentioned resource logics, proof search is not trivial mainly because of the management of context splitting and bunches in the related sequent calculi. Moreover, the design of semantic-based methods is difficult because the semantics of such logics (like Grothendieck topological semantics for BI), even if they are complete, are not always manageable in the context of proving or disproving formulae. Known methods, like tableaux or connections, dedicated to classical, intuitionistic or linear logics by using prefixes [13] cannot be easily extended to other resource logics. Therefore, in order to capture the particular interactions between connectives, for instance additive and multiplicative connectives in BI or commutative and non-commutative connectives in NL, our proposal is to start from a standard proof-search method like tableaux or connection and to define, for each logic, specific labels and (label) constraints that allow to capture and to deal with the underlying semantics. It leads to the design of new calculi with labelled signed formulae and constraints from which we define a new characterization of provability from standard notions, such as complementarity and closure conditions, extended with specific conditions about constraint satisfaction with respect to a particular set of constraints. This set is built during the proof-search process (tableau extensions or connection search) and can be easily represented as a graph, called dependency graph. It arises as the central syntactico-semantic structure from which the provability in some resource logics can be studied and allows us to generate countermodels, for instance, in Grothendieck topological semantics that is complete for BI. Another interesting point is to consider such a structure, with an appropriate valuation attached to some nodes, directly as a countermodel.

The relationships between semantics and syntax (labels and constraints) used to defined labelled calculi can be studied in both directions. For instance, in the case of BI without $\perp$, the labels and constraints directly reflect the elementary Kripke semantics of the logic [8] and thus the relationships between semantics and dependency graphs is clearly identified. In the case of BI (with $\perp$), the labels and constraints do not reflect the initial Grothendieck topological semantics, but considering the dependency (or resource) graph as the right representation of countermodels leads us to define a new simple resource semantics that is complete for BI [9] and for which the labelled calculus is a direct reflection of. A key point to mention is that these notions of labels, constraints and resource graphs are not exclusive to tableaux methods but can also be considered from the perspective of connection-based proof methods. It emphasizes that the semantic knowledge necessary to analyze provability is mainly covered by resource graphs built in parallel with the standard proof search methods. As said before, the approach is not only applicable for BI logic but also for Non-commutative logic (NL) for which we do not initially have a useful resource semantics but only a bunched calculus not well adapted to proof-search. In the case of NL, we are able to define a connection-based method using appropriate labels and resource graphs that capture its particular semantics [10]. Knowing that, in linear logics, connection methods and proof nets (a

standard semantic structure) are closely related, we then deduce an algorithm that builds proof nets. In case of non-provability, the partial proof net under construction and the resource graph both provide some explanations about the non-validity. Further work will be devoted in this context to build countermodels in the corresponding phase semantics and also to define a new semantics in which the resource graph can be seen as a countermodel.
In section 2 we focus on the notions of resource and dependency graph also called, under some conditions, a resource graph. This new semantic structure is central in the design of proof-search methods for resource logics like BI and allows us to generate countermodels in order to analyze the non-validity. In section 3, we emphasize the relationships between a resource logic, its semantics or its particular sequent calculus and the definition and construction of specific dependency graphs from which provability and non-provability can be discussed. In order to illustrate the main ideas and results about labels, constraints and resource graphs, we consider here the BI logic with an approach based on tableaux, but similar ideas can be applied to different resource logics such as MILL or NL and to different proof-search methods such as connections or natural deduction. In section 4, we describe the BILL system, an automated theorem prover for propositional BI that implements the previous results and builds proofs or countermodels in this logical fragment. In section 4.2, we consider the possibility of verifying models or countermodels through the description of a model-checker, called CheckBI. Further work will be devoted to the improvement of proof-search in resource logics by combining theorem-proving and model-checking approaches. For instance, we could improve the BILL and CheckBI tools and study how to combine their use for more efficient proving or disproving. Moreover, starting from these theoretical and practical results, we expect to propose similar methods and tools with countermodel generation for separation or spatial logics.

2 Resources and Resource Graphs

Let us formalize the notion of resource in an elementary way that is sufficient for our purpose. We start with a set R of resources with some properties that appear as characteristic: the existence of an *initial resource* or unit, denoted 1; the existence of a composition operator $\cdot$ that combines two resources x and y into a new one denoted $x \cdot y$; the existence of an operator $\leq$ which compares two resources x and y. At this level, our notion of resource is elementary because it does not consider the location or the owner of a resource. Then, we may state additional conditions on the comparison of resources, for instance, reflexivity ($x \leq x$) and/or transitivity ($x \leq y$ and $y \leq z$ imply $x \leq z$). We may also impose particular conditions on resource-composition such as associativity ($x \cdot (y \cdot z) = (x \cdot y) \cdot z$), commutativity ($x \cdot y = y \cdot x$), identity w.r.t. 1 ($1 = 1 \cdot x = x$) or compatibility with $\leq$ ($x \leq y$ imply $x \cdot z \leq y \cdot z$). Compatibility of resource-composition is a natural property in many resource-settings.

Let us consider now what we call a *resource graph*. It is a directed graph $G(N, E)$ with N a set of nodes and E a set of edges between nodes that satisfies some specific properties.
The nodes of the graphs are *labels*. The labelling language consists of the following symbols: a unit symbol 1, a binary function symbol $\circ$, a binary relation symbol $\leq$, a countable set of constants $c_1, c_2, \ldots$. *Labels* are inductively defined from the unit 1 and the constants as expressions of the form $x \circ y$ in which x and y are labels. *Atomic labels* are labels which do not contain any $\circ$, while *compound labels* contain at least one $\circ$. A *sublabel* of a label x is a subterm of x.
If we take $\circ$ to be monoidal then labels can be interpreted as multi-sets, 1 being the empty-set and $\circ$ being multi-set union. We can therefore omit the symbol $\circ$ when writing labels, for instance, $c_2c_2c_5$ represents the multi-set $\{c_2, c_2, c_5\}$ and the composition of the labels $c_1c_3c_4$ and c_2c_5 is the label $c_1c_2c_3c_4c_5$. Moreover, two labels are equivalent if they contain the same occurrences of constants. For instance, $c_1c_2c_2$, $c_2c_1c_2$, $1c_2c_1c_2$, $11c_1c_2c_2$ denote the same label (1 is not a constant). The notion of sublabel simply corresponds to the notion of sub-multi-set : y is a sublabel of x (notation $y \subseteq x$) if the multi-set denoted by y is included in the multi-set denoted by x. For instance, c_1c_3 is a sub-label of $c_1c_2c_3c_4$. Then, $\mathcal{P}(x)$ represents the set of the sublabels of x.
Let $G(N, E)$ a directed graph with the nodes of N labelled with the above defined labels. We denote $x \to y$ the oriented edge between the nodes x and y and $x \to^* y$ a path from x to y. From now on, we can consider that labels represent resources and edges represent links between resources.
To be a resource graph, a directed graph $G(N, E)$ must satisfy the following conditions: (i) $(\forall x \in N)(\mathcal{P}(x) \subseteq N)$ (closure by sublabel); (ii) If $yz \in N$ and $x \to y \in E$ then $xz \in N$ et $xz \to yz \in E$ (partial compatibility).
The first condition means that if a label is in the graph, its sublabels also are. The second condition corresponds to a weak form of compatibility of the resource composition with respect to the order. Figure 1 presents examples of resource graphs.

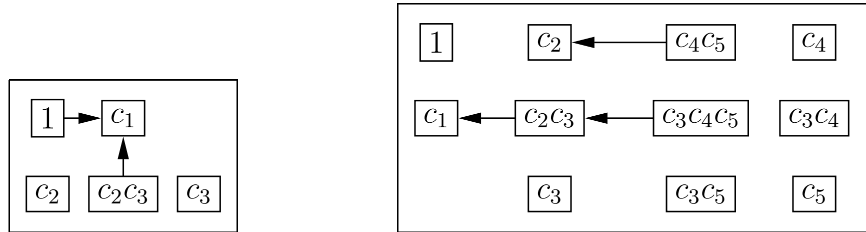


Figure 1. Examples of resource graphs

A resource graph is a graphical representation of a set of resources which can be composed and which verifies some particular conditions. In this abstract, we focus on the central role played by the resource graph as a semantic structure for proving and disproving formulae in resource-sensitive logics. It allows to generate countermodels and explanations about non-provability.

$$\frac{}{\phi \vdash \phi}\,ax \quad \frac{\Gamma \vdash \phi}{\Delta \vdash \phi}\,\Delta \equiv \Gamma \quad \frac{\Gamma(\Delta) \vdash \phi}{\Gamma(\Delta; \Delta') \vdash \phi}\,w \quad \frac{\Gamma(\Delta; \Delta) \vdash \phi}{\Gamma(\Delta) \vdash \phi}\,c \quad \frac{\Delta \vdash \phi \quad \Gamma(\Delta) \vdash \psi}{\Gamma(\phi) \vdash \psi}\,cut$$

$$\frac{}{\bot \vdash \phi}\,\bot_L \quad \frac{\Gamma(\emptyset_m) \vdash \phi}{\Gamma(\mathrm{I}) \vdash \phi}\,\mathrm{I}_L \quad \frac{}{\emptyset_m \vdash \mathrm{I}}\,\mathrm{I}_R \quad \frac{\Gamma(\emptyset_a) \vdash \phi}{\Gamma(\top) \vdash \phi}\,\top_L \quad \frac{}{\emptyset_a \vdash \top}\,\top_R$$

$$\frac{\Gamma(\phi, \psi) \vdash \chi}{\Gamma(\phi * \psi) \vdash \chi}\,*_L \quad \frac{\Gamma \vdash \phi \quad \Delta \vdash \psi}{\Gamma, \Delta \vdash \phi * \psi}\,*_R \quad \frac{\Delta \vdash \phi \quad \Gamma(\psi, \Delta') \vdash \chi}{\Gamma(\Delta, \phi \mathrel{-\!\!*} \psi, \Delta') \vdash \chi}\,\mathrel{-\!\!*}_L \quad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \mathrel{-\!\!*} \psi}\,\mathrel{-\!\!*}_R$$

$$\frac{\Gamma(\phi; \psi) \vdash \chi}{\Gamma(\phi \wedge \psi) \vdash \chi}\,\wedge_L \quad \frac{\Gamma \vdash \phi \quad \Delta \vdash \psi}{\Gamma; \Delta \vdash \phi \wedge \psi}\,\wedge_R \quad \frac{\Delta \vdash \phi \quad \Gamma(\psi; \Delta') \vdash \chi}{\Gamma(\Delta; \phi \to \psi; \Delta') \vdash \chi}\,\to_L$$

$$\frac{\Gamma; \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\,\to_R \quad \frac{\Gamma(\phi) \vdash \chi \quad \Delta(\psi) \vdash \chi}{\Gamma(\phi \vee \psi); \Delta(\phi \vee \psi) \vdash \chi}\,\vee_L \quad \frac{\Gamma \vdash \phi_{i\,(i=1,2)}}{\Gamma \vdash \phi_1 \vee \phi_2}\,\vee_{Ri}$$

Figure 2. The LBI sequent calculus

3 Proofs, Resource Graphs and Countermodels

Given a resource-aware logic, having (bunched or not) sequent calculi and related semantics, the design of proof-search methods is not trivial because of the management of formulae as resources (context splitting, interactions). We aim to illustrate the relationships between the way resources are managed, in bunched calculi or in semantics, and the definition of specific dependency or resource graphs from which proving or disproving can be studied.
In order to illustrate the main ideas and results about labels, constraints and dependency graphs, we consider here the BI logic with a tableau-based approach. But it is important to notice they can be also applied in different resource logics like MILL or NL and possibly with an alternative connection-based approach.

3.1 Resources and BI logic

The development of a mathematical theory of resource is one of the objectives of the programme of study of the logic of bunched implications (BI) [15,16]. The basic idea is to model directly the observed properties of resources and then to give a logical axiomatization. This logic provides a logical analysis of a basic notion of resource, quite different from linear logic's "number-of-uses" reading, which has proved rich enough to provide intuitionistic (*i.e.*, the additives) "pointer logic" semantics for programs which manipulate mutable data structures [12,14]. In this context, proof-search methods are necessary and the generation of countermodels in order to provide explanations of non-provability is very important.
The propositional language of BI consists of: a multiplicative unit I, the mul-

tiplicative connectives $*$, $-\!*$, the additive units $\top$, $\bot$, the additive connectives $\wedge$, $\rightarrow$, $\vee$, a countable set $L = \mathrm{p}, \mathrm{q}, \ldots$ of propositional letters. $\mathcal{P}(L)$, the collection of BI propositions over L, is given by the following inductive definition: $\phi ::= p \mid \mathrm{I} \mid \phi * \phi \mid \phi -\!* \phi \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \phi \vee \phi$.
The additive connectives correspond to those of intuitionistic logic (IL) whereas the multiplicative connectives correspond to those of multiplicative intuitionistic linear logic (MILL). The antecedents of logical consequences are structured as bunches, in which there are two ways to combine operations that respectively display additive and multiplicative behavior.
Bunches are given by the grammar: $\Gamma ::= \phi \mid \emptyset_a \mid \Gamma\,;\,\Gamma \mid \emptyset_m \mid \Gamma\,,\,\Gamma$. Equivalence of bunches, $\equiv$, is given by commutative monoid equations for "," and ";", whose units are $\emptyset_m$ and $\emptyset_a$ respectively, together with the evident substitution congruence for sub-bunches. $\Gamma(\Delta)$ denotes a subbunch Δ of Γ. Judgements are expressions of the form $\Gamma \vdash \phi$, where Γ is a "bunch" and ϕ is a proposition. The LBI sequent calculus is given on Figure 2. A proposition ϕ is a theorem iff $\emptyset_m \vdash \phi$ is provable in LBI.
BI has a natural semantics that is a Kripke-style semantics (interpretation of formulae) combining the Kripke semantics of IL and Urquhart's semantics of MILL [15]. This semantics deals with possible worlds, arranged as a commutative monoid and justified in terms of "pieces of information". It provides a way to read the formulae as propositions that are true or false relative to a given world. BI's Kripke semantics may be adapted to take into account for $\bot$ by moving from presheaves (elementary semantics) to sheaves on a topological space, namely Grothendieck topological semantics. Such a semantics considers an inconsistant world, at which $\bot$ is forced, together with the so-called indecomposable treatment of $\vee$: $m \models \phi \vee \psi$ if and only if $m \models \phi$ or $m \models \psi$.

3.2 Proof-search and Resource Graphs

Having in mind these results about completeness of BI's semantics, we aim to study the proof-theoretical foundations of (propositional) BI and to propose proof-search methods that build proofs or countermodels. For that, the challenge is to capture the interactions between connectives in the semantics or in the bunched calculi, through labels in the spirit of labelled deductive systems [4]. A key step in our semantic analysis is the use of so-called *dependency graphs* that are in fact particular resource graphs.
Let us illustrate this point with the BI logic. We define labels and sublabels as in section 2. *Label constraints* are expressions of the form $x \leq y$, where x and y are labels. We deal with *partially defined* labelling algebras, obtained from sets of labels and constraints by reflexive, transitive and partial compatible closure. We note $\overline{K}$ the closure of K, where K is a set of labels and constraints.
Having defined such labels and constraints, we can define new labelled calculi (sequent, tableaux or connections) such that, in parallel with the standard proof-search process, one generates a resource graph (set of particular constraints), from which

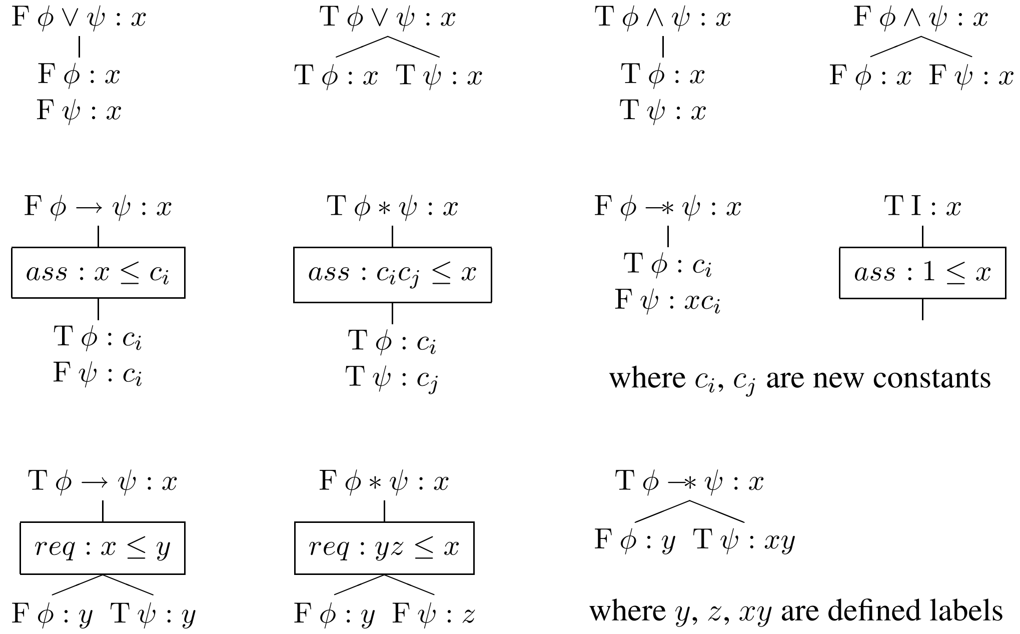


Figure 3. The tableau expansion rules

we can analyze the provability. We illustrate the main points with a tableau method that is well-adapted for a direct generation of countermodels. Compared to the standard method we consider *signed formulae* $\mathrm{Sg}\,\phi : l$ with Sg ($\in \{F, T\}$) being the sign of the formula ϕ and l its label. Then we define a labelled calculus that consists of the expansion rules of Figure 3.

Building a labelled tableau for an initial signed formula $\mathrm{F}\,\phi : 1$, by application of the above expansion rules, the key problem is to define some branch closure conditions such that either the tableau is closed and then ϕ is valid or there exists an open branch and then ϕ is not valid [5]. Moreover, in the latter case, we aim to build a countermodel of ϕ from an open branch. The example of Figure 4 illustrates the parallel construction of a tableau and a resource graph related to the set of assertions. We observe that we have $\pi\alpha$ rules that introduce constraints called *assertions* (including $F{-\!\!*}$ for which the assertion $c_i \leq c_i$ is implicit) and $\pi\beta$ rules that introduce constraints called *requirements*. Let us note that $\pi\alpha$ rules create new (atomic) labels while $\pi\beta$ have to reuse ones that already exist in the resource graph associated to a branch.

In order to define how to close a tableau branch $\mathcal{B}$ we define the resource graph of a branch $DG(\mathcal{B})$ as the directed graph such the nodes are the labels in $\mathcal{B}$ and there is an edge $x \rightarrow y$ iff there is an assertion $x \leq y$ in $\overline{Ass}(\mathcal{B})$.

In this context we say that two signed formulae $\mathrm{T}\,\phi : x$, $\mathrm{F}\,\phi : y$ are *complementary* in a branch $\mathcal{B}$ if and only if $x \leq y \in \overline{Ass}(\mathcal{B})$, i.e., there is a path from x to y in $DG(\mathcal{B})$. This definition can be seen as an extension of the standard notion of complementarity in standard labelled deductive systems. In order to define the

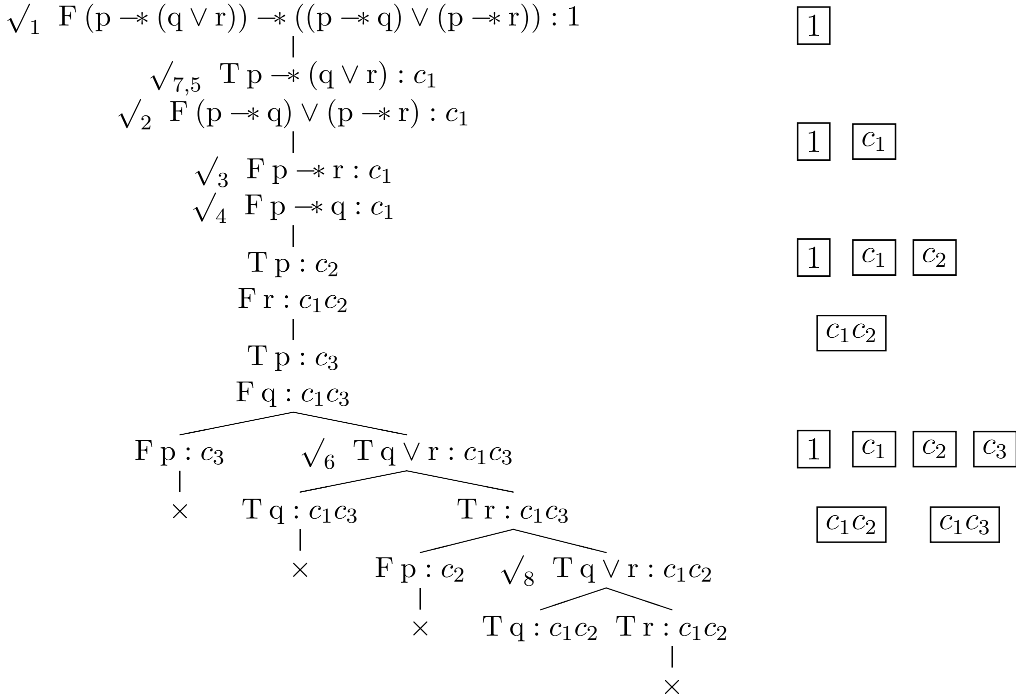


Figure 4. Tableau for $(\mathrm{p} \mathrel{-\!\!*} (\mathrm{q} \vee \mathrm{r})) \mathrel{-\!\!*} ((\mathrm{p} \mathrel{-\!\!*} \mathrm{q}) \vee (\mathrm{p} \mathrel{-\!\!*} \mathrm{r}))$

closure conditions for our labelled calculus, we need also to deal with units with a particular difficulty for the unit of $\vee$ denoted $\perp$. For that, we introduce the notion of inconsistent label in a tableau branch. We say that a label x is *inconsistent in* $\mathcal{B}$ if there exists a label y such that $y \leq x \in \overline{Ass}(\mathcal{B})$ and a sublabel z in of y, such that $\mathrm{T} \perp : z$ occurs in $\mathcal{B}$. More details are given in [9]. To complete the closure conditions, we impose that the requirements of a branch $\mathcal{B}$ should be instantiated in such a way that they are verified in $\overline{Ass}(\mathcal{B})$. To summarize, we have the following definition:
A tableau t is *closed* iff each branch $\mathcal{B}$ in t satisfies the following conditions:

- (i) (1) $\mathcal{B}$ contains two complementary formulae $\mathrm{T}\,\phi : x$ and $\mathrm{F}\,\phi : y$, or
  - (2) $\mathcal{B}$ contains a formula $\mathrm{F} \top : x$, or
  - (3) $\mathcal{B}$ contains a formula $\mathrm{F\,I} : x$ with $1 \leq x \in \overline{Ass}(\mathcal{B})$, or
  - (4) $\mathcal{B}$ contains a formula $\mathrm{F}\,\phi : x$ with x inconsistent in $\mathcal{B}$;
- (ii) $\forall\, x \leq y \in Req(\mathcal{B}),\, x \leq y \in \overline{Ass}(\mathcal{B})$.

If we suppress, in the above definition, the condition (4) in (i) then we have the closure conditions that fit well with BI without $\perp$ and its elementary Kripke semantics. We can also show that, with this additional condition, we can cope with BI and its Grothendieck topological semantics.
Coming back to our example of Figure 4, we observe that the tableau has four closed branches (marked with a cross $\times$) because of complementarity but also an open branch. Thus, we can conclude that the BI formula is not provable. In the

next subsection, we explain how to generate a countermodel from this open branch and the associated resource graph.

3.3 Completeness and Countermodel Generation

First, we can show that the resource tableau method is sound with respect to the Grothendieck topological semantics.

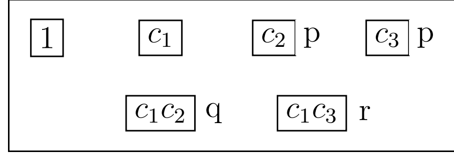
Theorem 3.1 (soundness) *Let ϕ be a proposition of BI, if there exists a closed tableau $\mathcal{T}$ for ϕ then ϕ is valid.*

Details of the proof are given in [9]. It is not a simple extension of the proof of [8] because, with $\perp$, we have to deal with Grothendieck topological semantics.
In order to prove the completeness of the method, we have to define how to build a countermodel of ϕ from an open branch in a labelled tableau and the resource graph for ϕ, which represents the reflexive, transitive and partial compatible closure of the assertions.
Therefore, if a formula ϕ happens to be unprovable, we should have enough information in the resource graph of an open branch to extract a countermodel for ϕ.
The idea behind the countermodel construction is to regard the resource graph itself as the desired countermodel, thereby considering it as a central semantic structure.
For that, we consider the nodes (labels) of the graph as the elements of a monoid whose multiplication is given by the label composition.
The key point is that, since the closure operator induces a *partially defined* labelling algebra, the resource graph only deals with those pieces of information (resources) that are relevant for deciding provability. Therefore, the monoidal product should be completed with suitable values for those compositions which are undefined. The problem of undefinedness is then solved by the introduction of a particular element, denoted π, to which all undefined compositions are mapped.
More precisely, in order to transform the resource graph $G(N, E)$ into a resource monoid $\langle \mathrm{R}, \cdot, 1, \leq \rangle$, we add a special node π to N, i.e., $\mathrm{R} \stackrel{\text{def}}{=} N \cup \{\pi\}$. Then, the monoidal product $\cdot$ is given by $x \cdot y \stackrel{\text{def}}{=} xy$ if $xy \in N$ and $x \cdot y \stackrel{\text{def}}{=} \pi$ otherwise. Notice that any composition with something undefined is itself undefined. The preordering relation is given by the arrows of the graph as follows: $x \leq y$ if and only if $x \rightarrow^* y$ or $y = \pi$, π being the greatest element. Finally, the forcing relation simply reflects the signed formulae of the open branch.

Theorem 3.2 (completeness) *Let ϕ be a proposition of BI, if ϕ is valid then there exists a closed tableau $\mathcal{T}$ for ϕ.*

The proof is detailed in [9] in which we also deduce the decidability and the finite model property for propositional BI as main results [9].
Returning to our example of Figure 4, we build a countermodel from the open branch $\mathcal{B}$ by considering the signed formulae of $\mathcal{B}$ of the form $\mathrm{T}\, At : x$ where At is an atom. The set of labels attached to At is considered as its valuation and we complete the resource graph with, at each node or label, the corresponding

atoms. Thus, we have the valuation v such that $v(\mathrm{p}) = \{c_2, c_3\}$, $v(\mathrm{q}) = \{c_1c_2\}$ and $v(\mathrm{r}) = \{c_1c_3\}$ and then represent the following resource graph:



We show that the node 1 does not force the formula ϕ, i.e., $G, 1 \not\models_v \phi$. We have both (i) $c_1 \models \mathrm{p} \multimap* (\mathrm{q} \vee \mathrm{r})$ and (ii) $c_1 \not\models (\mathrm{p} \multimap* \mathrm{q}) \vee (\mathrm{p} \multimap* \mathrm{r})$. Then, as we have $1c_1 = c_1$, we can deduce by definition of $\multimap*$ that $1 \not\models (\mathrm{p} \multimap* (\mathrm{q} \vee \mathrm{r})) \multimap* ((\mathrm{p} \multimap* \mathrm{q}) \vee (\mathrm{p} \multimap* \mathrm{r}))$. Let us show now (i) and (ii). For (i), we observe that $c_1c_2 \models \mathrm{q} \vee \mathrm{r}$ since $c_1c_2 \models \mathrm{q}$ and that $c_1c_3 \models \mathrm{q} \vee \mathrm{r}$ since $c_1c_3 \models \mathrm{r}$. The nodes c_2 and c_3 that force p can be combined with c_1 to provide c_1c_2 and c_1c_3. As we have $c_1c_2 \models \mathrm{q} \vee \mathrm{r}$ and $c_1c_3 \models \mathrm{q} \vee \mathrm{r}$, by definition of $\multimap*$, we deduce $c_1 \models \mathrm{p} \multimap* (\mathrm{q} \vee \mathrm{r})$. For (ii), we have $c_1 \not\models \mathrm{p} \multimap* \mathrm{q}$ since $c_3 \models \mathrm{p}$ and $c_1c_3 \not\models \mathrm{q}$. We also have $c_1 \not\models \mathrm{p} \multimap* \mathrm{r}$ since $c_2 \models \mathrm{p}$ and $c_1c_2 \not\models \mathrm{r}$. Thus, we have neither $c_1 \models \mathrm{p} \multimap* \mathrm{q}$, nor $c_1 \models \mathrm{p} \multimap* \mathrm{r}$, i.e., $c_1 \not\models (\mathrm{p} \multimap* \mathrm{q}) \vee (\mathrm{p} \multimap* \mathrm{r})$.

The previous results and examples illustrate the central role played by resource graphs for the countermodel generation. Thus, we can extract, from the resource graph, a countermodel in the related semantics, i.e., in the Kripke elementary semantics for BI without $\perp$ (our example) but also in the Grothendieck topological semantics for BI with $\perp$ [9]. As said before, for a resource logic like BI, we can relate a resource graph with a given complete semantics like Grothendieck topological semantics. But an interesting question arises: is it possible to deduce a new resource semantics from a deeper analysis of a given resource graph ? In the case of BI, the answer is yes. By considering resource graphs directly as countermodels, we have recently proposed a new semantics based on partially defined monoids (in which the composition is partial) that reflects the natural treatment of $\perp$ in a resource graph generated by our approach. The existence of such a semantics that generalizes the models of BI pointer logic [12] was an open question and it is clear that resource graph is the central notion allowing to give a positive answer [9].

3.4 A General Methodology for Resource Logics

Here we have extended the standard labelled tableau method [4] with (label) constraints related to resource semantics. It is based on two parallel processes: a syntactic decomposition of the formula to prove and a semantic construction of labels and constraints and then of a resource graph from which provability can be determined. Thus, we extend standard conditions (here closure of branches) with labels management and from this new semantic structure one can prove or disprove formulae and then generate proofs or countermodels.

A similar approach based on labels, constraints and resource graphs has been used in order to define a connection-based method for propositional BI [7]. It emphasizes that we propose a general methodology based on the construction of resource

graphs associated to standard proof-search methods such as tableaux or connections.

But a question arises: is this methodology restricted to BI logic ? A first answer comes from the fact that BI is conservative over intuitionistic logic (IL) and multiplicative intuitionistic linear logic (MILL) [15,16], our new proof-search methods can be restricted to both logics. It provides a new method for IL in which prefixes and unification [13] are replaced by constraints and resolution. Moreover, it is well adapted to the generation of countermodels. Further works will be devoted to deeper comparisons from the efficiency and countermodel construction perspectives. In addition, we obtain the first tableau (and connection) method for MILL that well illustrates the power of resource graphs for such resource logics. Knowing the relationships between connections and proof nets in linear logics [6], our results lead to a new algorithm for automated construction of MILL proof nets. Moreover, from the semantic point of view, the impact of these results will be analyzed and compared with previous proposals for models and countermodels analysis.

The previously mentioned resource logics are directly related to BI but we can also consider other resource logics like non-commutative logic (MNL), that is a conservative extension of both commutative (MLL) and non-commutative or cyclic (MCyLL) linear logic [1]. Specific labels, constraints and resource graphs can be defined in order to capture the interactions between commutative and non-commutative connectives (and thus its phase semantics) in this logic. They allow to define connection-based characterization of provability in MNL [10] and a related proof-search method. Finally, we aim to extend our results to proof-search and verification in separation logics [12,17] and spatial logics [2,3].

4 Countermodel Generation and Verification

Let us now consider first the proof-search approach and its implementation, the BILL system, that shows that this formula ϕ is not valid and thus generates a countermodel. Then, we complete it with a model-checking approach and its implementation, the CheckBI system, that verifies that the above resource graph is a countermodel of the formula ϕ.

4.1 BILL and Countermodel Generation

BILL is a prover for propositional BI (http://www.loria.fr/~dmery/BILL), written in CAML, that is able to decide whether a BI formula is provable or not and thus to build a countermodel under the form of a particular graph representation, that is a resource graph. In its current version, BILL can export the generated countermodels as GDL (Graph Description Language) files, GDL being a variant of XML adapted to graph descriptions. Thus, starting with a non-provable BI formula ϕ, a user can obtain, in a GDL file, a resource graph that is a countermodel for ϕ.

Let us describe the BILL prover and its main characteristics. It can be seen as an interpreter with simple commands. Its main command is `check <formula>`,

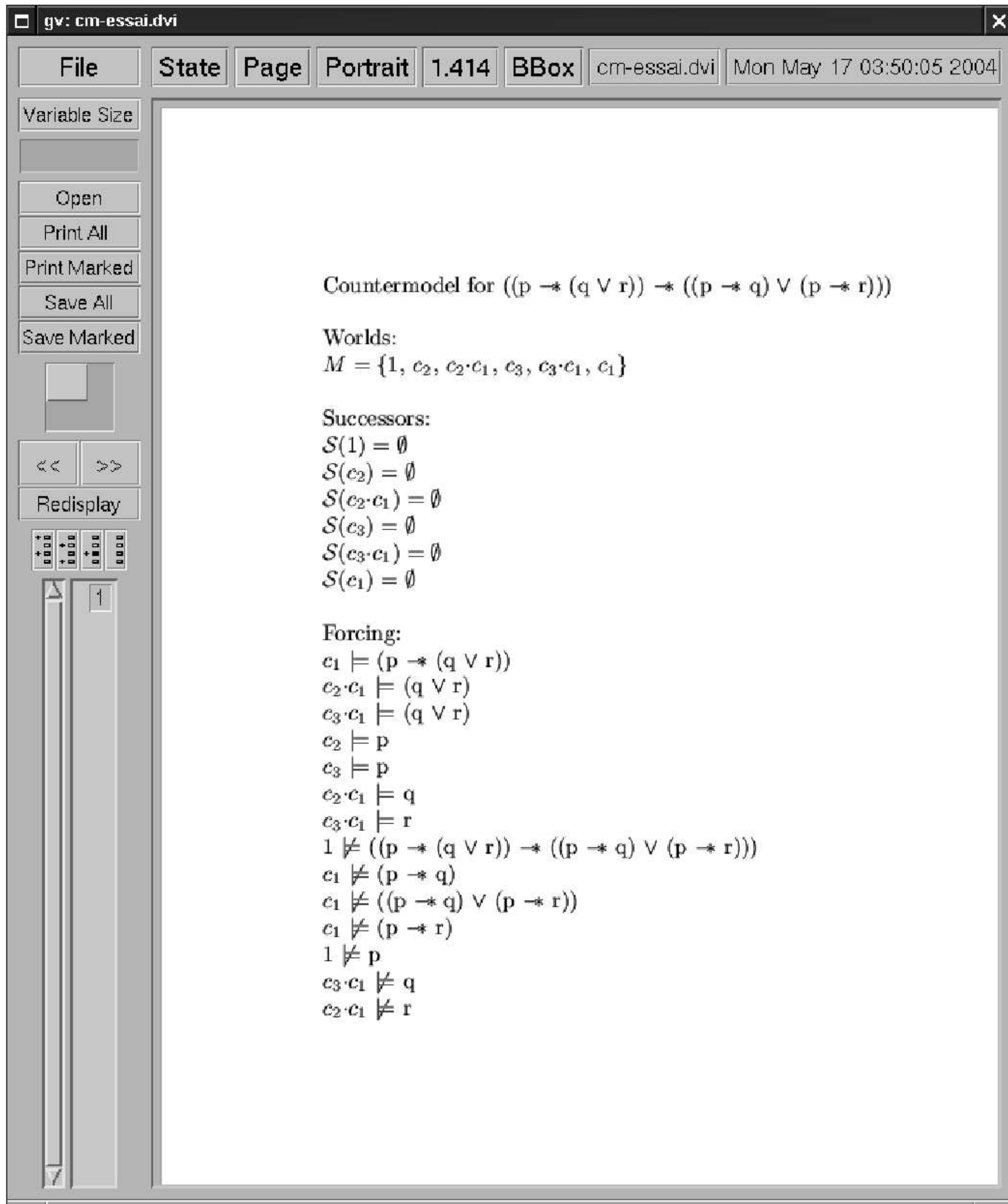
```
xterm
anubis ~/BILL-1.01b 53 % bill

Propositional BI
Version 1.01
Ctrl+D to quit

BILL> help
Variables: [A-Za-uw-z]
Connectives: -*, *, ->, ^, v
Precedence: ->, -* < v < ^, *
Commands:
  help: prints this page
  check <p>: decides proposition <p>
  depth <n>: sets search-depth to <n>
  stat: statistics about proof-search
  tex cm<file>: saves countermodel in a tex file
BILL>
BILL> check (p -* (q v r)) -* ((p -* q) v (p -* r))
false: ((p -* (q v r)) -* ((p -* q) v (p -* r)))
BILL>
BILL> stat
search-time: 0.000
search-calls: 20
BILL>
BILL> tex cm-formule
countermodel written in tmp/cm-formule.tex
BILL>
BILL> gdl cm-formule
countermodel written in tmp/cm-formule.gdl
BILL>
BILL>
anubis ~/BILL-1.01b 54 %
```

Figure 5. Example of a BILL session

that allows to decide the validity of `<formula>`. This `<formula>` parameter is written with the following syntax: the additive conjunctive unit is `1`; the additive disjunctive unit is `0`; the multiplicative unit is `I`; the additive connectives are: `^` (and), `v` (or), `->` (implication); the multiplicative connectives are `*` (star), `-*` (magicwand); the propositional variables are alphabetic characters, (`p`,`q`,...), except `I`, reserved for the multiplicative unit, and `v`, reserved for the additive disjunction. The command `stat` gives informations about proof search, for instance the time (in seconds) to decide the formula and the number of recursive calls of the proof-search loop. In case of non-validity, BILL can generate a countermodel from with commands `tex cm-<fichier>` and `gdl cm-<fichier>`. The first one generates a LaTeX file that describes and explain the semantic structure of the countermodel. The second one generates a countermodel as a resource graph with the GDL format. Such a graph can be exploited by graph manipulation tools such that `aiSee` ou `xvcg`.

Figure 5 illustrates the use of the BILL system. The `bill` command starts a BILL session and the `BILL>` prompt indicates that the user can write commands. With the `help` command we obtain a brief summary of the formula syntax and of the available commands. Then, with the command `check`, the user asks for validity of the given parameter that is the formula $(p \mathrel{-\!\!*} (q \vee r)) \mathrel{-\!\!*} ((p \mathrel{-\!\!*} q) \vee (p \mathrel{-\!\!*} r))$. Then, BILL reply that it is not valid and the `stat` command display the time and the number of recursive calls, necessary to conclude.

As the formula is not valid, we aim to generate a countermodel both as a LaTeX file



Countermodel for $((p -* (q \vee r)) -* ((p -* q) \vee (p -* r)))$

Worlds:
$M = \{1, c_2, c_2{\cdot}c_1, c_3, c_3{\cdot}c_1, c_1\}$

Successors:
$\mathcal{S}(1) = \emptyset$
$\mathcal{S}(c_2) = \emptyset$
$\mathcal{S}(c_2{\cdot}c_1) = \emptyset$
$\mathcal{S}(c_3) = \emptyset$
$\mathcal{S}(c_3{\cdot}c_1) = \emptyset$
$\mathcal{S}(c_1) = \emptyset$

Forcing:
$c_1 \models (p -* (q \vee r))$
$c_2{\cdot}c_1 \models (q \vee r)$
$c_3{\cdot}c_1 \models (q \vee r)$
$c_2 \models p$
$c_3 \models p$
$c_2{\cdot}c_1 \models q$
$c_3{\cdot}c_1 \models r$
$1 \not\models ((p -* (q \vee r)) -* ((p -* q) \vee (p -* r)))$
$c_1 \not\models (p -* q)$
$c_1 \not\models ((p -* q) \vee (p -* r))$
$c_1 \not\models (p -* r)$
$1 \not\models p$
$c_3{\cdot}c_1 \not\models q$
$c_2{\cdot}c_1 \not\models r$

Figure 6. A Countermodel in a Latex file

ans as a GDL resource graph. We do so using the commands `tex cm-formule` and `gdl cm-formule` that respectively provide the files `cm-formule.tex` and `cm-formule.gdl`.

Figure 6 shows what we obtain after the treatment of `cm-formule.tex` by LaTeX. The document contains the resource graph that is a countermodel with a list of worlds, and for each world, the lists of its immediate successors. We can observe that BILL has generated the resource graph described in section 3. Moreover the document provides the explanations of the non-validity of the formula by describing, for each subformula, what are the worlds for which it is verified and

falsified. Then, we recover the proof given in section 3.

4.2 CheckBI and Countermodel Verification

Another tool, called checkBI and written in Java, implements a dual functionality: from a resource graph encoded in GDL and a BI formula ϕ, to verify if the graph is a countermodel of ϕ. We aim to study how to combine model-checking and theorem proving appraoches in BI, i.e., to combine proof-search and countermodel-search, in order to have efficient decision procedures with countermodel generation.
Let us consider a resource graph $G(N, E)$, a valuation v and a formula ϕ, checkBI verifies if the given formula is true for the graph G with the valuation v, i.e., if we have $G, 1 \models_v A$. Thus we have the following command: The `<graph>` parameter corresponds to a file containing the GDL specification of the graph. The command verifies that this specification corresponds to a resource graph that is a graph that respects the conditions described in section 2. The `<formula>` parameter is a file that contains a BI formula written with the syntax used in BILL. Finally, the `<valuation>` parameter specifies the distribution of the formula atoms on the nodes of the graph under the form of a list of pairs (node, list of atomes forced by the node). This parameter is optional and if it is not present, it means that, for each node, there is no atom related to the node.
We illustrate the use of this tool with the countermodel generated by BILL for our example. If this resource graph correspond to a countermodel, the checkBI tool must verify that the graph falsifies the formula.
Figure 7 shows the contents of the `cm-formule.gdl` file, that is the specification in GDL of the resource graph. We observe that it is the graph described in section § 3. The `formule.txt` file contains the formula $(\mathrm{p} \mathrel{-\!\!*} (\mathrm{q} \vee \mathrm{r})) \mathrel{-\!\!*} ((\mathrm{p} \mathrel{-\!\!*} \mathrm{q}) \vee (\mathrm{p} \mathrel{-\!\!*} \mathrm{r}))$ and the `affec.aff` file describes the valuation $v(\mathrm{p}) = \{c_2, c_3\}$, $v(\mathrm{q}) = \{c_1 c_2\}$ and $v(\mathrm{r}) = \{c_1 c_3\}$ we previously mentioned. The result is obtained with the command `checkBI cm-formule.gdl formule.txt affec.aff` and is the one that is expected.

5 Conclusion and Perspectives

The aim here is to focus on a semantic structure, called resource graph, that is well adapted to design decision procedures for some propositional resource logics that generate countermodels. This structure, with an additional valuation attached to nodes, leads to a nice graphical representation of a countermodel and avoid representing it in some semantics, like for instance topological semantics, that are difficult to deal with. Such a resource graph arises from the definition of calculi including labels and label constraints that allow to capture the semantic interactions between connectives. This approach is well adapted to the treatment of "mixed" resource logics in which connectives of different kinds cohabit. As BI is used as an assertion logic for mutable data structures and is the logical kernel of so-called separation logics, our results and their implementation in the BILL system

```
xterm
anubis ~/BILL-1.01b 66 % cd tmp
anubis ~/BILL-1.01b/tmp 67 % more cm-formule.gdl
graph: {
label: "((p-*(qvr))-*((p-*q)v(p-*r)))"
node: { title: "1" label: "1" }
node: { title: "c2" label: "c2" }
node: { title: "c2c1" label: "c2c1" }
node: { title: "c3" label: "c3" }
node: { title: "c3c1" label: "c3c1" }
node: { title: "c1" label: "c1" }
}
anubis ~/BILL-1.01b/tmp 68 % more cm-formule.txt
(p -* (q v r)) -* ((p -* q) v (p -* r))
anubis ~/BILL-1.01b/tmp 69 %
anubis ~/BILL-1.01b/tmp 69 % more cm-formule.aff
(c2,[p]),(c3,[p]),(c1c2,[q]),(c1c3,[r])
anubis ~/BILL-1.01b/tmp 70 %
anubis ~/BILL-1.01b/tmp 70 % checkBI cm-formule

Parsing cm-formule.gdl ...
...done, building associated graph...
...done : graph for ((p -* (q v r))-*((p -* q) v (p -* r)))

Parsing cm-formule.txt ...
...done : formula ((p -* (q v r)) -* ((p -* q) v (p -* r)))

Parsing cm-formule.aff ...
...done, affecting resource-graph ...
...done : affectations

p is forced by [c2, c3]
q is forced by [c1c2]
r is forced by [c1c3]

Verifying closure under sublabels
...closure ok

Verifying partial compatibility
...compatibility ok

The formula ((p -* (q v r)) -* ((p -* q) v (p -* r))) is not forced by [1].

anubis ~/BILL-1.01b/tmp 71 %

VCG cm-formule.gdl
c3  c3c1  c1  c2  c2c1  1
```

Figure 7. An Example of GDL Countermodel Verification

are important to support the development of correct programs with pointers or the verification of properties for semi-structured data. In a practical perspective, the graphical representation of countermodels provides helpful and usable information for some failure analysis we will develop in further work. Another point to be studied is how to combine the theorem proving and model checking approaches in order

to improve the proof search process for our resource logics. It also means studying how to mutually use the BILL and checkBI tools for efficient proving and disproving and also how to extend them to deal for instance with pointer logic or some separation logics. The relationships between resource graphs and countermodels in new resource semantics will also be more deeply explored.

References

[1] M. Abrusci and P. Ruet. Non-commutative logic I : the multiplicative fragment. *Annals of Pure and Applied Logic*, 101:29–64, 2000.

[2] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). In *4th Int. Symposium on Theoretical Aspects of Computer Software, TACS 2001, LNCS 2215*, pages 1–37, Sendai, Japan, October 2001.

[3] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Int. Conference on Automata, Langages and Programming, ICALP'02, LNCS 2380*, pages 597–610, 2002.

[4] M. D'Agostino and D.M. Gabbay. A Generalization of Analytic Deduction via Labelled Deductive Systems. Part I: Basic substructural logics. *Journal of Automated Reasoning*, 13:243–281, 1994.

[5] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer Verlag, 1990.

[6] D. Galmiche. Connection Methods in Linear Logic and Proof nets Construction. *Theoretical Computer Science*, 232(1-2):231–272, 2000.

[7] D. Galmiche and D. Méry. Connection-based proof search in propositional BI logic. In *18th Int. Conference on Automated Deduction, CADE-18, LNAI 2392*, pages 111–128, 2002. Copenhagen, Danemark.

[8] D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI without bottom. *Journal of Logic and Computation*, 13(5):707–753, 2003.

[9] D. Galmiche, D. Méry, and D. Pym. Resource Tableaux (extended abstract). In *16th Int. Workshop on Computer Science Logic, CSL 2002, LNCS 2471*, pages 183–199, September 2002. Edinburgh, Scotland.

[10] D. Galmiche and J.M. Notin. Connection-based Proof Construction in Non-commutative Logic. In *1Oth Int. Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'03, LNCS 2850*, pages 422–436, September 2003. Almaty, Kazakhstan.

[11] J.Y. Girard. Linear Logic: its Syntax and Semantics. In J.Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 1–42. Cambridge University Press, 1995.

[12] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages, POPL 2001*, pages 14–26, London, UK, 2001.

[13] C. Kreitz and J. Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5(3):88–112, 1999.

[14] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th Int. Workshop on Computer Science Logic, CSL 2001, LNCS 2142*, pages 1–19, Paris, France, 2001.

[15] P.W. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[16] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[17] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Danemark, July 2002.

WoD 2004 Preliminary Version

Gödel-Dummett counter-models through matrix computation

Dominique Larchey-Wendling [1]

LORIA – CNRS
Vandœuvre-lès-Nancy, France

Abstract

We present a new method for deciding Gödel-Dummett logic. Starting from a formula, it proceeds in three steps. First build a conditional graph based on the decomposition tree of the formula. Then try to remove some cycles in this graph by instantiating these boolean conditions. In case this is possible, extract a counter-model from such an instance graph. Otherwise the initial formula is provable. We emphasize on cycle removal through matrix computation, boolean constraint solving and counter-model extraction.

Key words: Counter-models, conditional graphs and matrices.

1 Introduction

Gödel-Dummett logic LC is the intermediate logic (between classical logic and intuitionistic logic) characterized by linear Kripke models. It was introduced by Gödel in [10] and later axiomatized by Dummett in [6]. It is now one of the most studied intermediate logics for several reasons: among those, it is one of the *simplest "many-valued" logics,* whose semantics is captured by truth functions over the unit interval. It is one of the candidates (under the name "Gödel" logic) for use as a *fuzzy logic* [11]. With respect to decision procedures for intermediate logics, it witnesses some advantages of sequent calculi over hyper-sequent systems.

Proof-search in LC has benefited from the development of proof-search in intuitionistic logic IL with two important seeds: the *contraction-free calculus* of Dyckhoff [1,7,8] and the *hyper-sequent* calculus of Avron [2,14]. Two of the most recent contributions propose a similar approach based on a set of *local* and *strongly invertible* proof rules (for either sequent [13] or hyper-sequent [2] calculus,) and a semantic criterion to decide *irreducible (hyper)-sequents* and eventually build a counter-model.

[1] Email: larchey@loria.fr

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

We have recently proposed a combination of proof-search in sequent calculus and counter-model construction to provide a decision procedure for LC which is based on a new principle: we are able to gather all the useful information arising from all the proof-search branches into a semantic graph and then we use an efficient counter-model search algorithm based on cycle detection. We have reduced the decision problem in LC to a combination of boolean constraint solving and cycle detection. These results are presented in the upcoming paper [12], and the present paper comes as a complement to it. We will briefly recall the theoretical results, but we want to focus mainly on the description of the decision procedure with an emphasis on the counter-model generation algorithm.

Given a formula D of LC, the procedure proceeds in three steps at the end of which one obtains a counter-model (in case D is not provable.[2]) The first step which is described in full details in section 3 consists in building a particular *bi-colored graph* $\mathcal{G}_D$ based on the decomposition tree of D. The arrows of this graph may be indexed with *boolean conditions*. The second step consists in searching for an instantiation of the boolean conditions on arrows so that the *instance graph* has no remaining *r-cycle*.[3] This step is first described informally in section 3.3; then we provide a decision algorithm for this problem based on conditional matrix computation in section 4. For the third step, described in section 5, given a particular instance $\mathcal{G}_v$ with no r-cycle, we can extract a counter-model of D from this instance $\mathcal{G}_v$ by computing a *bi-height* for it.

2 The syntax and semantics of Gödel-Dummett logic

The set of propositional *formulae*, denoted Form is defined inductively, starting from a set of propositional *variables* denoted by Var and using the connectives $\wedge$, $\vee$ and $\supset$.[4] IL will denote the set of formulae that are provable in any intuitionistic propositional calculus (see [7]) and CL will denote the classically valid formulae. As usual an *intermediate propositional logic* [1] is a set of formulae $\mathcal{L}$ satisfying $\mathsf{IL} \subseteq \mathcal{L} \subseteq \mathsf{CL}$ and closed under the rule of modus ponens and under arbitrary substitution. LC is the smallest intermediate logic satisfying the axiom $(X \supset Y) \vee (Y \supset X)$.

On the semantic side, LC is characterized by linear Kripke models. In this paper, we will use the algebraic semantics characterization of LC [2] rather than Kripke semantics. The algebraic model is the set of natural numbers with its natural order $\leqslant$, augmented with a greatest element ∞. An interpretation of propositional variables $[\![\cdot]\!] : \mathsf{Var} \to \overline{\mathbb{N}}$ is inductively extended to formulae: the conjunction $\wedge$ is interpreted by the *minimum* function denoted

[2] As a decision procedure, it can also certify the validity of D in case it has a proof.

[3] A kind of cycle described later in section 6.

[4] We do not integrate the bottom $\perp$ constant. A specific treatment for $\perp$ is detailed in [13]. It can be easily integrated in the procedure described here.

$\wedge$, the disjunction $\vee$ by the *maximum* function $\vee$ and the implication $\supset$ by the operator $\twoheadrightarrow$ defined by $a \twoheadrightarrow b =$ if $a \leqslant b$ then ∞ else b. A formula D is *valid* for the interpretation $[\![\cdot]\!]$ if the equality $[\![D]\!] = \infty$ holds. This interpretation is complete for LC. A *counter-model* of a formula D is an interpretation $[\![\cdot]\!]$ such that $[\![D]\!] < \infty$.

3 A decision procedure for LC

In [12], we have described a procedure to decide the formulae of LC and to build a counter-model when a formula is not valid. The first step of this procedure is to build a graph with two kinds of arrows. Then the decision problem is reduced to the detection of particular cycles in this graph.

3.1 Conditional bi-colored graph construction

We introduce the exact notion of graph we use and then show how to build such a graph given a formula of LC.

Definition 3.1 A *bi-colored graph* is a (finite) directed graph with to kinds of arrows: *green* arrows denoted by $\rightarrow$ and *red* arrows denoted by $\Rightarrow$.

Definition 3.2 A *conditional bi-colored graph* is a bi-colored graph where arrows may be indexed with (propositional) boolean expressions.

We point out that we consider these boolean expressions up to classical equivalence, i.e. we consider them as representatives for boolean functions over atomic propositional variables. These variables can be instantiated by $\{0, 1\}$ with a valuation v and a boolean expression e gets a value $e_v \in \{0, 1\}$ computed in the obvious way. We thus obtain an instance graph: an arrow indexed with a boolean expression e belongs to this instance if and only if $e_v = 1$. The case of an unconditional (i.e. not indexed) arrow can be treated by considering that it has an implicit boolean conditional which is a tautology (and then always values 1) and non-existing arrows have an implicit boolean condition that always values 0.

Definition 3.3 Given a conditional bi-colored graph $\mathcal{G}$ and a valuation v of boolean variables in $\{0, 1\}$, we define the *instance graph* $\mathcal{G}_v$ as the bi-colored graph that one obtains when one evaluates boolean expressions indexing arrows and keeping exactly those whose valuation equals 1.

Given a LC formula D, we build a conditional bi-colored graph $\mathcal{G}_D$ by the following process. First, the nodes of $\mathcal{G}_D$ are obtained by considering the set of nodes of the decomposition tree of D, or equivalently, the set of occurrences of subformulae.

- If F is an occurrence of a subformula of D, we denote by $\mathcal{X}_F$ the corresponding node. Nodes are *signed* starting from $-$ at the root D^- and propagating

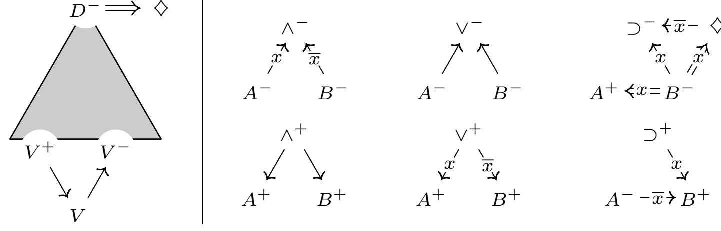


Fig. 1. Counter-model search system for LC

signs as usual.[5] We may write $\mathcal{X}_F^+$ or $\mathcal{X}_F^-$ to emphasize the sign.

- To this set of nodes, we add a node denoted V for each propositional variable V occurring in D. Hence, multiple occurrences of V only generate one node V but generate several $\mathcal{X}_V^+$ or $\mathcal{X}_V^-$ nodes.
- We add one new node denoted by $\Diamond$.

Then, the edges of $\mathcal{G}_D$ are obtained as follows: we describe the set of green and red arrows linking those nodes together and the boolean expressions indexing those arrows. We begin by unconditional arrows (i.e. arrows implicitly indexed with the tautology 1) introduced independently of the internal structure of the formula D:

- We add the (unconditional) red arrow $\mathcal{X}_D^- \Rightarrow \Diamond$ from the root node to $\Diamond$.
- For a negative occurrence V of a variable, we add the green arrow $V \rightarrow \mathcal{X}_V^-$.
- For a positive occurrence V of a variable, we add the green arrow $\mathcal{X}_V^+ \rightarrow V$.

These three rules are summarized on the left part of figure 1. Now we consider arrow introduction rules for internal nodes. First, the unconditional cases:

- For a positive occurrence $C \equiv A \wedge B$ of a subformula, we add the two following green arrows $\mathcal{X}_C^+ \rightarrow \mathcal{X}_A^+$ and $\mathcal{X}_C^+ \rightarrow \mathcal{X}_B^+$.
- For a negative occurrence $C \equiv A \vee B$ of a subformula, we add the two following green arrows $\mathcal{X}_A^- \rightarrow \mathcal{X}_C^-$ and $\mathcal{X}_B^- \rightarrow \mathcal{X}_C^-$.

We continue with conditional arrows. These arrows are indexed with *selectors*, i.e. boolean expressions of the form x or $\overline{x}$ where x is a boolean propositional variable. For each occurrence of subformula, we introduce a *new* boolean variable.[6]

- For a negative occurrence $C \equiv A \wedge B$ of a subformula, given a new boolean variable x, we introduce the two conditional green arrows $\mathcal{X}_A^- \rightarrow_x \mathcal{X}_C^-$ and $\mathcal{X}_B^- \rightarrow_{\overline{x}} \mathcal{X}_C^-$.
- For a positive occurrence $C \equiv A \vee B$ of a subformula, given a new boolean variable x, we introduce the two conditional green arrows $\mathcal{X}_C^+ \rightarrow_x \mathcal{X}_A^+$ and $\mathcal{X}_C^+ \rightarrow_{\overline{x}} \mathcal{X}_B^+$.

[5] The connectives $\wedge$ and $\vee$ preserve signs and $\supset$ preserves the sign on the right subformula and inverses the sign of the left subformula.

[6] Indexing these variables with the subformula occurrence is a way to ensure uniqueness.

- For a negative occurrence $C \equiv A \supset B$ of a subformula, given a new boolean variable x, we introduce the two following green arrows $\mathcal{X}_B^- \rightarrow_x \mathcal{X}_C^-$ and $\diamondsuit \rightarrow_{\overline{x}} \mathcal{X}_C^-$ and the two following red arrows $\mathcal{X}_B^- \Rightarrow_x \mathcal{X}_A^+$ and $\mathcal{X}_B^- \Rightarrow_x \diamondsuit$.
- For a positive occurrence $C \equiv A \supset B$ of a subformula, given a new boolean variable x, we introduce the two following green arrows $\mathcal{X}_C^+ \rightarrow_x \mathcal{X}_B^+$ and $\mathcal{X}_A^- \rightarrow_{\overline{x}} \mathcal{X}_B^+$.

All the rules introducing (un)conditional arrows for internal nodes (corresponding to subformulae of D that are not atomic) are summarized on the right part of figure 1.

Given this construction procedure, it should be clear that the construction of the graph $\mathcal{G}_D$ from a formula D take linear time as at most four arrows are introduced for each instance of a subformula of D. The validity of D is related to the existence of some particular cycles in instances of $\mathcal{G}_D$.

Definition 3.4 A *r-cycle* in a bi-colored graph is a cycle composed of either green ($\rightarrow$) or red ($\Rightarrow$) arrows, containing at least one red arrow. Equivalently, it is a chain of the form $l\,(\rightarrow + \Rightarrow)^\star \Rightarrow l$.

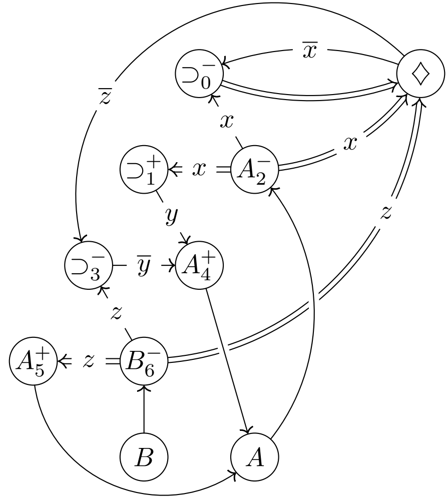
Theorem 3.5 *Let D be a formula of LC and $\mathcal{G}$ be its associated conditional bi-colored graph, built from the process previously described. Then D is provable in LC if and only if every instance graph $\mathcal{G}_v$ of $\mathcal{G}$ contains at least one r-cycle.*

This result is proved in [12]. So in order to refute D, we have to find an instance graph $\mathcal{G}_v$ which does not contain any r-cycle. Let us proceed with an example.

3.2 Graph construction example

We consider the case of the classically valid Peirce's formula $((A \supset B) \supset A) \supset A$. It is not provable in any intermediate logic but classical logic, so in particular, it should have a counter-model in LC.

We index this formula as follows: $\big((A_5^+ \supset_3^- B_6^-) \supset_1^+ A_4^+\big) \supset_0^- A_2^-$. We construct its associated conditional graph:



- We add the arrow $\supset_0^- \Rightarrow \diamondsuit$.
- We have two variables A and B for 4 occurrences, so we add $A \rightarrow A_2^-$, $A_4^+ \rightarrow A$, $A_5^+ \rightarrow A$ and $B \rightarrow B_6^-$.
- For the internal node $\supset_0^-$, we choose a new boolean variable x and add the four conditional arrows $A_2^- \rightarrow_x \supset_0^-$, $\diamondsuit \rightarrow_{\overline{x}} \supset_0^-$, $A_2^- \Rightarrow_x \supset_1^+$ and $A_2^- \Rightarrow_x \diamondsuit$.

- For the internal node $\supset_1^+$, we choose a new boolean variable y and add the two conditional arrows $\supset_1^+ \rightarrow_y A_4^+$ and $\supset_3^- \rightarrow_{\overline{y}} A_4^+$.
- For the last internal node $\supset_3^-$, we choose a new boolean variable z and add the four conditional arrows $B_6^- \rightarrow_z \supset_3^-$, $\Diamond \rightarrow_{\overline{z}} \supset_3^-$, $B_6^- \Rightarrow_z A_5^+$ and $B_6^- \Rightarrow_z \Diamond$.

3.3 Naive elimination of r-cycles

We now have to find a valuation v_x, v_y and v_z in $\{0,1\}$ such that the corresponding instance graph has no r-cycle. For this we identify all the r-cycles and we try to find a valuation that simultaneously breaks each of the r-cycles. We only have to consider r-cycles that do not repeat nodes because any r-cycle contains at least one that does not repeat nodes. We find four such r-cycles:

$$\begin{aligned}
&\supset_0^- \Rightarrow \Diamond \rightarrow_{\overline{x}} \supset_0^- \\
&\supset_0^- \Rightarrow \Diamond \rightarrow_{\overline{z}} \supset_3^- \rightarrow_{\overline{y}} A_4^+ \rightarrow A \rightarrow A_2^- \rightarrow_x \supset_0^- \\
&A_2^- \Rightarrow_x \Diamond \rightarrow_{\overline{z}} \supset_3^- \rightarrow_{\overline{y}} A_4^+ \rightarrow A \rightarrow A_2^- \\
&A_2^- \Rightarrow_x \supset_1^+ \rightarrow_y A_4^+ \rightarrow A \rightarrow A_2^-
\end{aligned}$$

The first r-cycle is broken if and only if the condition $\overline{x} = 0$ is satisfied, which is equivalent to satisfy x. The second r-cycle is broken if and only if the condition $z + y + \overline{x}$ is satisfied.[7] The third r-cycle is broken just in case $\overline{x} + z + y$ is satisfied and the last r-cycle is broken when $\overline{x} + \overline{y}$ is satisfied.

In order to break these four r-cycles in one valuation, we look for a valuation v which satisfies $x \cdot (\overline{x} + y + z) \cdot (\overline{x} + y + z) \cdot (\overline{x} + \overline{y})$. This gives us a unique solution: $v_x = 1, v_y = 0, v_z = 1$. Then, the reader could verify that the instance graph $\mathcal{G}_v$ obtained from this valuation has no r-cycle. See section 5 for a representation of this graph and the associated counter-model of the Peirce's formula.

The naive procedure we have described for computing a valuation with no r-cycles consists of searching all the possible r-cycles (without repeating nodes) and solving a boolean constraint system associated with these cycles. Unfortunately, such a procedure would be highly inefficient because there might be exponentially many r-cycles for a given formula. This problem has also been addressed in [12]. In the next section, we give a description of one possible solution to the elimination of r-cycles.

4 Removing r-cycles in conditional bi-colored graphs

In section 3.1, we have introduced the notion of conditional bi-colored graph. A natural way to represent a directed graph is by considering the matrix of the underlying *incidence* relation. Usually, these matrices take there values

[7] We denote by $+$ the boolean disjunction and by $\cdot$ the boolean conjunction.

$\to$	0	1	2	3	4	5	6	A	B	$\diamondsuit$
0										
1					y					
2	x									
3					$\overline{y}$					
4								1		
5								1		
6				z						
A			1							
B							1			
$\diamondsuit$	$\overline{x}$			$\overline{z}$						

$\Rightarrow$	0	1	2	3	4	5	6	A	B	$\diamondsuit$
0										1
1										
2		x								x
3										
4										
5										
6						z				z
A										
B										
$\diamondsuit$										

Fig. 2. The conditional matrices for Peirce's formula.

in the boolean algebra $\{0, 1\}$ and a 1 in the cell (i, j) means that there is an arrow from the node i to the node j.

4.1 Conditional matrices

To represent conditional bi-colored graphs, we use *conditional matrices*: the cells of these matrices take their values from the set of boolean functions. These functions are represented by boolean expressions built from the boolean selectors introduced during the conditional graph construction.

Definition 4.1 A *conditional matrix* on set $\mathcal{S}$ of size k is a $k \times k$-array with values in the free boolean algebra over the set of selectors.

There are two incidence relations for a bi-colored graph corresponding to the green ($\to$) and red ($\Rightarrow$) arrows. So a conditional bi-colored graph is represented by a pair of conditional matrices. We use the same denotation for the (conditional) incidence relation and for its corresponding matrix. So $\mathcal{G}_D$ is represented by a pair $(\to, \Rightarrow)$ of conditional matrices. Figure 2 presents the two matrices corresponding to the graph $\mathcal{G}_D$ when D is the Peirce formula of section 3.2. We only write the cells whose values are different from 0: the matrices are *sparse* because the number of non-zero cells is linear whereas the total number of cell is quadratic.

4.2 R-cycle removal as a trace computation

The boolean operator of conjunction (or multiplication) $\cdot$ and disjunction (or sum) $+$ extend naturally to conditional matrices. So we may consider the sum $\to + \Rightarrow$, product $\to \cdot \Rightarrow$ of conditional matrices and the reflexive and transitive closure $\to^\star = \sum_{i \geqslant 0} \to^i$. We also introduce the trace of a matrix: $\mathrm{tr}(M) = \sum_x M_{x,x}$. When boolean selectors are instantiated inside a conditional matrix, we get a matrix with values in $\{0, 1\}$ which is the incidence ma-

trix of the corresponding instance graph. Moreover, instantiation commutes with algebraic operations on matrices. This leads to the following result:

Theorem 4.2 *Let $\mathcal{G} = (\rightarrow, \Rightarrow)$ be a conditional bi-colored graph represented by a pair of conditional matrices. There exists a r-cycle in every instance $\mathcal{G}_v$ of $\mathcal{G}$ if and only if $tr\big((\rightarrow + \Rightarrow)^\star \Rightarrow\big) = 1$ holds.*

Moreover, when the boolean function $\mathrm{tr}\big((\rightarrow + \Rightarrow)^\star \Rightarrow\big)$ is not a tautology, there exists a valuation v on selectors in $\{0, 1\}$ such that this trace has value 0: $\mathrm{tr}\big((\rightarrow_v + \Rightarrow_v)^\star \Rightarrow_v\big) = 0$. Then, the corresponding instance graph $\mathcal{G}_v$ has no r-cycle.

Now, the problem is to compute this trace efficiently. Let us fix a size $k > 0$ of matrices. Let I denote the identity $k \times k$ matrix ($I_{x,x} = 1$ and $I_{x,y} = 0$ otherwise.) Let M be any conditional $k \times k$ matrix. Then $M^\star = (I + M)^k$ (because any path of size $k + 1$ contains a sub-path of size k.) Moreover, as $I \leqslant I + M$ (cell-wise), $I, I + M, (I + M)^2 \ldots$ is a (point-wise) increasing sequence of conditional matrices which stabilizes in at most k steps.

In order to evaluate $(\rightarrow + \Rightarrow)^\star \Rightarrow$, we compute $\alpha = I + \rightarrow + \Rightarrow$ and $\beta = \Rightarrow$ and then the increasing sequence $\beta, \alpha\beta, \alpha^2\beta, \alpha^3\beta, \ldots$ until it stabilizes to $\alpha^\star\beta$. This can be done column by column on β. Let β_i denote the column i of β then $\alpha^\star\beta_i$ is the column i of $\alpha^\star\beta$. The computation of the column 1 for Peirce's formula is the following:

α	0	1	2	3	4	5	6	A	B	$\diamondsuit$	β_1	$\alpha\beta_1$	$\alpha^2\beta_1$	$\alpha^3\beta_1$	$\alpha^4\beta_1$	$\alpha^\star\beta_1$
0	1									1						$x.\overline{y}.\overline{z}$
1		1			y									$x.y$	$x.y$	$x.y$
2	x	x	1							x	x	x	x	x	x	x
3				1	$\overline{y}$									$x.\overline{y}$	$x.\overline{y}$	$x.\overline{y}$
4					1			1					x	x	x	x
5						1		1					x	x	x	x
6				z		z	1			z				$x.z$	$x.z$	$x.z$
A			1					1				x	x	x	x	x
B							1		1						$x.z$	$x.z$
$\diamondsuit$	$\overline{x}$			$\overline{z}$						1					$x.\overline{y}.\overline{z}$	$x.\overline{y}.\overline{z}$

Most of the columns of β contain only 0 in which case there is no need for computation: the fixpoint is this zero column. In the Peirce example, only column 1, 5 and $\diamondsuit$ contain values which are different from zero.

When evaluating the trace, it is possible to share computation between the columns of β. Let T be the column matrix composed of 1 on each cells. We consider the following sequence: $t_0 = 0$ and $t_i = \big[\alpha^\star(t_{i-1}T + \beta_i)\big]_i$ for $i = 1, \ldots, k$. Then $t_k = \mathrm{tr}(\alpha^\star\beta)$. For example, in the case of Peirce's formula, we get $t_0 = 0$ and then $t_1 = x.y$. Columns β_2, β_3 and β_4 are empty (i.e. contain only 0) so $t_2 = t_3 = t_4 = t_1 = x.y$. Then $t_5 = \big[\alpha^\star(x.y.T + \beta_5)\big]_5$ and we obtain

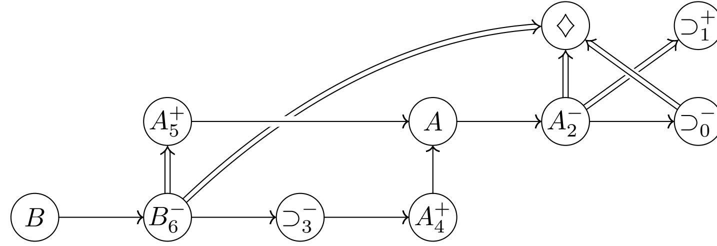
$t_5 = x.y$. Then columns β_6, β_A and β_B are empty and $t_6 = t_A = t_B = t_5 = x.y$. Finally we compute $t_\diamondsuit = \left[\alpha^\star(x.y.T + \beta_\diamondsuit)\right]_\diamondsuit$. Let $\gamma = x.y.T + \beta_\diamondsuit$:

α	0	1	2	3	4	5	6	A	B	$\diamondsuit$	γ	$\alpha\gamma$	$\alpha^2\gamma$	$\alpha^3\gamma$	$\alpha^\star\gamma$
0	1									1	1	1	1	1	1
1		1			y						$x.y$	$x.y$	$x.y$	$x.y$	$x.y$
2	x	x	1							x	x	x	x	x	x
3				1	$\overline{y}$						$x.y$	$x.y$	$x.y$	x	x
4					1			1			$x.y$	$x.y$	x	x	x
5						1		1			$x.y$	$x.y$	x	x	x
6				z		z	1			z	$x.y+z$	$x.y+z$	$x.y+z$	$x.y+z$	$x.y+z$
A			1					1			$x.y$	x	x	x	x
B							1		1		$x.y$	$x.y+z$	$x.y+z$	$x.y+z$	$x.y+z$
$\diamondsuit$	$\overline{x}$			$\overline{z}$						1	$x.y$	$\overline{x}+x.y$	$\overline{x}+x.y$	$\overline{x}+x.y$	$\overline{x}+x.y+x.\overline{z}$

and we obtain $t_\diamondsuit = \overline{x} + x.y + x.\overline{z}$. This the trace of $\alpha^\star\beta = (\rightarrow + \Rightarrow)^\star\Rightarrow$ and it is not a tautology. The only valuation that falsifies this trace is $v_x = 1, v_y = 0, v_z = 1$. This is of course the same valuation we obtained by hand (by looking up for r-cycles) in section 3.3.

5 Counter-model extraction

Now we explain how to extract a counter-model from the corresponding instance bi-colored graph $\mathcal{G}_v$. The reader can easily check that it can be represented by:



In this graph, red arrows are always strictly climbing up and green arrows never go down so no r-cycle could exist. The counter-model is very easy to compute: give the variable A and B their height in this graph. So $[\![A]\!] = 1$ and $[\![B]\!] = 0$ is a counter-model to the Peirce's formula which can be checked by $[\![((A \supset B) \supset A) \supset A]\!] = ((1 \twoheadrightarrow 0) \twoheadrightarrow 1) \twoheadrightarrow 1 = (0 \twoheadrightarrow 1) \twoheadrightarrow 1 = \infty \twoheadrightarrow 1 = 1 < \infty$

Now we explain how to extract a counter-model out of an instance graph lacking r-cycles in the general case. We give a characterization of the lack of r-cycles based on the notion of *bi-height*:

Definition 5.1 Let $\mathcal{G}$ be a bi-colored graph. A *bi-height* is a function $h : \mathcal{G} \to \mathbb{N}$ such that for any $x, y \in \mathcal{G}$, if $x \rightarrow y \in \mathcal{G}$ then $h(x) \leqslant h(y)$ and if $x \Rightarrow y \in \mathcal{G}$ then $h(x) < h(y)$.

It is clear that the preceding graph has a bi-height given by $h(B) = h(B_6^-) = h(\supset_3^-) = h(A_4^+) = 0$, $h(A_5^+) = h(A) = h(A_2^-) = h(\supset_0^+) = 1$ and $h(\diamondsuit) = h(\supset_1^+) = 2$. In [12], you will find a constructive proof of the following result which states the existence of a bi-height whenever no r-cycle exist:

Theorem 5.2 *Let D be a formula of* LC*, $\mathcal{G}$ the corresponding conditional bi-colored graph and v a valuation such that the instance graph $\mathcal{G}_v$ does not contains any r-cycle. Then it is possible to compute a bi-height h for $\mathcal{G}_v$ in linear time.*[8] *Moreover, if we define $[\![\cdot]\!] : \mathsf{Var} \to \overline{\mathbb{N}}$ by $[\![V]\!] = h(V)$ for V variable of D then $[\![\cdot]\!]$ is a counter-model of D, i.e. $[\![D]\!] < \infty$.*

6 Implementation remarks and conclusion

The procedure described throughout this paper has been implemented completely in the Objective Caml language and is accessible at

http://www.loria.fr/~larchey/LC

The reader interested in the proofs of the results presented here can also find them there.

For the prototype implementation, we have chosen to represent conditional matrices by *sparse arrays*. The boolean functions which compose them are represented by the nodes of a shared BDD [5] for efficient boolean computations and extraction of boolean counter-models. The algorithm for the computation of bi-heights is a slightly modified version of a depth first search procedure.

In further work, we will deeper investigate the relationships between the notion of r-cycle and the G-cycles of [3] and analyze if our conditional graphs also fit in the hyper-sequent setting. We will also investigate the relationships between our parallel counter-model search and other approaches based for example on parallel dialogue games [4,9].

References

[1] Alessendro Avellone, Mauro Ferrari, and Pierangelo Miglioli. Duplication-Free Tableau Calculi and Related Cut-Free Sequent Calculi for the Interpolable Propositional Intermediate Logics. *Logic Journal of the IGPL*, 7(4):447–480, 1999.

[2] Arnon Avron. A Tableau System for Gödel-Dummett Logic Based on a Hypersequent Calculus. In *TABLEAUX 2000*, volume 1847 of *LNAI*, pages 98–111, 2000.

[3] Arnon Avron and Beata Konikowska. Decomposition Proof Systems for Gödel-Dummett Logics. *Studia Logica*, 69(2):197–219, 2001.

[8] Linearity is measured with respect to either the size of D or the number of nodes and arrows of $\mathcal{G}_v$.

[4] Matthias Baaz and Christian Fermüller. Analytic Calculi for Projective Logics. In *TABLEAUX'99*, volume 1617 of *LNCS*, pages 36–50, 1999.

[5] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] Michael Dummett. A Propositional Calculus with a Denumerable matrix. *Journal of Symbolic Logic*, 24:96–107, 1959.

[7] Roy Dyckhoff. Contraction-free Sequent Calculi for Intuitionistic Logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.

[8] Roy Dyckhoff. A Deterministic Terminating Sequent Calculus for Gödel-Dummett logic. *Logical Journal of the IGPL*, 7:319–326, 1999.

[9] Christian Fermüller. Parallel Dialogue Games and Hypersequents for Intermediate Logics. In *TABLEAUX 2003*, volume 2796 of *LNAI*, pages 48–64, 2003.

[10] Kurt Gödel. Zum intuitionistischen Aussagenkalkül. In *Anzeiger Akademie des Wissenschaften Wien*, volume 69, pages 65–66. 1932.

[11] Petr Hajek. *Metamathematics of Fuzzy Logic*. Kluwer Academic Publishers, 1998.

[12] Dominique Larchey-Wendling. Counter-model search in Gödel-Dummett logics. To be published in the proceedings of IJCAR 2004.

[13] Dominique Larchey-Wendling. Combining Proof-Search and Counter-Model Construction for Deciding Gödel-Dummett Logic. In *CADE-18*, volume 2392 of *LNAI*, pages 94–110, 2002.

[14] George Metcalfe, Nicolas Olivetti, and Dov Gabbay. Goal-Directed Calculi for Gödel-Dummett Logics. In *CSL*, volume 2803 of *LNCS*, pages 413–426, 2003.

View publication stats