New Techniques for Handling Quantifiers in Boolean and First-Order Logic

PETER BACKEMAN
New Techniques for Handling Quantifiers in Boolean and First-Order Logic

Peter Backeman
peter.backeman@it.uu.se

December 2016

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

© Peter Backeman 2016
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden
Abstract

The automation of reasoning has been an aim of research for a long time. Already in 17th century, the famous mathematician Leibniz invented a mechanical calculator capable of performing all four basic arithmetic operators. Although automatic reasoning can be done in different fields, many of the procedures for automated reasoning handles formulas of first-order logic. Examples of use cases includes hardware verification, program analysis and knowledge representation.

One of the fundamental challenges in first-order logic is handling quantifiers and the equality predicate. On the one hand, SMT-solvers (Satisfiability Modulo Theories) are quite efficient at dealing with theory reasoning, on the other hand they have limited support for complete and efficient reasoning with quantifiers. Sequent, tableau and resolution calculi are methods which are used to construct proofs for first-order formulas, and can use more efficient techniques to handle quantifiers. Unfortunately, in contrast to SMT, handling theories is more difficult.

In this thesis we investigate methods to handle quantifiers by restricting search spaces to finite domains, explorable in a systematic manner. We present this method in two different contexts. First we introduce a function synthesis based on template-based quantifier elimination, applied to gene interaction computation. The function synthesis is capable of generating smaller representations of solutions than previous solvers, and by restricting the constructed functions to certain forms we can produce formulas which can more easily be interpreted by a biologist. Secondly we introduce the concept of Bounded Rigid $E$-Unification (BREU), a finite form of unification that can be used to define a complete and sound sequent calculus for first-order logic with equality. We show how to solve this bounded form of unification in an efficient manner, yielding a first-order theorem prover utilizing BREU that is competitive with other state-of-the-art tableau theorem provers.
Acknowledgments

I wish to thank my supervisors Philipp Rümmer and Christoph Wintersteiger for their guidance and endless provision of feedback. A big thank you (and an apology) is due to Aleksandar Zeljić, for all the times I have interrupted him to sort out the notions of logic. I also wish to thank all the people in my research group who makes for great company. Finally, I want to thank Linette Söderberg, who has always been supportive, even when I have been burning the midnight oil.
List of Papers

This thesis is based on the following papers

I  Peter Backeman, Philipp Rümmer *Theorem Proving with Bounded Rigid E-Unification* 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015 Springer-Verlag, LNCS 9195, pages 572-587

II Peter Backeman, Philipp Rümmer *Efficient Algorithms for Bounded Rigid E-unification* Automated Reasoning with Analytic Tableaux and Related Methods, 24th International Conference, TABLEAUX 2015 Springer-Verlag, LNCS 9323, pages 70-85


I am the main author of all the paper included in this thesis.
# Contents

1 Introduction 3  
1.1 Running Example ........................................ 5  
1.2 Research Questions ...................................... 7  

2 Propositional Logic 9  
2.1 Syntax ..................................................... 9  
2.2 Semantics .................................................. 10  
2.3 The SAT Problem .......................................... 12  
2.4 Sequent Calculus .......................................... 13  
2.5 DPLL ......................................................... 15  
2.6 Quantified Boolean Formulas ............................ 15  

3 Using QBFs to Reason about Boolean Networks 17  
3.1 Abstract Boolean Networks ................................ 19  
3.2 Contributions .............................................. 22  

4 First-Order Logic 23  
4.1 Syntax ..................................................... 24  
4.2 Semantics .................................................. 25  
4.3 Theories .................................................... 32  

5 Automatic Reasoning 35  
5.1 Automatic Theorem Proving ............................... 35  
5.2 Automatic Sequent Reasoning ............................. 36  
5.3 Equality in Sequent Tableaux ............................. 38  
5.4 Summary of Paper I ....................................... 39  

6 Unification 41  
6.1 Syntactical Unification ................................... 41  
6.2 Semantical Unification .................................... 43  
6.3 Summary of Paper II ...................................... 46
7 Conclusions 47
  7.1 Future Work 47

Paper I 54
Paper II 72
Paper III 95
Chapter 1

Introduction

“It is unworthy of excellent men to lose hours like slaves in the labour of calculation which could safely be relegated to anyone else if machines were used.”

Leibniz

The field of logic deals with reasoning about reasoning and how to formalize it. As with many scientific fields, already the thinkers of ancient Greece made the first steps. The famous syllogisms of Aristotle are an example of a template-based formalization of deduction. The automation of reasoning has also been an aim of research for a long time. Already in 17th century, the famous mathematician Leibniz invented a mechanical calculator capable of performing all four basic arithmetic operators.

Today we possess computers, capable of far more than the Leibniz machine, but the aim is nevertheless the same: freeing ourselves from the manual labor of doing reasoning that can be done mechanically, leaving us humans to deal with more “interesting things”. Since an universal method does not exist, there is a need for many different approaches to automated reasoning. Some of the well-known fields dealing with automatic reasoning are:

- **Satisfiability** (SAT) is the canonical NP-complete problem [13] of finding assignments of Boolean variables that satisfy (sets of) Boolean formula(s). Practical applications involve encoding of problems into Boolean formulas w.r.t. certain theories (e.g., integer arithmetic, the theory of arrays), and **SATisfiability Modulo Theories** (SMT) is the extension of the SAT problem to include theory reasoning over various theories (see [6]). There exists many sophisticated solvers based
on this paradigm (e.g., [17]).

- **Constraint Programming** (CP) also search for a satisfying assignment, but instead of working with Boolean assignments, variables over richer domains are used together with arbitrary constraints (see e.g., [2] for an introduction). Much of the complexity lies in designing propagators, special kinds of functions that infer restrictions on domains based on specific constraints from some domain. One example is the *distinct*-propagator which ensures that the values of a set of variables are distinct [46]. This is in contrast to SMT, where the theory-solver is designed to handle a whole theory.

- **Automated Theorem-Proving** (ATP) is the automatic search for proofs (for example, in first-order logic). The goal is to prove that a formula (the conclusion) follows from a set of formulas (the premises) by finding a suitable proof. Many different proof systems can be used for the underlying search (e.g., resolution, sequent calculus).

All of these technologies (and many more) are used in a variety of areas to perform automatized reasoning. The increasing performance of computers and the development of more efficient procedures make larger problem instances tractable. Examples of use cases are:

- **Hardware Verification** is a process of verifying that a digital circuit design computes the intended function. Since circuits are inherently Boolean, SAT has been used as a tool for a long time in hardware design [28].

- **Program Analysis** is used to prove properties, e.g., termination, of software programs (see, e.g. [23]). By automatically catching out-of-bounds errors or proving correctness of a function, certain bugs can be caught even without testing. Furthermore, using program analysis we can *guarantee* bug-free programs since, as famously put by Dijkstra, “Testing shows the presence, not the absence of bugs” [43].

- **Knowledge Representation** tries to accurately formalize knowledge to facilitate drawing valid conclusions and perform complex reasoning over a database of known facts. This can be done using special kinds of logic, e.g. Description Logic (see e.g., [3]). OpenGALEN [44] is a project utilizing Description Logics to formalize medical knowledge.

- **Biology** can also benefit from this kind of technique. In computational biology, automatic reasoning can help discover new knowledge. For example, a complete model of the organism Mycoplasma genitalium
[31] has been simulated, which allows for performing experiments and analysis which are impossible on an actual cell. In Section 3 we present another case where SMT-solvers have been used to assist biological research in gene regulatory networks.

1.1 Running Example

To illustrate some of the methods in this thesis we will use a running example (from [41]):

Example 1. “Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadbury Mansion, and are the only people who live therein. A killer always hates his victim, and is never richer than his victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone Aunt Agatha hates. No one hates everyone. Agatha is not the butler. Who killed Agatha?”

Example 1 illustrates many of the challenges in automatic theorem proving. Among these we find the formalization of natural language into some logic (for example into first-order logic), and also the automatic generation of a proof which shows that the conclusion follows from the premises. Example 1 can also be seen as an illustration of a problem of knowledge representation, where the clues are facts given and the problem is to deduce who is the killer. In Table 1.1 we see a formalization of the premises (taken from the TPTP-collection [58]). In the formulas we have constants such as Butler and Agatha identifying the people; predicates such as livesAtDreadbury(x) (which states that x lives at Dreadbury) and hates(Charles, x) (stating that Charles hates x), and Boolean connectives such as ∧ and ¬ connecting sub-formulas, for example killed(x, y) → ¬richer(x, y) (stating that if x killed y, then x is not richer than y).

This set of statements can be seen as a set of axioms (i.e. statements assumed to be true) and we wish to find out who is the killer (i.e. formalize “Who killed Agatha?”). We could make a guess and add the formula ¬killed(Agatha, Agatha) and see if the resulting set is contradictory (which would mean that killed(Agatha, Agatha) must be a consequence of the axioms). We could instead try and prove that the formula ¬killed(X, Agatha), where X is a free variable (whose value is decided during the proof-search), is a consequence of the axioms for some suitable value for X. We can then deduce who the killer is based on the value of X. In Example 1, two challenges which arises when trying to prove it automatically are quantifiers and equality reasoning.
Figure 1.1: Formalization of Example 1. Each statement is represented by one or more statements of first-order logic.
1.2. Research Questions

1.1.1 Quantified Formulas

Many of the procedures for automated reasoning work with formulas of first-order logic. One of the fundamental challenges in first-order logic is handling quantifiers. Quantification is a language construct which allows for making statements which talks about one, more or all individuals of a population. For example, “every man is mortal”, which represents the fact that every object which has the property of being a man also possesses the property of being mortal, can be expressed using a quantifier. In Example 1, the statements “A killer always hates his victim” and “No one hates everyone” are quantified statements.

SMT, being quite efficient at handling existential theory reasoning, does have a limited support for reasoning with quantifiers. There are several proposed techniques for handling quantifiers (many based on using triggers, see [20]), but so far, none of the techniques for solving SMT lends itself to efficient quantifier-reasoning which is also complete (i.e. which can guarantee to find a proof if one exists).

Since quantified formulas are a fundamental part of first-order logic, there have been many methods towards solving problems involving quantified formulas. In this thesis we will focus on the two approaches, (i) function synthesis based on quantifier elimination which works with Quantified Boolean Formulas formulas over Boolean variables; and (ii) using Free Variables in sequent calculus, which is a method of introducing meta-variables whose instantiation is delayed until a proper term can be found.

1.1.2 Equality Reasoning

Another important challenge in automated theorem proving is handling the equality predicate. In Example 1 the statements “Agatha is not the butler” and “Agatha hates everyone except the butler” are instances of reasoning with equality (this is clear when looking at the formalization).

Sequent calculi are methods within the automated theorem proving field which are used to construct proofs for first-order formulas and can, compared to SMT-solvers, use more efficient techniques to handle quantifiers. Unfortunately, in contrast to the SMT context, handling theories is more difficult. In Section 4.3.2 we present an overview of how to handle equalities in sequent calculus.

1.2 Research Questions

In this thesis we are introducing new techniques for handling quantifiers (in conjunction with the equality predicate) in Boolean and first-order logic.
1.2.1 Research question 1

Finding techniques for handling formulas with quantifiers and equality in a complete and efficient way. The usual method of handling equality in a sequent calculus is by using free variables. However, this requires the solution the Simultaneous Rigid E-Unification (SREU) problem, the problem of finding assignment to variables such that pairs of terms becomes equal w.r.t. sets of equations. The SREU problem has been proven to be undecidable [18]. Replacing this unification problem by a restricted (bounded) problem, which is decidable, enables optimizations otherwise impossible, and allows for a more efficient solving procedure. In this thesis we present such a bounded unification problem and show how it can be implemented into a proof procedure.

1.2.2 Research question 2

Finding succint closed-form descriptions of Boolean networks by means of Quantifier Elimination. A major task in science is to establish connection between different fields, allowing one field to reap the benefits of another fields success. In this thesis we take a look at a biological application and investigate if there is a way of synthesizing functions which have a more readable description of solutions than what have previously been computed. This is an example of Quantifier Elimination which is the process of transforming a formula with quantifiers to an equisatisfiable formula which has no quantifiers (another instance is Coopers method for linear arithmetic [14]).
In this section we give a formalization of propositional logic. This introduction is necessarily brief; for more extensive descriptions there are plenty of options, e.g. [56, 5]. The core part of propositional logic are propositions - statements which are either true or false. This rather basic formalization of logic has been considered for a long time. One of the first scientific studies in logic was the introduction of syllogisms by Aristotle in the 4th century BC. We begin by giving an example of reasoning in propositional logic.

**Example 2.** The sentences “The deadline is on Wednesday or on Thursday. It is not on Thursday, so it must be on Wednesday.” is an example of a deduction in propositional logic.

“The deadline is on Wednesday” and “The deadline is on Thursday” are the two propositions involved, and the conclusion that the deadline is on a Wednesday is drawn from the two premises that it is either on a Wednesday or Thursday and that it is not on a Thursday.

### 2.1 Syntax

Example 2 is given in natural language, i.e., plain English. However, when dealing with logical systems it is often necessary to use a more strict notation to remove ambiguities. In this section we define a syntax for propositional logic — a description of what propositional formulas look like. We let $p, q, \ldots$ stand for propositions, also known as atoms, and $\phi, \psi, \ldots$ stand for formulas, where $\mathcal{F}$ denotes the set of all formulas. Atoms can be combined using
the Boolean connectives: \{\land, \lor, \neg, \implies\} into formulas, which are defined inductively (we will treat \(\phi \implies \psi\) as \(\neg \phi \lor \psi\)).

**Definition 1.** A literal is either an atom \(p\) or the negation of an atom \(\neg p\).

**Definition 2.** \(\tt\) and \(\ff\) are formulas. If \(p\) is a literal, then \(p\) and \((\neg p)\) are formulas. If \(s\) and \(t\) are formulas, then \((s \circ t)\) is a formula, where \(\circ \in \{\lor, \land, \implies\}\).

We will often omit parentheses from formulas if not necessary for truth-evaluation. For example, the connectives \(\lor\) and \(\land\) are associative, so we can write \(((s \land t) \land u)\) as \(s \land t \land u\).

### 2.2 Semantics

The *semantics* of propositional logic defines how the formulas should be interpreted. Each atom \(p\) can take exactly one one of the values true and false, denoted by \(\tt\) and \(\ff\). We define the notion of Boolean valuations and truth values of formulas.

**Definition 3.** A Boolean valuation is a function \(\beta\) which assigns one of the truth values \(\{\tt, \ff\}\) to each atom.

**Definition 4.** The truth value of a formula w.r.t. to a Boolean valuation \(\beta\) is given by the function \(\text{val}_\beta : \mathcal{F} \mapsto \{\tt, \ff\}\), defined inductively:

- \(\text{val}_\beta(p) = \beta(p)\) if \(p\) is an atom.
- \(\text{val}_\beta(\neg \phi) = \tt\) if \(\text{val}_\beta(\phi) = \ff\)
- \(\text{val}_\beta(\phi \lor \psi) = \tt\) if \(\text{val}_\beta(\phi) = \tt\) or \(\text{val}_\beta(\psi) = \tt\)
- \(\text{val}_\beta(\phi \land \psi) = \tt\) if \(\text{val}_\beta(\phi) = \tt\) and \(\text{val}_\beta(\psi) = \tt\)

**Example 3.** The sentences in Example 2 can be formalized as follows.

- “The deadline is on Wednesday or on Thursday” can be formalized as \(D_w \lor D_t\).
- “The deadline is not on Thursday” can be formalized as \(\neg D_w\).
- “The deadline is on Wednesday or on Thursday. It is not on Thursday, so it must be on Wednesday.” can be formalized as \(\varphi = (D_w \lor D_t) \land \neg D_w \implies D_t\)

where \(D_t\) stands for “The deadline is on Wednesday” and “The deadline is on Thursday” respectively.
In Example 3, if \( \beta \) is defined as \( \beta(D_w) = \text{tt} \) and \( \beta(D_t) = \text{ff} \) then \( \text{val}_\beta(\varphi) = \text{tt} \). Actually, irregardless of our choice \( \beta \), it will always be the case that \( \text{val}_\beta(\varphi) = \text{tt} \). This is a special property of the formula \( \varphi \).

**Definition 5.** A propositional formula \( s \) is a tautology, if for every Boolean valuation \( \beta \), \( \text{val}_\beta(s) = \text{tt} \).

**Definition 6.** A propositional formula \( \phi \) is satisfiable if there exists a Boolean valuation \( \beta \) s.t. \( \text{val}_\beta(s) = \text{tt} \). A formula which is not satisfiable is called unsatisfiable.

Every tautology is satisfiable, but there are satisfiable formulas which are not tautologies. More interestingly, if a formula \( \phi \) is a tautology, then \( \neg \phi \) is unsatisfiable. This allows us to check tautologies by using procedure for checking satisfiability. Furthermore, to check whether one formula implies another one, e.g., checking if \( (D_w \lor D_t) \land \neg D_w \) implies \( D_t \) in Example 3, can be done by checking whether the conjunction \( (((D_w \lor D_t) \land \neg D_w) \land D_t) \) is unsatisfiable.

**Normal Forms**

We call two formulas equivalent if they evaluate to \( \text{tt} \) under the same Boolean valuations. If we wish to decide whether a formula \( s \) is satisfiable, we can instead investigate an equisatisfiable formula \( t \), which might be easier to reason about. If \( t \) is an equisatisfiable formula to \( s \), it means that if \( s \) is satisfiable, then so is \( t \), though not necessarily in the same valuations. This allows for the usage of normal forms — a normal form is a syntactic set of formulas such that for every formula there exists an equivalent formula of this set. These are important because their existence allows algorithms to be restricted to formulas in a normal form. We introduce three main examples of normal forms:

**Definition 7.** A formula is in Negation Normal Form (NNF) if the negation connective only occur in front of atoms.

**Definition 8.** A formula is in Conjunction Normal Form (CNF) if it is a conjunction of disjunctions, i.e. \( (p_{0,0} \lor \ldots p_{0,n_0}) \land \ldots (p_{m,0} \lor \ldots p_{0,n_m}) \).

**Definition 9.** A formula is in Disjunctive Normal Form (DNF) if it is a disjunction of conjunctions, i.e. \( (p_{0,0} \land \ldots p_{0,n_0}) \lor \ldots (p_{m,0} \land \ldots p_{0,n_m}) \).

CNF is very commonly used it automated theorem proving and is the normal form for several major methods. One of the appealing reasons of CNF is that there exists a polynomial-time algorithm, using Tseitin-encoding (see [60]), which can reduce any formula to a polynomial-sized CNF which is equisatisfiable by introducing new propositions.
Example 4. The formula given in Example 3 could be transformed to the three different normal forms:

- **NNF** \((-D_w \land D_t) \lor D_w \lor D_t\)
- **CNF** \((-D_w \lor D_w \lor D_t) \land (-D_t \lor D_w \lor D_t)\)
- **DNF** \((-D_w \land D_t) \lor D_w \lor D_t\)

It is interesting to note that the NNF and DNF in Example 4 are equivalent, however this is not generally the case.

### 2.3 The SAT Problem

To establish if a propositional formula is satisfiable is one of the core problems of Computer Science. Known as the SAT problem (short for satisfiability), it was introduced in [13] as the first NP-complete problem. This suggests that there are no “efficient” (polynomial-time) algorithms for solving this problem (unless \(P = NP\)). However, many different methods exist which show good performance in practice.

#### 2.3.1 Truth-Tables

One well-known method of establishing the satisfiability of a formula is using a truth-tables. The idea is to make a table where each row represents one particular Boolean valuation. Given a formula with \(n\) atoms, there are \(n\) Boolean valuations, therefore such a table will have \(2^n\) rows, each of which row contains one Boolean valuation, together with truth value for the formula. If any row contains a **tt** for the formula, then there is a satisfying assignment. If all rows are **ff**, however, the formula is unsatisfiable.

Constructing a truth-table for a formula will always give a definitive answer whether the formula is satisfiable or not. However, in the worst case all rows of the table must be computed, which amounts to \(2^n\) rows, where \(n\) is the number of distinct propositions in the formula. This is not feasible in practice, therefore other more efficient methods have been developed.

Example 5. The formula \(\phi = (D_w \lor D_t) \land \neg D_w \implies D_t\) has its truth table shown in Table 2.1. By looking at the right-most column, we can see that \(\phi\) is indeed satisfiable (looking at e.g., the first row), and also a tautology (since every row contains **tt**).
2.4. Sequent Calculus

Here follows a brief description of a sequent calculus for propositional logic. A sequent calculus is a set of rules which can be used to construct a proof of a formula. A proof can show that a formula is a tautology, without the enumeration of Boolean valuations which is needed for truth-tables. We begin by showing an example proof of the formula given in Example 3.

**Example 6.** A formalization of Example 3 is \((D_w \lor D_t) \land \neg D_w \implies D_t\), or expressed as a sequent: \(D_w \lor D_t, \neg D_w \vdash D_t\). A proof using the calculus in Table 2.2 is:

\[
\begin{array}{c}
\vdash D_t, D_w \\
\vdash D_t, \neg D_w \\
\vdash D_t \\
\end{array}
\]

**Table 2.1:** Truth table for \((D_w \lor D_t) \land \neg D_w \implies D_t\)

<table>
<thead>
<tr>
<th>(D_w)</th>
<th>(D_t)</th>
<th>(D_w \lor D_t)</th>
<th>(\neg D_w)</th>
<th>((D_w \lor D_t) \land \neg D_w)</th>
<th>((D_w \lor D_t) \land \neg D_w \implies D_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{ff})</td>
<td>(\text{ff})</td>
<td>(\text{ff})</td>
<td>(\text{tt})</td>
<td>(\text{ff})</td>
<td>(\text{tt})</td>
</tr>
<tr>
<td>(\text{ff})</td>
<td>(\text{tt})</td>
<td>(\text{tt})</td>
<td>(\text{tt})</td>
<td>(\text{tt})</td>
<td>(\text{tt})</td>
</tr>
<tr>
<td>(\text{tt})</td>
<td>(\text{ff})</td>
<td>(\text{tt})</td>
<td>(\text{ff})</td>
<td>(\text{ff})</td>
<td>(\text{tt})</td>
</tr>
<tr>
<td>(\text{tt})</td>
<td>(\text{tt})</td>
<td>(\text{tt})</td>
<td>(\text{ff})</td>
<td>(\text{ff})</td>
<td>(\text{tt})</td>
</tr>
</tbody>
</table>

Proving that the formula is a tautology.

Before explaining its structure we must introduce the fundamental notion in a sequent calculus:

**Definition 10.** A sequent is a pair \(\Gamma \vdash \Delta\), where \(\Gamma\) and \(\Delta\) are finite sets of formulas. \(\Gamma\) is called the antecedent and \(\Delta\) the succedent.

Intuitively, a sequent \(\Gamma \vdash \Delta\) should be interpreted as being true (valid) if the conjunction of the formulas in \(\Gamma\) implies at least one of the formulas in \(\Delta\), i.e.,

\[
\gamma_1 \land \ldots \land \gamma_n \implies \delta_1 \lor \ldots \lor \delta_m
\]

where \(\Gamma = \{\gamma_1, \ldots, \gamma_n\}\) and \(\Delta = \{\delta_1, \ldots, \delta_m\}\).

A proof consists of a tree of sequents, where each node is related to its parents according to a set of inference rules (see Table 2.2). If a proof exists with a sequent at the root and each leaf node is an instance of the close-rule, then the root sequent is proven to be a tautology. We call a set of inference rules sound if there only exists proofs for sequents that are indeed a tautology, and complete if there exists a proof for every sequent.
which is a tautology. With a set of sound and complete inference rules, we can establish whether a formula is a tautology by searching over proofs (i.e., applying the rules exhaustively).

Sequent calculus was introduced by Gentzen [25] and has been one of the most influential logical systems since. A similar system, Semantic Tableaux, traces back to Beth in 1955 [10] and was made popular by Smullyan [56] for classical logics. The two systems are similar, indeed Carnielli observed that “tableaux systems are sequent systems upside down, and vice versa” ) [11]. Therefore we will much in the same spirit as in [19], treat results in one system as results in the other.

2.4.1 Analytic and Synthetic proofs

Proofs can be divided into two categories: analytic and synthetic proofs. An analytic proof starts from the formula to be proven and reduces it to smaller parts (i.e., sub-formulas) until self-evident truths has been reached (i.e., axioms). A synthetic proof, instead, begins with the self-evident truths and combine them until the formula to be proven has been reached. The sequent calculus is an analytic proof method, which implies that every formula occurring in the proof will be a sub-formula of the original formula. This makes sequent calculi very suitable for automatic theorem proving since the set of formulas which needs to be considered in proof construction is confined to this set.

Let \( S_p \) denote the calculus composed of the rules in 2.2. An inference rule states that given the sequent(s) above the line, the sequent below the line can be inferred. A proof in \( S_p \) of a formula \( \phi \) can be seen as a upwards growing tree with (i) the sequent \( \Gamma \vdash \phi \) at the root; (ii) each internal (non-leaf) node has as its premise the conclusion of its parent; (iii) and each leaf-node is an instance of the close-rule. Since this calculus is sound (see e.g., [56]
for a proof), if we are given a proof of a formula $\phi$, we know that $\phi$ is a tautology.

## 2.5 DPLL

A different approach, which tries to prove the satisfiability of propositional formulas, was introduced by Davis and Putnam [16], dubbed DP. They defined a set of rules (One-literal rule, Affirmative-Negative rule and Eliminating Atomic Formulas) which are applied to a formula in CNF transforming it to an equisatisfiable formula. Two years later, together with Logemann and Loveland, Davis published an improved version [15] where the Eliminating Atomic Formulas-rule was replaced by Case Distinction. This version, referred to as the Davis-Putnman-Logemann-Loveland procedure (DPLL). Its successor CDCL [35] is the basis of many state-of-the-art SAT/SMT-solvers (e.g., [17]).

## 2.6 Quantified Boolean Formulas

Given a propositional formula $s$, if we ask for its satisfiability, we are implicitly existentially quantifying all atoms (atoms in this context is usually called variables) in the formula, i.e. we are asking if there is a Boolean valuation giving values to each of the atoms in such a way that the formula is valuated to $tt$. However, we could imagine wanting some property to hold for all possible values for some atoms.

We can make this notion clearer (and also allow for more succinct ways to express formulas) by the introduction of two quantifiers, the existential quantifier, denoted by $\exists$, and the universal quantifier, denoted by $\forall$. The intuitive meaning is that a formula $\exists x. \varphi(x)$ states that $\varphi(x)$ is satisfied for some $x$ while $\forall x. \varphi(x)$ states that $\varphi(x)$ is satisfied by every $x$.

**Example 7.** The formula $\varphi = \forall x \forall y. (\neg x \lor y) \land (x \lor \neg y)$ is not valid satisfiable, since for $x = tt, y = tt$ the formula $\neg tt \lor tt \land (tt \lor \neg tt)$ clearly does not evaluate to $tt$. However, the formula $\varphi = \forall x \exists y. (\neg x \lor y) \land (x \lor \neg y)$ is valid, since for every value of $x$, we can pick a value of $y$ to make the formula $\neg x \lor y \land (x \lor \neg y)$ evaluate to $tt$.

We introduce the following definition for representing an assignment of a propositional atom, and use this to define the semantics of a quantified Boolean formula (QBF).

**Definition 11.** Given a formula $s$, let $s|_{x=tt}$ denote the formula $s$ with every occurrence of $x$ replaced by $tt$ (and analogously for $ff$).
Definition 12. The truth value of a formula w.r.t. to a Boolean valuation \( \beta \) is given by function \( \text{val}_\beta : \mathbb{F} \mapsto \{\text{tt}, \text{ff}\} \), defined inductively:

- \( \text{val}_\beta(\neg \phi) = \text{tt} \) if \( \text{val}_\beta(\phi) = \text{ff} \)
- \( \text{val}_\beta(\phi \lor \psi) = \text{tt} \) if \( \text{val}_\beta(\phi) = \text{tt} \) or \( \text{val}_\beta(\psi) = \text{tt} \)
- \( \text{val}_\beta(\phi \land \psi) = \text{tt} \) if \( \text{val}_\beta(\phi) = \text{tt} \) and \( \text{val}_\beta(\psi) = \text{tt} \)
- \( \text{val}_\beta(\phi \implies \psi) = \text{tt} \) if \( \text{val}_\beta(\phi) = \text{ff} \) or \( \text{val}_\beta(\psi) = \text{tt} \)
- \( \text{val}_\beta(\exists x. \phi) = \text{tt} \) if \( \text{val}_\beta(\phi|x=\text{tt}) = \text{tt} \) or \( \text{val}_\beta(\phi|x=\text{ff}) = \text{tt} \)
- \( \text{val}_\beta(\forall x. \phi) = \text{tt} \) if \( \text{val}_\beta(\phi|x=\text{tt}) = \text{tt} \) and \( \text{val}_\beta(\phi|x=\text{ff}) = \text{tt} \)

Deciding the satisfiability of a QBF is PSPACE-complete [57]. Several methods for checking satisfiability of QBF exists. The naive solution is expansion-based quantifier elimination. The idea is that every existential quantifier can be eliminated by replacing the quantified variable by a free variable, and every universal quantifier can be handled by handling each case. The formula \( \forall x. \varphi \) can be translated into an equivalent formula \( \varphi|_{x=\text{tt}} \land \varphi|_{x=\text{ff}} \). In this manner, every universal quantifier can be removed (at the expense of doubling the size of the formula). In the end we receive a formula which is quantifier-free (but possibly exponential in size) and can be handled by the techniques used for ordinary SAT problems. This process if often referred to as Shannon expansion [54].

A more advanced method is by using counter-example guided abstraction refinement (CEGAR). In a CEGAR setting, an abstraction of the original formula is created (an over-approximation) which might yield assignments which does not satisfy the original formula. Such an assignment is called a counter-example, and is used to refine the abstraction (by making it more detailed) such that (at least) that counter-example should not be found again. One solver implementing this method is the Recursive Abstraction Refinement QBF Solver [29].
Chapter 3

Using QBFs to Reason about Boolean Networks

“DNA is like a computer program but far, far more advanced than any software ever created.”

Bill Gates

In this section we present a biological problem which can be investigated using quantified Boolean formulas. We present the formalization and briefly discuss how we can use QBF solvers to find solutions (more details are in Paper III). An area which has had a rising demand for computational reasoning is biology, especially within genetics. Several examples exist where problems can be formulated in propositional logic, which then can be proved to have particular properties.

An interesting research problem is the investigation of genetic regulation, the interaction between genes in cellular processes and can be modeled using Gene Regulatory Networks (GRNs). The idea is that each gene is governed by a regulatory function, which as its inputs takes the state of other genes. The challenge is to abstract these GRNs in a meaningful manner. A very popular abstraction was presented in the seminal paper of Kauffman [32]. He introduced a model of genetic nets, which would represent genes as two-state entities governed by Boolean functions, later dubbed Boolean Networks:

**Definition 13.** A Boolean Network (BN) is a tuple \((G, I, R)\) with a set of genes \(G\) with \(|G| = n\), a set of interactions \(I \subseteq G \times G\), and a regulation function \(r_g \in R\) for each node \(r_g : G' \rightarrow B\) with \(G' \subseteq B^n\).

Intuitively, a Boolean network \((G, I, R)\) is a graph where each node \(g \in G\) corresponds to a single gene, each directed edge \((g_i, g_j) \in I\) corresponds to
an interaction between gene $g_i$ and $g_j$, each regulatory function $r_g \in R$ expresses how the gene $g$ is affected by its interacting genes (i.e., all genes $g'$ s.t. there is an interaction $(g', g) \in I$). In this model, each gene state is abstracted to “on” and “off”, and each gene $g_i$ is assigned a regulatory condition which is restricted to be a (quantifier-free) Boolean formula. In this thesis, we will focus on synchronous BN, i.e., all nodes are updated simultaneously.

In Figure 3.1 we see an example of a Boolean network. Since each node represents a gene and each edge an interaction between two genes, we can see that for example there is an interaction from LIF to Klf4. This means that in each time-step the state of gene LIF affects the new state of gene Klf4. As we will not discuss the regulatory functions in detail here, we just simply observe that a regulatory function should be a function of all interacting genes, e.g., in Figure 3.1 the regulatory function $r_{Klf4}$ of Klf4 should be some Boolean formula containing $s_{LIF}$ and $s_{Esr}$, (where $s_g$ refers to the state of gene $g$). Given a Boolean network it is possible to formulate QBF which expresses statements about it. For example, it is easy to create a formula which evaluates to true if and only if a specific gene in the network can ever reach the state “on”; or given a specific input state, is a certain state reachable.

The Boolean network abstraction for GRNs has become a very popular method of modeling gene states to perform various computations. The challenge lies in picking interactions and regulatory functions such that it presents a reasonable abstraction of how these would interact in an actual cell. Of course, this is far from an easy problem, one can conduct wet-lab experiments to form hypotheses of how the BN might behave. The challenges in constructing a BN model of a GRN is to (i) identify what interactions should be in the graph, and (ii) what regulation condition should be assigned to each gene. In [21] Dunn et. al. showed that restricting (ii) by allowing
just a few number of distinct regulation conditions one can focus on (i). The idea is that only regulatory conditions from a small set of predefined regulatory conditions are allowed to be assigned to nodes in a BN. Then the structure of the network (the interactions) are investigated over this restricted set of regulatory conditions. For this purpose they introduce a new concept: Abstract Boolean Networks.

3.1 Abstract Boolean Networks

While Boolean Networks are useful for representing a single concrete genetic net, however it is sometimes interesting to postulate queries about several networks Dunn et. al. introduced the concept of Abstract Boolean Networks [21]:

**Definition 14.** An Abstract Boolean Network (ABN) is a Boolean network with set of genes $G$ with $|G| = n$, state-space $Q$, initial states $Q_0$, final states $F$, regulation function sets $R_g \subseteq \mathcal{P}(\mathbb{B}^n \times \mathbb{B})$, and further equipped with restrictions on interactions $I := I_{\text{def}} \cup I_{\text{opt}}$ with

$$i_k \in I := \langle g, f, p, o \rangle,$$

where $g, f \in G$, $p \in \{+, -\}$, $o \in \{\text{opt, def}\}$.

Intuitively an ABN represents a set of (Concrete) Boolean Networks (CBN) which share the same set of genes and the set of definitive interactions $I_{\text{def}}$. However, each CBN can include any subset of the optional interactions $I_{\text{opt}}$ and must pick one regulatory condition for each gene $g_i$ from the set $R_{g_i}$. In contrast to Boolean networks, each interaction now has a polarity, it is either positive (+) or negative (-). In this description we ignore the state-space, the set of initial states and the final states.

In Figure 3.3 we a see a list of all Concrete Boolean Networks represented by the Abstract Boolean Network presented in Figure 3.2. This illustrates how we can use the concept of an ABN to represent a set of possible Boolean Networks. However, two important choices have to be made when constructing an ABN, which interactions should be chosen as optional and definitive, and how should we create the sets of allowed regulatory conditions? We start by describing how the latter is tackled in [21].

3.1.1 Restricting Regulatory Condition

A problem when declaring the set of regulatory conditions for a gene $g$ in an ABN, is that it should only depend on genes which are interacting with $g$. However, with optional interactions it is not the case that the
Figure 3.2: An example of an ABN. A solid edge represents a definitive interaction from one gene to another, while dashed edge represents an optional interaction. Sets of regulatory conditions for each gene are not presented in this figure.

set of interacting genes is definitive. Dunn et. al. solved this by defining regulatory conditions that did not depend on specific genes, but instead on all interacting genes. These are divided into two classes, the positive interacting and negative interacting genes. The idea is to define regulatory conditions such as “\( r_g \) is true if at least one positive interacting gene is on and none of the negative interacting genes are on”. A total of 18 regulatory conditions were formulated and it was motivated why it was a reasonable set to choose as the set of possible, Boolean regulatory conditions for all genes.

By doing this, it is possible to focus on the interactions, trying to investigate what optional interactions are most plausible in a CBN representing a real gene regulatory network. Experimental data is taken from already performed experiments, showing how the states of genes are changing over time. By constructing an ABN with all involved genes, definitive interactions between all genes which are known to interact in a certain way (positively or negatively) from earlier experiments, optional interactions between all genes where an interaction is suspected, and assigning the set of regulator conditions are picked as described above, we can construct a propositional formula which describes whether a certain CBN conforms to experiments, and by using a QBF-solver we can ask whether several CBNs of an ABN conform to the experimental data. Then, as a conservative approach, all CBNs are analyzed and all interactions common to all networks which conformed to the data are posted as a hypothesis for being scientifically true.

Studying the set of all CBNs in Figure 3.3, representing the ABN found in Figure 3.2, if CBNs c, d, g and h are the only ones modeling the experimental data, we can see that the edge from Esrrb to Klf is included in all of them. This leads us to conclude that the interaction from Esrrb to Klf is required to model the system behaviour. However, the edge from Klf to Oct4 is only present in some of the models, and therefore not concluded to be a required
Figure 3.3: All CBNs represented by the ABN in Figure 3.2
interaction.

The problem with this method is the enumeration of all CBNs. The number of possible models grows exponentially with the problem size, which leads to an insurmountably number of networks to analyze. An alternative approach is to use function synthesis.

### 3.1.2 Function Synthesis

A more top-down is to synthesize a formula describing the set of CBNs fulfilling a certain property. This approach is investigated in Paper III, where we describe an approach, similar to Counter-Example Guided Abstraction Refinement (see e.g., [12]), which tries to create such a function. The main purpose is to create a function on an interpretable template s.t. experts of the application domain (in this case biologists) can study them to find interesting properties.

### 3.2 Contributions

In Paper III we developed a function synthesis approach towards solving the analysis of Abstract Boolean Networks. By using a functional description of sets of concrete Boolean networks we produced descriptions of possible models without enumerating all of them, thus allowing us to represent solutions in a more concise format. This opens up the possibility of facilitating analysis of the results by enforcing templates which makes sense in a biological setting as well as allowing partial results to be interpreted (if not all models have been enumerated the conservative approach can not draw any conclusions, while a partial function can describe certain facts).
Chapter 4

First-Order Logic

“Life is not impossible, but it is exponentially difficult and sometimes worse.”

Melvin Fitting

Although propositional logic is powerful and can model many interesting problems, there are many more problems for which the propositional language is not strong enough. The next step is First-Order Logic. The difference between First-Order Logic and propositional logic is the introduction of predicates, functions, and (non-Boolean) variables. Furthermore, variables are now quantified over an arbitrary domain rather than just the set of \{\texttt{tt}, \texttt{ff}\}. We begin with an example:

Example 8. Consider the statement “No one hates everyone”, taken from Example 1, we presented the formalization as $\forall x \exists y. \neg \text{hates}(x, y)$. This formula is impossible to formalize without using quantifiers.

Three features are highlighted: (i) the variable $x$ which stands for objects of an underlying universe; (ii) the quantifier $\forall$ which quantifies the formula giving the meaning of $x$ referring to every object of the underlying universe; and (iii) the predicates Number, Odd and Even, applied to $x$ which are evaluated to \texttt{tt} or \texttt{ff} depending on the domain of $x$. Before describing the syntax and semantics of first-order logic we introduce the notion of a substitution. It is a fundamental part in both defining the semantics of first-order logic as well as describing the unification problem.

Definition 15. A substitution $\sigma$ is a mapping of a finite set of variables to terms. We use postfix notation of substitution, i.e. $\varphi \sigma$ denotes the
application of the substitution \( \sigma \) to the formula \( \varphi \). If \( x_1, \ldots, x_n \) are all variables \( x \) s.t. \( x \sigma \neq x \), and for each \( x_i \) we have \( x_i \sigma = t_i \) we can write \( \sigma \) as \( \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \).

**Example 9.** If we have the substitution \( \sigma = \{x \mapsto a, y \mapsto f(x)\} \) and \( \phi = g(f(x)) \lor \neg g(y) \), then \( \phi \sigma = g(f(a)) \lor \neg g(f(a)) \).

## 4.1 Syntax

We now present a formalization of first-order logic (for a more extensive description check e.g., [56, 22]). The syntax is similar to the propositional logic, however, first-order logic also has (non-Boolean) variables, terms, predicates, functions, and quantifiers. A first-order signature is a four-tuple \( \Sigma = \langle V, F, P, \alpha \rangle \), where \( V \) is a set of variables, \( F \) a set of function symbols, \( P \) a set of predicate symbols, and \( \alpha : (F \cup P) \mapsto \mathbb{N} \) is a function mapping each function and predicate symbol to its *arity*. Zero-arity functions are called constants.

**Definition 16.** The set of terms is defined inductively:

- any variable is a term;
- if \( t_1, \ldots, t_n \) are terms, and \( \alpha(f) = n, f \in F \) then \( f(t_1, \ldots, t_n) \) is a term.

**Definition 17.** The set of formulas is defined inductively. If \( x \) is a variable, \( t_1, \ldots, t_n \) are terms, \( p \in P \) s.t. \( \alpha(p) = n \), and \( \phi, \psi \) are formulas, then:

- \( p(t_1, \ldots, t_n) \) is a formula;
- \( \neg \varphi \) is a formula;
- \( \phi \lor \psi \) is a formula;
- \( \phi \land \psi \) is a formula;
- \( \phi \implies \psi \) is a formula;
- \( \exists x.\phi \) is a formula;
- \( \forall x.\phi \) is a formula;

The notion of a sub-formula is defined in the usual way. We also define the set of free variables:

**Definition 18.** The set of free variables \( \text{fv}(\phi) \) is defined inductively:

- \( \text{fv}(t) = \emptyset \) if \( t \not\in V \).
4.2 Semantics

A first-order structure is a tuple \( \langle U, I \rangle \) where \( U \) is a non-empty underlying universe of objects and \( I \) is an interpretation, s.t. for each function \( f \in F \) it assigns a function on the underlying universe \( I(f) = U^n \to U \) where \( n = \alpha(f) \), and for each predicate \( p \in P \) it assigns a predicate relation on the underlying universe \( I(p) \subseteq U^n \) where \( n = \alpha(p) \).

The interpretation of a predicate with arity \( n \) can be seen as a predicate over \( n \)-tuples of \( U \), which is the same as a total function from the \( n \)-tuples to Booleans. To define the truth-value of a sentence of first-order logic, we need both a first-order structure as well as a variable assignment.

**Definition 20.** A variable assignment \( \mathcal{A} \) is a function which assigns to each variable \( v \in V \) an element \( \mathcal{A}(v) = c \in U \) of the underlying universe. If \( \mathcal{A}' \) assigns to every variable (except possibly \( x \)) the same constant as \( \mathcal{A} \), then we call it an \( x \)-variant of \( \mathcal{A} \).

We next define what it means for a first-order structure and a variable assignment to satisfy a first-order formula:
Definition 21. Given a first-order structure $S = (U, I)$ and assignment $A$, we define a valuation $val_{S, A}$ over formulas as:

- $val_{S, A}(p(t_1, \ldots, t_n)) = I(p)(I(t'_1 A, \ldots, I(t'_n A)))$, where each $t'_i = A(t_i)$ if $t_i \in V$ else $t'_i = t_i$.
- $val_{S, A}(\neg \phi) = tt$ iff $val_{S, A}(\phi) = ff$
- $val_{S, A}(\phi \lor \psi) = tt$ iff $val_{S, A}(\phi) = tt$ or $val_{S, A}(\psi) = tt$
- $val_{S, A}(\phi \land \psi) = tt$ iff $val_{S, A}(\phi) = tt$ and $val_{S, A}(\psi) = tt$
- $val_{S, A}(\phi \rightarrow \psi) = tt$ iff $val_{S, A}(\phi) = ff$ or $val_{S, A}(\psi) = tt$
- $val_{S, A}(\exists x. \phi) = tt$ iff $val_{S, A'}(\phi) = tt$ for some $x$-variant $A'$ of $A$
- $val_{S, A}(\forall x. \phi) = tt$ iff $val_{S, A'}(\phi) = tt$ for all $x$-variant $A'$ of $A$

If a first-order formula is true under all first-order structures and all assignments then it is valid and if it is true under one first-order structure for some assignment, then it is satisfiable. For propositional logic we have the method of truth tables to decide the satisfiability (or tautology) for any propositional formula (since domains are finite, we can enumerate all Boolean valuations). Unfortunately, there exists no such procedure for first-order logic (one of Turing’s seminal results [61]). Nevertheless, it is possible to find subsets of problems in first-order logic which are decidable, or create a procedure which is semi-decidable for all of first-order logic (i.e., will always find a proof if there is one, but has no guarantee of termination). We will present some different approaches to proving first-order formulas, but the main focus of this thesis is on sequent calculi for first-order logic, which will be treated in the next chapter.

4.2.1 Resolution

Resolution was introduced in 1965 by Robinson [50], where a calculus and proof procedure was designed for automatic reasoning is presented. In contrast to earlier procedures, which would more often try to mimic human reasoning, the resolution method is simpler for implementation since it consists of a single inference rule. In the first-order case factoring is also needed, i.e., applying substitutions to make literals in a clause identical (and then removing redundant copies).

Resolution is a refutation-style system, that is to prove a formula $\phi$, one takes the negation $\neg \phi$ and proves that it is unsatisfiable, therefore implying that the original formula $\phi$ is valid. A formula in CNF is a conjunction of disjunctions, where each disjunction is called a clause. The resolution
method is based on a single inference rule working on a formula in CNF which can be seen as a set of clauses. The inference rule is based upon the notion of *resolvents*.

**Definition 22** ([1]). *The clause C is a resolvent of clauses A and B if there are a substitutions σ and θ s.t. Aσ = L ∨ A’ and Bθ = ¬L ∨ B’ and C = A’ ∨ B’.*

It is quite clear that given a first-order structure S and variable assignment A s.t. \(val_{S,A}(A \land B) = \text{tt}\) then \(val_{S,A}(C) = \text{tt}\). Resolution is applied to the set of clauses until the empty clause has been derived, proving that the original set of clauses was unsatisfiable. Resolution has been extended in many directions, e.g., using a set-of-support strategy [62], include equality by using paramodulation [49], and much more. It is used by many first-order theorem provers [37, 48, 53].

### 4.2.2 Sequent Calculus and Tableaux Methods

While resolution is a generalization of conjunction normal form (working on clauses), semantic tableaux can be seen as a generalization of disjunctive normal form (working on the duality of clauses). It is usually presented as a tree, where each branch represents a conjunction of its node, and the tree represents the disjunction of its branches. A branch is closed if it contains a contradiction, which corresponds to two literals in a conjunction which are contradictory in the DNF. And if each branch is closed (each disjunct is contradictory) then the original formula is unsatisfiable (and its negation valid).

We present here the extension of the calculus in Chapter 2 to include rules for first-order logic. It is in many ways similar, but now there are rules for handling quantifiers, and closure of branches is done with predicates rather than propositional atoms. Table 4.1 shows all of the rules. Special care must be taken with the rules \(\exists L\) and \(\forall LR\) where the constant \(c\) has to be fresh, i.e., it must not occur in the proof below. The notion of synthetic and analytic proof also applies to the first-order case.

In Example 4.1 we now give an example of a proof of Example 1 (with abbreviations for functions and predicates), showing that Agatha killed herself. This is represented by taking the root sequent as

\[ \varphi_{pre} \vdash killed(Agatha, Agatha) \]

where \(\varphi_{pre}\) is the set of premises. The proof can understood as a case-by-case analysis of whether Someone (i.e. the murderer) is Agatha, the Butler or Charles (since Someone must live at Dreadbury). Then the proof shows
that each of these cases leads to a contradiction (using the close-rule), which tells us that our conclusion is correct. Observe that the close-rule used in this example is taking into account equalities (see Section 4.3.2) by allowing a sequent \( \Gamma, E, p(s) \vdash p(t), \Delta \) to be closed if \( E \models s \simeq t \), where \( E \) is a set of ground equalities. Important properties of any first-order calculus are completeness and soundness.

**Definition 23.** A first-order calculus is complete if for every set of closed formulas \( \Gamma \) and a single closed formula \( \phi \), if \( \Gamma \implies \phi \) is valid then there exist a proof in the calculus of the sequent \( \Gamma \vdash \phi \).

**Definition 24.** A first-order calculus is sound if for every set of closed formulas \( \Gamma \) and a single closed formula \( \phi \), only if \( \Gamma \implies \phi \) is valid then there exists a proof of the sequent \( \Gamma \vdash \phi \).

Intuitively, if a calculus is complete, then everything that is true can be proved within our system, while soundness means that everything that can be proved is actually true. Of course most calculi would require both of these properties to be useful. The rules in Table 4.1 are indeed both sound and complete (but we will not prove that here, see for similar proofs e.g. [22].
\[
\begin{align*}
\neg\text{lives}(Ag), \text{lives}(Ag) \vdash \ldots & \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \\
S_0 \simeq Ag, Kd(So, Ag) \vdash Kd(Ag, Ag) & \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \quad \ldots \quad \text{CLOSE} \\
\neg\text{lives}(So) \lor Ag \simeq Ag \lor Ag \simeq Bu \lor Ag \simeq Ch, \ldots \vdash \ldots & \quad \exists L \quad \exists L \\
\forall x. \neg\text{lives}(x) \lor x \simeq Ag \lor x \simeq Bu \lor x \simeq Ch, \ldots \vdash \ldots & \quad \exists L \quad \exists L
\end{align*}
\]

Figure 4.1: Proof of Example 1
Figure 4.2: Left subproof

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]

\[ \frac{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H }{ \ldots \vdash (\bar{h} \cdot x) \mid H \land (\bar{h} \cdot x) \vdash \bar{h} \cdot x \vdash x \cdot \bar{x} \cdot (\bar{h} \cdot x) \mid H } \]
Figure 4.3: Right subproof
4.3 Theories

Many interesting problems need first-order reasoning in conjunction with theories. Examples of theories include equality or the theory of arrays. It is therefore interesting to find methods how to reason about formulas with these theories in an efficient manner.

4.3.1 Integer Arithmetic

Presburger arithmetic is the first-order theory with natural or integer numbers, the addition function and the equality predicate. It is decidable [38] and many software verification conditions can be expressed in terms of integer arithmetic (e.g., loop bounds in programs) which makes it a clear candidate for integration into sequent calculus. By adding inference rules dealing with (in-)equalities of integer arithmetic, one can create a procedure for handling Presburger formulas [52]).

4.3.2 Equality

Many interesting problems use the notion of equality, therefore the addition of equality reasoning is essential for creating a useful calculus. In the semantic tableaux setting, several different methods have been proposed (see e.g., [7], [9], [45]):

Adding Equality Axioms

Equality, denoted by $\simeq$ can be defined by its three properties:

- **Reflexivity**, $x \simeq x$.
- **Symmetry** $x \simeq y$ implies $y \simeq x$.
- **Transitivity** $x \simeq y$ and $y \simeq z$ implies $x \simeq z$.
- **Congruence** For all functions $f$ and predicates $p$, if $s_1 \simeq t_1, \ldots, s_n \simeq t_n$ then $f(s_1, \ldots, s_n) \simeq f(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n) \simeq p(t_1, \ldots, t_n)$

Each of these three can be axiomatized in first-order logic:

- **Reflexivity**, $\text{Ref} = \forall x. \; x \simeq x$.
- **Symmetry** $\text{Sym} = \forall x, y. x \simeq y \implies y \simeq x$.
- **Transitivity** $\text{Tran} = \forall x, y, z. (x \simeq y \land y \simeq z) \implies x \simeq z$.
- **Congruence**
- For each function symbol $f$, $\text{Cong}_f = \forall x_1, \ldots x_n, y_1, \ldots y_n. (x_1 \simeq y_1, \ldots x_n \simeq y_n) \implies f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n)$
- For each predicate symbol $p$, $\text{Cong}_p = \forall x_1, \ldots x_n, y_1, \ldots y_n. (x_1 \simeq y_1, \ldots x_n \simeq y_n) \implies p(x_1, \ldots, x_n) \simeq p(y_1, \ldots, y_n)$.

We can now take a formula $\phi$ containing equality literals and prove it under the axiomization of the equality predicate by trying to prove: $\text{Ref} \land \text{Sym} \land \text{Tran} \land \text{Cong}_f \land \text{Cong}_p \vdash \phi$. However, the search space becomes extremely large and can make proof search infeasible.

**Adding inference rules**

One of the earliest methods of handling equality is by introducing new inference rules which corresponds to equality reasoning. In [45] Reeves presents an overview of the first methods introduced and proposes a method which adds an equality-rule that takes equality constraints, however, these method also has problems (see [7]). To illustrate some of the problems with inference rules, consider the inference rules presented in Smullyan [56] for handling equality in a sequent calculi:

\[
\frac{\Gamma, s \simeq t \vdash \phi[s]\sigma, \Delta \quad \Gamma, s \simeq t \vdash \phi[s]\sigma, \Delta}{\Gamma, t \simeq s \vdash \phi[t'], \Delta} \quad \frac{\Gamma, s \simeq t \vdash \phi[t'], \Delta}{\Gamma, t \simeq s \vdash \phi[t'], \Delta}
\]

where $\phi[s]$ is the formula obtained by substituing $s$ for a occurrence of $t'$ in $\phi[t']$, and $\sigma$ is the mgu of $t$ and $t'$. Intuitively, the rule can be applied to use an equation to replace one term with another equal one. However, without any restrictions this rule can be applyde back-and-forth generating an infinite chain of the same sequents. There are means of remeding these flaws, for example by introducing a term order and only allow rewriting a term to a smaller term (thus making sure infinite chains can not be created).
Chapter 5

Automatic Reasoning

“A good algorithm never lies.”

Deep Impact

In this section we will give a brief overview of some different approaches to automated theorem-proving (ATP) and delve a bit further into sequent calculus methods. Logic has been used for proving theorems in different contexts. Since the dawn of computers, much research effort has been put into trying to automate this process. Many different so called proof procedures have been developed, targeting different kinds of logics (e.g., see [33]). A decision procedure is a procedure which can always give an answer to a yes/no question. For example, truth-tables are a decision procedure for testing satisfiability of a propositional formula. A decision procedure for first-order logic should be able determine the truth-value of any first-order formula, which is in the general case undecidable. Nevertheless many different (non decision-)procedures for trying to establish the satisfiability of these formulas exists. We will take a look at some different automations of theorem proving, but mostly focus on automatic reasoning in a sequent calculus since Paper I and Paper II are presenting results in that context.

5.1 Automatic Theorem Proving

In general the decision problem for first-order logic is undecidable. For this reason, we could limit ourselves to looking at a subset of the first-order formulas. In certain fragments of first-order logic a decision procedure exists, examples includes: the effectively propositional fragments [42] - formulas with a $\exists^*\forall^*$ quantifier-prefix (where $\exists^*$ means a prefix of any number of existential quantifiers) and the monadic predicate calculus [34] - formulas
where all predicates only takes on argument and there are no function symbols.

Another approach would be content with a semi-decision procedure, i.e. a procedure which only guarantees termination if the answer is yes. In a first-order setting this would mean a procedure which guarantees to find a proof if one exists, but if a formula is not valid then the procedure might not terminate. This is the route taken by many solvers, since for many problems many procedures does find a counter-example for invalid formulas.

A third approach in deciding the satisfiability of formulas is using Interactive Theorem Proving (ITP) which is a hybrid approach using both automated reasoning and human interaction. The idea is to automate the trivial (or easy) parts of the proof while the user can provide assistance at key points in the proof (selecting invariants, instantiations of quantifiers, etc.). Popular tools include Isabelle/HOL [39]. ITP is not very popular for plain first-order logic since it is not as expressive as other logics (e.g., higher-order logics).

5.2 Automatic Sequent Reasoning

One naive procedure would be to just enumerate all possible proofs (for example by considering them as strings, and enumerating all strings, checking those that are valid proof. However, this is of course extremely inefficient and impractical. A sequent calculus, instead we start with the root of the tree (corresponding to the formula to be proved) and then check what rules can be applied properly. If there is only one choice, then that rule can be applied and the algorithm continues. However, many times there are plenty of applicable rules (especially with quantifiers, see below) and therefore our proof search might have to branch.

5.2.1 Backtracking

When searching for a proof a decision has to made at each point of which proof rule to use for expansion next. In some calculi this choice can be destructive, i.e. by making a wrong choice the proof tree might not be completable anymore, i.e. there is no extension of the proof which could be closed. This can be seen in Example 11. If such a wrong move has been made, a procedure would need to backtrack and undo the destructive decision to try a different choice. Taken to its extreme, this corresponds to the branching search of DPLL, where each decision is over a Boolean variable, and found to be wrong the opposite can be assumed. However, some calculi have a certain property that makes backtracking unnecessary, namely the notion of proof confluence:
Definition 25. In a calculus, if every unfinished proof of a valid formula can be extended to a valid proof, the calculus is said to be proof confluent.

Given a proof confluent calculus, since we can always extend the proof to a valid one, we never need to backtrack, thus giving us a backtracking proof procedure.

5.2.2 Fair Proof Search

It can be shown that for many calculi, a fair proof search is a sufficient condition to guarantee completeness.

Definition 26. A proof search strategy is fair if every applicable rule-application will be applied at least once and each $\forall$-quantifier in the antecedent, and each $\exists$-quantifier in the succedent will be instantiated an arbitrary number of times if run for enough time.

5.2.3 Free Variables

The second major decision done during proof search is to decide which term should be instantiated using the $\forall L$ and $\exists R$. At each point a potential infinite amount of terms could be chosen, which yields the potential of ending up in an infinite “guessing game”. The most popular solution to this problem is the use of free variables (or “dummy variables” in Kangers terms [30].

The idea behind free variables is to introduce a meta-variable when a $\gamma$-rule is applied to delay the choice of the term until the branch is closed. Then substitutions are sought which close one or more branches.

Example 10.

\[
\begin{align*}
\forall x.P(x), P(X_1), P(X_2) & \vdash P(a) & \forall x.P(x), P(X_1), P(X_2) & \vdash P(b) \\
\forall x.P(x), P(X_1), P(X_2) & \vdash P(a) \land P(b) & \forall x.P(x) & \vdash P(a) \land P(b) \\
\forall x.P(x), P(X_1) & \vdash P(a) \land P(b) & \forall x.P(x) & \vdash P(a) \land P(b)
\end{align*}
\]

In Example 10 we can see an example of the usage of free variables. The sub-formula $\forall x.P(x)$ is instantiated two times with the free variables $X_1$ and $X_2$. This allows us to choose a term for which the $\forall L$ should be applied (in this case, the correct choices would be $a$ and $b$). At the top of the left branch, it can be seen that using the substitution $\sigma_l = \{X_1 \mapsto a\}$ allows us to close this branch. For the right branch $\sigma_r = \{X_2 \mapsto b\}$ is a closing substitution. These two substitutions are then combined $\sigma = \{X_1 \mapsto a, X_2 \mapsto b\}$ which is a closing substitution for the whole tableau. Note that it is not always
the case that the substitutions of several branches can be merged. Another
option would be to apply a closing substitution, whenever found, to the
tableaux. However, applying a substitution to the whole tableaux is
a potential destructive operation in the sense that proof confluency can be
lost.

Example 11.

\[ \{ X \mapsto a \} \]
\[ \frac{P(a, X), P(X, b) \vdash P(a, a) \lor P(a, b)}{\text{CLOSE}} \]
\[ \frac{P(a, X), P(X, b) \vdash (P(a, a) \lor P(a, b)) \land P(b, b)}{\lor R} \]

In Example 11 we have found a closing substitution for the left-hand
branch \( \sigma = \{ X \mapsto a \} \). However, if we apply this to the whole tableaux, the
right-hand branch would consist of the top-most sequent:

\[ P(a, a), P(a, b) \vdash P(b, b) \]

which can not be closed. Thus the proof is no longer closable (the correct
substitution would be \( \sigma' = \{ X \mapsto b \} \)). Therefore, if the first substitution
\( \sigma \) was applied to the tableaux, the procedure would need to backtrack.
There are several approaches how to handle these problems and to avoid
backtracking with application of substitutions. One way is to close all the
branches at the same time (solving a simultaneous unification problem, see
Section 6.1.2) as described in [7]. Another approach is by using incremental
closure [26] where substitutions for each branch are computed one-by-one
in such away that a global substitution is constructed incrementally (for
eaxmple, as described in [26]).

Another illustration of free variables is seen in our running example. The
proof given in 4.1 adds the “guess” \textit{killed(Someone, Agatha)} as the conclu-
sions and proves this to be valid. However, this presupposes that we know
which guess to make. Instead we can assert the conclusion \textit{killed(X, Agatha)}
(which represents all “guesses”) where \( X \) is a free variable, and then search
over proofs and find substitutions over \( X \) which would close the proof. If a
substitution is found, then the term which \( X \) is mapped to would identify
the killer.

5.3 Equality in Sequent Tableaux

As noted in the previous chapter, handling equality in conjunction with first-
order logic is a major challenge. Of course, doing this when automatically
constructing proofs is no easier. In addition to the two methods described
earlier, adding axioms and adding new inference rules, there is another ap-
proach when using free variables.
5.3.1 E-Unification

One method of handling equality is by using free variables and E-unification. E-unification will be treated in the next chapter (Chapter 6) but it can be seen as a procedure of deciding whether there is a substitution such that two terms become equal (modulo a set of equations). The idea is that when trying to close branches (see Section 5.2.3) instead of searching for a substitution using regular unification, E-unification is used, where the unification is done modulo the equation literals on the branch.

Example 12. We can now describe the unification used in the proof of our running example more accurately. In Figure 4.1, the second branch from the left, the closing sequent is:

\[ \text{\star} \quad \text{Someone} \simeq \text{Agatha, killed(Someone, Agatha)} \ldots \vdash \text{killed(Agatha, Agatha)} \quad \text{CLOSE} \]

which using normal (syntactic) unification can not be closed, since that requires a solution to the unification problem:

\[ \text{killed(Someone, Agatha)} = \text{killed(Agatha, Agatha)} \]

However, using E-unification, we have to find a unifier modulo the equation Someone \(\simeq\) Agatha. It is easy to see that the empty substitution is a unifier, and thus the branch can be closed.

This method can be an effective one, but as we will see in the next chapter, E-unification is a hard problem, especially when trying to close several branches simultaneously. It is worth noting that, while resolution-based provers are better at handling theories and free variables, they are less good at handling the Boolean structure of a problem. In contrast, sequent calculi can handle the Boolean structure efficiently, but, are worse at dealing with the combination of quantifiers and theories (especially equality).

5.4 Summary of Paper I

In Paper I we present a modified way of handling equality literals in a sequent tableaux. The idea is to use a special form of unification, where dummy-variables are ranging over finite domains, thus making it a finite process. We show that this can be done while still retaining a complete proof procedure. This process was inspired by work from Kanger [30] who presented these ideas as early as fifty years ago. We present a complete and sound calculus and an implementation which produces competitive results compared with other tableaux-provers.
Chapter 6

Unification

“In the ‘addition and multiplication’ of deduction work.”

J. A. Robinson

In this section we present the problem of unification, some variations, and their application. Unification is a well researched field (see [4] for an overview) which focuses on the problem of unifying two terms. It can informally be explained as finding a substitution such that two terms become equal. Several variations exists, among others $E$-unification, simultaneous unification and rigid unification (as well as their combinations). Free variable methods, as introduced in Sec. 5, are heavily dependent on the process of unification. When closing one branch of a proof tree, two terms have to be syntactically unified (see Section 6.1). If we want to close several branches at once, for example when using a non-backtracking calculus, we need to use simultaneous unification (see Section 6.1.2). If we are also dealing with equality, we need to unify terms modulo equations, using $E$-unification (see Section 6.2.1).

6.1 Syntactical Unification

The problem of syntactical unification was introduced by Robinson in 1965 [50]. The unification problem is syntactical in the sense that the two terms should become syntactically identical. The problem was introduced as a crucial part of the resolution method (see Section 4.2.1) and defined as:

Definition 27 ([50]). If $A$ is any set of terms and $\theta$ is a substitution, then $\theta$ is said to unify $A$, or to be a unifier of $A$, if $A\theta$ is a singleton. Any set of well-formed expressions which has a unifier is said to be unifiable.
An important class of unifiers are the most general unifiers.

**Definition 28.** A unifier $\gamma$ is called a most general unifier (or mgu for short) if for every other unifier $\theta$ there is a substitution $\sigma$ s.t. $\theta = \gamma\sigma$.

It is shown in [50] that for every unification problem as defined in Definition 27 there exists a unique mgu up to variable renaming. As we will see, this notion does not carry over when discussing $E$-unification.

### 6.1.1 Robinson

When the concept was first introduced in [50], Robinson also gave an algorithm for computing a most general unifier for two terms. It works by parsing both expressions from left to right and tries to substitute variables with suitable terms to make the two expressions equal. It allows variables to take any value and is complete in the sense that it will find a unifier if one exists.

**Algorithm 1:** Robinson original unification algorithm [50]

<table>
<thead>
<tr>
<th>Input</th>
<th>A set of terms $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>A mgu of the terms in $A$</td>
</tr>
</tbody>
</table>

1. $\sigma \leftarrow \varepsilon$
2. while $A\sigma$ is not singleton do
3. Find two terms $s, t \in A$ s.t. $s\sigma \neq t\sigma$
4. Find the first position $i$ where $s\sigma$ and $t\sigma$ differs.
5. $s' \leftarrow s[i]$
6. $t' \leftarrow t[i]$
7. if $s'$ is a variable and $s'$ does not occur in $t'$ then
8. $\sigma \leftarrow \sigma \cup \{s' \leftarrow t'\}$
9. else if $t'$ is a variable and $t'$ does not occur in $s'$ then
10. $\sigma \leftarrow \sigma \cup \{t' \leftarrow s'\}$
11. else
12. Fail
13. end

A common example (taken from [4]) illustrating the problem with Algorithm 1:

**Example 13 ([4]).** Consider the problem of unifying the two terms:

\[ h(x_1, x_2, \ldots, x_n, f(y_0, y_0), \ldots, f(y_{n-1}, y_{n-1}), y_n) \]

\[ h(f(x_0, x_0), \ldots, f(x_{n-1}, x_{n-1}), y_1, \ldots, y_n, x_n) \]

A unifier for these terms would have to include exponential sized terms, since $x_1 = f(x_0, x_0), x_2 = f(f(x_0, x_0), f(x_0, x_0)), \ldots$. 
6.2. Semantical Unification

Example 13 illustrates the problem that sometimes the resulting mgu is exponential in size. There are several approaches how to mitigate this: Robinson proposes [51] using a table to represent substitutions, which can represent the above kind of solutions in linear space (by sharing common subterms between terms). A more sophisticated approach using multi-sets of terms is proposed in [36], which is linear-time in the length of the terms to be unified.

6.1.2 Simultaneous Unification

As described in Chapter 5, closing a sequent tableaux with multiple branches requires us to find a substitution that unifies several pairs of terms simultaneously.

Example 14. If we wish to unify \( s_1 = f(g(a), Y) \) with \( t_1 = f(X, b) \) simultaneously as \( s_2 = g(f(X), c) \) is unified with \( t_2 = g(Z, c) \) we can rewrite it as a single unification problem by introducing a new function-symbol \( f' \) and unify:

\[
\begin{align*}
    s &= f'(s_1, s_2) = f'(f(g(a), Y), f((X), c)) \\
    t &= f'(t_1, t_2) = f'(f(X, b), g(Z, c))
\end{align*}
\]

If we have a unifier \( \sigma \) s.t. \( s\sigma = t\sigma \) then it is easy to see that also \( s_1\sigma = t_1\sigma \) and \( s_2\sigma = t_2\sigma \).

As seen in Example 14, simultaneous syntactical unification can be simulated using regular unification. However, when we are dealing with more complicated forms of unification, it is not always the case that this transformation is applicable.

6.2 Semantical Unification

Above we have discussed syntactical unification, that is two terms are only considered equal if they are syntactically identical. However, there are also many interesting problems that require unification under some theory, e.g., equality or associativity. We illustrate this with an example:

Example 15. The two terms \( s = f(X, a) \) and \( t = f(Y, b) \) are not syntactically unifiable, however under the theory of commutativity, the substitution \( \sigma = \{ X \rightarrow b, Y \rightarrow a \} \) is a unifier since \( s\sigma = f(b, a) \models_C f(a, b) = t\sigma \).

In a sequent tableaux proof, a branch can be closed if we have a sequent

\[
\Gamma, p(f(s, t)) \vdash p(f(s', t')), \Delta
\]
and a unifier $\sigma$ s.t.:
\[ s'\sigma = s'\sigma \text{ and } t\sigma = t'\sigma \]

If we know that the predicate $P$ is commutative, then we can perform unification modulo commutativity, i.e.,
\[ (s\sigma = s'\sigma \land t\sigma = t'\sigma \lor s\sigma = t'\sigma \text{ and } t\sigma = s'\sigma) \]

It has been shown [4] that unification modulo commutativity and unification modulo associativity (and the combination) are both NP-hard in the general case. An important point is that when doing semantic unification modulo some theory, concepts like mgu does not exists. For example, in rigid $E$-unification (see Section 6.2.1, there will not be a single mgu, but there will be a set of mgus [8].

### 6.2.1 $E$-Unification

As mentioned above, the theory of equality has a special role in theorem proving and occurs in many problems. Therefore it is an fundamental problem how to do unification modulo equations, as illustrated by the following example:

**Example 16.** Consider a sequent $\Gamma, a = b, P(f(X, X)) \vdash (f(a, b)), \Delta$ with the free variable $X$. To show that this is valid we need to find a substitution $\sigma$ s.t. $f(X, X)\sigma = f(a, b)\sigma$. By syntactical unification we can easily see that this is impossible. However, observe that it is possible to find a substitution if we consider the equation $a = b$. Since $a$ and $b$ are equal, we can allow $X$ to be substituted for either $a$ or $b$ (e.g., $f(a, a) = f(a, b)$ modulo $a = b$).

Example 16 clearly shows a situation where we wish to unify two terms w.r.t. the equalities found in the same sequent. This gives rise to the formulation of $E$-unification.

**Definition 29.** Let $E$ be a set of equations, and $s, t$ be terms. A substitution $\sigma$ is called a (universal) $E$-unifier of $s$ and $t$ if $E \models s\sigma = t\sigma$, where all equations in $E$ are treated as implicitly universally quantified. A (universal) simultaneous $E$-unifier $\sigma$ is a common $E$-unifier for a set $(E_i, s_i, t_i)_{i=1}^{n}$ of $E$-unification problems.

If we consider ground $E$-unification, i.e. the equations in $E$ contain no variables, it can be shown to be decidable [55] while the general case is undecidable. Now the Robinson algorithm (and its improvements) does not apply since we can no longer assume that we need syntactical identity. Other methods must be used. A popular method for handling the general case is narrowing [40] which is incomplete (since the problem is undecidable).
convergent rewrite systems, narrowing informally works by replacing sub-terms by “smaller” sub-terms until a normal form has been achieved (and thus terms can be compared).

**Rigid $E$-unification**

When considering tableau systems with free variables, a special kind of $E$-unification has to be applied:

**Definition 30** ([59]). Let $E$ be a set of equations, and $s, t$ be terms. A substitution $\sigma$ is called a rigid $E$-unifier of $s$ and $t$ if $s\sigma = t\sigma$ follows from $E\sigma$ via ground equational reasoning. A simultaneous rigid $E$-unifier $\sigma$ is a common rigid $E$-unifier for a set $(E_i, s_i, t_i)_{i=1}^n$ of rigid $E$-unification problems.

The difference between rigid and non-rigid $E$-unification is that in the former the variables in equations are only instantiated with one term, while in the non-rigid case they are assumed to be implicitly universally quantified. In sequent tableaux, we have to replace each occurrence of a free variable with the same constant in the entire tableaux, which means we have to use rigid $E$-unification instead of universal $E$-unification. Rigid $E$-unification as in Definition 30 is decidable, in fact it is NP-complete [24].

**Simultaneous (Rigid) $E$-unification**

When extending (rigid) $E$-unification to the simultaneous case for use with sequent tableaux, each pair of terms to be unified will have its own set of equations $E$ which should be calculated modulo. This yields the following definition:

**Definition 31.** A substitution $\sigma$ is a simultaneous rigid $E$-unifier if it is a common rigid $E$-unifier for a set $(E_i, s_i, t_i)_{i=1}^n$ of rigid $E$-unification problems.

If the sets $E_i$ are equal then this problem can be reduced to an instance of non-simultaneous rigid $E$-unification and is thus decidable, but in the general case it is undecidable [18].

**Definition 32.** A bounded rigid $E$-unification (BREU) problem is a triple $U = (\preceq, E, e)$, with $\preceq$ being a partial order over atomic terms such that for all variables $X$ the set $\{s \mid s \preceq X\}$ is finite; $E$ is a finite set of flat equations; and $e = s = t$ is an equation between atomic terms (the target equation). An atomic substitution $\sigma$ is called a bounded rigid $E$-unifier of $s$ and $t$ if $s\sigma \leftrightarrow_{E\sigma}^* t\sigma$ and $X\sigma \preceq X$ for all variables $X$. 
While simultaneous rigid $E$-unification is undecidable in the general case, BREU is decidable (since we have finite domains). It is shown in Paper I to be NP-complete. BREU has the presented application of handling the (simultaneous) closing of branches in sequent tableaux. There are also other potential uses. One use-case is $e$-matching, which is used in other unification procedures where an $E$-graph is computed keeping track of which terms are equal [20]. $E$-matching is a special case of $E$-unification, where one side is ground, and therefore the modifications needed to apply BREU are simple. Another potential use-case would be in conflict instance generation [47], where one tries to find instances of quantified formulas such that a contradiction is found. Here one could utilize the concept of free variables, and by using BREU to find substitutions that would create a contradiction.

6.3 Summary of Paper II

In Paper I we present a proof search procedure utilizing BREU and show that it is capable of producing competitive results on standard benchmarks. In Paper II we show two methods of solving the BREU problem, one based on lazily generating blocking clauses and the other based on searching over possible congruence closure relations represented by tables similar to the tables defined in [51].
Chapter 7

Conclusions

In this thesis we have presented two approaches towards handling quantifiers in (Boolean and) first-order logic with equality. One the one hand is the top-down function synthesis approach which tries to construct an equivalent formula in a CEGAR manner which can avoid the enumeration of the quantifiers domains. On the other hand we introduce the Bounded Rigid $E$-Unification, which we show can be used to construct a complete and sound first-order calculus.

7.1 Future Work

Both methods have potential in being extended in different direction. The function synthesis is using template-based synthesis, i.e. we are restricting on what form the generated function can take. We present in Paper III algebraic polynomials as a example of a template with some desirable properties with respect to termination. However, there is room for constructing other kinds of templates which could either be more efficient or to be on a form which is logical in the biological world. It would also be interesting to see how this template-based technique would be applied in other domains (e.g. model checking).

The BREU-problem also inspires interesting research questions. Firstly, it is interesting to see how BREU can be applied in other domains, examples were given with conflicting clause-generation and $E$-matching. There could also be other areas where a bounded unification algorithm is desirable. Secondly, there is the improvement of how BREU is utilized in a sequent tableaux, it would be interesting to see how concepts such as lemma-learning can be incorporated or how to propagate information deduced in the BREU-procedure to the proof-search and vice versa. Thirdly, it is the improvement of the BREU-procedure itself. In Paper II we present two methods of which
we can use to find solutions to BREU-problems. There is room for optimiz-
ing the algorithm for implementing these, and also for devicing completely
new methods of solving BREU.
Bibliography


Theorem Proving with
Bounded Rigid $E$-Unification

Peter Backeman and Philipp Rümmer
Uppsala University, Sweden

Abstract. Rigid $E$-unification is the problem of unifying two expressions modulo a set of equations, with the assumption that every variable denotes exactly one term (rigid semantics). This form of unification was originally developed as an approach to integrate equational reasoning in tableau-like proof procedures, and studied extensively in the late 80s and 90s. However, the fact that simultaneous rigid $E$-unification is undecidable has limited practical adoption, and to the best of our knowledge there is no tableau-based theorem prover that uses rigid $E$-unification.

We introduce simultaneous bounded rigid $E$-unification (BREU), a new version of rigid $E$-unification that is bounded in the sense that variables only represent terms from finite domains. We show that (simultaneous) BREU is NP-complete, outline how BREU problems can be encoded as propositional SAT-problems, and use BREU to introduce a sound and complete sequent calculus for first-order logic with equality.

1 Introduction

The integration of efficient equality reasoning in tableaux and sequent calculi is a long-standing challenge, and has led to a wealth of theoretically intriguing, yet surprisingly few practically satisfying solutions. Among others, a family of approaches related to the (undecidable) problem of computing simultaneous rigid $E$-unifiers have been developed, by utilising incomplete unification procedures in such a way that an overall complete first-order calculus is obtained. To the best of our knowledge, however, none of those procedures has led to competitive theorem provers.

We introduce simultaneous bounded rigid $E$-unification (BREU), a new version of rigid $E$-unification that is bounded in the sense that variables only represent terms from finite domains. BREU is significantly simpler than ordinary rigid $E$-unification, in terms of computational complexity as well as algorithmic aspects, and therefore a promising candidate for efficient implementation. BREU still enables the design of complete first-order calculi, but also makes combinations with techniques from the SMT field possible, in particular the use of congruence closure to handle ground equations.

* This work was partly supported by the Microsoft PhD Scholarship Programme and the Swedish Research Council.
1.1 Background and Motivating Example

We start by illustrating our approach using the following problem (from [5]):

\[
\phi = \exists x, y, u, v. \left( (a \not= b \lor g(x, u, v) \approx g(y, f(c), f(d))) \land (c \not= d \lor g(u, x, y) \approx g(v, f(a), f(b))) \right)
\]

To show validity of \(\phi\), a Gentzen-style proof (or, equivalently, a tableau) can be constructed, using free variables for \(x, y, u, v\):

\[
\begin{align*}
A : & \quad a \approx b \vdash g(X, U, V) \approx g(Y, f(c), f(d)) \\
& \quad c \approx d \vdash g(U, X, Y) \approx g(V, f(a), f(b)) \\
\hline
& \quad (a \not= b \lor g(X, U, V) \approx g(Y, f(c), f(d))) \land (c \not= d \lor g(U, X, Y) \approx g(V, f(a), f(b))) \vdash \phi
\end{align*}
\]

To finish this proof, both \(A\) and \(B\) need to be closed by applying further rules, and substituting concrete terms for the variables. The substitution \(\sigma_1 = \{X \mapsto Y, U \mapsto f(c), V \mapsto f(d)\}\) makes it possible to close \(A\) through equational reasoning, and \(\sigma_r = \{X \mapsto f(a), U \mapsto V, Y \mapsto f(b)\}\) closes \(B\), but neither closes both. Finding a substitution that closes both branches is known as simultaneous rigid E-unification (SREU), and has first been formulated in [9]:

**Definition 1 (Rigid E-Unification).** Let \(E\) be a set of equations, and \(s, t, \) be terms. A substitution \(\sigma\) is called a rigid E-unifier of \(s\) and \(t\) if \(s \sigma \approx t \sigma\) follows from \(E \sigma\) via ground equational reasoning. A simultaneous rigid E-unifier \(\sigma\) is a common rigid E-unifier for a set \((E_i, s_i, t_i)_{i=1}^n\) of rigid E-unification problems.

In our example, two rigid E-unification problems have to be solved:

\[
E_1 = \{a \approx b\}, \quad s_1 = g(X, U, V), \quad t_1 = g(Y, f(c), f(d)), \\
E_2 = \{c \approx d\}, \quad s_2 = g(U, X, Y), \quad t_2 = g(V, f(a), f(b)).
\]

We can observe that \(\sigma_s = \{X \mapsto f(a), Y \mapsto f(b), U \mapsto f(c), V \mapsto f(d)\}\) is a simultaneous rigid E-unifier, and suffices to finish the proof of \(\phi\). In general, of course, the SREU problem famously turned out undecidable [4], which makes the style of reasoning shown here problematic.

Different solutions have been proposed to address this situation, including potentially non-terminating, but complete E-unification procedures [8], and terminating but incomplete algorithms that are nevertheless sufficient to create complete proof procedures [5, 11]. The practical impact of such approaches has been limited; to the best of our knowledge, there is no (at least no actively maintained) theorem prover based on such explicit forms of SREU.

This paper introduces a new approach, bounded rigid E-unification (BREU), which belongs to the class of “terminating, but incomplete” algorithms for SREU. In contrast to ordinary SREU, our method only considers E-unifiers where substituted terms are taken from some predefined finite set. This directly
implies decidability of the unification problem; as we will see later, the problem is in fact NP-complete, even for the simultaneous case, and can be handled efficiently using SAT technology. In our experiments, cases with hundreds of simultaneous unification problems and thousands of terms were well in reach, and future advances in terms of algorithm design and efficient implementation are expected to further improve scalability.

For sake of presentation, BREU operates on formulae that are normalised by means of flattening (observe that $\phi$ and $\phi'$ are equivalent):

$$
\phi' = \forall z_1, z_2, z_3, z_4. (f(a) \not\equiv z_1 \lor f(b) \not\equiv z_2 \lor f(c) \not\equiv z_3 \lor f(d) \not\equiv z_4 \lor
\exists x, y, u, v. \forall z_5, z_6, z_7, z_8. (g(x, u, v) \not\equiv z_5 \lor g(y, z_3, z_4) \not\equiv z_6 \lor g(u, x, y) \not\equiv z_7 \lor g(v, z_1, z_2) \not\equiv z_8 \lor ((a \not\equiv b \lor z_5 \approx z_6) \land (c \not\equiv d \lor z_7 \approx z_8)))
$$

A proof constructed for $\phi'$ has the same structure as the one for $\phi$, with the difference that all function terms are now isolated in the antecedent:

$$
\begin{align*}
\vdash A' & = \ldots, g(X, U, V) \approx o_5, a \approx b \vdash o_5 \approx o_6 \\
\vdash B' & = \ldots, g(U, X, Y) \approx o_7, c \approx d \vdash o_7 \approx o_8 \\
& \vdash \forall z_1, z_2, z_3, z_4, \ldots
\end{align*}
$$

To obtain a bounded rigid $E$-unification problem, we now restrict the terms considered for instantiation of $X, Y, U, V$ to the symbols that were in scope when the variables were introduced (at $(\ast)$ in the proof): $X$ ranges over constants $\{o_1, o_2, o_3, o_4\}$, $Y$ over $\{o_1, o_2, o_3, o_4, X\}$, and so on. Since the problem is flat, those sets contain representatives of all existing ground terms at point $(\ast)$ in the proof. It is therefore possible to find a simultaneous $E$-unifier, namely the substitution $\sigma_b = \{X \mapsto o_1, Y \mapsto o_2, U \mapsto o_3, V \mapsto o_4\}$.

It has long been observed that this restricted instantiation strategy gives rise to a complete calculus for first-order logic with equality. The strategy was first introduced as dummy instantiation in the seminal work of Kanger [13] (in 1963, i.e., even before the introduction of unification), and later studied under the names subterm instantiation and minus-normalisation [6, 7]; the relationship to SREU was observed in [5]. The impact on practical theorem proving was again limited, however, among others because no efficient search procedures for dummy instantiation were available [7]. The present paper addresses this topic and makes the following main contributions:

- we define bounded rigid $E$-unification, as a restricted version of SREU, and investigate its complexity (Sect. 3);
- we present a sound, complete, and backtracking-free BREU-based sequent calculus for first-order with equality (Sect. 4–6);
- we give a preliminary experimental evaluation, comparing with other tableau-based theorem provers (Sect. 7).
1.2 Further Related Work

For a general overview of research on equality handling in sequent calculi and related systems, as well as on SREU, we refer the reader to the detailed handbook chapter [6]. The following paragraphs survey some of the more recent work.

Our work is partly motivated by a recent line of research on backtracking-free tableau calculi with free variables [10], capturing unification conditions as constraints that are attached to literals or tableau branches. This calculus was extended to handle equality using superposition-style inferences in [11], building on results from [5]. Our work resembles both [5, 11] in that we define an incomplete version of SREU, but show it to be sufficient for complete first-order reasoning. Our variant BREU is incomparable in completeness to the SREU solving in [5, 11]: BREU is able to derive a solution for the example shown in Sect. 1.1, which [5, 11] cannot; on the other hand, the procedures in [5, 11] are able to synthesise new terms of unbounded size as unifiers, whereas our procedure only considers terms from predefined bounded domains. The calculus in [11] was further extended to handle linear integer arithmetic in [14], however, excluding functions (but including uninterpreted predicates, to which functions can be reduced via axioms), leading to a further unification problem that is incomparable in expressiveness.

Equality handling was integrated into hyper tableaux in [2], again using superposition-style inferences, and also including redundancy criteria. This work deliberately avoids the use of rigid free variables shared between multiple tableau branches, so that branches can be closed one at a time, and there is no need for simultaneous E-unification. The calculus was implemented in the Hyper prover, against which we compare our implementation in Sect. 7.

2 Preliminaries

We assume familiarity with classical first-order logic and Gentzen-style calculi (see e.g., [8]). Given countably infinite sets \( C \) of constants (denoted by \( c, d, \ldots \)), \( V_b \) of bound variables (written \( x, y, \ldots \)), and \( V \) of free variables (denoted by \( X, Y, \ldots \)), as well as a finite set \( F \) of fixed-arity function symbols (written \( f, g, \ldots \)), the syntactic categories of formulae and terms \( t \) are defined by

\[
\phi ::= \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \forall x. \phi \mid \exists x. \phi \mid t \approx t , \quad t ::= c \mid x \mid X \mid f(t, \ldots, t) .
\]

Note that we distinguish between constants and zero-ary functions for reasons that will become apparent later. We generally assume that bound variables \( x \) only occur underneath quantifiers \( \forall x \) or \( \exists x \). Semantics of terms and formulae without free variables is defined as is common using first-order structures \( (U, I) \) consisting of a non-empty universe \( U \), and an interpretation function \( I \).

We call constants and (free or bound) variables atomic terms, and all other terms compound terms. A flat equation is an equation between atomic terms, or an equation of the form \( f(t_1, \ldots, t_n) \approx t_0 \), where \( t_0, \ldots, t_n \) are atomic terms. A flat formula is a formula \( \phi \) in which functions only occur in flat equations. A
formula $\phi$ is **positively flat** (**negatively flat**) if it is flat, and every occurrence of a function symbol is underneath an even (odd) number of negations. Note that every formula can be transformed to an equivalent positively flat (negatively flat) formula; we will usually assume that such preprocessing has been applied to formulae handled by our procedures. This kind of preprocessing is also standard for congruence closure procedures [1], and similarly used in SMT solvers.

If $\Gamma$ is a finite set of positively flat formulae (the \textit{antecedent}), and $\Delta$ a finite set of negatively flat formulae (the \textit{succeedent}), then $\Gamma \vdash \Delta$ is called a \textit{sequent}. A sequent $\Gamma \vdash \Delta$ without free variables is called \textit{valid} if the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. A calculus rule is a binary relation between finite sets of sequents (the premises) and sequents (the conclusion).

A substitution is a mapping of variables to terms, such that all but finitely many variables are mapped to themselves. Symbols $\sigma, \theta, \ldots$ denote substitutions, and we use post-fix notation $\phi \sigma$ or $\sigma t$ to denote application of substitutions. An \textit{atomic substitution} is a substitution that maps variables only to atomic terms. We write $u[r]$ do denote that $r$ is a sub-expression of a term or formula $u$.

**Definition 2 (Replacement relation [16]).** The replacement relation $\rightarrow_E$ induced by a set of equations $E$ is defined by: $u[l] \rightarrow u[r]$ if $l \approx r \in E$. The relation $\leftrightarrow_E^*$ represents the reflexive, symmetric and transitive closure of $\rightarrow_E$.

### 3 Bounded Rigid $E$-Unification

We present \textit{bounded} rigid $E$-Unification, a restriction of rigid $E$-unification in the sense that we now require solutions to be atomic substitutions such that variables are only mapped to smaller atomic terms according to a given partial order $\preceq$. This order takes over the role of an occurs-check of regular unification.

**Definition 3 (BREU).** A bounded rigid $E$-unification (BREU) problem is a triple $U = (\preceq, E, e)$, with $\preceq$ being a partial order over atomic terms such that for all variables $X$ the set $\{s \mid s \preceq X\}$ is finite; $E$ is a finite set of flat equations; and $e = s \approx t$ is an equation between atomic terms (the target equation). An atomic substitution $\sigma$ is called a bounded rigid $E$-unifier of $s$ and $t$ if $s \sigma \leftrightarrow^*_E t\sigma$ and $X\sigma \preceq X$ for all variables $X$.

Note that the partial order $\preceq$ is in principle an infinite object. However, only a finite part of it is relevant for defining and solving a BREU problem, which ensures that BREU problems can effectively be represented.

**Definition 4 (Simultaneous BREU).** A simultaneous bounded rigid $E$-unification problem is a pair $(\preceq, (E_i, e_i)_{i=1}^n)$ such that each triple $(\preceq, E_i, e_i)$ is a bounded rigid $E$-unification problem. An atomic substitution $\sigma$ is a simultaneous bounded rigid $E$-unifier for $(\preceq, (E_i, e_i)_{i=1}^n)$ if $\sigma$ is a bounded rigid $E$-unifier for each problem $(\preceq, E_i, e_i)$.

A solution to a simultaneous BREU problem can be used to close all branches in a proof tree. In Sect. 4 we present the connection in detail.
Example 5. We revisit the example introduced in Sect. 1.1, which leads to the following simultaneous BREU problem (\( \leq, \{(E_1, e_1), (E_2, e_2)\} \)):

\[
\begin{align*}
E_1 &= E \cup \{a \approx b\}, \quad e_1 = o_5 \approx o_6, \quad E_2 = E \cup \{c \approx d\}, \quad e_2 = o_7 \approx o_8, \\
E &= \{ f(a) \approx o_1, f(b) \approx o_2, f(c) \approx o_3, f(d) \approx o_4, \\
g(X, U, V) \approx o_5, g(Y, o_3, o_4) \approx o_6, g(U, X, Y) \approx o_7, g(V, o_1, o_2) \approx o_8 \}
\end{align*}
\]

with \( \{a, b, c, d\} \prec o_1 \prec o_2 \prec o_3 \prec o_4 \prec X \prec Y \prec U \prec V \prec o_5 \prec o_6 \prec o_7 \prec o_8 \).

A unifier to this problem is sufficient to close all goals of the tree up to equational reasoning; one solution is \( \sigma = \{X \mapsto o_1, Y \mapsto o_2, U \mapsto o_3, V \mapsto o_4\} \).

While SREU is undecidable in the general case, BREU is decidable; the existence of bounded rigid \( E \)-unifiers can be decided in non-deterministic polynomial time, since it can be verified in polynomial time that a substitution \( \sigma \) is a solution of a (possibly simultaneous) BREU problem (and since an \( E \)-unifier only has to consider variables that occur in the problem, it can be represented in space linear in the size of the BREU problem). Hardness follows from the fact that propositional satisfiability can be reduced to BREU, by virtue of the following construction.

3.1 Reduction of SAT to BREU

Consider propositional formulae \( \phi_b \), which are assumed to be constructed using the following operators:

\[
\phi_b ::= p \mid \neg \phi_b \mid \phi_b \lor \phi_b
\]

where \( p \) is a propositional symbol.

A formula \( \phi_b \) of this kind is converted to a BREU problem by introducing two constants \( 0 \) and \( 1 \); two function symbols \( f_{or} \) and \( f_{not} \); for each propositional symbol \( p \) in \( \phi_b \), a variable \( X_p \) such that \( 0 \prec X_p \) and \( 1 \prec X_p \); and for each sub-formula \( \psi \) of \( \phi_b \), a constant \( c_\psi \) and an equation:

\[
\begin{align*}
X_p &\approx c_\psi & \text{if } \psi = p, \\
f_{not}(c_\psi_1) &\approx c_\psi & \text{if } \psi = \neg \psi_1, \\
f_{or}(c_\psi_1, c_\psi_2) &\approx c_\psi & \text{if } \psi = \psi_1 \lor \psi_2.
\end{align*}
\]

The above, together with the set of equations \( \{f_{or}(0, 0) \approx 0, f_{or}(0, 1) \approx 1, f_{or}(1, 0) \approx 1, f_{or}(1, 1) \approx 1, f_{not}(0) \approx 1, f_{not}(1) \approx 0\} \) defining the semantics of the Boolean operators, and a target equation \( c_{\phi_b} \approx 1 \) yields a BREU problem that is naturally equivalent to the problem of checking satisfiability of \( \phi_b \). Indeed, every \( E \)-unifier can be translated to an assignment \( A \) of the propositional symbols such that \( A \models \phi_b \).

Theorem 6. Satisfiability of BREU problems is NP-complete.
3.2 Generalisations

A number of generalisations in the definition of BREU are possible, but can uniformly be reduced to BREU as formulated in Def. 3, without causing a blow-up in the size of the BREU problem.

General target constraints. Most importantly, there is no need to restrict BREU to single target equations \( e \), instead arbitrary positive Boolean combinations of equations can be solved; this observation is useful for integration of BREU into calculi. Any such combination of equations can be transformed to a single target equation using a construction resembling that in Sec. 3.1, at the cost of introducing a linear number of new symbols and defining equations.

For the remainder of the paper, we assume that \( e \) in Def. 3 can indeed be any positive Boolean combination of atomic equations.

Arbitrary equations. BREU problems containing arbitrary (i.e., possibly non-flat) equations in \( E \) or as target equation can be handled by reduction to equisatisfiable BREU problems with only flat equations, in a manner similar to [1]. Any non-flat equation of the form \( t[f(\bar{c})] \approx s \) can be replaced by two new equations \( t[d] \approx s \) and \( f(\bar{c}) \approx d \), where \( d \) is a fresh constant; the symmetric case, and non-flat target equations are handled similarly. Iterating this reduction eventually results in a problem with only flat equations.

Non-atomic E-unifiers. It is further possible to consider partial orders \( \preceq \) over arbitrary terms, as long as the set \( \{ s \mid s \preceq X \} \) is still finite for all variables \( X \). Reduction to problems as in Def. 3 is done by introducing a fresh constant \( c_t \) and a (possibly non-flat) equation \( t \approx c_t \) for each compound term \( t \) occurring in a set \( \{ s \mid s \preceq X \} \) for some variable \( X \) in the BREU problem. A new order \( \preceq' \) is defined by replacing compound terms \( t \) with constants \( c_t \), in such a way that

\[
\{ s \mid s \preceq' X \} = \{ s \mid s \preceq X, s \text{ is atomic} \} \cup \{ c_t \mid t \preceq X, t \text{ is compound} \}.
\]

With this in mind, it is possible to relax Def. 3 by including non-atomic unifiers \( \sigma \) (which might map variables to compound terms) as solutions to a BREU problem, as long as the condition \( X\sigma \preceq X \) holds for all variables \( X \).

Example 7. Consider the generalised BREU problem \( B = (\preceq, E, e) \) defined by

\[
E = \{ f(f(a,b),c) \approx g(b), f(X,Y) \approx c, g(b) \approx a \}, \quad e = a \approx c,
\]

\[
a < b < c < f(a,a) < f(a,b) < f(b,a) < f(b,b) < X < Y.
\]

Intuitively, the order \( \preceq \) encodes the fact that an E-unifier has to be constructed that maps every variable to a term with at most one occurrence of \( f \), and no occurrence of \( g \). A solution is the substitution \( \sigma = \{ X \mapsto f(a,b), Y \mapsto c \} \).

An equisatisfiable BREU problem according to Def. 3 is \( B' = (\preceq', E', e') \):

\[
E' = \{ f(d_1,c) \approx d_2, f(a,b) \approx d_1, g(b) \approx d_2, f(X,Y) \approx c, g(b) \approx a, f(a,a) \approx d_3, f(a,b) \approx d_4, f(b,a) \approx d_5, f(b,b) \approx d_6 \},
\]

\[
e' = e = a \approx c, \quad a \preceq' b \preceq' c \preceq' d_3 \preceq' d_4 \preceq' d_5 \preceq' d_6 \preceq' X \preceq' Y,
\]

with the E-unifier \( \sigma' = \{ X \mapsto d_4, Y \mapsto c \} \).


3.3 Encoding of $E$-Unification into SAT

Since satisfiability of BREU problems is NP-complete, a natural approach to compute solutions is an encoding as a propositional SAT problem, so that the performance of modern SAT solvers can be put to use. A procedure for solving a BREU problem will consist of three steps: (i) generating a candidate $E$-unifier $\sigma$; (ii) using congruence closure [1] to calculate the equivalence relation induced by the candidate $\sigma$ and the equations of the BREU problem; and (iii) checking if the BREU target equation is satisfied by this relation.

Each of these steps can be encoded into SAT. Candidate $E$-unifiers $\sigma$ are represented by a set of bit-vector variables storing the index of the term $X\sigma$ that each variable $X$ is mapped to. To guess candidate $E$-unifiers, it is then just necessary to encode the conditions $X\sigma \leq X$ as a propositional formula.

A congruence closure procedure can be modelled by representing intermediate results (i.e., equivalence relations) as a sequence of union-find data structures. To represent such a data structure in SAT, it suffices to introduce one bit-vector variable per atomic term $t$ occurring in the BREU problem, storing the index of the parent of $t$ in the union-find forest. Propositional constraints are added to characterise well-formed union-find forests, and to define the derivation of each forest from the previous one.

Lastly, to check the correctness of the candidate $\sigma$, it is asserted that the target equation is satisfied in the last union-find structure.

4 A First-order Logic Calculus with $E$-Unification

We will now introduce our sequent calculus for first-order logic with equality. The calculus operates only on flat formulae, and is kept quite minimalist to illustrate the use of free variables and BREU for delayed instantiation; for practical purposes, many refinements are possible, some of which are outlined in Sect. 6. The BREU procedure is utilised to define a global closure rule that discharges all goals of a proof tree simultaneously. Proof construction is intended to be done in upward direction and backtracking-free manner, following the proof procedures presented in [10, 14]; this is possible because all calculus rules are non-destructive and the overall calculus proof-confluent. We will show that fair application of the proof rules is complete.

The propositional, first-order, and equational rules of the calculus are shown in Table 1. Propositional and first-order rules mostly correspond to the classical system LK [8], however, keeping all structural rules implicit ($T$ and $A$ are sets of formulae). The first-order rules use Skolem symbols $c \in C$ for existential quantifiers in the antecedent, and fresh free variables $X \in V$ for universal quantifiers; and similarly for formulae in the succedent.

The equational rules simplify terms by means of ordered ground rewriting. Given a proof tree, we introduce a strict partial order $\prec \subseteq (C \cup V)^2$ over constants and free variables reflecting the order in which symbols are introduced by the rules $\forall L$, $\forall R$, $\exists L$, $\exists R$: we define $s \prec t$ if the constant or variable $t$ was introduced...
Table 1. Our sequent calculus for first-order logic with equality. In rules \( \forall L \) and \( \exists R \), \( X \) is a fresh variable, whereas the rules \( \exists L \) and \( \forall R \) introduce a fresh constant \( c \). In \( \approx L \) and \( \approx R \), the equation \( (t' \approx s') [t/s] \) is the result of replacing all occurrences of \( t \) with \( s \).

\[
\begin{align*}
\Gamma, \phi, \psi & \vdash \Delta \quad \Rightarrow \quad \Gamma, \phi \wedge \psi \vdash \Delta \\
\Gamma, \phi & \vdash \Delta \quad \Rightarrow \quad \Gamma, \psi \vdash \Delta \\
\Gamma, \forall x, \phi & \vdash \Delta \\
\Gamma, \phi[x/c] & \vdash \Delta \\
\Gamma, \exists x, \phi & \vdash \Delta \\
\Gamma, t & \approx s \quad \Rightarrow \quad \Gamma, t \approx s \vdash \Delta \\
\Gamma, t & \approx s, (t' \approx s') [t/s] \vdash \Delta \\
\Gamma, t & \approx s \vdash (t' \approx s') [t/s], \Delta \\
\frac{\Gamma_i \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta} & \text{BREU}
\end{align*}
\]

above the symbol \( s \), or if \( s \) is a symbol already occurring in the root sequent and \( t \) is introduced by some rule in the proof. For instance, for the proof shown in Sect. 1.1, the partial order shown in Example 5 is derived.

By slight abuse of notation, we also write \( s < f(t_1, \ldots, t_n) \) if \( s \) does not start with a function symbol. The rule \( \approx \text{ORIENT} \) moves the bigger term to the left-hand side of an equation. \( \approx L \) and \( \approx R \) can be used to replace occurrences of the (bigger) left-hand side term of an equation with the smaller right-hand side term; this rewriting is purely ground and does not unify expressions containing free variables (unification is entirely left to the BREU closure rule discussed in the next paragraph). As a consequence, and since \( < \) is well-founded, rewriting is terminating and confluent, and in fact implements a congruence closure procedure \([1]\) that eventually replaces every term with a unique representative term of its equivalence class modulo equations in the antecedent.

The BREU rule operates globally and closes all remaining goals of a proof if a global \( E \)-unifier \( \sigma \) exists that solves some succedent equation in each goal. The rule makes use of the non-strict partial order \( \preceq \) corresponding to \( < \), with the implication that every variable \( X \) can be mapped to symbols that were introduced prior to \( X \) in the proof. To encode non-emptiness of the universe, we assume that there is some constant \( c_\bot \in C \) below all variables \( X \in V \) in a proof.
(c_\perp \prec X \text{ for all } X \in V); \text{ if the proof itself does not contain such a constant, it is assumed that } c_\perp \text{ is some fresh constant with } c_\perp \prec X \text{ for all variables } X.

## 5 Properties of the Calculus

### 5.1 Soundness

The soundness of the calculus from Table 1 can be shown by substituting constants for all free variables, and observing the local soundness of each rule.

**Lemma 8.** Suppose \( \Gamma \vdash \Delta \) is a sequent without free variables. If a closed proof can be constructed for \( \Gamma \vdash \Delta \) using the calculus in Table 1, then \( \Gamma \vdash \Delta \) is valid.

**Proof.** We assume that a proof for \( \Gamma \vdash \Delta \) was closed using rule \textsc{breu}, with a unifier \( \sigma \) that maps every variable \( X \) occurring in the proof to a constant \( X \in C \) with \( X\sigma \prec X \). In case all goals were closed using \( \approx \text{close} \), \( X\sigma \) can be some arbitrary constant with \( X\sigma \prec X \).

By induction, it can be shown that the instance \( (\Gamma' \vdash \Delta')\sigma = \Gamma'\sigma \vdash \Delta'\sigma \) of every sequent \( \Gamma' \vdash \Delta' \) occurring in the proof is valid. This is the case for every goal discharged using rule \textsc{breu} by definition. For all other rules, it is the case that if the \( \sigma \)-instance of the premises is valid, then also the \( \sigma \)-instance of the conclusion is valid. We show two cases, the other rules are verified similarly:

- \( \exists l \): assume that the instantiated premise \( (\Gamma, \phi[x/c] \vdash \Delta)\sigma \) is valid. Since \( c \) is fresh, we know that \( X \prec c \) for all free variables \( X \) in \( \Gamma, \exists x.\phi \vdash \Delta \). Therefore \( X\sigma \prec c \), and it follows that \( (\Gamma, \exists x.\phi \vdash \Delta)\sigma \) does not contain \( c \). Validity of \( (\Gamma, \phi[x/c] \vdash \Delta)\sigma \) then implies validity of \( \forall x. (\Gamma \land \phi \rightarrow \bigvee \Delta)\sigma \), and equivalently of \( (\Gamma, \exists x.\phi \vdash \Delta)\sigma \).
- \( \approx l \): assume that \( (\Gamma, t \approx s, (t' \approx s')[t/s] \vdash \Delta)\sigma \) is valid. Then the conclusion \( (\Gamma, t \approx s, t' \approx s')[t/s] \vdash \Delta)\sigma \) is valid, too, since the conjunctions \( (t \approx s \land t' \approx s')\sigma \) and \( (t \approx s \land (t' \approx s')[t/s])\sigma \) are equivalent.

Since the root sequent \( \Gamma \vdash \Delta \) does not contain any free variables, it is implied that \( (\Gamma' \vdash \Delta)\sigma = \Gamma \vdash \Delta \) is valid. \( \square \)

### 5.2 Completeness

The completeness of the calculus can be shown using a model construction argument (e.g., [8]), which also implies that every attempt to construct a proof of a valid sequent in a “fair” manner will ultimately be successful; this ensures that proofs can always be found without the need for backtracking (although backtracking might sometimes lead to success more quickly, of course).

We call a proof search strategy for the calculus in Table 1 \textit{fair} if the propositional and first-order rules \( \land l, \land r, \lor l, \lor r, \neg l, \neg r, \forall l, \forall r, \exists l, \exists r \) are always eventually applied when they are applicable to some formula, and if every proof goal in which one of those rules is applicable is eventually expanded. This implies,
Lemma 9 (Completeness of fair proof search). Suppose \( \Gamma \vdash \Delta \) is a sequent without free variables, and suppose that a proof is constructed in a fair manner. If \( \Gamma \vdash \Delta \) is valid, then eventually a proof tree will be obtained that can be closed using the rule \( \mathrm{breu} \).

In order to prove this lemma, we first consider a “ground” version \( GC \) of our calculus, obtained by removing the rule \( \mathrm{breu} \), and by replacing \( \forall \) and \( \exists \) with the following ground rules:

\[
\begin{align*}
\Gamma, \forall \phi, \phi[x/c] & \vdash \Delta \quad \forall L_g, \\
\Gamma & \vdash \exists x. \phi[x/c], \Delta \quad \exists R_g
\end{align*}
\]

where \( c \) is an arbitrary constant. \( GC \) has the property that systematic application of the rules will either eventually produce a closed proof, or lead to a saturated (possibly infinite) branch from which a model can be derived:

**Definition 10.** An open proof branch in \( GC \) labelled with sequents \( \Gamma_0 \vdash \Delta_0, \Gamma_1 \vdash \Delta_1, \ldots \) (where \( \Gamma_0 \vdash \Delta_0 \) is the root of the proof) is called saturated if

(i) the branch is finite and no rule is applicable in the goal sequent \( \Gamma_n \vdash \Delta_n \); or

(ii) the branch is infinite, and for the limit sets \( \Gamma^\infty, \Delta^\infty \) of formulae occurring on the branch, as well as the sets \( \Gamma^p, \Delta^p \) of persistent formulae

\[
\Gamma^\infty = \bigcup_{i \geq 0} \Gamma_i, \quad \Delta^\infty = \bigcup_{i \geq 0} \Delta_i, \quad \Gamma_p = \bigcup_{i \geq 0} \bigcap_{j \geq i} \Gamma_j, \quad \Delta_p = \bigcup_{i \geq 0} \bigcap_{j \geq i} \Delta_j
\]

it is the case that (a) \( \Gamma_p \) only contains equations and \( \forall \)-quantified formulae; (b) \( \Delta_p \) only contains equations and \( \exists \)-quantified formulae; (c) none of the rules \( \approx \mathrm{elim}, \approx \mathrm{close}, \approx \mathrm{orient}, \approx \mathrm{L}, \approx \mathrm{R} \) is applicable in \( \Gamma_p \vdash \Delta_p \); (d) at least one constant \( c \) occurs on the branch; (e) for every formula \( \forall x. \phi \in \Gamma_p \) and every constant \( c \) occurring on the branch, there is an instance \( \phi[x/c] \in \Gamma^\infty \); and (f) for every formula \( \exists x. \phi \in \Delta_p \) and every constant \( c \) there is an instance \( \phi[x/c] \in \Delta^\infty \).

The ability to construct saturated branches follows directly from the observation that application of the \( GC \)-rules other than \( \forall L_g \) and \( \exists L_g \) terminates (because \( \prec \) is well-founded), and that \( \forall L_g \) and \( \exists L_g \) can be managed in a fair way using a work queue. The property (ii)–(d) encodes non-emptiness of universes, and is ensured by instantiating every formula \( \forall x. \phi \in \Gamma_p \) and \( \exists x. \phi \in \Delta_p \) at least once on every branch (e.g., using the \( \prec \)-smallest constant \( c_\bot \)).

**Lemma 11.** If a (finite or infinite) \( GC \) proof contains a saturated branch, then the root sequent \( \Gamma \vdash \Delta \) has a counter-model (is invalid).
Proof. We use persistent equations to construct a structure \( S = (U, I) \). In case of a finite saturated branch, persistent formulae are the ones in the goal; without loss of generality, we assume that also finite branches contain at least one constant. \( U \) is chosen as the set of constants that do not occur as left-hand side of some persistent antecedent equation; left-hand side terms are interpreted as the right-hand side constants. In case the value of some function application \( f(c_1, \ldots, c_n) \) is not determined by the equations, we set the value to some arbitrary constant \( c \in U \):

\[
U = \{ c \in C \mid c \text{ occurs in } \Gamma^\infty \cup \Delta^\infty \} \setminus \{ c \mid c \approx d \in \Gamma_p \}
\]

\[
I(c) = \begin{cases} 
  d & \text{if there exists an equation } c \approx d \in \Gamma_p \\
  c & \text{otherwise}
\end{cases}
\]

\[
I(f)(c_1, \ldots, c_n) = \begin{cases} 
  d & \text{if there exists an equation } f(c_1, \ldots, c_n) \approx d \in \Gamma_p \\
  c & \text{otherwise, for some arbitrary } c \in U
\end{cases}
\]

Since no equational rule is applicable in \( \Gamma_p \vdash \Delta_p \), it is clear that \( \text{val}_S(t \approx s) = \text{true} \) for every \( t \approx s \in \Gamma_p \), and \( \text{val}_S(t \approx s) = \text{false} \) for every \( t \approx s \in \Delta_p \).

By well-founded induction over the equations in \( \Gamma^\infty \), it can then be shown that in fact all equations in \( \Gamma^\infty \) evaluate to true under \( S \). For this we define a well-founded order \( \prec' \) over flat equations (for \( c, d \in C, \bar{c}, \bar{c}' \in C^\ast, f, g \in F \), and \( \prec_{\text{lex}} \) the well-founded lexicographic order induced by \( \prec \)):

\[
(c \approx d) \prec' (c' \approx d') \iff (d, c) \prec_{\text{lex}} (d', c'), \quad (c \approx d) \prec' (f(\bar{c}) \approx d'),
\]

\[
(f(\bar{c}) \approx d) \prec' (g(\bar{c}') \approx d') \iff f = g \text{ and } (d, \bar{c}) \prec_{\text{lex}} (d', \bar{c}').
\]

In particular, note that in any application of rule \( \approx_L \) we have \( (t \approx s) \prec' (t' \approx s') \) and \( (t' \approx s')[t/s] \prec' (t' \approx s') \); this implies that if all equations \( \prec' \)-smaller than \( t' \approx s' \) hold, then also \( t' \approx s' \) holds. In the same way, it can be proven that all equations in \( \Delta^\infty \) evaluate to false.

By induction over the depth of formulae we can conclude that all formulae (not only equations) in \( \Gamma^\infty \) evaluate to true, and all formulae in \( \Delta^\infty \) to false. □

Proof (Lem. 9). Assume that an (unsuccessful) attempt was made to construct a proof \( P \) for the valid sequent \( \Gamma \vdash \Delta \) by fair application of the rules in Table 1. We define a global mapping \( v : V \rightarrow C \) of variables occurring in \( P \) to constants, and use \( v \) to map \( P \) to a GC-proof with a saturated branch. The mapping \( v \) is defined successively by depth-first traversal of \( P \), visiting sequents closer to the root earlier than sequents further away. Note that for each branch that has not been closed by applying \( \approx_{\text{CLOSE}} \), fairness implies that \( \forall L \ (\exists r) \) has been applied infinitely often to every universally quantified formula in the antecedent (existentially quantified formula in the succedent).

When a node is visited where a new variable \( X \) is introduced by \( \forall L \) or \( \exists r \) for a quantified formula \( \phi \), set \( v(X) = c \) for some constant \( c \prec X \) that is \( \prec \)-minimal among the constants that have not yet been assigned for the same formula \( \phi \) on this branch. If no such constant exists, an arbitrary constant \( c \prec X \) is chosen. On
every infinite branch, this ensures that for every quantified formula $\phi$ handled via $\forall l$ or $\exists r$, and every constant $c$ occurring on the branch, there is some application of $\forall l$ or $\exists r$ to $\phi$ such that the introduced variable $X$ is mapped to $c = v(X)$.

The function $v$ can then be used to translate $P$ to a GC-proof $P'$, replacing each variable $X$ with the constant $v(X)$, and inserting exhaustive applications of the equational rules wherever they are applicable. By Lem. 11 and since $\Gamma \vdash \Delta$ is valid, each branch in $P'$ can be closed after finitely many steps through $\approx \text{CLOSE}$. This implies that it has to be possible to close the corresponding finite prefix of the original proof $P$ using rule $\text{BREU}$, with the mapping $v$ restricted to the variables occurring in the prefix as $E$-unifier. 

\[ \square \]

### 6 Refinements of the Calculus

The presented calculus can be refined in many practically relevant ways; in the scope of this paper, we only outline three modifications that we use in our implementation (also see Sect. 7).

**General instantiation.** Similar the subterm instantiation method proposed by Kanger [13], our system explicitly generates constants representing all terms possibly required for instantiation of quantified formulae, through application of $\exists l$ and $\forall r$. While subterm instantiation is complete, it has been observed (e.g., in [6]) that resulting proofs can sometimes be significantly longer than the shortest proofs that can be obtained when considering arbitrary instances of quantified formulae. Instantiation with new terms can be simulated in our systems by adding a rule $\text{TOT}$ representing the totality axiom $\forall x. \exists y. f(\bar{x}) \approx y$, which iteratively increases the range of terms considered for substitution by the $\text{BREU}$ rule. In $\text{TOT}$, $f$ is a function symbol, $X_1, \ldots, X_n$ are fresh variables, and $c$ is a fresh constant (and we set $X_i < c$ for all $i \in \{1, \ldots, n\}$):

\[
\frac{\Gamma, f(X_1, \ldots, X_n) \approx c \vdash \Delta}{\Gamma \vdash \Delta} \quad \text{TOT}
\]

**Local closure.** The closure rule $\text{BREU}$ can be generalised to operate not only on complete proof trees, but also on arbitrary sub-trees, and thus be used to guide proof expansion. For any sub-tree $t$, it can be checked (i) whether all goals in $t$ contain equations that are simultaneously $E$-unifiable; as long as this is not the case, proof expansion can focus on $t$, since rules applied to branches outside of $t$ will not be helpful; and (ii) whether the goals in $t$ are $E$-unifiable with a unifier $\sigma$ such that $X\sigma = X$ for all variables $X$ that occur outside of $t$; in this case, $t$ can be closed permanently and does not have to be considered again. It is also possible to define a notion of unsatisfiable cores for $E$-unification problems, which can further refine the selection of goals to be expanded.

**Ground instantiation.** It has also been observed that handling of quantifiers using free variables is very powerful, but is excessively expensive in case of simple
Table 2. Comparison of our prototypical implementation on TPTP benchmarks. The numbers indicate how many benchmarks in each group could be solved; the runtime per benchmark was limited to 240s (wall clock time). All experiments were done on an AMD Opteron 2220 SE machine, running 64-bit Linux, heap space limited to 1.5GB.

<table>
<thead>
<tr>
<th></th>
<th>FOF with eq.</th>
<th>FOF w/o eq.</th>
<th>CNF with eq.</th>
<th>CNF w/o eq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Princess + BREU</td>
<td>211</td>
<td>325</td>
<td>203</td>
<td>252</td>
</tr>
<tr>
<td>Hyper 1.0,16112014 [2]</td>
<td>119</td>
<td>378</td>
<td>160</td>
<td>305</td>
</tr>
<tr>
<td>leanCoP 2.2 (CASC-J7)</td>
<td>153</td>
<td>379</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

quantified formulae that have to be instantiated many times, and provides little guidance for proof construction. Possible solutions include the use of connection conditions, universal variables, or simplification rules [3, 12]. In our implementation, we use a more straightforward hybrid approach that combines free variables with ground instantiation through E-matching [15]; in combination, free variables and e-matching can solve significantly more problems than either technique individually. E-matching can be integrated naturally in our calculus without losing completeness, following [15]; in general this requires the use of the rule $\text{tot}$ shown above.

7 Experimental Results

We are in the process of implementing our BREU algorithm, and the calculus from Sect. 4, as an extension of the Princess theorem prover [14]. Our implementation uses the SAT encoding outlined in Sect. 3.3, and the Sat4j solver to solve the resulting constraints; we also include the refinements discussed in Sect. 6. Considered benchmarks were randomly selected TPTP v.6.1.0 problems with status Theorem or Unsatisfiable. To illustrate strengths and weaknesses of the compared tools, the benchmarks were categorised into FOF (first-order) problems with equality, FOF problems without equality, CNF (clause normal form) problems with equality, and CNF problems without equality. 500 benchmarks from all of TPTP were chosen in each group.

We compared our BREU implementation with the tableau provers Hyper and leanCoP from the CASC-J7 competition. Hyper uses the superposition-based equality reasoning from [2], whereas leanCoP relies on explicit equality axioms. The experimental results shown in Table 2 are still preliminary, and expected to change as further optimisations in our BREU procedure are done. However, it can be seen that even our current implementation of BREU shows performance that is comparable with the other tableau systems in all groups of benchmarks, and outperforms the other systems on benchmarks with equality.

1 leanCoP cannot process benchmarks in the TPTP CNF dialect.
2 http://user.it.uu.se/~petba168/breu/
Conclusion

We have introduced bounded rigid $E$-unification, a new variant of SREU, and illustrated how it can be used to construct sound and complete theorem provers for first-order logic with equality. We believe that BREU is a promising approach to handling of equality in tableaux and related calculi. Apart from improved algorithms for solving BREU, and an improved implementation, in future work we plan to consider the combination of BREU with other theories, in particular arithmetic, and integration of BREU with DPLL($T$)-style clause learning.

Acknowledgements We would like to thank Christoph M. Wintersteiger for comments on this paper, and the anonymous referees for helpful feedback.

References

Efficient Algorithms for Bounded Rigid $E$-Unification*

Peter Backeman and Philipp Rümmer

Uppsala University, Sweden

Abstract. Rigid $E$-unification is the problem of unifying two expressions modulo a set of equations, with the assumption that every variable denotes exactly one term (rigid semantics). This form of unification was originally developed as an approach to integrate equational reasoning in tableau-like proof procedures, and studied extensively in the late 80s and 90s. However, the fact that simultaneous rigid $E$-unification is undecidable has limited the practical relevance of the method, and to the best of our knowledge there is no tableau-based theorem prover that uses rigid $E$-unification. We recently introduced a new decidable variant of (simultaneous) rigid $E$-unification, bounded rigid $E$-unification (BREU), in which variables only represent terms from finite domains, and used it to define a first-order logic calculus. In this paper, we study the problem of computing solutions of (individual or simultaneous) BREU problems. Two new unification procedures for BREU are introduced, and compared theoretically and experimentally.

1 Introduction

The integration of efficient equality reasoning in tableaux and sequent calculi is a long-standing challenge, and has led to a wealth of theoretically intriguing, yet surprisingly few practically satisfying solutions. Among others, a family of approaches related to the (undecidable) problem of computing simultaneous rigid $E$-unifiers [7] have been developed, by utilising incomplete unification procedures in such a way that an overall complete first-order calculus is obtained. To the best of our knowledge, however, none of those procedures has led to competitive theorem provers.

We recently introduced simultaneous bounded rigid $E$-unification (BREU) [2], a new version of rigid $E$-unification that is bounded in the sense that variables only represent terms from finite domains, thus preserving decidability even for simultaneous $E$-unification problems. As demonstrated in [2], BREU can be used to design sound and complete calculi for first-order logic with equality, and to implement theorem provers that compare favourably to state-of-the-art tableau provers in terms of performance on problems with equality. In this paper we introduce two new unification algorithms for BREU problems.

* This work was partly supported by the Microsoft PhD Scholarship Programme and the Swedish Research Council.
1.1 Background and Motivating Example

We start by illustrating our approach using an example from [3, 2]:

\[ \phi = \exists x, y, u, v. \left( (a \neq b \vee g(x, u, v) \approx g(y, f(c), f(d))) \land (c \neq d \vee g(u, x, y) \approx g(v, f(a), f(b))) \right) \]

For sake of presentation, the formula is flattened to ensure that every literal contains at most one function symbol (for more details, see [2]):

\[ \phi' = \forall z_1, z_2, z_3, z_4. (f(a) \neq z_1 \vee f(b) \neq z_2 \vee f(c) \neq z_3 \vee f(d) \neq z_4 \vee \exists x, y, u, v. \forall z_5, z_6, z_7, z_8. \left( g(x, u, v) \neq z_5 \vee g(y, z_3, z_4) \neq z_6 \vee g(u, x, y) \neq z_7 \vee g(v, z_1, z_2) \neq z_8 \vee ((a \neq b \vee z_5 \approx z_6) \land (c \neq d \vee z_7 \approx z_8)) \right) ) \]

To show that \( \phi' \) is valid, a Gentzen-style proof (or, equivalently, a tableau) can be constructed, using free variables for \( x, y, u, v \):

\[
\begin{array}{c}
A \\
\ldots, g(X, U, V) \approx o_5, a \approx b \vdash o_5 \approx o_6 \\
\ldots, g(U, X, Y) \approx o_7, c \approx d \vdash o_7 \approx o_8 \\
\vdots \\
f(a) \approx o_1, f(b) \approx o_2, f(c) \approx o_3, f(d) \approx o_4 \vdash \exists x, y, u, v. \forall z_5, z_6, z_7, z_8. \ldots \quad (\ast) \\
\vdots \\
\vdash \forall z_1, z_2, z_3, z_4. \ldots
\end{array}
\]

To finish this proof, both \( A \) and \( B \) need to be closed by applying further rules, and substituting concrete terms for the variables. In our bounded setting, we restrict the terms considered for instantiation of \( X, Y, U, V \) to the symbols that were in scope when the variables were introduced (at \( \ast \) in the proof): \( X \) ranges over constants \( \{o_1, o_2, o_3, o_4\} \), \( Y \) over \( \{o_1, o_2, o_3, o_4, X\} \), and so on. Since the problem is flat, those sets contain representatives of all existing ground terms at point \( \ast \) in the proof. We can observe that the proof can be concluded by applying the substitution \( o_6 = \{X \mapsto o_1, Y \mapsto o_2, U \mapsto o_3, V \mapsto o_4\} \).

It has long been observed that this restricted instantiation strategy gives rise to a complete calculus for first-order logic with equality. The strategy was first introduced as dummy instantiation in the seminal work of Kanger [8] (in 1963, i.e., even before the introduction of unification), and later studied under the names subterm instantiation and minus-normalisation [4, 5]; the relationship to general Simultaneous Rigid \( E \)-unification (SREU) was observed in [3]. The present paper addresses the topic of solving a problem using the restricted strategy in an efficient way and makes the following main contributions:

- we define congruence tables and present an eager procedure for solving BREU using a SAT encoding (Sect. 4);
- we define complemented congruence closure, a procedure for abstract reasoning over sets of equivalence relations, and present a lazy solving procedure utilising this method (Sect. 5 and 6);
- we give an experimental comparison between the two methods (Sect. 7).
Further related work. For a general overview of research on equality handling in sequent calculi and related systems, as well as on SREU, we refer the reader to the detailed handbook chapter [4]. To the best of our knowledge, we are the first to develop algorithms for the BREU problem.

2 Preliminaries

We assume familiarity with classical first-order logic and Gentzen-style calculi (see e.g., [6]). Given countably infinite sets $C$ of constants (denoted by $c, d, \ldots$), $V_b$ of bound variables (written $x, y, \ldots$), and $V$ of free variables (denoted by $X, Y, \ldots$), as well as a finite set $F$ of fixed-arity function symbols (written $f, g, \ldots$), the syntactic categories of formulae $\phi$ and terms $t$ are defined by

$$
\phi ::= \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \forall x.\phi \mid \exists x.\phi \mid t \approx t,
\quad t ::= c \mid x \mid X \mid f(t, \ldots, t).
$$

Note that we distinguish between constants and zero-ary functions for reasons that will become apparent later. We generally assume that bound variables $x$ only occur underneath quantifiers $\forall x$ or $\exists x$. Semantics of terms and formulae without free variables is defined as is common.

We call constants and (free or bound) variables atomic terms, and all other terms compound terms. A flat equation is an equation between atomic terms, or an equation of the form $f(t_1, \ldots, t_n) \approx t_0$, where $t_0, \ldots, t_n$ are atomic terms. A congruence pair is a pair of two flat equations $(f(\tilde{a}) \approx b, f(\tilde{a}') \approx b')$ with $b \neq b'$.

A substitution is a mapping of variables to terms, s.t. all but finitely many variables are mapped to themselves. Symbols $\sigma, \theta, \ldots$ denote substitutions, and we use post-fix notation $\phi \sigma$ or $\sigma t$ to denote application of substitutions. An atomic substitution is idempotent if $\sigma \sigma = \sigma$. We write $u[r]$ do denote that $r$ is a sub-expression of a term or formula $u$, and $u[s]$ for the term or formula obtained by replacing the sub-expression $r$ with $s$.

Definition 1 ([11]). The replacement relation $\rightarrow_E$ induced by a set of equations $E$ is defined by: $u[l] \rightarrow u[r]$ if $l \approx r \in E$. The relation $\leftrightarrow_E^*$ represents the reflexive, symmetric, and transitive closure of $\rightarrow_E$.

2.1 Congruence Closure

We characterise the concept of congruence closure (CC) [9, 1] as fixed-point computation over equivalence relations between symbols. Let $S \subseteq C \cup V$ denote a finite set of constants and variables. The equivalence closure $\text{Cl}_{Eq}(R)$ of a binary relation $R \subseteq S^2$ is the smallest equivalence relation (ER) containing $R$.

Let further $E$ be a finite set of flat equations over $S$ (and arbitrary functions from $F$). Without loss of generality, we assume that every equation in $E$ contains a function symbol; equations $a \approx b$ between constants or variables can be rewritten to $f() \approx a, f() \approx b$ by introducing a fresh zero-ary function $f$. The
congruence closure $CC_E(R)$ of a relation $R \subseteq S^2$ with respect to $E$ is the smallest ER that is consistent with the equations $E$, and defined as a least fixed-point over binary relations as follows:

$$CC^1_E(R) = \text{Cl}_{E \uplus \{ (b,b') \mid \exists f(a) \approx b, f(a') \approx b' \in E \text{ with } (a,a') \in R \}}$$

$$CC_E(R) = \mu X \subseteq S^2. CC^1_E(R \cup X)$$

where we write $(\bar{a}, \bar{a}') \in R$ for the inclusion $\{(a_1,a_1'), (a_2,a_2'), \ldots, (a_n,a_n')\} \subseteq R$, provided $\bar{a} = (a_1, \ldots, a_n)$ and $\bar{a}' = (a_1', \ldots, a_n')$.

### 2.2 The Bounded Rigid $E$-Unification Problem

Bounded rigid $E$-unification is a restriction of rigid $E$-unification in the sense that solutions are required to be atomic substitutions s.t. variables are only mapped to smaller atomic terms according to some given partial ordering $\preceq$. This order takes over the role of an occurs-check of regular unification.

**Definition 2 (BREU).** A bounded rigid $E$-unification (BREU) problem is a triple $U = (\preceq, E, e)$, with $\preceq$ being a partial order over atomic terms s.t. for all variables $X$ the set $\{ s \mid s \preceq X \}$ is finite; $E$ is a finite set of flat formulae; and $e = s \approx t$ is an equation between atomic terms (the target equation). An atomic substitution $\sigma$ is called a bounded rigid $E$-unifier of $s$ and $t$ if $s \sigma \leftrightarrow^{*}_{E, \sigma} t \sigma$ and $X \sigma \preceq X$ for all variables $X$.

**Definition 3 (Simultaneous BREU).** A simultaneous bounded rigid $E$-unification problem is a pair $(\preceq, (E_i, e_i)_{i=1}^n)$ s.t. each triple $(\preceq, E_i, e_i)$ is a bounded rigid $E$-unification problem. A substitution $\sigma$ is a simultaneous bounded rigid $E$-unifier if it is a bounded rigid $E$-unifier for each problem $(\preceq, E_i, e_i)$.

A solution to a simultaneous BREU problem can be used in a calculus to close all branches in a proof tree. While SREU is undecidable in the general case, simultaneous BREU is decidable, in fact it is NP-complete [2]; the existence of bounded rigid $E$-unifiers can be decided in non-deterministic polynomial time, since it can be verified in polynomial time that a substitution $\sigma$ is a solution of a (possibly simultaneous) BREU problem. Hardness follows from the fact that propositional satisfiability can be reduced to BREU. Also, a number of generalisations are possible, but can be reduced to BREU as in Def. 2.

**Example 4.** We revisit the example introduced in Sect. 1.1, which can be captured as the following simultaneous BREU problem $(\preceq, \{(E_1, e_1), (E_2, e_2)\})$:

$$E_1 = E \cup \{a \approx b\}, \quad e_1 = o_5 \approx o_6, \quad E_2 = E \cup \{c \approx d\}, \quad e_2 = o_7 \approx o_8,$$

$$E = \{ f(a) \approx o_1, f(b) \approx o_2, f(c) \approx o_3, f(d) \approx o_4,$$

$$g(X, U, V) \approx o_5, g(Y, o_3, o_4) \approx o_6, g(U, X, Y) \approx o_7, g(V, o_1, o_2) \approx o_8 \}$$

with $a \prec b \prec c \prec d \prec o_1 \prec o_2 \prec o_3 \prec o_4 \prec X \prec Y \prec U \prec V \prec o_5 \prec o_6 \prec o_7 \prec o_8$.

A unifier to this problem is sufficient to close all goals of the tree up to equational reasoning; one solution is $\sigma = \{X \mapsto o_1, Y \mapsto o_2, U \mapsto o_3, V \mapsto o_4\}$. 

4
Input: BREU problem $B = (\preceq, E, s \approx t)$
1: while candidates remains do
2: \hspace{1em} $\sigma \leftarrow$ new candidate // Guessing
3: \hspace{1em} ER $\leftarrow$ $CC_E\{(X, X\sigma) \mid X \in S \cap V\}$ // Congruence Closure
4: \hspace{1em} if $(s, t) \in ER$ then // Verifying
5: \hspace{1em} \hspace{1em} return $\sigma$
6: \hspace{1em} end if
7: end while
8: return UNSAT

Algorithm 1: Generic search procedure for BREU

3 Solving Bounded Rigid $E$-Unification

Suppose $B = (\preceq, E, e)$ is a BREU problem, and $S \subseteq V \cup C$ the set of all atomic terms occurring in $B$ (“relevant terms”). On a high level, our procedures for solving BREU problems consist of three steps: Guessing a candidate substitution; using Congruence Closure to calculate the corresponding equivalence relation; and Verifying that the target equation is satisfied by this relation (see Alg. 1). This schema derives from the basic observation that $s \sigma \leftrightarrow^*_E t \sigma$ if and only if $(s, t) \in CC_E\{(X, X\sigma) \mid X \in S \cap V\}$, provided that $\sigma$ is an idempotent substitution [11]. Since an $E$-unifier $\sigma$ with $X \sigma \preceq X$ for all $X \in V$ can be normalised to an idempotent $E$-unifier, search can be restricted to the latter.

This paper introduces two different methods of performing these steps; an eager encoding of the problem into SAT that encodes the entire procedure as a SAT-problem, and a lazy encoding that uses SAT to generate candidate solutions. Common to both methods is the representation of the candidate substitution.

3.1 Candidate representation

We introduce a bijection $Ind : S \to \{1, \ldots, |S|\}$, s.t. for each $s, t \in S$ we have $s \preceq t \Rightarrow Ind(s) \leq Ind(t)$; the mapping $Ind$ will be used for the remainder of the paper. We also introduce a pseudo-integer variable$^1$ $v_s$ for each $s \in S$, together with a SAT-constraint restricting the domains:

$$\bigwedge_{c \in S \cap C} v_c = Ind(c) \wedge \bigvee_{X \in S \cap V \atop t \preceq X} (v_X = Ind(t) \wedge v_t = Ind(t)) \quad \text{(SAT DOMAIN)}$$

Any idempotent substitution $\sigma$ satisfying $X \sigma \preceq X$ for the variables $X \in V$ (as in Def. 2) can be represented by $v_X = Ind(X \sigma)$, and thus gives rise to a SAT model of the domain constraint; and vice versa. A search procedure over the models is thus sufficient for solving the Guessing step of Alg. 1. The SAT Domain constraint will be used in both methods presented in this paper.

$^1$ A pseudo-integer variable is a bit-wise representation of an integer in the range $\{1, \ldots, n\}$ by introducing $\lceil \log n \rceil$ Boolean variables.
4 Eager Encoding of Bounded Rigid $E$-Unification

In this section we describe how to eagerly encode a (simultaneous) BREU problem into SAT based on the procedure shown in Alg. 1. We note that a fairly intricate encoding is necessary to accommodate the combination of variables, constants, and congruence reasoning. For instance, the classical Ackermann reduction can be used to encode congruence closure and constants, but is not applicable in the presence of both variables and constants.

4.1 Congruence Tables

A congruence table is a table where each column represents a union-find data structure in a step of the congruence closure procedure, and each row corresponds to an atomic term, the “representative” for each step. The initial column is defined by a substitution while every internal column is constrained by its previous column modulo the given set of equations. From the final column of the table, an equivalence relation, equal to the congruence closure of the given substitution modulo the given equations, can be extracted.

Definition 5. Suppose $E$ is a set of flat equations, each containing exactly one function symbol, and $\sigma$ is a substitution s.t. $X\sigma \preceq X$ for all $X \in V$. As before, let $S = \{t_1, \ldots, t_m\} \subseteq C \cup V$ be the relevant terms, and $\text{Ind}(t_i) = i$ ($i \in \{1, \ldots, m\}$).

Then a congruence table $T$ of size $n$ for $E$ and $\sigma$ is a list of column vectors $[\bar{c}_1, \ldots, \bar{c}_n]$, with $\bar{c}_i \in \{1, \ldots, |S|\}^m$, where $\bar{c}_1 = (\text{Ind}(t_1\sigma), \ldots, \text{Ind}(t_m\sigma))$ and for each pair of consecutive vectors $\bar{c}_i$ and $\bar{c}_{i+1}$ and each $j \in \{1, \ldots, m\}$:

1. if $\bar{c}_i(j)^2 \neq j$ then $\bar{c}_{i+1}(j) = \bar{c}_{i+1}(\bar{c}_i(j))$.
2. if $\bar{c}_i(j) = j$ then:
   
   (a) $\bar{c}_{i+1}(j) = \bar{c}_{i+1}(k)$ if $k < j$, and there are equations $f(a_1, \ldots, a_l) \approx b$, $f(a_1', \ldots, a_l') \approx b' \in E$ s.t. $\bar{c}_i(\text{Ind}(b)) = j$ and $\bar{c}_i(\text{Ind}(b')) = k$, and furthermore $\bar{c}_i(\text{Ind}(a_h)) = \bar{c}_i(\text{Ind}(a_h'))$ for all $h \in \{1, \ldots, l\}$.
   (b) $\bar{c}_{i+1}(j) = j$ if no such pair of equations exists.

To illustrate the definition, observe first that all entries of the first vector point upwards, i.e., $\bar{c}_1(j) \leq j$ for $j \in \{1, \ldots, m\}$ (due to the definition of $\text{Ind}$ in Sect. 3.1), and define a union-find forest. The rules relating consecutive vectors (union-find data structures) to each other in Def. 5 correspond to three different cases: (1) defines path shortening, stating that each term can point directly to its representative term; (2a) states that if the arguments of two function applications are equal, the results must also be equal, and enables merging of the two equivalence classes s.t. the new representative is the smaller term; and (2b) states that if no such merging is possible, a term retains its identity value. All definitions are acyclic because the property $\bar{c}_i(j) \leq j$ is preserved in all columns $i$ (see Lem. 8 below).

\footnote{We write $\bar{c}(j)$ for the $j$th component of a vector $\bar{c}$.}
Example 6. Consider the simultaneous BREU problem and unifier $\sigma$ introduced in Example 4. Table 1 shows a complete congruence table of size 4 for $E_1$ (the left branch) and $\sigma$; for sake of presentation, the table contains symbols $t$ rather than their index $Ind(t)$, and in each column bold font indicates modified entries. The represented union-find forests are shown in Fig. 1, in which each edge is annotated with number of the column in which the edge was introduced. We can see that the fourth column defines an equivalence relation partitioning $ER$ of the set of relevant terms $S$ into seven sets. More importantly, under this equivalence relation the two terms in the target equation $e_1 = o_5 \approx_{ER} o_6$ are equal, implying that the substitution is a unifier to this sub-problem.

Definition 7. A congruence table $T = [\vec{c}_1, \ldots, \vec{c}_n]$ of size $n$ is complete if for every table $T' = [\vec{c}'_1, \ldots, \vec{c}'_{n+1}]$ of size $n+1$, if $\vec{c}_1 = \vec{c}'_1, \ldots, \vec{c}_n = \vec{c}'_n$ then $\vec{c}'_{n+1} = \vec{c}'_n$.

Intuitively, a congruence table $T$ is complete, if every additional column added would be identical to the last one.

Lemma 8. For every congruence table $T = [\vec{c}_1, \ldots, \vec{c}_n]$ of size $n$ $\forall i \in \{1, \ldots, n - 1\}$. $\forall j \in \{1, \ldots, |\vec{c}_i|\}$. $\vec{c}_{i+1}(j) \leq \vec{c}_i(j)$.

Table 1. $\begin{array}{|c|c|c|c|c|}
\hline
S & 1 & 2 & 3 & 4 \\
\hline
a & a & a & a & a \\
b & b & a & a & a \\
o_1 & o_1 & o_1 & o_1 & o_1 \\
o_2 & o_2 & o_1 & o_1 & o_1 \\
o_3 & o_3 & o_3 & o_3 & o_3 \\
o_4 & o_4 & o_4 & o_4 & o_4 \\
o_5 & o_5 & o_5 & o_5 & o_5 \\
o_6 & o_6 & o_6 & o_6 & o_5 \\
o_7 & o_7 & o_7 & o_7 & o_7 \\
o_8 & o_8 & o_8 & o_8 & o_8 \\
\hline
\end{array}$
only necessary to consider the congruence tables of a large enough size for every substitution to find a solution, and if none of them represents a solving substitution, the given BREU problem is unsatisfiable. This leads to the construction of a SAT model that encodes all possible congruence table of a certain size. However, this upper bound will in general be very pessimistic, so we introduce an iterative procedure that replaces this upper bound by checking an incompletion constraint.

4.2 Modeling Congruence Tables using SAT

In the remainder of this section we present the variables (the congruence matrix and the active congruence pairs) as well as the constraints introduced to model congruence tables for a given BREU problem $B = (\leq, E, e)$ using SAT.

Congruence matrix. The congruence matrix $M \in \{1, \ldots, m\}^{m \times n}$ is a matrix of pseudo-integer variables with $m$ rows and $n$ columns, corresponding to the vectors $[\vec{c}_1, \ldots, \vec{c}_n]$ in Def. 5. We write $M^i_j$ for the cell in row $j$ and column $i$. Intuitively, the matrix represents congruence tables of size $n$ for a set of relevant symbols $S$ with $|S| = m$, and cell $M^i_j$ represents the entry $\bar{c}_i(j)$.

Active congruence pairs. The set of congruence pairs is the set $CP = \{(f(\vec{a}) \approx b, f(\vec{a}') \approx b') \in E^2\}$. For each column $i > 1$ in the congruence matrix, there is also a set $\{v^i_{cp} \mid cp \in CP\}$ of auxiliary Boolean variables that indicate the active congruence pairs $cp = (f(a_1, \ldots, a_k) \approx b, f(a'_1, \ldots, a'_k) \approx b')$, constrained by:

$$v^i_{cp} \iff M^{i-1}_{\text{Ind}(a_1)} = M^{i-1}_{\text{Ind}(a'_1)} \land \cdots \land M^{i-1}_{\text{Ind}(a_k)} = M^{i-1}_{\text{Ind}(a'_k)} \land M^i_{\text{Ind}(b)} > M^i_{\text{Ind}(b')},$$

(TABLE CP)

Intuitively, if some $v^i_{cp}$ is true, the congruence pair $cp$ represents two equations in which the arguments are equal in the equivalence relation of column $i - 1$, but the results are different.

Initial column. In the initial column, we constrain each cell $M^1_j$ to be consistent with the variables $v_s$ introduced in Sect. 3.1 to represent solution candidates:

$$\bigwedge_{t \in S} M^1_{\text{Ind}(t)} = v_t$$

(TABLE INIT)

Internal column. In the internal columns with index $i > 1$, each cell must obey the following constraints, for every $j \in \{1, \ldots, m\}$:

$$\bigvee_{k \in \{1, \ldots, j-1\}} (M^{i-1}_j = k \land M^i_j = M^i_k) \lor$$

(TABLE INT)

$$M^{i-1}_j = j \land \left(\bigwedge_{cp \in CP} (\neg v^i_{cp} \lor M^{i-1}_{\text{Ind}(b)} \neq j ) \land M^i_j = j \lor \bigvee_{cp \in CP} \left(v^i_{cp} \land M^{i-1}_{\text{Ind}(b)} = j \land \bigvee_{k \in \{1, \ldots, j-1\}} (M^{i-1}_{\text{Ind}(b')} = k \land M^i_j = M^i_k) \right)\right)$$

with $cp = (f(\vec{a}) \approx b, f(\vec{a}') \approx b')$. The topmost constraint models condition (1) while the bottom constraint models condition (2) in Def. 5.
**Input:** BREU problem $B = (\preceq, E, s \approx t)$

1: Add initial table constraint (Sat Domain, Table CP, Init, Int, Goal)

2: while $\neg$ solver.isSat() do

3: Remove goal constraint (Sat Goal)

4: Add incompletion constraint (Table Incomp)

5: if $\neg$ solver.isSat() then

6: return UNSAT

7: else

8: Remove incompletion constraint (Table Incomp)

9: Add internal column and goal constraints (Table Int, Goal)

10: end if

11: end while

12: return SAT

**Algorithm 2:** Search procedure for the table encoding of a BREU problem

**Goal Constraint.** The final constraint asserts that the two rows corresponding to the two terms in the target equation contain the same atomic term in the final column.

$$M_{\text{Ind}(s)}^n = M_{\text{Ind}(t)}^n \quad \text{(Table Goal)}$$

where the target equations is $e = s \approx t$ and the table has $n$ columns.

### 4.3 Eager procedure

Our eager procedure (outlined in Alg. 2) creates constraints for an initial table, and then in an iterative fashion adds columns until either a solution is found, or an incompletion constraint is not satisfied. Incompletion constraints make it unnecessary to provide an a-priori upper bound on the size of constructed tables, and instead check whether some congruence pair can be used to merge further equivalence classes in the last column:

$$\bigvee_{c_{p} \in \text{CP}} v_{c_{p}}^{n+1} \quad \text{(Table Incomp)}$$

To handle a simultaneous BREU problem $B = (\preceq, (E_i, e_i)_{i=1}^n)$, one table is created for each sub-problem $(\preceq, E_i, e_i)$, s.t. the variables $x_t$ are shared. However, for many simultaneous BREU problems only a few of sub-problems are required to prove unsatisfiability. Therefore we use an iterative approach, where initially there is only a table for the first sub-problem. Once the constraints of the first table could be satisfied, the encoding is extended in an iterative fashion with tables for the other sub-problems, until either all tables are satisfied, or a subset of complete but unsatisfiable tables has been found.

### 5 Complemented Congruence Closure

The congruence closure algorithm (Sect. 2) efficiently decides entailment between ground equations, and can therefore be used to check (in polynomial time)
whether a given substitution σ is a solution to a BREU problem: σ translates to the equivalence relation \( \{(a, b) \in S^2 \mid a\sigma = b\sigma\} \) over the symbols \( S \subseteq C \cup V \) occurring in the problem, and can be completed to the smallest ER solving the BREU equations via CC.

As main building block for the lazy BREU algorithm introduced in the next section, we defined a generalised version of CC that can be applied to whole sets of relations over \( S \), in a manner similar to abstract interpretation (the new algorithm can indeed be identified as an abstract domain for CC, within the framework of abstract interpretation, but the details are beyond the scope of this paper). This notion of complemented congruence closure (CCC) can also be used as an optimisation for the SAT-based algorithm in Sect. 4, since it can often quickly rule out the existence of solutions to a BREU problem (Example 12).

CCC reasons about disequalities that are preserved by CC: while CC is defined as a least fixed-point over relations \( R \subseteq S^2 \) representing equalities between symbols (constants or variables), CCC corresponds to the computation of greatest fixed-points over relations \( D \subseteq S^2 \) representing disequalities between symbols. The definition of CCC is similar in shape to the one of CC in Sect. 2.1; as before, we assume that \( E \) is a finite set of flat equations over \( S \) in which each equation contains exactly one function symbol.

\[
C^{3,1}_E(D) = \left\{ (c, c') \in D \mid c \neq c', \text{ and for all } f(\bar{a}) \approx b, f(\bar{a}') \approx b' \in E \text{ it holds that } D \cap Cl_{Eq}\{(\bar{a}, \bar{a'}), (b, c), (b', c')\} \neq \emptyset \right\}
\]

\[
C^3_E(D) = \nu X \subseteq S^2. C^{3,1}_E(D \cap X)
\]

The one-step function \( C^{3,1}_E \) removes all pairs \((c, c')\) (representing disequalities \( c \neq c' \)) from the relation \( D \) that can no longer be maintained, i.e., if there are equations \( f(\bar{a}) \approx b \) and \( f(\bar{a}') \approx b' \) s.t. in some ER (consistent with the disequalities \( D \)) it is the case that \( \bar{a} \approx \bar{a}', b \approx c, \) and \( b' \approx c' \). This criterion is expressed by checking whether the equivalence closure \( Cl_{Eq}\{(\bar{a}, \bar{a'}), (b, c), (b', c')\} \) has some elements in common with the relation \( D \) representing assumed disequalities. The function \( C^{3,1}_E \) is clearly monotonic, and can therefore be used to define \( C^3_E \) as a greatest fixed-point over the complete lattice of binary relations; \( C^3_E \) itself is then also monotonic.

### 5.1 Properties of Complemented Congruence Closure

In this and later sections, we write \( R^C = S^2 \setminus R \) for the complement of a relation over \( S \). Most importantly, we can show that CC and CCC yield the same result when starting from equivalence relations, illustrating that CCC is a strict generalisation of CC:

**Theorem 10.** Suppose \( R \subseteq S^2 \) is an ER. Then \( CC_E(R)^C = C^3_E(R^C) \).

For arbitrary relations \( R \), congruence closure \( CC_E(R) \) will be an ER, whereas the result \( C^3_E(R^C)^C \) in general is not; consider in particular the case \( E = \emptyset \), in which \( CC_E \) will not have any effect beyond removing pairs \((c, c)\) from a relation. This implies that the assumption of \( R \) being an ER is essential in the theorem.
Sets $C_E^3(D)$ for relations $D$ whose complement is not an ER can be used to approximate the effect of CC, and in particular summarise the effect of applying CC to whole families of relations:

**Corollary 11.** Suppose $R \subseteq S^2$ is an ER, and $D \subseteq S^2$ a relation s.t. $R \cap D = \emptyset$. Then $CC_E(R) \cap C_E^3(D) = \emptyset$.

**Example 12.** Consider $S = \{c, d, e, X\}$, equations $E = \{f(X) \approx X, f(c) \approx d\}$, and the equivalence relation $R = Cl_{Eq}\{(X, c)\}$ that identifies $X$ and $c$ and keeps the other symbols distinct. CC on this input will also identify $X$ and $d$, and thus $c$ and $d$, but keep $e$ in a separate class: $CC_E(R) = Cl_{Eq}\{(X, c), (X, d)\}$.

The complement is $R^C = \{(c, d), (d, e), (e, c), (X, d), (X, e)\}$, where we write $A^{\leftrightarrow} = A \cup A^{-1}$ for the symmetric closure of a relation. CCC on $R^C$ will remove $(X, d)$ from the relation, since $Cl_{Eq}\{(X, c), (X, X), (X, d)\}$ is disjoint from $R^C$, and similarly $(c, d)$: $C_E^3(R^C) = \{(d, e), (e, c), (X, e)\}^{\leftrightarrow} = CC_E(R)^C$.

Consider now the BREU problem $B = (\preceq, E, c \approx e)$ with $c \prec d \prec e \prec X$. Note that every substitution $\sigma$ with $X \sigma \preceq X$ preserves the disequalities

$$D = \{(c, d), (d, e), (e, c)\}^{\leftrightarrow} = \bigcap_{\sigma \text{ a substitution }} \{a, b \in S^2 \mid a \sigma \neq b \sigma\}.$$

As before, CCC will remove $(c, d)$ from $D$; but CCC will keep $(c, e)$, because both $Cl_{Eq}\{(X, c), (X, c), (d, e)\}$ and $Cl_{Eq}\{(X, c), (X, e), (d, c)\}$ overlap with $D$, and similarly $(d, e)$: $C_E^3(D) = \{(d, e), (c, e)\}^{\leftrightarrow}$. This shows that $c$ and $e$ are not $E$-unifiable, and neither are $d$ and $e$.

## 6 Lazily Solving Bounded Rigid $E$-Unification

When dealing with large simultaneous BREU problems, e.g., containing many parallel problems as well as many equations, just constructing a monolithic SAT model (possibly containing much redundancy) as in Sect. 4 can be time-consuming, even if the subsequent solving might be fast. Our second algorithm for solving BREU problems works in the style of lazy SMT solving: starting from a compact SAT encoding that coarsely over-approximates the BREU problem, additional constraints are successively added, until eventually a correct $E$-unifier is derived, or the encoding becomes unsatisfiable. Following Alg. 1, the overall idea is to repeatedly generate candidate solutions $\sigma$, check whether the candidate is a genuine solution, and otherwise generate a blocking constraint that excludes (at least) this solution from the search space.

**Overall procedure.** Consider a simultaneous BREU problem $(\preceq, (E_i, e_i)_{i=1}^n)$. The overall procedure is shown in Alg. 3, and based on the three steps described in Sect. 3, but directly solving simultaneous BREU problems. The algorithm uses an underlying solver process for reasoning incrementally about the SAT encoding. The GUESSING step is implemented using the SAT DOMAIN constraints from Sect. 3.1 (line 1). When a candidate solution $\sigma$ has been found, congruence closure is used to verify that $\sigma$ solves each sub-problem $(\preceq, E_i, e_i)$ (line 4), executing the CONGRUENCE CLOSURE and VERIFYING steps in Alg. 1.
Add domain constraints (SAT DOMAIN)

while solver.isSat() do
  σ ← solver.model
  if σ solves all sub-problems then
    return σ
  else
    Let (≤, E, e) be an unsolved sub-problem
    \( D \in \{(s, t) \in S^2 \mid s \neq t\sigma\} \)
    \( D' ← \text{minimise}(D, (\leq, E, e)) \)
    Add blocking constraint \( \bigvee \{v_s = v_t \mid (s, t) \in D'\} \)
  end if
end while
return UNSAT

Algorithm 3: Lazy search procedure for a simultaneous BREU problem.

Input: Disequality set \( D \)
Input: BREU problem \((\leq, E, s \approx t)\) with \((s, t) \in C^2_E(D)\)
1: Compute set BaseD for \( \leq \)    // by construction, BaseD ⊆ D
2: for \( dq \in D \setminus \text{BaseD} \) do
3: \( D' ← C^3_E(D \setminus \{dq\}) \)
4: if \((s, t) \in C^3_E(D')\) then
5: \( D ← D' \cup \text{BaseD} \)
6: end if
7: end for
8: return \( D \)

Algorithm 4: Minimisation of disequality sets

Blocking constraints. Given a candidate \( \sigma \) that violates \((\leq, E_i, s_i \approx t_i)\), a blocking constraint for \( \sigma \) is a formula \( \phi \) over the solution variables \( \{v_t \mid t \in S\} \) introduced in Sect. 3.1 with the property that 1. \( \phi \) evaluates to false for the assignment \( \{v_t \mapsto \text{Ind}(t\sigma) \mid t \in S\} \), and 2. \( \phi \) evaluates to true for all genuine \( E \)-unifiers \( \sigma' \) and assignments \( \{v_t \mapsto \text{Ind}(t\sigma') \mid t \in S\} \). In other words, \( \phi \) excludes the incorrect solution \( \sigma \), but it does not rule out any correct \( E \)-unifiers. The most straightforward blocking constraint excludes the incorrect candidate \( \sigma \):

\[
\bigvee_{X \in S \cap V} v_X \neq \text{Ind}(X\sigma)
\]

(1)

This constraint leads to a correct procedure, but is inefficient since it does not generalise from the observed conflict (in SMT terminology), and does not exclude any candidates other than \( \sigma \). More efficient blocking constraints can be defined by using the concept of complemented congruence closure. For this, observe that (1) can equivalently be expressed in terms of disequalities implied by \( \sigma \):

\[
\bigvee_{(s, t) \in D} v_s = v_t, \quad D = \{(s, t) \in S^2 \mid s\sigma \neq t\sigma\}
\]

(2)
Example 13. is not helpful, since such disequalities are already implied by the serving (in Alg. 4, which successively attempts to remove elements minimisation. therefore (s, t) is a solution candidate violating Lazy BC, which by construction implies \( R \cap D' = \emptyset \) for \( R = \{ (s, t) \in S^2 \mid s \sigma = t \sigma' \} \). By Corollary 11, we then have \( CC_{E_i}(R) \cap C^3_{E_i}(D') = \emptyset \), and therefore \( (s_i, t_i) \not= CC_{E_i}(R) \), so that \( \sigma' \) cannot be an \( E \)-unifier of \( (\preceq, E_i, s_i \approx t_i) \).

The constraint Lazy BC is implemented in lines 8–10 in Alg. 3.

Minimisation. Greedy systematic minimisation of disequality sets \( D \) is described in Alg. 4, which successively attempts to remove elements \( dp \) from \( D \), but preserving \( (s, t) \in C^3_{E_i}(D) \). Certain disequalities \( s \sigma \not= t \sigma \) are known to hold under any substitution \( \sigma \), and are handled using a special set \( BaseD \) and kept in \( D \):

\[
BaseD = \bigcap_{\sigma \text{ a substitution} \atop \forall X \in V. \ X \sigma \preceq X} \{ (a, b) \in S^2 \mid a \sigma \not= b \sigma \}
\]

\( BaseD \) can easily be derived from \( \preceq \). Elimination of disequalities from \( BaseD \) is not helpful, since such disequalities are already implied by the SAT DOMAIN constraint; at the same time, they are useful as input for CCC.

Example 13. We consider again \( (\preceq, \{(E_1, e_1), (E_2, e_2)\}) \) from Example 4, which is solved by the run of Alg. 3 shown in Table 2. Note that various executions

<table>
<thead>
<tr>
<th>Candidate ( \sigma )</th>
<th>((E_1, e_1)) ((E_2, e_2))</th>
<th>Minimised set ( D' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( X \mapsto X, Y \mapsto Y, U \mapsto U, V \mapsto V )</td>
<td>( \times )</td>
<td>( { (Y, o_4), (V, o_4) } \cup BaseD )</td>
</tr>
<tr>
<td>2: ( X \mapsto X, Y \mapsto Y, U \mapsto U, V \mapsto o_4 )</td>
<td>( \times )</td>
<td>( { (Y, o_4), (U, o_4) } \cup BaseD )</td>
</tr>
<tr>
<td>3: ( X \mapsto X, Y \mapsto o_4, U \mapsto U, V \mapsto V )</td>
<td>( \times )</td>
<td>( { (U, o_4), (V, o_4) } \cup BaseD )</td>
</tr>
<tr>
<td>4: ( X \mapsto X, Y \mapsto o_4, U \mapsto U, V \mapsto o_4 )</td>
<td>( \times )</td>
<td>( { (U, o_4), (U, o_3) } \cup BaseD )</td>
</tr>
<tr>
<td>5: ( X \mapsto X, Y \mapsto o_4, U \mapsto o_3, V \mapsto o_4 )</td>
<td>( \times )</td>
<td>( { (X, Y), (Y, a), (Y, b), (Y, o_1), (Y, o_2), (U, o_4) } \cup BaseD )</td>
</tr>
<tr>
<td>6: ( X \mapsto o_4, Y \mapsto o_4, U \mapsto o_3, V \mapsto o_4 )</td>
<td>( \checkmark )</td>
<td>( { (X, o_2), (Y, o_2) } \cup BaseD )</td>
</tr>
<tr>
<td>7: ( X \mapsto o_2, Y \mapsto o_1, U \mapsto o_3, V \mapsto o_4 )</td>
<td>( \checkmark )</td>
<td>( { (Y, o_2), (V, o_2) } \cup BaseD )</td>
</tr>
<tr>
<td>8: ( X \mapsto o_1, Y \mapsto o_2, U \mapsto o_3, V \mapsto o_4 )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>
exist, since the sets $D'$ and the candidates $\sigma$ are not uniquely determined. Sets $D'$ directly translate to blocking constraints, for instance $\{(Y,o_4),(V,o_4)\} \cup BaseD$ is encoded as $v_Y = v_{o_4} \lor v_V = v_{o_4} \lor \cdots$. In iterations 1–5, the subproblem $(\preceq,E_1,e_1)$ is violated, and used to generate a blocking constraints; in 6–7, $(\preceq,E_2,e_2)$ is used. It can be observed that the algorithm is able to derive very concise blocking constraints, and quickly focuses on interesting assignments.

### 7 Experiments

We implemented both procedures as described in Sect. 4 and Sect. 6 and integrated them into the ePrincess theorem prover (based on [10]) using the calculus presented in [2]. The Sat4j solver was used to reason about the propositional encoding used in the procedures. To measure the performance of the two methods, we used randomly selected benchmarks from TPTP v.6.1.0 to generate BREU problems: when constructing a proof for a TPTP problem, ePrincess repeatedly extracts and attempts to solve BREU problems in order to close the constructed proof. ePrincess was instrumented to output and collected those BREU problems, so that altogether 6626 instances were in the end available for benchmarking. Those 6626 BREU problems were then separately processed by the Table and Lazy procedure, with a timeout of 60s.
7.1 Results and Discussion

The two procedures can handle most of the BREU problems generated. Table 3 tells us that the table procedure can solve all but three, while the lazy timeouts on slightly above 50. However, the three problems which the table method could not handle where all solved by the lazy method. The fact that almost all BREU problems could be solved indicates the efficiency of the two BREU procedures, but also that the BREU problems generated by ePrincess are not excessively large (which can be considered a strength of the calculus implemented by ePrincess [2]).

The cactus plot in Fig. 2 shows the distribution of runtime needed by either procedure to solve the BREU problems. It can be observed that more than half of the problems can be solved in less than 0.1s, and most of the problems in less than 1s. Fig. 3 shows that increasing complexity of BREU problems \((\leq, (E_i, e_i)_{i=1}^n)\), measured in terms of the maximum number of equations in any BREU subproblem \((E_i, e_i)\), also leads to increased solving time. The graph illustrates that the lazy procedure is more sensitive to this form of complexity than the table procedure. The high runtime for equation count \(> 35\) corresponds to timeouts. In contrast, we found that neither procedure is very sensitive to the number of sub-problems that a BREU problem consists of.

From Fig. 2 and Fig. 3, it can be seen that the table procedure is on average a bit faster than the lazy procedure. The scatter plot in Fig. 4 gives a more detailed comparison of runtime, and shows that the correlation of runtime of the procedures is in fact quite weak, but there is a slight trend towards shorter
runtime of the table method. Note that this is a comparison between procedures for solving BREU problems, for an evaluation of the overall performance of ePRINCESS on TPTP problems we refer the reader to [2].

On average, the lazy procedure produces 4.3 blocking clauses before finding an $E$-unifier, or proving that no unifier exists. The major bottleneck of the lazy method lies in the minimisation step of blocking constraints. The procedure spends most of its time in this part, and could be improved by creating a more efficient algorithm for CCC. For the table method, most of the runtime is spent in SAT solving, in particular in calls concluding with UNSAT.

8 Conclusion

In this paper we have presented two different procedures for solving the BREU problem. Both of them are shown to be efficient and usable in an automated theorem proving environment. Apart from further improving the proposed procedures, in future work we plan to consider the combination of BREU with other theories, in particular arithmetic.

Acknowledgements We thank the anonymous referees for helpful feedback.

References

A Proof of Lemma 9

Proof. We can first observe that \( \bar{c}_i(j) \leq j \) for every \( i \in \{1, \ldots, n\} \) and \( j \in \{1, \ldots, m\} \), due to Lem. 8, and because the first column has this property.

In the whole proof, we write \( R_\sigma = CC_E\{(t, t\sigma) \mid t \in S\} \). We show the two directions of the lemma: \( ER(\bar{c}_n) \subseteq R_\sigma \) (soundness), and \( R_\sigma \subseteq ER(\bar{c}_n) \) (completeness).

**Soundness** By well-founded induction over pairs \((i, j) \in \mathbb{N}^2\), with the lexicographic ordering \((i, j) < (i', j') \iff i < i' \lor (i = i' \land j < j')\), we show the property

\[
(Ind^{-1}(j), Ind^{-1}(\bar{c}_i(j))) \in R_\sigma.
\]  

From this soundness follows directly, because \( R_\sigma \) is an equivalence relation. Suppose (3) holds for all pairs \((i', j')\) with \((i', j') < (i, j)\).

- If \( i = 1 \), then \( \bar{c}_i(j) = Ind(Ind^{-1}(j)\sigma) \) by Def. 5, that means \( Ind^{-1}(\bar{c}_i(j)) = Ind^{-1}(j)\sigma \), and (3) holds by construction.

- For \( i > 1 \), according to Def. 5 there are different cases:

1. \( \bar{c}_{i-1}(j) \neq j \), which implies \( \bar{c}_{i-1}(j) < j \) and \( \bar{c}_i(j) = \bar{c}_i(\bar{c}_{i-1}(j)) \).
   
   \( \Rightarrow \) Through induction, we know \( (Ind^{-1}(j), Ind^{-1}(\bar{c}_{i-1}(j))) \in R_\sigma \) and \( (Ind^{-1}(\bar{c}_i(j)), Ind^{-1}(\bar{c}_i(\bar{c}_{i-1}(j)))) \in R_\sigma \). It is then also the case that \( (Ind^{-1}(j), Ind^{-1}(\bar{c}_i(\bar{c}_{i-1}(j)))) \in R_\sigma \) because \( R_\sigma \) is an ER.

2. \( \bar{c}_{i-1}(j) = j \), and
   
   (a) \( \bar{c}_i(j) = \bar{c}_i(k) \) with \( k < j \), and there are equations \( f(a_1, \ldots, a_l) \approx b, f(a'_1, \ldots, a'_l) \approx b' \in E \) such that \( \bar{c}_{i-1}(Ind(a_1)) = \bar{c}_{i-1}(Ind(a'_1)), \ldots, \bar{c}_{i-1}(Ind(a_l)) = \bar{c}_{i-1}(Ind(a'_l)) \), and \( \bar{c}_{i-1}(Ind(b)) = j, \bar{c}_i(Ind(b')) = k \).
   
   \( \Rightarrow \) Through the induction, and because \( R_\sigma \) is an ER that is consistent with the equations \( E \), we know that \( (b, b') \in R_\sigma \). Because of \( \bar{c}_{i-1}(Ind(b)) = j \) and induction, we furthermore know

\[
(Ind^{-1}(Ind(b)), Ind^{-1}(\bar{c}_{i-1}(Ind(b)))) \in R_\sigma
\]

and similarly \( (b', Ind^{-1}(k)) \in R_\sigma \). Because \( R_\sigma \) is an ER, this altogether implies \( (Ind^{-1}(j), Ind^{-1}(k)) \in R_\sigma \).

Due to \( k < j \), we finally have \( (Ind^{-1}(k), Ind^{-1}(\bar{c}_i(k))) \in R_\sigma \) by induction, and therefore \( (Ind^{-1}(j), Ind^{-1}(\bar{c}_i(k))) \in R_\sigma \).

(b) no such pair of equations exists, and \( \bar{c}_i(j) = j \)

\( \Rightarrow \) It follows that \( (Ind^{-1}(j), Ind^{-1}(j)) \in R_\sigma \) because \( R_\sigma \) is an ER.

**Completeness** Observe that all columns \( \bar{c}_i \) with \( i > 1 \) are idempotent in the following sense: if \( \bar{c}_i(j) \neq j \), then \( \bar{c}_i(\bar{c}_i(j)) = \bar{c}_i(j) \) (this follows directly from Def. 5). This implies that \( ER(\bar{c}_n) = \{(s, t) \in S^2 \mid \bar{c}_n(Ind(s)) = \bar{c}_n(Ind(t))\} \). Completeness of a table then implies that \( ER(\bar{c}_n) \) is also a fixed-point of the function \( CC_E \) defining the congruence closure function \( CC_E \). Because \( R_\sigma = CC_E\{(t, t\sigma) \mid t \in S\} \) is the least fixed-point of \( CC_E \) (that includes \( \{(t, t\sigma) \mid t \in S\}\)), we can conclude \( R_\sigma \subseteq ER(\bar{c}_n) \). \( \square \)
B Proof of Theorem 10

We first show a weaker lemma:

**Lemma 14.** If $R \subseteq S^2$ is an ER, then $C_E^3(R^C)^C$ is an ER as well.

**Proof.** Consider a sequence $D_0, D_1, D_2, \ldots$ of relations over $S$, defined by $D_0 = R^C$; and for each $i \in \mathbb{N}$, if there is some pair $(b, b') \in D_i$ such s.t. are equations $f(\bar{a}) \approx b, f(\bar{a}') \approx b' \in E$ and $\{(\bar{a}, \bar{a}')\} \cap D_i = \emptyset$, set

$$D_{i+1} = D_i \setminus \{(c, c'), (c', c) \mid c, c' \in S \text{ such that } (b, c) \notin D_i \text{ and } (b', c') \notin D_i\}.$$ 

Otherwise (no such pair $(b, b')$ exists), set $D_{i+1} = D_i$.

We derive the following properties about the sequence:

1. the sequence is decreasing, and since $S$ is finite the sequence eventually becomes constant. That means that there is some $n \in \mathbb{N}$ s.t. for all $n' \geq n$ we have $D_n = D_n$.
2. by induction over $i$, show that each $D_i$ is an ER.
3. by induction, show that $D_i \supseteq C_E^3(R^C)$ for every $i \in \mathbb{N}$:
   - clearly, $D_0 = R^C \supseteq C_E^3(R^C)$.
   - assume $D_i \supseteq C_E^3(R^C)$, and conclude that $D_{i+1} \supseteq C_E^{3,1}(D_i) \supseteq C_E^{3,1}(C_E^3(R^C)) = C_E^3(R^C)$.

4. for the $n$ from point 1, we also have $D_n \subseteq C_E^3(R^C)$, because $D_n$ is a fixed-point of $C_E^{3,1}$ and $D_n \subseteq R^C$, and $C_E^3(R^C)$ is a greatest fixed-point. To see that $C_E^{3,1}(D_n) = D_n$, show that if $C_E^{3,1}(D_n) \neq D_n$ then also $D_{n+1} \neq D_n$.

   Altogether, we can conclude that $D_n$ is an ER, and $D_n = C_E^3(R^C)$. \hfill \qed

**Proof (Theorem 10).** We show the two directions $CC_E(R)^C \subseteq C_E^3(R^C)$ and $CC_E(R)^C \supseteq C_E^3(R^C)$ separately.

$\subseteq$: By proving that $CC_E(R)^C$ is a fixed-point of $C_E^{3,1}$ and $CC_E(R)^C \subseteq R^C$ (the latter of which holds because of $CC_E(R) \supseteq R$):

$$C_E^{3,1}(CC_E(R)^C)$$

$$= \left\{(c, c') \in CC_E(R)^C \mid c \neq c', \text{ and for all } f(\bar{a}) \approx b, f(\bar{a}') \approx b' \in E \text{ it holds that } CC_E(R)^C \cap Cl_{Eq}\{(\bar{a}, \bar{a}'), (b, c), (b', c')\} \neq \emptyset \right\}$$

$\subseteq$ $$\left\{(c, c') \in CC_E(R)^C \mid \text{for all } f(\bar{a}) \approx b, f(\bar{a}') \approx b' \in E \text{ it holds that } Cl_{Eq}\{(\bar{a}, \bar{a}'), (b, c), (b', c')\} \subseteq CC_E(R) \right\}$$

$\subseteq$ $$CC_E(R)^C$$

$\subseteq$ holds because $c \neq c'$ for all $(c, c') \in CC_E(R)^C$, since $CC_E(R)$ is an ER. For $\subseteq$, observe that if $Cl_{Eq}\{(\bar{a}, \bar{a}'), (b, c), (b', c')\} \subseteq CC_E(R)$, then also $(c, c') \in CC_E(R)$, contradicting the fact that $(c, c') \in CC_E(R)^C$. 


\[ \begin{align*}
\exists: & \text{ By proving that } C^3_E(R^C)^C \text{ is a fixed-point of } CC^1_E, \text{ and } R \subseteq C^3_E(R^C)^C \iff R^C \supseteq C^3_E(R^C): \\
& C^1_E(C^3_E(R^C)^C) \\
& = Cl_Eq \left( C^3_E(R^C)^C \cup \left\{ (b, b') \mid \text{ there are } f(\bar{a}) \approx b, f(\bar{a}') \approx b' \in E \text{ with } (\bar{a}, \bar{a}') \in C^3_E(R^C)^C \right\} \right) \\
& \overset{(*)}{=} Cl_Eq \left( C^3_E(R^C)^C \cup \left\{ (b, b') \mid \text{ there are } f(\bar{a}) \approx b, f(\bar{a}') \approx b' \in E \right\} \right) \\
& \overset{(**)}{=} Cl_Eq \left( C^3_E(R^C)^C \right) \\
& \overset{(*)}{=} C^3_E(R^C)^C
\end{align*} \]

(*) hold because \( C^3_E(R^C)^C \) is an ER by Lem. 14. For (**), suppose there is a pair \((b, b') \not\in C^3_E(R^C)^C\) with \( Cl_Eq\{(\bar{a}, \bar{a}'), (b, b), (b', b')\} \subseteq C^3_E(R^C)^C \). Then also \((b, b') \in C^3_E(R^C)^C\) and \( C^3_E(R^C)^C \cap Cl_Eq\{(\bar{a}, \bar{a}'), (b, b), (b', b')\} = \emptyset \), and by definition \((b, b') \not\in C^3^1_E(C^3_E(R^C))\), contradicting the fact that \( C^3_E(R^C)^C \) is a fixed-point of \( C^3^1_E \). \( \square \)


c C Table procedure algorithm

\textbf{Input:} BREU problem $B$ with subproblems $P = \{p_1, \ldots, p_n\}$

1: $A \leftarrow \emptyset$  \hspace{0.5cm} \text{// Active Problems}
2: Add domain constraints (SAT DOMAIN)
3: \textbf{while} $A \neq P$ \textbf{do}
4: \hspace{0.5cm} \textbf{if} $\text{solver.isSat()}$ \textbf{then}
5: \hspace{1cm} $\sigma \leftarrow \text{solver.model()}$
6: \hspace{1cm} \textbf{if} $\sigma$ solves all sub-problems \textbf{then}
7: \hspace{1.5cm} \textbf{return} SAT
8: \hspace{1cm} \textbf{else}
9: \hspace{1.5cm} Let $p$ be an unsolved problem
10: \hspace{1.5cm} $A \leftarrow A \cup \{p\}$
11: \hspace{1.5cm} Add constraints for $p$ (TABLE CP, INIT, INT, GOAL)
12: \hspace{0.5cm} \textbf{end if}
13: \hspace{0.5cm} \textbf{else}
14: \hspace{1cm} \textbf{for all} $p \in A$ \textbf{do}
15: \hspace{1.5cm} Remove goal constraints for $p$ (TABLE GOAL)
16: \hspace{1.5cm} Add incompletion constraint for $p$ (TABLE INCOMP)
17: \hspace{1cm} \textbf{end for}
18: \hspace{1cm} \textbf{if} $\text{solver.isSat()}$ \textbf{then}
19: \hspace{1.5cm} \textbf{for all} $p \in A$ \textbf{do}
20: \hspace{1.5cm} Remove incompletion constraint for $p$ (TABLE INCOMP)
21: \hspace{1.5cm} Add additional column for $p$ (TABLE INT)
22: \hspace{1.5cm} Add goal constraint $p$ (TABLE GOAL)
23: \hspace{1cm} \textbf{end for}
24: \hspace{1cm} \textbf{else}
25: \hspace{1.5cm} \textbf{return} UNSAT
26: \hspace{1cm} \textbf{end if}
27: \hspace{0.5cm} \textbf{end if}
28: \hspace{0.5cm} \textbf{end while}
29: \textbf{return} UNSAT

\textbf{Algorithm 5:} The table procedure for a Simultaneous BREU problem
Paper III
Algebraic Polynomial-based Synthesis for Abstract Boolean Network Analysis

Peter Backeman$^{1,2}$, Sara-Jane Dunn$^2$, Boyan Yordanov$^2$, and Christoph M. Wintersteiger$^2$

$^1$ Uppsala University, Uppsala, Sweden  
$^2$ Microsoft Research

Abstract. Function synthesis is the problem of automatically constructing functions that fulfil a given specification. Using templates to limit the form of those functions is a popular way of reducing the search-space while still allowing interesting functions to be found. We present an investigation of restrictions to templates over Boolean functions of polynomial shape, based on their algebraic normal form. These polynomials are then lazily created in such a way that completeness of the search is still guaranteed, while performance is improved. This method is then implemented using an SMT-solver (Z3) and illustrated on a biological problem, the goal of which is to synthesise (Boolean) gene regulatory networks that capture specifications derived from experimental measurements.

1 Introduction

Synthesis techniques aim towards the automated construction of correct-by-design systems from specifications of desired behaviour, often expressed as temporal logic specifications [14]. These techniques have been successfully applied in a number of fields, including software [7, 18] and hardware synthesis [1], but also in biology (e.g., [11, 5, 10]). For biological applications, and more broadly in the field of computational science, the construction of computational models that reproduce known or observed behaviour of a natural system is a major challenge. However, this can be framed as a synthesis problem in which the specification is the experimentally-observed behaviour (as observed in a ‘wet-lab’ setting), together with known or assumed constraints, and the goal is the construction of a model that is consistent with these observations and assumptions.

Abstraction techniques are often applied as part of computational modelling, both to describe the system at a sufficient level of detail, and to increase the runtime performance of synthesis algorithms. For example, the genetic regulation that arises via complex biochemical processes, which governs many cellular processes can be represented by Gene Regulatory Networks (GRNs). Further, these networks can be modelled as Boolean networks (BNs), under the assumption that each involved gene can be abstracted into active or inactive states [9]. Considering such a model as a transition system, a Boolean function defines the next state of each component in terms of the states of its regulators (activators
and repressors). The resulting models capture an abstract, qualitative representation of the detailed biochemical mechanisms involved, and have proven useful for studying different types of cellular behaviour, such as cell differentiation.

The challenge of constructing a BN for a given biological system lies in identifying the network topology, i.e., the interactions between components which comprise a model that reproduces all observed behaviour. To address this challenge, Dunn et al. [4] propose Abstract Boolean Networks (ABNs), which allow the construction of models where only some interactions are known, and putative ones are included only as optional interactions. An ABN thus describes a set of unique, concrete topologies, in which the optional interactions are instantiated as present or absent.

An SMT-based synthesis approach, implemented as part of the RE:IN tool [4], is used to enumerate individual, concrete BNs from the ABN, and to study the constrained interaction network, for example by identifying required interactions, which appear in every concrete model that satisfies the specification. While the RE:IN approach reveals important properties, the enumeration of all valid models from the ABN is infeasible due to the usually large number of concrete solutions capable of reproducing the observed behaviour. A naive follow-up approach lies in the enumeration of this large set of solutions and then to search for patterns in the results. Finally, RE:IN does not provide a concise representation of the ABN constrained against the experimental observations, which can be queried efficiently or analysed further, e.g., to develop new biological experiments for validation of the model.

An alternative strategy to the iterative enumeration of concrete BNs from a constrained ABN is to synthesise a template function, parametrised by unknown, optional interactions, that describes all concrete models consistent with the observations. Such a representation of all consistent models is easily interrogated to reveal dependencies between interactions and expose all different mechanisms capable of producing the observed behaviour. If sufficiently concise, this representation also serves as a valuable tool for understanding system properties and guiding further biological studies. For the specific application to ABNs, function synthesis focuses on restricted sets of Boolean functions, through the instantiation of parametrised function templates. However, the problem of choosing appropriate templates is non-trivial. Here, we focus on this problem and propose a lazy algorithm that combines templates in a sound and complete manner.

2 Related Work

Pnueli and Rosner were among the first to suggest a theoretical framework for program synthesis and accompanying algorithms, based on automata theory and temporal logic [14]. Since then, this and similar techniques have been used in many applications, with varying degrees of success. Examples include software synthesis, such as the recent work by Gulwani et al. [7] and Solar-Lezama [18], who both employ different algorithms and abstraction techniques based on template instantiation to reduce the vast space of functions that would otherwise
have to be investigated. It is interesting to note that in recent years, there has also been an increased interest in software synthesis from natural language specifications, such as by Gvero and Kuncak [8] and Raza et al. [15].

Hardware synthesis has been a subject of research for many years. Given that some systems are required to be finite-state, or have a very limited number of inputs and output, the resulting synthesis problems are of moderate degree and often practically feasible. To this end, various modelling, verification, and synthesis techniques are applied, many of them based on automata-theoretic principles like the various classes of automata used in synthesis of temporal logic specifications; recent examples include distributed synthesis algorithms by Chatterjee et al. [3] and robust system synthesis by Bloem et al. [2]. Other formal bases are of course considered as well, e.g., Asarin et al. [1] abstract the systems to be synthesised by a finite collection of linear systems. In computational biology there are a number of different approaches to synthesis problems for various specification logics. For instance, Kugler et al. [11] consider the synthesis of Live Sequence Charts. As mentioned, Dunn et al. [4] consider the synthesis of BNs to model gene regulatory networks, and others like Fisher et al. [5] consider techniques tailored toward the efficient reconstruction of BNs from large collections of single-cell experimental data. Köksal et al. [10] propose a modelling and specification language, as well as an embedding thereof into Scala, and they describe an efficient synthesiser able to find a BN for cell fate determination of C. elegans, based on wet-lab mutation experiments.

The synthesis algorithm that we propose in Sec. 4.2 is strongly inspired by more general quantifier instantiation techniques in the wider area of automated theorem proving. Our algorithm can be seen as an instance of Model-Based Quantifier Instantiation [6] with automated (and complete) function template refinement [19]. However, we implicitly exploit other heuristic techniques, such as E-matching [12], and their implementation in the Z3 SMT solver [13]. Recently approaches have emerged to support synthesis algorithms that are integrated directly into a theorem prover or SMT solver, e.g., by Reynolds et al. [17, 16].

3 Background

Originally introduced by Kauffman [9], Boolean networks (BNs) represent one particular class of gene regulatory network models, where every gene is represented by a Boolean variable indicating a gene’s state as enabled or disabled.

For many (if not most) interactions between genes, it is not known whether they are indeed present in a particular system and if so whether they are positive or negative, and it is very much on the agenda of computational biology to discover and establish the type of these interactions. For our models to encompass such partial knowledge, we add to each interaction a label that is either optional or definite. Where the presence of an optional interaction is unknown, a definite is assumed to be present. We thus define abstract Boolean networks over a set $G := \{g_1, \ldots, g_n\}$ (in accordance with the semantics attached to them by Dunn et al. [4]), as an extension of non-deterministic finite state machines:
Definition 1 (Abstract Boolean Network). An Abstract Boolean Network (ABN) is a non-deterministic finite-state machine with

- finite state-space $Q = \mathbb{B}^n = \langle g_1, \ldots, g_n \rangle$
- empty input alphabet,
- set of initial states $Q_i \subseteq Q$,
- set of final states $F \subseteq Q$, and
- transition relation $\delta : Q \times Q = (q, (r_1(q), \ldots, r_n(q)))$, for all $q \in Q$ with an update function $r_i$ for each gene $g_i$.

ABN models allow us to incorporate uncertainty about the precise topology of a BN. Some of the many concrete BNs (CBNs) captured by an ABN may produce behaviour that is consistent with experimental observations of the biological system, while others might not. Therefore a set of experimentally-derived constraints is imposed over the behaviour of ABNs to exclude networks that are inconsistent with observations. We then say that an ABN satisfies an observation iff there exists at least one concrete BN that satisfies all such constraints.

Experimental observations are represented as reachability predicates over the states of some or all components at different time steps during the execution of the system. Every concrete execution of an ABN is a sequence of states $q_0, \ldots, q_k$. An observation is a set of concrete traces, which may be specified by a predicate that restricts the set of all traces, e.g., all traces starting in a particular starting state $q_0$ and that reach a final state $q_f$.

Definition 2 (Observation satisfaction). An ABN satisfies an observation iff there exists at least one concrete BN, for which there exists at least one trace that satisfies all conditions on initial, final, and other states.

Example 3. Consider the network in Fig. 1(a). This depicts a small ABN where each circle represents a gene, each solid edge represents a definite interaction, and each dashed edge represents an optional interaction (the bidirectional edge between Klf4 and Essrb represents two optional interactions). Fig. 1(b) shows all unique instantiations of optional interactions forming a CBN consistent with some set of observations (not shown here). Each column represents one CBN, where a green square indicates that the corresponding interaction was included, and a black square indicates that it was not. Fig. 1(c) shows the CBN corresponding to column 1 of Fig. 1(b) where only the first interaction is instantiated.

In previous work, Dunn et al. [4] aimed at enumerating and analysing a set of CBNs (as well as finding a minimal CBN) consistent with all experimental observations. They enumerated all possible network topologies, then formulated a query that posits that a network satisfies the observational constraints, and dispatched each of those queries to an SMT solver (Z3 [13]). The set of solutions obtained using this approach describes a set of CBNs consistent with all observations.
4 Polynomial-based Synthesis

We start by defining parametrised templates that enable us to transform the problem of synthesising (finding) a function into an equivalent problem of finding parameters or coefficient values. Note that our goal is not to synthesise or find update functions \( r_i \) of ABNs. Instead, the goal is to synthesise a concise description of the topology and the properties of ABNs (e.g., observational constraints), also encoded in Boolean functions. Such a template can be used to construct SMT queries to find the coefficients required to model certain properties of a system.

To this end, we observe that all Boolean functions have an Algebraic Normal Form (ANF) polynomial, which provides a canonical representation of functions of \( n \) inputs in terms of \( 2^n \) coefficients (in the worst case). Further, this representation also allows us to order Boolean functions in a manner that enables us to define a class of abstract function refinement procedures. We begin by stating the basic completeness property of ANF.

**Theorem 4 (Folklore).** Every Boolean formula \( f \) over variables \( \{x_1, \ldots, x_n\} \) has an equivalent and unique Algebraic Normal Form (ANF) taking the form of

\[
f(x_1, \ldots, x_n) = a_0 \oplus (a_1 \land x_1) \oplus \cdots \oplus (a_n \land x_n) \oplus (a_{1,2} \land x_1 \land x_2) \oplus \cdots \oplus (a_{n-1,n} \land x_{n-1} \land x_n) \oplus \cdots \oplus a_{1,\ldots,n} \land x_1 \land \cdots \land x_n
\]

for some \( a_1, \ldots, a_{1,\ldots,n} \in \{0, 1\} \) where \( \oplus \) is the exclusive-or operator.

We chose this representation of Boolean functions, because it allows us to easily identify and to specify some simple sub-classes of functions; for instance, constants are indeed just a single constant term, linear functions contain only
monomials, etc. Mathematically, any other representation of Boolean functions may be equivalent, but we suspect that presenting functions in ANF may help computational biologists to interpret our results. To make this more explicit, we define algebraic polynomials as a symbolic representation of Boolean functions:

**Definition 5.** Let \( X = \{x_1, \ldots, x_n\} \) be a set of Boolean variables. A Boolean formula \( p \) over \( X \), i.e., \( p(x_1, \ldots, x_n) \), is an algebraic normal form polynomial (or simply algebraic polynomial) if it is of the form

\[
p(x_1, \ldots, x_n) = c_0 \oplus t_1 \oplus \cdots \oplus t_k
\]

where \( k < 2^n \), each \( t_i \) is distinct from all others, and of the form \( c_i \land \bigwedge_{i=0}^{\left|X_j\right|} x_i \), where \( X_j \) is the \( j \)-th subset of \( 2^X \) and \( x_i \) is the \( l \)-th item in \( X_j \), and \( \{c_i \mid 0 \leq i < 2^n\} \) is a set of distinct Boolean constants which we call the ‘coefficients’.

An algebraic polynomial is conveniently described as a subset of the ANF monomials in its representation, and it represents a set of Boolean functions (those representable by concrete choices of values for the coefficients \( c_i \)). This definition also allows us to order functions based on the lexicographical ordering of the coefficients. The maximal polynomial is the one equivalent to the \((2^n - 1)\)-monomial ANF (with all coefficients \( c_i = 1 \)).

**Lemma 6.** For every non-maximal algebraic polynomial \( p \), there is a polynomial \( p' \) s.t. every monomial in \( p \) is included in \( p' \).

### 4.1 Searching for Boolean Functions

Given that we have established an ordering of ANF polynomials, we are now able to define directed search strategies that are designed to exhaust a particular class of functions. The search for a particular function representable within an algebraic polynomial is ultimately performed by checking a propositional formula for satisfiability (in practice by an SMT solver), such that a satisfying assignment to all \( c_i \) identifies a specific function.

**Example 7.** The algebraic polynomial \( c \oplus (c_1 \land x_1 \land x_2) \) corresponds to the Boolean functions \( \{f(x_1, x_2) = 0, f(x_1, x_2) = 1, f(x_1, x_2) = x_1 \land x_2, f(x_1, x_2) = 1 \oplus (x_1 \land x_2)\} \).

Looking at Ex. 7, a query could be posted to a SMT solver that checks whether any of the functions corresponding to the given ANF is equivalent to a sought after Boolean function \( (\phi(\overline{x})) \). Such a query could be on the following form:

\[
\varphi = \exists c, c_1 \forall \overline{x}. \phi(x) \leftrightarrow c \oplus (c_1 \land x_1 \land x_2)
\]

This query will only consider two of the arguments in \( \overline{x} \), this might of course not be enough, in which case a larger ANF must be tried. The key to efficiency lies in knowing which monomials to include in the ANF to make it feasible to find suitable coefficients quickly.
The universal quantifier is necessary to ensure the ANF instantiated with coefficients behaves like the given function. It could be eliminated by using enumeration of the whole domain (or partially, using some kind of bounded model checking), this has not been further developed at this point. However, this does not ensure a quantifier-free formula, since $(\forall x)$ might contain quantifiers (which is the case in many applications, including ABN synthesis).

Lem. 6 tells us that every Boolean function represented by $p$ has an equivalent representation in some $p' \succ p$. If $p$ is maximal, then all Boolean functions are included. We may therefore traverse any set of algebraic polynomials to search for a particular function, $f$, that has a representation in $p$, yet we are still able to guarantee to find any Boolean function if we include (at least) the maximal polynomial. In practice, this is of course (too) expensive. Often, a much smaller set of functions, or a set of smaller functions, is sufficient.

This leads us to a simple, sound, and complete search procedure: start with any algebraic polynomial $p$ and see if it can represent $f$. If it cannot, then pick the next polynomial $p' \succ p$ according to the ordering and retry. This is repeated until a suitable polynomial has been found. The process is guaranteed to terminate since there is only a finite number of algebraic polynomials over a fixed set of variables, and the maximal algebraic polynomial is guaranteed to be able to represent $f$.

4.2 FIND and FIX

Let $\varphi(x)$ be a Boolean function and suppose that the problem is to synthesise another function $f_{\varphi}(x) = \varphi(x)$, i.e., for each assignment to $x \in X$, we want $f_{\varphi}(x)$ to compute the same result as $\varphi(x)$, but preferably in a smaller or more ‘general’ representation. The idea is to slice the domain of assignments into subsets that are easily solvable, have small representations, and ultimately yield a description of $f_{\varphi}$ that pairs predicates (functions that map inputs to 1 or 0), identifying parts of functions, with (function)-terms that efficiently describe the desired functional relationships between variables in $\varphi$. Formally, we are looking for a model, that is a formula equivalent to $\varphi(x)$ but in a different form or representation.

We propose an iterative procedure: we begin by stating the formula to be modelled, $f_{\varphi}$, and we construct an initial arbitrary model function $m = 0$, for $f_{\varphi}(x)$. We then search for a counter-example, $c$, to the current model, $m$, i.e. an assignment to the variables $x$ such that $m(x) \neq \varphi(x)$. Then a predicate FIND and a function FIX is constructed s.t. $\text{FIND}(c) = 1 \land \forall x. \text{FIND}(x) \Rightarrow m(x) \neq \varphi(x)$ and $\forall x. \text{FIND}(x) \Rightarrow (\text{FIX}(x) = \varphi(x))$. A new, improved model is then easily constructed from those parts: $m'_{\varphi}(x) = \text{ITE}(\text{FIND}(x), \text{FIX}(x), m(x))$, where $\text{ITE}$ is the if-then-else operator. The algorithm is summarised in Alg. 1.

This procedure can be relaxed in two ways while still maintaining completeness, approximating either the FIND or the FIX predicate. This means that in each iteration a counter-example $c$ is generated, w.r.t. the current model $m$. The task is then to “patch” the current model s.t. $m(c)$ now computes the correct value. We synthesise FIND and FIX, where FIND must identify at least one counter-example $c$, but is allowed to be imprecise elsewhere. The new model
m(x) ← 0;
while ∃x.m(x) ≠ φ(x) do
  c ← x s.t. m(x) ≠ φ(x);
  FD ← fd s.t. fd(c) ∧ (∀x.fd(x) ⇒ m(x) ≠ φ(x));
  FX ← fx s.t. ∀x.fd(x) ⇒ fx(x) = φ(x);
  m(x) ← ITE(fd(x), fx(x), m(x));
end

Algorithm 1: The FIND and FIX loop

will be correct for c (per construction) and for no other function points will it
be incorrect where it previously was correct (FIX being exact). Therefore, the
number of inputs for which the new model is correct, will be guaranteed to be
larger than for the older model (by at least one). The algorithm is described in
Alg. 2. The reasoning for approximate FIX is similar and summarised by Alg. 3.

m(x) ← 0
while ∃c.m(c) ≠ φ(c) do
  FD ← fd s.t. fd(c)
  FX ← fx s.t. ∀x.
  fd(x) ⇒ fx(x) = φ(x)
  m(x) ← ITE(fd(x), fx(x), m(x))
end

Algorithm 2: Approximate FIND

m(x) ← 0
while ∃c.m(c) ≠ φ(c) do
  FD ← fd s.t. fd(c)
  ∀x.fd(x) ⇒ m(x) ≠ φ(x)
  FX ← fx s.t. fx(x) = φ(x)
  m(x) ← ITE(fd(x), fx(x), m(x))
end

Algorithm 3: Approximate FIX

4.3 Function Synthesis Reasoning

Combining FIND and FIX with algebraic polynomial function templates leads to
a Boolean function synthesis procedure, as outlined in Alg. 4. It first looks for a

m ← 0;
while ∃ce . m(ce) ≠ φ(ce) do
  p ← findRepresentingPolynomial(ce, p₀);
  while ∃k . p(k) ∧ (φ(k) = 0) do
    p ← nextRepresentingPolynomial(k, p);
    m ← ITE(p, 1, m);
  end
end

Algorithm 4: Approx. FIND, Exact FIX-model search for formula φ
counter-example \( ce \) to the initial model \( m \).\(^3\) \( \text{findRepresentingPolynomial}(ce, p_0) \) generates an algebraic polynomial that contains \( ce \) (starting from an arbitrary but fixed initial polynomial \( p_0 \)) and \( \text{nextRepresentingPolynomial}(k, p) \) generates an algebraic polynomial greater than \( p \) that contains \( k \).

In \( \text{findRepresentingPolynomial} \) as well as \( \text{nextRepresentingPolynomial} \) there is a lot of room for heuristics. A simple way of finding the next representing polynomial is by considering all possible ANFs and selecting the next one according to a lexicographical ordering (which is easy to construct) until a suitable is found. This will often be very slow and, so better heuristics are necessary. Our prototype implementation (FSREIN) implements a method based on identifying relevant variables from unsatisfiable core reasoning, and only adds monomials over those variables that appear in the core. An important insight here is that any heuristic that always produces larger polynomials is guaranteed to be complete.

5 Comparison

Using the techniques presented in Sec. 4.3 a tool was implemented, FSREIN, which is capable of synthesizing a Boolean function describing all concrete Boolean network, for a given ABN, that are valid with respect to some observational constraints. We present in this section a comparison between this and the state-of-the-art tool for ABN reasoning (RE:IN [4]). Both tools, RE:IN and FSREIN, are sound and complete, and they will find all valid networks given sufficient time. Both tools have exponential worst-case runtime complexity.

Reduced Models. Consider the problem presented earlier in Fig. 1. The ABN shown in Fig. 1(a) is a module of the more extensive gene regulatory network that governs the mouse pluripotent embryonic stem cell (ESC) state dynamics. Experimental measurement of gene expression under different inputs allows us to define experimental observations of the pluripotency network, which we impose as constraints. Using RE:IN to enumerate consistent CBNs yields the table shown in Fig. 1(b). Eight unique CBNs are consistent with the experiments. From the table it is quite easy to see that the interaction \( \text{Esrrb} \rightarrow \text{Oct4} \) is part of every solution, and it is thus a required interaction. The output of FSREIN are the two \text{find}-functions

\[
(\text{Esrrb} \rightarrow \text{Oct4}) \text{ OR } ((\text{not Esrrb} \rightarrow \text{Oct4}) = (\text{Esrrb} \rightarrow \text{Klfnn4 and Esrrb} \rightarrow \text{Oct4}))
\]

where the corresponding \text{fix} functions are both 1, from which it is trivial to see (or to deduce) that \( \text{Esrrb} \rightarrow \text{Oct4} \) is a required interaction.

For another, larger example (16 genes, 8 optional interactions, 2 experiments) a similar result was found. When RE:IN was executed it enumerated 96 solutions, while FSREIN produces the following functions:

\(^3\) In essence, this is the query RE:IN uses to enumerate solutions.
((Nanog $\rightarrow$ Sox2) AND (Klf2 $\rightarrow$ Oct4)) OR
((Sall4 $\rightarrow$ Sox2) AND (Klf2 $\rightarrow$ Oct4))

The running time for the FSREIN was in this instance three orders of magnitude slower than running the RE:IN-tool, which can of course be a major obstruction, but there is plenty of room to greatly improve the efficiency of the procedure. However, the major improvement is the readability of the model. The FSREIN model is easy to understand and clearly shows the interactions between genes that is in a more ‘natural’ form for computational biologists than the list of counter-examples obtained from RE:IN.

**Fig. 2.** A comparison of the model size produced by RE:IN and FSREIN.

**Benchmarks.** To compare more quantitatively, we crafted a set of 39 simple benchmark problems on which to test both tools. Fig. 2 summarises the results after running each of the tools, comparing the size of the model expressions (the number of enumerated models for RE:IN and the number of FIND and FIX-pairs for FSREIN) for each tool. Observe that FSREIN never produces a model larger than that of RE:IN, but it is interesting to see that in some cases it produces significantly smaller representations.

### 6 Conclusion

In this paper we present a function synthesis approach based on the FIND and FIX-strategy, using algebraic polynomials to describe sets of Boolean functions to be synthesised. We apply this technique to problems that arise in computational biology; specifically the abstract Boolean network synthesis problem, and show that in small instances it is able to produce functions describing the set of concrete Boolean networks that are consistent with the abstract network, and with experimental observations obtained through wet-lab experiments. The average runtime of our implementations is not better than the state-of-the-art, but it has the benefit of removing the burden of having to analyse a large set of enumerated models to find patterns within the data. Rather, the functions describing the networks are kept in a compact and expressive representation, which is much more amenable to scientific testing and interpretation.

**References**


Recent licentiate theses from the Department of Information Technology

2016-011  Andreas Svensson:  *Learning Probabilistic Models of Dynamical Phenomena Using Particle Filters*

2016-010  Aleksandar Zeljić:  *Approximations and Abstractions for Reasoning about Machine Arithmetic*

2016-009  Timofey Mukha:  *Inflow Generation for Scale-Resolving Simulations of Turbulent Boundary Layers*

2016-008  Simon Sticko:  *Towards Higher Order Immersed Finite Elements for the Wave Equation*

2016-007  Volkan Cambazoglou:  *Protocol, Mobility and Adversary Models for the Verification of Security*

2016-006  Anton Axelsson:  *Context: The Abstract Term for the Concrete*

2016-005  Ida Bodin:  *Cognitive Work Analysis in Practice: Adaptation to Project Scope and Industrial Context*

2016-004  Kasun Hewage:  *Towards a Secure Synchronous Communication Architecture for Low-power Wireless Networks*

2016-003  Sven-Erik Ekström:  *A Vertex-Centered Discontinuous Galerkin Method for Flow Problems*

2016-002  Rubén Cubo:  *Mathematical Modeling for Optimization of Deep Brain Stimulation*

2016-001  Victor Shcherbakov:  *Radial Basis Function Methods for Pricing Multi-Asset Options*

2015-006  Hanna Holmgren:  *Towards Accurate Modeling of Moving Contact Lines*