THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

# Extending the Automated Reasoning Toolbox

ANN LILLIESTRÖM


CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019

Extending the Automated Reasoning Toolbox
ANN LILLIESTRÖM

ISBN 978-91-7905-205-8



Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4672
ISSN 0346-718X
Technical Report 178D
Department of Computer Science and Engineering
Research group: Functional Programming

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2019

# Abstract

Due to the semi-decidable nature of first-order logic, it can be desirable to address a wider range of problems than the standard ones of satisfiability and derivability. We extend the automated reasoning toolbox by introducing three new tools for analysing problems in first-order logic.

*Infinox* aims to show finite unsatisfiability, i.e. the absence of models with finite domains, and is a useful complement to finite model-finding. Infinox can also be used to reason about the relative sizes of model domains in sorted first-order logic.

*Monotonox* uses a novel analysis that can identify sorts with extendable domains, improving on well-known existing translations between sorted and unsorted logic. This enables reasoning tools for unsorted logic to tackle problems in sorted logic. Conversely, finite model finders benefit from sort information which Monotonox can add to unsorted problems.

*Equalox*, the third tool in our toolbox, can improve the performance of first-order provers on problems involving transitive relations. The insight is that first-order provers are poor at applying the transitivity axiom effectively, but that the problem can always be transformed to safely remove the transitivity axiom.

Finally, we explore the field of computational linguistics as an application of automated reasoning. The tool *Morfar* uses a constraint solver to analyse the morphology of an input language. The result is a novel automatic method for segmentation and labelling that works well even when there is very little training data available.

# List of publications

This thesis is based on the work contained in the following papers:

**Paper 1** Automated Inference of Finite Unsatisfiability. *Koen Claessen and Ann Lilliestr̈om. Journal of Automated Reasoning*, pages 111–132, Springer, 2011.

**Paper 2** Sort it Out with Monotonicity: Translating between Many-Sorted and Unsorted First-Order Logic. *Koen Claessen, Ann Lilliestr̈om and Nicholas Smallbone.* In *Proceedings of the 23rd International Conference on Automated Deduction*, pages 207–221, Springer, 2011.

**Paper 3** Handling Transitive Relations in First-Order Automated Reasoning. *Koen Claessen and Ann Lilliestr̈om.* In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*, pages 11–23, Springer, 2016.

**Paper 4** Inferring Morphological Rules from Small Examples using 0/1 Linear Programming. *Koen Claessen, Ann Lilliestr̈om and Nicholas Smallbone.* In *Proceedings of the 22nd Nordic Conference on Computational Linguistics*, pages 164–174, Linköping University Electronic Press, 2019.

# Contents

# Acknowledgements

I wish to thank my supervisor Koen Claessen for all the fun and inspiring discussions we have had, but also for his friendship, understanding, and for believing in me. Thank you for giving me the opportunity to start working with you and for convincing me to continue. Many thanks to my co-supervisor Wolfgang Ahrendt, whose support has meant a lot to me. I also wish to thank Aarne Ranta and the language technology group for welcoming me into the community and encouraging me to pursue my interest in natural language. A special thanks to Bengt Nordström who, during the writing of my bachelor's thesis many years ago, gave me the confidence to become a researcher. Thanks to my family, and to all my friends both inside and outside the department. Among them, I especially want to mention my officemate and co-author Nick, for combining entertaining discussions on logic with ridiculous amounts of cake, and for being an awesome friend. I also want to thank Sara, for always being there for me. Last but not least, thanks to Jean-Philippe, Irmeli and Miranda for absolutely everything.

# Introduction

## A motivating example

An automated reasoner's toolbox contains two basic kinds of tools: theorem provers and theorem disprovers. Let us demonstrate their use, and more importantly, their limitations, by means of an example. Below, we have formalised some laws that would typically hold for sets, involving the operations union and set difference (diff).

$$\mathsf{member}(X, \mathsf{empty}) \iff \mathsf{false} \tag{1}$$

$$\mathsf{member}(X, \mathsf{singleton}(Y)) \iff X = Y \tag{2}$$

$$\mathsf{member}(X, \mathsf{union}(A, B)) \iff \mathsf{member}(X, A) \vee \mathsf{member}(X, B) \tag{3}$$

$$\mathsf{member}(X, \mathsf{diff}(A, B)) \iff \mathsf{member}(X, A) \wedge \neg\mathsf{member}(X, B) \tag{4}$$

$$(\forall X(\mathsf{member}(X, A) \iff \mathsf{member}(X, B))) \implies A = B \tag{5}$$

Suppose that we wish to find out if, given the above laws, reassociating the operations union and diff yields the same result. We conjecture the following:

$$\mathsf{union}(A, \mathsf{diff}(B, C)) = \mathsf{diff}(\mathsf{union}(A, B), C) \tag{6}$$

Let us try an automated theorem prover to see if the conjecture can be proven from the theory. E [50] is an automated theorem prover, based on the superposition calculus, which is refutation-complete. In short, this means that E will eventually find a proof if there is one, unless limited by computer memory, time constraints or the user's impatience. When we put E to the task, it is still working on the problem after several minutes and no answer is reported back. How should we interpret this? Does it mean that the conjecture is false (E didn't find a proof because there isn't any), or does the theorem prover just need more time to complete a proof? There is no way to know, and we may be left with the same question even if we wait for a year, a lifetime or more.

We can instead try a different angle. Supposing that the conjecture is false, we could use a model finder to try to find a counter-model. A counter-model would serve as a counter-proof to the conjecture, showing that the property does not hold. We run the finite model finder Paradox on the problem, and after a few minutes it is still busy, but we have the following output:

```
Paradox, version 4.0, 2010-06-29.
+++ PROBLEM: sets.fof
Reading 'sets.fof' ...  OK
+++ SOLVING: sets.fof
domain size 1
domain size 2
domain size 3
domain size 4
domain size 5
domain size 6
domain size 7
domain size 8
domain size 9
domain size 10
domain size 11
```

What the above means is that Paradox has ruled out counter-models with domains of all sizes up to 10, and is currently trying to find a model of size 11. But we are not much wiser than before. Would the conjecture be provable if we gave E some more time? Would Paradox find a counter-model of more than 10 elements if we waited a little longer?

Because first-order logic is semi-decidable, theorem provers can be complete for theorems, but theorem disprovers can never be complete for non-theorems. In practice, this means that there will always be cases where it is impossible to find out whether or not a conjecture holds. While a theorem prover may always find a proof when one exists, it may not be able to do so in a reasonable time. It is situations like these that motivate the development of tools that can do more than proving and disproving. This thesis is devoted to extending the toolbox of automated reasoning.

The thesis presents four different tools. The common theme is the invention of new ways to formulate, encode or translate problems in order to tackle them with formal reasoning tools. The main focus of the first three tools is reasoning about formulas in first-order logic. The final tool in this thesis, *Morfar*, instead shows an application of automated reasoning to a problem in computational linguistics.

1. **Infinox** is a tool for first-order logic that can show that a conjecture has no finite counter-models.

2. **Monotonox** is a tool that translates sorted first-order logic into unsorted, so that unsorted theorem provers can solve sorted problems.

3. **Equalox** is a tool that improves the performance of first-order theorem provers on problems involving transitive relations, by transforming the problem to safely remove the transitivity axiom.

4. **Morfar** infers the morphological features of a natural language by encoding the morphology as a constraint problem, to be solved by an integer linear programming solver.

### Infinox

Infinox is a tool for first-order logic that specialises in showing finite (counter-)unsatisfiability, i.e. disproving the existence of finite counter-models. If Infinox shows that a problem is finitely counter-unsatisfiable, it means that it is either a theorem (no counter-model exists), or it has only infinite counter-models. The main purpose of the tool is to serve as a complement to finite model finders. If Infinox classifies a problem as finitely unsatisfiable, there is no point in searching for a finite model.

When we try Infinox on our example from above, it yields the following output:

```
Infinox, version 1.0, 2009-07-20.
+++ PROBLEM: sets.fof
Reading 'sets.fof' ...  OK
+++ SOLVING: sets.fof
InjNotSurj
t:  singleton(X)
r:  X = Y
+++ RESULT: FinitelyCounterUnsatisfiable
```

It tells us that any possible counter-models must be infinite, i.e. the problem is either a theorem or has an infinite counter-model. Although we still don't know whether the conjecture is true or not, Infinox has proved that no finite counter-model exists and thus there is no use in trying Paradox or other finite model finders. Even if given more time and resources, a finite model finder is not going to find a counter-proof. In this way, Infinox is a useful complement to finite model finding. But Infinox has more to tell us. Take a look again at the following lines of the output:

```
InjNotSurj
t:  singleton(X)
r:  X = Y
```

It says that Infinox has found a function that is injective and not surjective, namely singleton. (The line `"r:  X = Y"` tells us that it is injective and not surjective with respect to equality, as other relations are sometimes possible.) Because the theory implies the existence of a function with these properties, there cannot be any finite counter-models. We shall explain why:

- *Non-surjectivity* of singleton means that there exists some element $e$ such that for any $X$, $\mathsf{singleton}(X) \neq e$. In fact, it follows from axioms 1, 2 and 5 that empty is such an element.

$$\mathsf{singleton}(X) \neq \mathsf{empty} \tag{7}$$

- *Injectivity* of singleton means that it never maps two distinct elements to the same element, or equivalently:

$$\mathsf{singleton}(X) = \mathsf{singleton}(Y) \implies X = Y \tag{8}$$

Given the above properties of singleton, we can construct the following infinite sequence of elements:

$$\mathsf{singleton}^0(\mathsf{empty}), ..., \mathsf{singleton}^i(\mathsf{empty}), ... \tag{9}$$

Because singleton is injective, no two elements in the sequence can be equal to each other, as can be seen by the following argument: Suppose that two elements were the same. Then,

$$\mathsf{singleton}^m(\mathsf{empty}) = \mathsf{singleton}^n(\mathsf{empty}) \tag{10}$$

for two distinct $m$ and $n$, with $m > n$. By repeated application of injectivity, we get

$$\mathsf{singleton}^{m-n}(\mathsf{empty}) = \mathsf{empty} \tag{11}$$

which can be rewritten as:

$$\mathsf{singleton}(\mathsf{singleton}^{m-n-1}(\mathsf{empty})) = \mathsf{empty} \tag{12}$$

which contradicts the non-surjectivity constraint of 7. Thus, if there is a model of the problem, it must contain all of the elements in the sequence, and as a consequence be infinite. □

### Monotonox

Let us again consider the elements of the infinite sequence in 9. Any singleton set can be a member of another set. It looks like we have mixed up sets and elements in the definition. Is this what we intended? If our intention was to model a set theory similar to that of Zermelo-Fraenkel [5], where elements and sets are indistinguishable, the fact that a counter-model is necessarily infinite is not surprising, since the sets represent the natural numbers. But let us assume that we meant for sets and elements to be two distinct kinds of objects. In this case Infinox has helped us reveal a mistake in our definition: sets and elements should be of different sorts!

In our example above, we make no distinction between sets and elements, and any element of the model domain would act both as an element and as a set. Because elements and sets share the same domain, the number of elements and the number of sets are necessarily the same. (In our example, the shared domain has infinite size.)

In many-sorted first-order logic, the domain is partitioned into subdomains that each correspond to a sort. The arguments and return values of functions and predicates are each given a sort, and all variables come with a sort.

To express our problem in many-sorted first-order logic, we add the following sort signature to the original problem.

$$\mathsf{empty} : \mathsf{Set} \tag{13}$$
$$\mathsf{singleton} : \mathsf{Element} \to \mathsf{Set} \tag{14}$$
$$\mathsf{union} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \tag{15}$$
$$\mathsf{member} : \mathsf{Element} \to \mathsf{Set} \to \mathsf{Bool} \tag{16}$$
$$\mathsf{diff} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \tag{17}$$

In addition, the quantifiers in each equation must range over the appropriate sorts only. For example, equation 3 becomes:

$$\forall X : \mathsf{Element}.\, \forall A, B : \mathsf{Set}.$$
$$\mathsf{member}(X, \mathsf{union}(A, B)) \iff \mathsf{member}(X, A) \vee \mathsf{member}(X, B) \tag{18}$$

E, which supports sorted logic since version 2.0, again gets stuck on the new sorted problem. Because Paradox solves only problems in unsorted logic, we cannot use it to try to find a counter-model. But there is a way to get around this: many-sorted first-order logic can be reduced to unsorted first-order logic. A standard way to do this is to use sort predicates. This involves introducing a new unary predicate for every sort, letting any quantification over a sort be bounded by the predicate associated with the sort. It is also necessary to add axioms stating the return sort of each function. For example, using sort predicates, equation 18 translates to:

$$\mathsf{isElement}(X) \wedge \mathsf{isSet}(A) \wedge \mathsf{isSet}(B) \Longrightarrow$$
$$\mathsf{member}(X, \mathsf{union}(A, B)) \iff \mathsf{member}(X, A) \vee \mathsf{member}(X, B) \tag{19}$$

This technique introduces a lot of clutter, which can affect a theorem prover negatively, and is sometimes unnecessary. Monotonox, the next tool in our toolbox, can help us overcome this by a novel analysis which improves on existing well-known translations.

In short, we say that a sort is monotone in a given theory if, for any model of the theory, the domain of that sort can be made larger without affecting satisfiability. The result of the translation for monotone sorts turns out to be much simpler than for non-monotone sorts, as monotone sorts do not require sort predicates to preserve satisfiability. The monotonicity analysis exploits the fact that the only way to limit domain size is to use equality. For example, equality can be used to state that all elements must be equal to each other and thus that the domain size must be 1. When equality is not present, we can extend the domain by adding a new element that behaves identically to an already existing one. Monotone sorts can simply be removed in the translation, while non-monotone sorts still require special care. In Chapter 2, we describe the analysis in further detail and explain how it can be implemented using a SAT solver.

For now, let us see how Monotonox translates our problem:

$$\mathsf{member}(X, \mathsf{empty}) \iff \mathsf{false} \tag{20}$$

$$\begin{aligned}\mathsf{isElement}(X) \wedge \mathsf{isElement}(Y) \implies \\ \mathsf{member}(X, \mathsf{singleton}(Y)) \iff X = Y\end{aligned} \tag{21}$$

$$\mathsf{member}(X, \mathsf{union}(A, B)) \iff \mathsf{member}(X, A) \vee \mathsf{member}(X, B) \tag{22}$$

$$\mathsf{member}(X, \mathsf{diff}(A, B)) \iff \mathsf{member}(X, A) \wedge \neg\mathsf{member}(X, B) \tag{23}$$

$$\begin{aligned}\mathsf{isSet}(A) \wedge \mathsf{isSet}(B) \implies \\ (\forall X(\mathsf{member}(X, A) \iff \mathsf{member}(X, B))) \implies A = B\end{aligned} \tag{24}$$

Conjecture:

$$\mathsf{union}(A, \mathsf{diff}(B, C)) = \mathsf{diff}(\mathsf{union}(A, B), C) \tag{25}$$

In addition, there are sort axioms which specify the result sorts of each function:

$$\mathsf{isSet}(\mathsf{union}(X, Y)) \tag{26}$$

$$\mathsf{isSet}(\mathsf{diff}(X, Y)) \tag{27}$$

$$\mathsf{isSet}(\mathsf{empty}) \tag{28}$$

$$\mathsf{isSet}(\mathsf{singleton}(X)) \tag{29}$$

And axioms that make sure that the sorts are non-empty:

$$\exists A.\mathsf{isSet}(A) \tag{30}$$

$$\exists X.\mathsf{isElement}(X) \tag{31}$$

Sort predicates are needed for both sorts, which suggests that none of the sorts were found to be monotone. However, only two of the axioms, 21 and 24, both involving equality, use sort predicates to restrict the range of the quantifications. (In fact, axiom 24 does not need sort predicates either, as would be detected by a new addition to the scheme which is discussed as future work on page 16.) Monotonox can thus reduce the clutter that would have been present in standard sorted to unsorted translations.

Now we can try Paradox on the new version of the problem:

```
Paradox, version 4.0, 2010-06-29.
+++ PROBLEM: sorted_sets.fof
Reading 'sorted_sets.fof' ...  OK
+++ SOLVING: sorted_sets.fof
domain size 1
domain size 2
+++ RESULT: CounterSatisfiable
```

This time, Paradox finds a finite counter-model of size 2, proving the conjecture false. Indeed, a counter-example to the conjecture is given by

$$A = \mathsf{a}, B = \mathsf{empty}, C = \mathsf{a}$$

which can be verified given the following interpretation:

$$\begin{aligned} \mathsf{union}(\mathsf{a},\mathsf{a}) &= \mathsf{a} \\ \mathsf{union}(\mathsf{a},\mathsf{empty}) &= \mathsf{a} \\ \mathsf{union}(\mathsf{empty},\mathsf{a}) &= \mathsf{a} \\ \mathsf{union}(\mathsf{empty},\mathsf{empty}) &= \mathsf{empty} \end{aligned} \qquad \begin{aligned} \mathsf{diff}(\mathsf{a},\mathsf{a}) &= \mathsf{empty} \\ \mathsf{diff}(\mathsf{a},\mathsf{empty}) &= \mathsf{a} \\ \mathsf{diff}(\mathsf{empty},\mathsf{a}) &= \mathsf{empty} \\ \mathsf{diff}(\mathsf{empty},\mathsf{empty}) &= \mathsf{empty} \end{aligned}$$

$$\begin{aligned} \mathsf{singleton}(\mathsf{a}) &= \mathsf{a} \\ \mathsf{singleton}(\mathsf{empty}) &= \mathsf{a} \end{aligned} \qquad \begin{aligned} \mathsf{member}(\mathsf{a},\mathsf{a}) &\iff \mathsf{true} \\ \mathsf{member}(\mathsf{a},\mathsf{empty}) &\iff \mathsf{false} \\ \mathsf{member}(\mathsf{empty},\mathsf{a}) &\iff \mathsf{true} \\ \mathsf{member}(\mathsf{empty},\mathsf{empty}) &\iff \mathsf{false} \end{aligned}$$

$$\begin{aligned} \mathsf{isSet}(\mathsf{a}) &\iff \mathsf{true} \\ \mathsf{isSet}(\mathsf{empty}) &\iff \mathsf{true} \end{aligned} \qquad \begin{aligned} \mathsf{isElement}(\mathsf{a}) &\iff \mathsf{true} \\ \mathsf{isElement}(\mathsf{empty}) &\iff \mathsf{false} \end{aligned}$$

The counter-example consists of two sets (empty and a) and one element (a), where a stands for *either* a set *or* an element. Because empty is not an element, neither of singleton(empty), member(empty, a) or member(empty, empty) exist in the sorted problem, but when sorts are removed, we must include an interpretation for them to make the model well-defined. As these terms do not correspond to anything in the sorted problem, their values are in that regard unimportant. To obtain the sorted model, these terms are simply removed from the interpretation, and the model domain is separated into one domain for sets and one for elements.

In the example above, satisfiability is not affected by these "bogus terms", because in the interpretation above, every constant of the "wrong" sort mimics some constant of the "correct" sort. In some cases, e.g. axioms which contain equality, this does not work, as a mimicking element of the wrong sort would be exposed. In this case, the axiom must be guarded by a sort predicate. Chapter 2 describes how to analyse a problem to detect where sort predicates are needed.

We have seen an example that motivates the development of tools that can do more than proving and disproving. Infinox complements finite model finders by disproving the existence of finite models. It can also show the reason why there is no finite model, which may provide more intuition to the user. For example, when a finite (counter-)model is expected, Infinox can reveal mistakes in the problem definition. In our example, by analysing the injective and non-surjective function that was discovered by Infinox, we found that we had failed to separate elements and sets. After rewriting the problem to sorted first-order logic, separating the sorts, we could no longer apply the same tools as before. Monotonox efficiently translated the sorted problem back to unsorted logic, using a novel translation algorithm that introduces sort predicates only in the necessary places. Finally, Paradox found a counter-example of size 2, proving the conjecture false.

### Equalox

Let us look at a different example in the domain of sets. $\mathsf{union}(A, B)$ is defined as the smallest set containing both $A$ and $B$ as subsets (35, 36, 37). We axiomatise the reflexive, transitive and antisymmetric properties of the subset relation (32,33,34).

$$\mathsf{subset}(A, A) \tag{32}$$

$$\mathsf{subset}(A, B) \wedge \mathsf{subset}(B, C) \Longrightarrow \mathsf{subset}(A, C) \tag{33}$$

$$\mathsf{subset}(A, B) \wedge \mathsf{subset}(B, A) \Longrightarrow A = B \tag{34}$$

$$\mathsf{subset}(A, \mathsf{union}(A, B)) \tag{35}$$

$$\mathsf{subset}(B, \mathsf{union}(A, B)) \tag{36}$$

$$\mathsf{subset}(A, C) \wedge \mathsf{subset}(B, C) \Longrightarrow \mathsf{subset}(\mathsf{union}(A, B), C) \tag{37}$$

Suppose that we wish to find out if, given the above laws, union is associative:

$$\mathsf{union}(A, \mathsf{union}(B, C)) = \mathsf{union}(\mathsf{union}(A, B), C) \tag{38}$$

This time, let us try the automated theorem prover SPASS [62], which like E uses the superposition calculus. SPASS does not find a solution given an execution time of ten minutes. A reason for this may be the transitivity axiom, which can cause a theorem prover to get lost in a proof due to the many consequences it generates. For example, resolving the transitivity axiom (33) with itself yields

$$\mathsf{subset}(A, B) \wedge \mathsf{subset}(B, C) \wedge \mathsf{subset}(C, D) \Longrightarrow \mathsf{subset}(A, D)$$

which in turn yields

$$\mathsf{subset}(A, B) \wedge \mathsf{subset}(B, C) \wedge \mathsf{subset}(C, D) \wedge \mathsf{subset}(D, E) \Longrightarrow \mathsf{subset}(A, E)$$

and so on, producing a never-ending chain of consequences.

SPASS has a built in strategy called *chaining* [4], which it applies when transitive axioms are present. Chaining is a family of methods that limit the use of transitivity-like axioms in proofs by only allowing certain chains of them to occur in the proof. The chaining calculus applies a special chaining rule to combine two clauses that both involve a literal with a transitive relation. For example, given the two clauses $\mathsf{subset}(A, B)$ and $\mathsf{subset}(B, C)$, chaining applies transitivity to derive the new clause $\mathsf{subset}(A, C)$.

For this particular problem, however, chaining does not seem to overcome the difficulties of transitivity.

Equalox, the third tool in our toolbox, can improve the performance of first-order provers on problems involving transitive relations. The insight is that first-order provers are poor at applying the transitivity axiom effectively, but that the problem can always be transformed to safely remove the transitivity axiom. Equalox implements several such transformations, with

more efficient encodings available for special forms of transitivity such as equivalence relations and total orders.

For our example problem, the available encoding is named "detransification". It can be applied to any theory that involves a transitivity axiom. The transformation removes the transitivity, but adds for every positive occurrence of a transitive relation $R(X, Y)$ an implication that says:

$$R(S, X) \Longrightarrow R(S, Y)$$

i.e. "for any $S$, if you could reach $X$ from $S$, now you can reach $Y$ too". Thus, we have specialised the transitivity axiom for every positive occurrence of $R$. Negative occurrences of the relation are left unchanged.

Detransification can be seen as performing one resolution step with each positive occurrence of the transitive relation and the transitivity axiom. A positive occurrence $R(a, b)$ of a transitive relation $R$, resolved with the transitivity axiom $R(X, Y) \wedge R(Y, Z) \Rightarrow R(X, Z)$ becomes $R(X, a) \Rightarrow R(X, b)$ under the substitution $Y :=$ a, $Z :=$ b.

This transformation is equisatisfiable to the original theory, and thus if we find a proof of the transformed problem, it is a proof also of the original problem.

The transformed version of our example becomes:

$$\mathsf{subset}(A, A) \tag{32'}$$

$$\mathsf{subset}(A, B) \wedge \mathsf{subset}(B, A) \Longrightarrow A = B \tag{34'}$$

$$\mathsf{subset}(A, \mathsf{union}(A, B)) \wedge \forall S.\mathsf{subset}(S, A) \Longrightarrow \mathsf{subset}(S, \mathsf{union}(A, B)) \tag{35'}$$

$$\mathsf{subset}(B, \mathsf{union}(A, B)) \wedge \forall S.\mathsf{subset}(S, B) \Longrightarrow \mathsf{subset}(S, \mathsf{union}(A, B)) \tag{36'}$$

$$\begin{array}{l}\mathsf{subset}(A, C) \wedge \mathsf{subset}(B, C) \Longrightarrow \\ \quad \mathsf{subset}(\mathsf{union}(A, B), C) \wedge \forall S.\mathsf{subset}(S, \mathsf{union}(A, B)) \Longrightarrow \mathsf{subset}(S, C)\end{array} \tag{37'}$$

The conjecture is left unchanged:

$$\mathsf{union}(A, \mathsf{union}(B, C)) = \mathsf{union}(\mathsf{union}(A, B), C) \tag{38'}$$

Note that the negative occurrences of subset (occurring to the left of the implication sign) are left unchanged, while for any positive occurrence $\mathsf{subset}(A, B)$, the resolvent of $\mathsf{subset}(A, B)$ and the transitivity axiom, i.e. $\forall S.\mathsf{subset}(S, A) \wedge \mathsf{subset}(S, B)$, must also hold. For reflexivity (32), the transformed clause can be simplified to produce the original clause. The transitivity axiom (33) is removed completely.

Although the problem looks messier to the eye, SPASS is now able to prove the conjecture in around 10 seconds. The example demonstrates that even though detransification and chaining have a similar purpose, they perform differently in practice. The chaining rule is more liberal in that it can be applied to any two clauses that contain the transitive relation. The effect of using detransification is similar to that of using the chaining rule, with the

restriction that one of the input clauses has to be an axiom. Detransification also removes the transitivity axiom altogether, while preserving soundness and completeness. By narrowing down the set of valid inferences, it seems that detransification often makes the problem easier to solve compared to chaining.

Equalox offers specialised transformations for more specific transitive relations. For equivalence relations, we can apply a transformation that makes use of the built-in equality reasoning of many theorem provers. Another transformation, which works for total orders, exploits the SMT solvers' built-in support for real numbers and the $\leqslant$ operator.

In Chapter 3, we describe the transformations in detail and present the experimental results. Depending on the tool and the nature of the problem, the transformations sometimes make the problem easier and sometimes harder to solve. The transformation for total orders significantly improved the results for the two SMT solvers [6, 18] that we used in our evaluation, while all of the transformations turned out to be generally worse for E [50]. Our evaluation showed that, after applying the transformations, 5 problems from the TPTP problem library [56] were solved that had never been solved automatically before.

## Morfar

The tools previously described analyse or transform logical problems. The final tool in this thesis, *Morfar*, instead shows an application of automated reasoning to a problem in computational linguistics.

Morphological segmentation is the task of dividing each word in a given list into smaller meaning-bearing segments (morphemes), and assigning each morphological feature of the word to one of those segments.

For example, the input may consist of the following set of Swedish inflected nouns, together with their standard forms and features:

| **inflected** | **features** |
|---|---|
| hästarnas | Pl; Def; Gen |
| fiskar | Pl; Indef; Nom |
| kattens | Sg; Def; Gen |

| **standard** | **features** |
|---|---|
| häst | Sg; Indef; Nom |
| fisk | Sg; Indef; Nom |
| katt | Sg; Indef; Nom |

The output of the tool may then look as follows, where the stems are marked in bold:

| **inflected** | **standard** |
|---|---|
| **häst**\|ar\|na\| s<br>Pl Def Gen | **häst**\| ∅<br>Sg,Indef,Nom |
| **fisk**\|ar\| ∅<br>Pl Indef, Nom | **fisk**\| ∅<br>Sg,Indef,Nom |
| **katt**\|en\| s \|∅<br>Def Gen Sg | **katt**\| ∅<br>Sg,Indef,Nom |

In natural language, the set of morphemes associated with a feature is generally limited. With this idea in mind, from a given input we aim to reuse as many feature-morpheme pairs as possible. In our example above, the feature-morpheme pairs (ar, Pl) and (s, Gen) are both used twice.

In a valid segmentation there should be no overlapping segments, and an optimal segmentation minimises the number of feature-morpheme pairs. Such criteria can be formally defined and expressed mathematically as a system of constraints. A solution can then be found using *zero-one linear programming* (0/1 LP).

A 0/1 LP problem consists of a set of variables and a set of linear inequalities over those variables. Given such a problem, a 0/1 LP solver finds an assignment of values to the variables that makes all the inequalities hold, and where all variables have the value 0 or 1. By defining a variable for each possible feature-morpheme pair, we can define the necessary constraints that need to be fulfilled to produce a valid segmentation of each word in the input. In addition, a 0/1 LP problem also specifies a linear term whose value should be minimised, called the *objective function*. In our case, the objective function aims to minimise the set of feature-morpheme pairs. A 0/1 LP solver is guaranteed to find the solution that minimises the objective function. Because of how our problem is constructed, a solution is sure to exist. The solver that we use is CPLEX [26]. Once CPLEX returns a solution, we look at which variables were assigned a value of 1. These variables describe an optimal segmentation of the input and the set of feature-morpheme pairs used in it.

**Refinements** We extend the basic algorithm with two novel refinements, which allow us to describe the morphology of the language more precisely.

Firstly, Morfar introduces constraints which help distinguish between stems and morphemes, restricting how these can be placed in relation to each other. For example, the morpheme *s* marks plural in English, but only as a suffix of the stem; seashell|s is a valid segmentation, but not s|eashells or sea|s|hells. By distinguishing between these variants, Morfar can achieve a more precise segmentation.

Secondly, Morfar detects inflection rules that involve replacing one morpheme with another. For example, in English, a final -y is typically replaced by -ie- when a word is inflected. Such rules are found by first running the segmentation algorithm, and then introducing extra features that describe the parts of the standard form that are changed when the word is inflected.

As an example, assume the input includes the following entries:

| **inflected** | **features** |
|---|---|
| cars | Pl; Nom |
| babies | Pl; Nom |

| **standard** | **features** |
|---|---|
| car | Sg; Nom |
| baby | Sg; Nom |

The resulting segmentation includes two different morphemes for plural, *s* and *ies*.

| **inflected** | **standard** |
|---|---|
| **car**\|s\| ∅ (Pl, Nom) | **car**\| ∅ (Sg, Nom) |
| **bab** \|ies\| ∅ (Pl, Nom) | **bab**\|y\| ∅ (Sg, Nom) |

We can capture the fact that the *ies* morpheme is a special case, which occurs only when the standard form ends in a *y*. This is done by automatically adding an extra feature to the input and taking advantage of the machinery we already have in place. For all entries of the input where the standard form is distinct from its stem, a new feature is automatically added to the inflected word that describes the variable part of the standard form. In our example, the variable part of the standard form *baby* is the suffix *y*. The tool thus introduces a new feature *From_Suffix_y* to the features of *babies*, and the new input becomes:

| **inflected** | **features** | **standard** | **features** |
|---|---|---|---|
| cars | Pl; Nom | car | Sg; Nom |
| babies | Pl; Nom; From_Suffix_y | baby | Sg; Nom |

With these new features added, we use the same algorithm as before to obtain a new segmentation. The morphemes that are mapped to the new features are specific to the parts of the standard form that were changed when inflected.

| **inflected** | **standard** |
|---|---|
| **car**\|s\| ∅ (Pl, Nom) | **car**\| ∅ (Sg, Nom) |
| **bab**\| ie (From_Suffix_y) \|s\| ∅ (Pl, Nom) | **bab**\|y\| ∅ (Sg, Nom) |

The morpheme pair (*From_Suffix_y*, ie) can be seen as a function taking the suffix *y* in the standard form and changing it into *ie*. We picture this as (y → ie). The segmentation of *babies* is thus shown as:

$$\textbf{bab}(\text{y} \rightarrow \text{ie})\,|\underset{\text{Pl}}{\text{s}}|\,\underset{\text{Nom}}{\varnothing}$$

The new segmentation captures the -y to -ie- rule of plural nouns in English, allowing the morpheme pair (Pl, s) to be reused.

We can also find inflection rules that are specific to a part of the stem, by adding features that specify what characters the stem ends or begins with. One such example is the doubling of the letter *g* in the comparative form of *big*, pictured as: bi(g → gg) $|\underset{\text{Comp}}{\text{er}}$.

**Morphological Reinflection** Morfar has been adapted to do morphological reinflection [14]. Given a lemma and set of morphological features, the task is to generate a target inflected form. For example, given the source form *release* and target features PTCP and PRS, the task is to predict the target form *releasing*.

Our approach requires a set of labelled training data, which we segment to obtain a list of morphemes and their associated features. To predict the target inflected form of a word, we: 1) find the stem of the word, 2) find a word in the training data whose features match the target features, and 3) replace the stem of that word with that of the input word. Our reinflection algorithm is very simple, but still competes with state-of-the-art systems, indicating that the underlying morphological analysis provided by our tool is of good quality.

## Impacts and Future Work

This thesis consists of four papers that respectively describe the tools Infinox, Monotonox, Equalox and Morfar. Below is a short summary of some additional work that has been done after their publication, and some stories of people who have been in touch about using Infinox and Monotonox in their work.

### Infinox

Since the original version of Infinox was released in 2009, it has been used by several people in their own research. A few mathematicians have found Infinox through web search while looking for criteria on infinite models, which suggests that the problems that Infinox attempts to solve are indeed highly relevant. One of the people who have been in touch is David Stanovsky, a mathematician studying loop theory. He was originally looking for a tool that can determine that a problem does not have a model of a certain finite size. While Paradox turned out to be a better tool for this particular purpose, our discussion generated a nice exchange of ideas. David was at first disappointed to find that Infinox was unable to show finite unsatisfiability of some problems from Group Theory known to be finitely unsatisfiable. However, as it turned out, the problems were solvable by tweaking the parameters and choosing the right method, namely the one that searches for serial relations (in other words, a strict partial order with no maximal elements). David was also pleased that Infinox found an alternative proof to a problem to which he was only aware of an argument using Lagrange's theorem: Infinox was quick to find that a group containing an element of order 2 and having square roots must be infinite, by pointing out that the function $X \to X * X$ is surjective but not injective.

Another Infinox user is Mark Greer, who came across Infinox while trying to solve an open problem in Loop Theory as part of his dissertation. Mark

wants to know if every finite alternative loop has a 2-sided inverse. There is a known infinite counter-example, but in the finite case, no counter-example has been found. Infinox was not able to rule out finite counter-models to his problem, but we could show using Paradox that any possible counter-model would have to be of size 25 or larger. Mark has only case-by-case arguments for different parameters $k$, but no clear overall pattern to complete a proof. Such situations can motivate the addition of new methods to Infinox, where one searches for different functions with "infinity properties" at the same time. Since Mark is looking for different arguments for different values of $k$, it may be the case that there are different functions or relations with different infinity properties for different $k$. As an example, suppose we have a problem with two functions $f$ and $g$, where $f$ is injective and non-surjective on domains with even size, and $g$ is surjective and non-injective on domains with odd size. Infinox will not be able to show the infinity properties for either of the functions on their own, but it could show that either $f$ or $g$ must have the infinity properties in each case, and hence that any model must be infinite. Variations on this are of course without limit, and it is hard to know what extensions of Infinox would be useful in practice. Problems like the one Mark is trying to solve can serve as a guide in extending Infinox with new methods that are worthwhile.

Jesse Alama, a researcher in Mathematical Logic and Proof Theory, has used Infinox in his research while working with the large Mizar Mathematical Library [36]. In one of his projects, he is interested in making sure that the models of the problems he is working with are finite. Using Infinox, he detects cases where this is not the case, so that no time is wasted wondering whether using a finite model finder would provide a finite model. Jesse has made a tool called "Tipi", which aims at providing support in theory development. He mentions Infinox in his paper [3] as a possible extension to complement Tipi. Both Infinox and Monotonox could play a role in detecting unintended models, as exemplified in the previous section. His goal is similar to ours in that he wishes to provide answers to a wider range of questions than the standard ones of satisfiability and derivability.

Infinox also has some more practical uses. Geoff Sutcliffe, the developer of the TPTP problem library [56], and organizer of the annual CASC competition for theorem provers and model finders [57], has used Infinox to evaluate the test problems for the competition, checking what ones do not have a finite model.

## Monotonox

**Monotonox and the monotonicity calculus** The paper in this thesis, published in 2011, develops the theory of monotonicity for sorted first-order logic. The domain of a monotone sort can always be extended with an extra element. Monotone sorts result in less clutter when translating to unsorted logic, as sort information can be erased without affecting satisfiability. In first-order logic, the only possible source of non-monotonicity is the use

of a variable next to an equality sign. To determine exactly what occurrences of equality limit domain size is trickier, and detecting monotonicity of a sort in general is not decidable. We have introduced two algorithms approximating the answer, one linear in the size of the problem, and one improved algorithm solving an NP-complete problem using a SAT solver. The algorithms have been implemented in our tool Monotonox. Our results show that the improved algorithm detects many cases of monotonicity, and that the NP-completeness is not a problem in practice.

**Previous work that inspired us** The original ideas behind the monotonicity calculus were first implemented in 2003 in a simpler form in the finite model finder Paradox [12], which performs sort inference in order to simplify the underlying SAT problem. With additional sort information, clauses can be added to the SAT problem to reduce symmetries related to isomorphic interpretations. The analysis was based on the observation that the domain of a sort can be extended when there are no positive equality literals involving naked (i.e. not occurring as a subterm) variables of that sort.

Monotonicity for higher-order logic was invented by Blanchette and Krauss [9] in 2010, and implemented as part of the higher-order model finder NitPick, to prune the search space in a similar way to how it is done in Paradox. The differences in the logics makes the problem of inferring monotonicity considerably different even though it is related to ours. For example, in higher-order logic, since equality can be encoded using quantification over predicates, it is not true that a formula without equality is always monotone.

**Work inspired by Monotonox** In a follow up paper from 2013 [10], the notion of first-order monotonicity is extended to include polymorphic sorts. The paper introduces two different main approaches to encoding polymorphic symbols in first-order logic. One works by monomorphising the problem, which is done by heuristically instantiating all sort variables with ground sorts. Monotonox can then be used to remove the sorts as usual. Monomorphisation is incomplete as an upper bound has to be set to limit term depth and the number of additional axioms. The second approach works by encoding sorts by a term, which is passed as an argument to the sort predicate. As an example, $p(list(A), X)$ restricts the range of the variable $X$ to lists. Despite incompleteness, the monomorphic method turned out to be the most successful, which suggests that the clutter introduced by the sort arguments considerably slows down automated theorem provers. The paper also makes a major improvement to the monotonicity calculi which considerably reduces the clutter associated with the translation of non-monotone sorts. The new scheme, which can be applied to both polymorphic and first order monotonicity, requires guards only in the particular axioms that make the sort non-monotone. This is based on the observation that when all "dangerous" axioms are protected, all sorts in the translated problem are

monotone, so the remaining axioms can be left unchanged. Monotonox has been updated to use the new improved encoding.

Reger et al. [43] have used ideas from both Infinox and Monotonox to improve the performance of finite model finders in sorted first order logic. Their model finder tries different combinations of model sizes for each sort. They build on our ideas to limit the number of different combinations that need to be tried. When the problem contains monotone sorts, the number of combinations can be reduced in two ways. Firstly, all monotone sorts can be assumed to have the same size. Secondly, when a monotone sort is found to have no model of size $n$, then no model with a size smaller than $n$ can exist. They also use an Infinox-like approach to discover constraints between the sizes of different sorts. For example, if an injective and non-surjective function exists from sort $A$ to sort $B$, then it can be inferred that the size of $B$ must be bigger than the size of $A$ in any finite model. Their evaluation shows that these techniques are useful in practice.

An application where Monotonox is used is QuickSpec [53], a tool that finds algebraic properties of functional programs by testing. To remove redundant laws, an equational theorem prover for first-order logic is used. Since the laws that are discovered are for typed programs, Monotonox is used to soundly remove the types without introducing additional clutter. Using Monotonox instead of the naive encoding has resulted in more redundant laws being discovered and removed.

**Future Work on Monotonox** Left as future work is to improve the encodings to detect more sorts that can be erased without introducing sort predicates. To remove a sort safely, monotonicity is an unnecessarily strong criteria as it suffices that the domain of the sort can be made as big as that of the biggest sort. When the domain of a sort can be extended without bound, we do not need sort predicates to restrict the sort's range. Searching for a chain of injections between the sorts can help us find the biggest one. (If there is an injection from a sort $A$ to $B$, the domain of $B$ must be at least as big as the domain of $A$.) Infinox could be adapted to do this analysis. In our running example, the function singleton is an injection from Element to Set, which means that Set has the biggest domain and does not need sort predicates.

Another possible extension of Infinox is to adapt it to sorted problems. This would allow monotonicity inference of sorts by showing that their domains must be infinite. Infinity inference for monotonicity is exploited in the higher-order proof assistant Isabelle, as discussed in [10]. Isabelle has the advantage of having datatypes registered with their constructors. If a constructor is recursive or takes infinite arguments, the associated sort is trivially infinite and thus monotone.

### Equalox

Our paper presents 5 transformations that can be applied to theories with certain transitive relations. A transformed theory is equisatisfiable to the original theory; a proof of the transformed problem is also a proof of the original problem.

**Detransification** works for any transitive relation.

**Detransification with reflexivity** specialises detransification for relations that are both transitive and reflexive.

**Equalification** works on equivalence relations, and makes use of the built-in equality reasoning of many theorem provers.

**Pequalification** applies equalification on partial equivalences, i.e. relations that act as equivalence relations on a subset of the domain.

**Ordification** works for total orders. It introduces arithmetic operators and real numbers, and therefore requires the solver to support arithmetic reasoning.

The purpose of our paper is to investigate how different representations of transitive relations affect the performance of theorem provers. Overall, the results vary between each transformation and reasoning tool. For many of the transformations, there are both problems that become easier and problems that become harder to solve. For some combinations of transformations and theorem provers, there is a clear improvement. For example, applying ordification to problems with total orders improved the results greatly for the two SMT solvers Z3 and CVC4. We also show that a time-slicing strategy can be advantageous, where the reasoning tool is run on both the original and the transformed problem, with a suitably chosen time-limit for each.

For some combinations of transformations and provers (such as detransification for Vampire, and equalification for Z3), the overall results are clearly better on the transformed problems, and we would thus recommend these transformations as preprocessors for these provers. Theorem provers that already have the time slicing machinery in place would for some transformations benefit from a time slicing strategy, which given a proportion of the time each tries to solve both the original and the transformed theory.

These ideas could also be integrated in a theorem prover. If a transitive relation is detected during proof search, an appropriate transformation could be applied at that point. Another possibility is to change the chaining rule that is used in SPASS so that one of the resolvents must be an axiom. By doing so, we would restrict the valid inferences to achieve the same effect as detransification.

### Morfar

Morphological segmentation has been extensively studied. The most popular approaches use statistical methods, but these typically require large amounts of data. Integer linear programming has not, as far as we know, been used for morphological segmentation before. By requiring the segmentation to follow strict constraints, we can achieve good results with less training data. Compared to previous work, our method improves the precision of the morphological analysis in two ways. Firstly, we can use the constraint solver to restrict where a given morpheme can appear in a word, e.g. prefix, infix or suffix. Secondly, by automatically adding new features to the problem, we can describe more precisely in which situations each morpheme should be used.

The use of a constraint solver for morphological segmentation has many benefits. Firstly, we are forced to make the problem well defined and specify exactly what problem we want to solve. Having done so, the solution that we obtain is guaranteed to be optimal. In the case that the solution is not satisfactory, it means that the problem definition was not adequate. The process of formally describing the problem helps us to better understand it, which in turn helps us to fine-tune the constraints and add new features to precisely capture the morphological structure of the language.

Our paper demonstrates that constraint solving is a useful alternative to machine learning for segmentation and reinflection, particularly for low-resource languages where one must make the most of a small set of data. We believe that the use of automated reasoning and integer linear programming in computational linguistics is an underexplored topic, and hope that our paper spurs more research in this direction.

## Conclusions

This thesis introduces four different tools, which extend the automated reasoning toolbox in two different ways.

Firstly, Infinox, Monotonox and Equalox all have in common that they are aimed at complementing and augmenting existing reasoning tools by solving a wider range of problems than the traditional ones of satisfiability and derivability. The tools themselves, as well as the ideas behind them, have inspired new research in automated reasoning and related fields.

Secondly, Morfar extends the scope of applications of automated reasoning, demonstrating the benefits of formalising the problem and finding an exact solution in natural language processing, an area currently dominated by approximate methods such as neural networks.

Because first-order logic is semi-decidable, there will always be problems that we cannot solve. We can always improve our tools to solve more problems, but we can never complete the task and solve them all. In the same way, the automated reasoning toolbox can always be extended, but never completed.

## Contributions of the Author

**Paper I - Automated Inference of Finite Unsatisfiability** I came up with the different methods, extensions and optimisations. I implemented the tool described in the paper, and did the evaluation. The sections related to this, and the motivating examples are written by me, while the remaining parts were written jointly. I presented this work at the CADE conference in Montreal, Canada in 2009. At the time of writing, the journal and conference versions of the paper together have a total of 21 citations.

**Paper II - Sort it Out with Monotonicity: Translating between Many-Sorted and Unsorted First-Order Logic** The implementation was made jointly with Nicholas Smallbone. The monotonicity calculi and proofs were worked out by us together and the paper was written jointly. Nicholas Smallbone presented the work at the CADE conference in Wrocław, Poland in 2011. At the time of writing, the paper has 46 citations.

**Paper III - Handling Transitive Relations in First-Order Automated Reasoning** The generalised method and the partial equalification are both my ideas. I designed and implemented the tool described in the paper, and did the evaluation. The related sections are written by me, while the remaining parts were written jointly with my co-author. I presented this work at the PAAR workshop in Coimbra, Portugal in 2016.

**Paper IV - Inferring Morphological Rules from Small Examples using 0/1 Linear Programming** I designed and implemented the tool described in the paper and did the evaluation. The technique to detect stem changes is my own idea, as well as the semantics for patterns and the ideas for future work. The problem description, the description of the algorithm and optimisations, the evaluation and examples are all written by me. The paper was published in the proceedings of the NoDaLiDa'19 conference in Turku, Finland.