UNIVERSITY OF GOTHENBURG

The Infinox Infinity Inferrer

A Complement to Finite Model Finding

Master of Science Thesis in Computer Science and Engineering

ANN LILLIESTRÖM

University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2009

The Infinox Infinity Inferrer
A Complement to Finite Model Finding

Ann Lillieström

Examiner: Koen Claessen

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2009

Abstract

Infinox is an automated reasoning tool that can disprove the existence of finite models of first-order theories. It is a tool of both theoretical and practical value and is especially well suited as a complement to finite model finders. The main idea behind Infinox is to search for function and predicate symbols with certain properties that imply the non-existence of finite models. A standard automated theorem prover is used to check if these properties hold.

We describe the methods used to identify terms that possess the desired properties, and explain in detail how these can be combined and applied to concrete problems. Some very promising first results are presented; Infinox has classified a large number of problems from the TPTP problem library as finitely unsatisfiable. Many of these problems have never before been solved (nor classified) by an automated system.

Acknowledgements

I wish to thank my supervisor Koen Claessen, for many inspiring discussions, and the guidance and encouragement that he has given me during my work with this thesis. Thanks to everyone who shared their comments and ideas, and thanks to Geoff Sutcliffe for his work on the TPTP library. I also wish to thank my friends and family for their love and support, and my dog Julius for reminding me to take breaks. A special thanks to Lotty and Zenita, for helping me through some difficult times. Finally, I dedicate this work to the memory of my mother Christina, who will never cease to inspire me.

Contents

Chapter 1

Introduction

1.1 Automated Reasoning

Automated reasoning deals with the development of computer programs that can assist in solving problems requiring reasoning. The problems must be phrased in a language that the reasoning program accepts. A typical language acceptable to various automated reasoning programs is first-order logic (FOL), which I will assume the reader is familiar with. I will use the notation of [16], which is a standard book on the topic.

The importance of a formal language Many real world problems in a multitude of fields can be formulated in FOL. Such a formulation typically consists of a statement expressing the question being asked (the *conjecture* or *conclusion*), and a set of statements expressing the available information (the *axioms* or *assumptions*).

A major benefit of using a formal language to describe a problem is that there is no ambiguity, as is often the case when using natural language. Take, for example the sentence "Doctor helps dog bite victim". -Does the doctor help the dog or the victim?

In order to avoid ambiguities, we must state the problem in a precise way, and supply all information explicitly. This has the advantage of exposing questionable or even incorrect assumptions, and may in itself lead to a deeper understanding of the problem.

An appropriate formulation of a problem enables the use of a computer program; an *automated theorem prover (ATP system)* to attempt to solve it. An ATP system mechanises different forms of reasoning, while following the laws of the formal language being used.

Applications Perhaps the most obvious application of ATP systems is to prove mathematical theorems. In fact, many other fields can benefit from ATP systems' ability to reason flawlessly.

Some examples are hardware and software verification [7], the identification of faults in electronic equipment [6], and expert systems that can reason, for example medical diagnosis systems. [8] Even fields such as Social Science can benefit from the use of ATP systems, by "translating" theories expressed in natural language to formal logic. [1]

Limitations ATP systems can solve extremely hard problems that are essentially impossible to prove by hand. However, they will sometimes fail to terminate while searching for a proof. By Church's theorem [2], there is no general algorithm that can determine whether a first-order sentence is universally valid. If a proof exists, however, it can always be found in finite time, since proofs are finite sequences of formulas. But it is not always possible to produce a counter model of a problem if a proof does not exist. In other words, first-order logic is semi-decidable.

There will always be problems that cannot be solved by any ATP system, regardless of future methods and improvements to performance. By Trakhtenbrot's theorem [21], even the problem of validity in finite domains is semi-decidable. This means that there exists no algorithm that, for any given problem, is able to in finite time correctly decide whether or not a finite model exists. If a problem has a finite model, it can be found in finite time by exhaustive search. If, on the other hand, it has no finite model and no counter proof, there is no algorithm that is able to determine this and terminate.

Thus, there is a group of problems that cannot be solved. In this thesis, we show how some of these problems can be classified automatically.

Example Below is an example of a problem formulated in first-order logic.

Axioms

$\forall X,Xs : cons(X,Xs) \neq nil$

$\forall X,Xs :$
$cons(X,Xs) = cons(Y,Xs) \Longrightarrow X = Y$

$\forall X : \neg member(X,nil)$

$\forall X,Y,Xs :$
$member(X,cons(Y,Xs)) \Longleftrightarrow$
$X = Y \vee member(X,Xs)$

Conjecture

$\exists X : member(X,X)$

The axioms introduce the constructor functions for lists; the constant *nil,* that represents the empty list, and *cons*, that constructs a new list by adding an element to the front of a list. The axioms also give a recursive definition of the *member*-predicate; an element is a member of a list *iff* it is either equal to the head of the list, or it is a member of the tail of the list. Given these axioms, the problem reads "*is there an element X, such that X is a member of itself?*". To show that a set of axioms implies some conjecture is is a typical task for an ATP system.

The E Equational Theorem Prover [6] is a standard automated theorem prover. It was run on the above problem on a 2x Dual Core processor, operating at 1 GHz, and 4 GB RAM. After about 15 minutes of computation, E ran out of memory. With more resources, it *might* be possible to find a proof. It could also be the case that no proof exists. In such a situation, one may consider a different strategy. If we can find a counter-model of the problem, this means that the negation of the problem is satisfiable, and thus that the axioms do not imply the conjecture.

To see if there is a counter-model, the finite model finder *Paradox* [3] was run on the same problem. Paradox failed to find a counter-model up to size 78 before it gave up.

We were not able to find a proof nor a counter-model of this problem with the given tools and resources. We will discuss ATP systems and their limitations further in section 2. We can, however, show that if this problem has a counter-model, then it must be infinite. Thus, Paradox will never succeed in finding a counter-model. We show how this is done in chapters 3 and 4.

1.2 Infinox

In this thesis, we present *Infinox*; an automated tool that specialises in disproving the existence of finite models (or finite countermodels). The aim of this tool is to automatically classify problems as *finitely unsatisfiable*, i.e. as having no finite model. All unsatisfiable problems are by definition finitely unsatisfiable.

Infinox can be used as a complement to existing automated reasoning software, in particular to finite model finders. If a problem lacks finite models, a finite model finder will potentially never terminate in the search for one. This can be avoided if Infinox can conclude that no finite model exists.

The main idea behind Infinox is to identify certain properties of the given problem, which make the existence of a finite model impossible. The E Equational Theorem Prover [6] is used to check if these properties hold.

1.3 Thesis outline

The rest of the thesis is organised as follows:

A brief overview of various automated reasoning systems, and how these relate to Infinox, is given in chapter two.

In chapter three, we present some examples of axiom sets that lack finite models, and identify the properties that they have in common.

In chapter four, we describe the naive algorithm to prove finite unsatisfiability of a theory. We show in steps how this algorithm can be generalised to classify all of the example problems.

In chapter five, we present some experimental results, compare the performance of the different methods.

Further improvements and future work are discussed in chapter six.

Finally, we present our conclusions and contributions in chapter seven.

A short user manual to Infinox is available in the appendix.

Chapter 2

Related Work

In this chapter, we give a brief overview of some of the techniques employed by automated reasoning systems, and discuss their strengths and weaknesses.

2.1 Automated theorem provers

Automated theorem provers typically solve problems through contradiction. By inferring a contradiction from the negation of a theory, we can draw the conclusion that the original theory is valid in all interpretations.

Saturation One of the techniques most widely used by automated theorem provers is saturation. A saturation-based theorem prover searches for a contradiction by saturating the given clause set, i.e. systematically and exhaustively applying all inference rules. Vampire [12] and SPASS [22] are examples of theorem provers based on saturation.

Resolution One of the major theorem proving techniques based on saturation is resolution. The resolution algorithm was published by J.A. Robinson in 1965 [14], and is based on a single inference rule.

There exist several strategies for reducing the search space of a resolution system without compromising completeness. One such strategy is *unification,* which is used to identify appropriate substitutions for variables. [13]

As a consequence of Herbrand's theorem (1930)[11], resolution is refutation-complete, which means that if a theory is unsatisfiable, then resolution will always be able to derive a contradiction.

2.2 Finite model finders

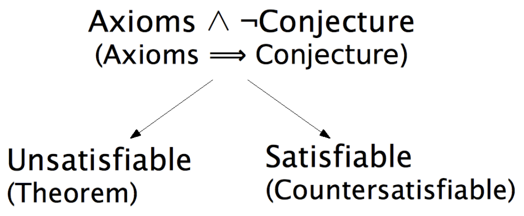
Finite model finders are often used as a complement to automated theorem provers. When an automated theorem prover fails to find a proof of a theory, a finite model finder may be used to try to find a model of the theory's negation, and thus show that no proof exists.

SAT-solvers A SAT-solver is a model finder for propositional logic. It attempts to prove satisfiability of a propositional formula, by checking if there is an assignment of truth-values of the variables such that the formula evaluates to true under that assignment. Satisfiability is NP-complete, as proved by Stephen Cook in 1971. [4] However, there exist many algorithms that solve SAT-problems efficiently in practice. A successful SAT-solver is *MiniSat.* [5]

Finite model finders for first-order logic With a fixed domain size n, initially n = 1, a finite model finder decides if there is a consistent interpretation of the problem with this domain size. If not, we increment n and start over. Using this technique, we obtain a concrete model of our problem (if one of a fixed size exists). The two most successful styles of methods of model finding are *Sem-style,* named after Zhang and Zhang's tool *Sem* [23], and *Mace-style,* named after McCune's *Mace.* [9]

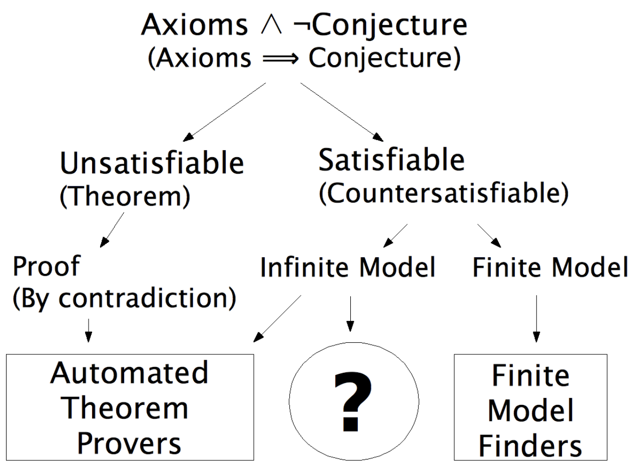
2.3 Limitations

Automated theorem provers typically solve problems through contradiction. The validity of a theory is shown by proving that the negation of the theory is unsatisfiable, i.e. that it leads to a contradiction. If, on the other hand, the negation is satisfiable, there is a counter-model of the theory.



As explained in section 2.1, if a contradiction exists, it can always be found in finite time by an automated theorem prover based on resolution.

If the negation of the theory is satisfiable, it has, by definition, a model. A model is either finite or infinite.



If the model is finite, it can be found in finite time by a finite model finder, as described in section 2.2. If there are only infinite models of the theory, a finite model finder will never terminate its search. Theories with only infinite models can in some cases be proved to be satisfiable by a saturation based theorem prover, but this is far from always possible. In section 1.1 we saw that many problems in first-order logic are, in fact, not provable.

In the next chapter, we shall take a look at some examples of theories that lack finite models, and identify the properties that these theories have in common.

Chapter 3

Examples

In this section, we identify the properties that preclude the existence of finite models. Furthermore, we illustrate by a number of examples ways in which these properties can be generalised.

3.1 Injectivity and non-surjectivity

Example *Consider the following two axioms* for a function symbol *suc:*

A1. $\forall X : suc(X) \neq zero$

A2. $\forall X, Y : suc(X) = suc(Y) \Longrightarrow X = Y$

In any model with domain D of these axioms, D is infinitely large.

Proof. From the first axiom we know:

- There exists a constant *zero*.
- For all X in our domain, there exists a successor of X, which is not equal to *zero*. Thus, there are at least two elements in D; *zero* and $suc(zero)$.

Now, assume that

$$zero, suc(zero), ..., suc^n(zero)$$

are unique in D for some $n \geq 1$. We will show that $suc^{(n+1)}(zero)$ is not equal to any of these. Assume the contrary. We know by A1 that

$$suc(zero) \neq zero.$$

Thus, we have, for for some i s.t. $1 \leq i \leq n$.

$$suc^{(n+1)}(zero) = suc^i(zero)$$

or, equivalently

$$suc(suc^n(zero)) = suc(suc^{(i-1)}(zero))$$

By A2, this yields:

$$suc^n(zero) = suc^{(i-1)}(zero)$$

This contradicts our assumption that the elements $zero, .., suc^n(zero)$ are unique in D. We conclude that every element X in the set gives rise to a new, unique element $suc(X)$. In other words, only infinite domains exist for these axioms. □

From this example, we can conclude the following:

Theorem 1 *Any domain on which an injective and non-surjective function operates is infinite.*

Proof. The axioms of the above example state precisely that the successor function is *non-surjective* (by A1), and *injective* (by A2). The proof above can thus be applied to any function with these properties. This shows that any domain on which an injective and non-surjective function operates must be infinite. Thus, any set of axioms that implies injectivity and non-surjectivity of a function must either be unsatisfiable, or have only infinite models. □

3.2 Existential quantification

We consider a set of axioms, which define selection of the first and second elements of a pair. Though not as easily spotted as in the previous example, these axioms hide an injective and surjective function. This function is not lexically present in the axioms, however, it can be constructed from the functions and constants that are given.

Example *The following axioms have no finite model*

B1 $\forall X, Y : fst(pair(X, Y)) = X$

B2 $\forall X, Y : snd(pair(X, Y)) = Y$

B3 $a \neq b$

Proof. Axiom B3 states that there are at least two elements in the domain; the constants a and b. Now, suppose

$$pair(X, a) = pair(Y, a)$$

for some $X, Y \in D$. Applying fst to both sides of the equality sign yields

$$fst(pair(X, a)) = fst(pair(Y, a))$$

By B1 this implies $X = Y$. We have derived

$$\forall X, Y : \mathit{pair}(X, a) = \mathit{pair}(Y, a) \Longrightarrow X = Y$$

meaning that $\mathit{pair}(X, a)$ is an injective function. Next, we note that, by B2,

$$\forall X : \mathit{snd}(\mathit{pair}(X, a)) = a$$

and

$$\mathit{snd}(\mathit{pair}(a, b)) = b$$

Since, by B3, $a \neq b$, it must be that

$$\forall X : \mathit{pair}(X, a) \neq \mathit{pair}(a, b)$$

meaning that $\mathit{pair}(X, a)$ is a *non-surjective* function. Now, we can simply apply theorem 1 to conclude that no finite model for these axioms can exist. □

3.3 Reflexive relations

In this example, we show that it is sufficient that a function is injective and non-surjective with respect to a reflexive relation.

Example *The following axioms have no finite model*

D1 $\forall X : \mathit{lte}(X, X)$

D2 $\forall X : \neg \mathit{lte}(\mathit{suc}(X), \mathit{zero})$

D3 $\forall X, Y : \mathit{lte}(\mathit{suc}(X), \mathit{suc}(Y)) \Longrightarrow \mathit{lte}(X, Y)$

Remark. Since the relation need not be symmetric, we distinguish between left-related and right-related; *lte(X, Y)* means that X is left-related to Y, while Y is right-related to X, by *lte*.

Proof. From D1, we know:

D1' $X = Y \longrightarrow \mathit{lte}(X, Y)$

This, in turn, implies that

D1" $\neg \mathit{lte}(X, Y) \longrightarrow X \neq Y$

From D2, we know:

- There exists a constant *zero*.

- For all X in our domain, there exists a successor of X, which is not left-related to *zero* by *lte*. Thus, $\neg lte(suc(zero), zero)$, and by D1", $suc(zero) \neq zero$.

Now, assume that

$$zero, suc(zero), ..., suc^n(zero)$$

are unique in D for some n > 1. We will show that $suc^{(n+1)}(zero)$ is not equal to any of these elements. Assume the contrary. We know by D1' that $suc^{(n+1)}(zero)$ is not equal to *zero*, since this would, by D1', imply $lte(suc^{(n+1)}(zero), zero)$, which contradicts axiom D2. Thus, we assume that

$$suc^{(n+1)}(zero) = suc^i(zero)$$

for some i s.t. $0 < i \leq n$. By D1', this implies

$$lte(suc^{(n+1)}(zero), suc^i(zero))$$

which, by applying axiom D3 i times yields

$$lte(suc(suc^{(n-i)}(zero), zero))$$

which contradicts axiom D2. We conclude that every element X in the set gives rise to a new, unique element $suc(X)$. In other words, only infinite domains exist for these axioms. Note that the same reasoning applies if the arguments to the function in axiom D2 are flipped. Simply replace "*left-related*" by "*right-related*" in the proof.

Theorem 2 *Any domain on which a function f is injective and either left- or right-non-surjective, with respect to a reflexive relation r, is infinitely large.*

Proof. The axioms of the example above state precisely that the function *suc* is injective and non-surjective with respect to the reflexive relation *lte*. The proof of the example can thus be applied to any function f and relation r with these properties. □

3.4 Infinite subdomains

In order to show that a set is infinite, it is sufficient to show that a subset of the set is infinite. Often, a function is injective and non-surjective only on a part of its domain. In these cases, it is necessary to locate the infinite subdomain.

Example *The following axioms have no finite model.*

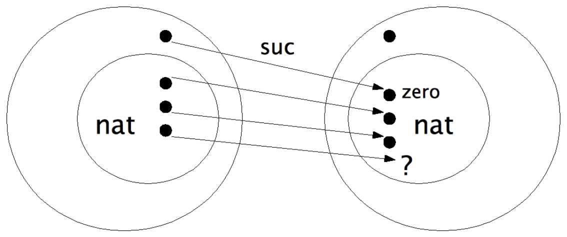
E1 $nat(zero)$

E2 $\forall X : nat(X) \Longrightarrow suc(X) \neq zero$

E3 $\forall X : nat(X) \Longrightarrow nat(suc(zero))$

E4 $\forall X, Y : nat(X) \wedge nat(Y) \Longrightarrow (X = Y \Longrightarrow suc(x) = suc(Y))$

Proof. The above axioms include a predicate *nat*, which defines a subset of the domain. When considering the full domain, the axioms do not imply injectivity and non-surjectivity of the function *suc*, since they say nothing about the behaviour of *suc* outside the subdomain defined by *nat*. However, by disregarding any elements X, for which $nat(X) = false$, we can use the proof of example 1 to show injectivity and non-surjectivity of *suc* on this subdomain. Since any domain that contains an infinite subdomain must be infinite, it follows that these axioms cannot have finite models. □



3.5 Non-injectivity and surjectivity

In this example, we show how *non-injectivity* and *surjectivity* of a function implies infinity of its domain.

Example *The following axioms have no finite model.*

F1 $\forall Y : \exists X : f(X) = Y$

F2 $\exists X, Y : X \neq Y \wedge f(X) = f(Y)$

Proof. Suppose there is a model with a finite domain D, of size n, that satisfies the above axioms. The axioms describe a mapping $f : D \mapsto D$, that is *surjective* and *non-injective.* F2 states that there are at least two elements in the domain that map to the same element. Now, take one element in D that is mapped to by at least two elements. Call these elements $e_1, .., e_k$(where $k \geq 2$). We can now construct a new function, f', that is exactly the same as f, with the exception that it is not defined for $a_1, .., a_k$, and thus $f(a_1) = ... = f(a_k)$ is not in its codomain:

$$f' : D \setminus \{a_1, .., a_k\} \mapsto D \setminus \{f(a_1)\}$$

The surjectivity of f is preserved in f'; any element in D that was covered by f is also covered by f', with the exception of $f(a_1)$, which is not in f':s codomain.

However, a mapping from a set of arity $(n - k)$,with $k \geq 2$ to a set of arity $(n-1)$ can, by the pigeonhole principle, clearly not be surjective. It must either be that our assumption that f is surjective is false, or that D is infinitely large.□

Theorem 3 *Any domain on which a function that is non-injective and surjective, with respect to equality, is infinite.*

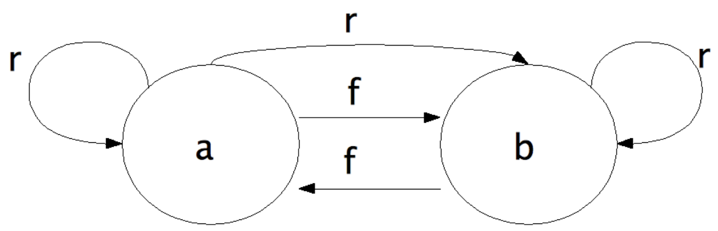
Proof. The axioms of the above example state precisely that the function f is non-injective and surjective. The proof above can thus be applied to any function having these properties. □
Remark. Unlike in example 3, we cannot replace equality by a reflexive relation, as shown by the following counter-example:

F1' $\forall X : r(X, X)$

F2' $\forall Y : \exists X : r(f(X), Y)$

F3' $\exists X, Y : \neg r(X, Y) \wedge r(f(X), f(Y))$

The above axioms state that the function $f$is non-injective and surjective with respect to the reflexive relation r. These axioms have a finite model with domain $D = \{a, b\}$, and the following interpretation:



r(a,a) = true
r(b,b) = true
r(a,b) = true
r(b,a) = false

f(a) = b
f(b) = a

3.6 Robbins Problem

A problem that is not as easy to solve by hand is *Robbins problem*; "are all Robbins algebras boolean?". In 1933, E.V. Huntington presented the following equations as a basis for Boolean algebra:

Commutativity $\forall X, Y : plus(X, Y) = plus(Y, X)$

Associativity $\forall X, Y, Z : plus(plus(X, Y), Z) = plus(X, plus(Y, Z))$

Huntington $\forall X, Y : neg(plus(neg(plus(X, Y)), neg(plus(X, neg(Y))))) = X$

Shortly thereafter, Huntington's student Herbert Robbins conjectured that the Huntington equation can be replaced with the following equation:

Robbins $\forall X, Y : plus(neg(plus(neg(X), Y))), neg(plus(neg(X), neg(Y)))) = X$

Neither Huntington nor Robbins was able to find a proof or a counter-example. Several mathematicians, including Alfred Tarski and his students, have since worked on this problem. Despite their efforts, it has remained open until 1996, when EQP, an automated theorem prover for equational logic, found a proof after eight days of computation [10].

Eventhough Robbins Problem is difficult to solve, it is easily classified as finitely unsatisfiable. Given the axioms of boolean algebra and Robbins conjecture, Infinox establishes in under 30 seconds that *neg(X)* is a surjective and non-injective function, and thus that the problem has no finite models.

3.7 Summary

We have seen that a theory that contains a function that is either *injective and non-surjective* or *non-injective and surjective* cannot have a finite domain.

In example 2 we concluded that these properties can be found even in functions that are not lexically present in the axioms. We detect these functions by instantiating variables with constants.

In example 3, we learned that injectivity and non-surjectivity with respect to any reflexive relation implies infinity of the domain.

Example 4 showed how we can consider subdomains to weaken the properties that imply infinity of the domain.

In example 5 we saw how non-injectivity and surjectivity of a function implies that the domain cannot be finite.

In example 6 we saw how a problem that is difficult to solve can be easy to classify using these properties.

In the next chapter, we will see how these methods can be implemented, in order to classify theories automatically.

Chapter 4

Infinox

Infinox is an automated tool that is used to prove that no finite model (or counter-model) for a given first-order theory can exist. Infinox will, given a set of clauses C, either establish a conclusion of the form *"if* a model of C exists, *then* this model cannot be finite.", or simply give up. It thus infers that the theory is either unsatisfiable, or lacks finite models. We refer to this classification of theories as *finitely unsatisfiable.*

Infinox is a useful complement to model finders. If a theory lacks finite models, a finite model finder will potentially search for a model until it runs out of memory. By classifying a theory as finitely unsatisfiable, we know that searching for a finite model is pointless.

4.1 The ideas

The ideas behind Infinox are based on the following two facts:

- Any domain on which an *injective* and *non-surjective* function operates, must be infinite. This is shown in section 3.1.
- Any domain on which a *non-injective* and *surjective* function operates, must be infinite. This is shown in section 3.5.

Thus, if we can implement an algorithm to detect such terms, this algorithm could then be used to disprove the existence of finite models.

4.2 The algorithm

In this section, we describe the algorithm in detail. We begin with a simpler, naive algorithm, and describe how to make the necessary generalisations to classify the examples in section 3 as finitely unsatisfiable.

4.2.1 A first attempt

We cannot directly ask a first-order logic theorem prover: “Does this axiom set include injective and non-surjective functions?”. The reason for this is that it is not possible to quantify over functions in first-order logic. What we can do, however, is to provide the theorem prover with the specific functions that we want to check. A simple automated method to solve the above problem is this:

1. Identify all functions of arity 1 in the problem.
2. For each such function, construct the conjecture that the function is injective and non-surjective, with respect to equality (“=”). This can be done automatically.
3. For each conjecture, use an automated theorem prover to check whether it is a logical consequence of the axioms.

If we succeed for any function, we have shown that the problem does not have finite models.

4.2.2 The need for existential quantification

In section 3.2, we saw that terms that are not lexically present in the theory can still imply infinity of the theory’s model. We found the injective and non-surjective function $pair(X,a)$ by instantiating the variable Y in $pair(X,Y)$ with a constant from the domain.

The same reasoning can be applied to any function f of $n \geq 1$ variables. We view f as a function of one variable, while the remaining (n-1) variables are instantiated with constants present in the domain. The good news is that we do not need to provide these constants ourselves. As long as we know that there *exist* constants that make our conjecture true, we do not need to know what these constants are. Since *existential quantification* is a construct of first order logic, an automated theorem prover can tell us whether such constants exist.

The generalised algorithm using existential quantification

1. Identify all functions of arity n in the theory, for $n \geq 1$.
2. For each such function, we create $2^n - 1$ new functions, by fixing one variable (that can occur more than once), and existentially quantifying over the others. For example, from $f(X,Y,Z)$, we create the functions

   $$f(X,*,*),\ f(*,X,*),\ f(*,*,X),\ f(*,X,X),$$

   $$f(X,*,X),\ f(X,X,*),\ f(X,X,X)$$

   where each “*” stands for a different existentially quantified variable (with a unique name).

3. For each of these functions, we construct the conjecture of injectivity and non-surjectivity.
4. An automatic theorem prover is used to check whether any of these conjectures is implied by the axioms of the theory.

Since the number of functions generated from a function f is exponential in the arity of f, it is often not feasible to check all of them. Since we are dealing with a computationally hard (impossible) problem, this is inevitable. It is, however, desirable to investigate ways to increase the likelihood of checking the right functions. One way to do this is to use *zooming,* which is described in section 4.3.

Example We illustrate how the algorithm can be applied to the example in section 3.2: In step 1, we identify all the functions of arity greater than zero. We find the functions $fst/1$, $snd/1$ and $pair/2$.

fst and *snd* are checked as described in 4.2.1. However none of these functions have the desired properties. Now, let us focus on the function $pair(X,Y)$. In step 2, we create the functions $pair(X,*)$, $pair(*,X)$ and $pair(X,X)$, where $*$ represents an existentially quantified variable. In step 3, let us focus on $pair(X,*)$, and create the conjecture for injectivity:

$$\exists C : \forall X, Y : pair(X,C) = pair(Y,C) \implies X = Y$$

the conjecture for non-surjectivity becomes:

$$\exists C, Y : \forall X : pair(X,C) \neq Y$$

Since it is necessary that it is the *same* function that is both injective and non-surjective, the existential quantification must range over both conjectures. Thus, we merge them into one:

$$\exists C : ((\forall X, Y : pair(X,C) = pair(Y,C) \implies X = Y) \wedge$$

$$(\exists Y : \forall X : pair(X,C) \neq Y))$$

We showed in section 3.2 that given the axioms B1-B3, this conjecture is true for $C = a$ and $Y = pair(a,b)$. In step four, we let an automated theorem prover check that the axioms of the theory do, indeed, imply the conjecture of injectivity and non-surjectivity of the given function. By the use of existential quantification, we do not need to specify the constants with which to instantiate the variables. We simply provide the theorem prover with the given axioms, together with the conjecture of injectivity and non-surjectivity, and repeat for each function that we wish to check.

4.2.3 Generalising equality

In example 3.3, we see that a function that is injective and non-surjective with respect to any reflexive relation implies infinity of the functions domain. By considering all reflexive relations, rather than just equality, as done in the naive algorithm, we significantly increase the likelihood of finding an injective and non-surjective function. Given a relation of arity $n \geq 2$, we can construct a number of new relations of arity 2, by adapting the technique of existential quantification in example 2. Each of these relations can then be checked for reflexivity, and used as equality relation in the conjecture of injectivity and non-surjectivity for each function we check. Naturally, for larger scale problems, this yields a huge number of test cases, and in practice it is not feasible to go through them all. We show a way to tackle this problem by the use of *zooming,* in section 4.3.

The generalised algorithm using reflexive relations

1. Use steps 1 and 2 of the algorithm in section 4.2.2 to generate test functions.

2. Identify all relations of arity n in the problem, for $n \geq 2$.

3. For each such relation, we create all of the 2-variable relations obtained by fixing two variables (each of them can occur more than once), and existentially quantifying over the others. For example, from $r(X, Y, Z)$, we obtain the relations

   $$r(X, Y, *),\ r(*, X, Y),\ r(X, *, Y)$$

   where each "*" stands for a different existentially quantified variable (with a unique name).

4. For each of these relations, we check if there is an assignment of constants to the variables that are represented by a *, that makes the relation reflexive. For example, for $r(X, Y, *)$,we construct the conjecture

   $$\exists C : \forall X : r(X, X, C)$$

   and use a theorem prover to test if this is implied by the axioms.

5. When we find a relation that is reflexive for some assignment of constants to the existentially quantified variables, we use this relation instead of equality when constructing the conjecture of injectivity and non-surjectivity. Note that we in the relation need to use precisely the constants that we proved make it reflexive. Thus, it is necessary to use the same quantification of this relation in the new conjecture. We need to both check for reflexivity *and* injectivity and surjectivity in the same scope:

   $$\exists C : (\forall X, Y : r(X, X, C) \wedge (r(X, Y, C) \Longrightarrow$$

   $$r(f(X), f(Y), C)) \wedge (\exists Z : \neg r(f(X), Z, C)))$$

It may seem unnecessary to check for reflexivity twice, however, since it is likely that a relation fails the first reflexivity check, we often do not need to perform the second check, which is a lot more expensive, since it is repeated once for each function.

6. For each conjecture we construct, we let an automated theorem prover check whether it follows from the axioms. If we for any conjecture get a positive answer, this means that there are no finite models of the given problem.

4.2.4 Searching for infinite subdomains

As seen in the example of section 3.4, if a function is injective and non-surjective on a subdomain, this implies infinity of the full domain. When looking at subdomains, we use a predicate to limit the set of elements that we wish to consider. When doing so, anything that happens outside of this set can be disregarded. When replacing equality for a reflexive relation, as shown in the previous section, it does not matter if the relation is not reflexive outside of the set. Thus, we only need to check for reflexivity on the subset that is limited by our predicate. Similarly, the desired properties of the functions we check need not hold outside of the set. By weakening the constraints in this way, we increase the likelihood of finding a function that fits our description.

The predicates used to define such subdomains are taken from those syntactically present in the problem. It is also possible to use the negation of a predicate in order to define the subset of elements for which the predicate is false, or to use the conjunction or disjunction of any number of predicates. In fact, we could create any predicate we want. However, sticking to the predicates that are present in the problem is a good limitation. The number of test cases is the product of the number of functions, equality predicates and limiting predicates that we consider.

Intuitively, the domain of an injective and non-surjective function is unlikely to be random; it seems more probable that the function's domain would consist of elements satisfying a predicate which occurs syntactically in the axioms.

The generalised algorithm using limiting predicates

1. Use steps 1 and 2 of the algorithm in section 4.2.2 to generate test functions.

2. Generate predicates of arity 1 in the same way as functions are generated in the previous step.

3. Construct all the possible pairs of functions generated in step 1, and predicates generated in step 2. For each such pair, we use an automated theorem prover to check if the subset defined by the predicate is closed under the function.

4. Generate equality relations in the same way as in step 3 of the algorithm in section 4.2.3. For each limiting predicate that is a member of at least one pair in step 3, we check if the equality predicate is reflexive under that limiting predicate.
5. We now have a number of pairs of compatible limiting predicates and functions, and a number of pairs of compatible limiting predicates and equality predicates. We merge the pairs of matching limiting predicates to obtain triples of limiting predicates, equality predicates and functions.
6. For each triple obtained in step 5, we use an automated theorem prover to check if the function is injective and non-surjective with respect to the equality predicate and the limiting predicate. It is also necessary to check again that the equality relation is reflexive with respect to the limiting predicate, and that the set defined by the limiting predicate is closed with respect to the function. This must be done simultaneously with the testing of injectivity and non-surjectivity of the function. This is to ensure that any existentially quantified variable scopes over the entire formula and thus instantiates to the same value for all clauses. If we find any triple, for which we receive a positive answer, this means that the problem cannot have finite models.

4.2.5 Searching for non-injective and surjective functions

As seen in section 3.5, non-injectivity and surjectivity of a function implies infinity of its domain. To use this fact to show that an axiom set lacks finite models, we can apply the algorithm described in section 4.2.2, but replace *injectivity* and *non-surjectivity* by *non-injectivity* and *surjectivity.*

Generalising equality, in the way explained in section 4.2.3 cannot be done to this method, however, as shown by the counter example in section 3.5. When applying the algorithm of limiting predicates in section 4.2.4, we skip step 4 and use equality ("=") as the only relation.

4.3 Zooming

When considering theories that contain a large number of terms and predicates, it is often not feasible to test all of the combinations for the desired properties. The use of existential quantification, as described in section 4.4.2, generates even more combinations, making it impossible to deal with them all, within a reasonable time limit. We must limit the number of combinations that we check. The naive algorithm simply checks the terms in the order in which they appear, and gives up with the time-out.

Given a theory for which no finite model has been found, we can weaken the theory by removing some of the axioms. If we cannot find a finite model of this new theory, we know that the removed axioms were not responsible for the absence of a finite model. We can continue removing axioms until we reach the smallest axiom set for which a finite model cannot be found. Now, we check only the combinations of terms and predicates found in this smaller theory. In this way, we are able to zoom in on the relevant part of the theory, and, in many cases, significantly reduce the number of test-cases.

The algorithm

1. The input is a set S of axioms for which no finite model has been found. We want to find the smallest subset of these axioms that lacks a finite model. We split the set of axioms into two halves, A and B. This is simply done by listing them in the order in which they occur, and splitting the list in half. Now, run the finite model finder *Paradox* on each of A and B. There are two possibilities:

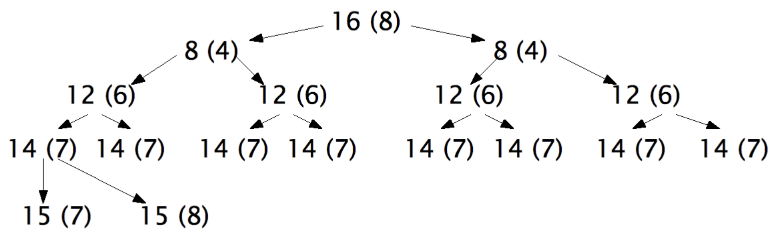
2. If, for any half, Paradox is *not* able to find a finite model within the given time-out, (we use a time-out of 2 seconds as default), we assume that there is no finite model of this subset, and go back to step 1 with this subset as input.

3. If, on the other hand, Paradox finds finite models of both A and B, this means that we have removed too many axioms. For each half, we *put back* half of the removed axioms. We thus split A and B further into four sets of axioms; $A1$, $A2$, $B1$ and $B2$, and run Paradox on each of $A \cup B1$, $A \cup B2$, $B \cup A1$ and $B \cup A2$. If Paradox is unable to find a finite model for any one of these axiom sets, use this set as input in step 1. Otherwise, we continue with step 4.

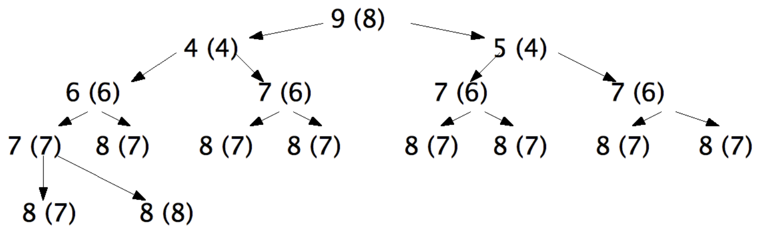
4. We repeat the procedure of step 3 recursively on each of the subsets for which we found a finite model; if the set of removed axioms is non-empty, we split the set of absent axioms in half and add each of them to the subset, to produce two larger subsets. If Paradox is unable to find a model for any one of them, we use it as input in step 1. Otherwise we repeat step 4 for these new subsets. If the set of removed axioms is empty, we have put back all of the axioms and restored the last input to step 1. We have then obtained an axiom set for which no axioms can be removed without making the theory finitely satisfiable. This is thus the smallest subset of the original input, for which no finite models can be found.

5. Now, we consider the terms (and predicates, where applicable) that occur in this subtheory only, and choose an algorithm from sections 4.2.2 - 4.2.5 to check the terms for the desired properties.

Complexity The complexity of zooming is in its worst case quadratic in the number of axioms. The worst case occurs when half of the axioms together make the model infinite. Consider the following example: a theory consists of 16 axioms, out of which 8 make the existence of a finite model impossible. In the worst case, it takes 8 splits, and 16 runs of Paradox, to gather all of the 8 axioms in the same subset:



With this, we have reduced the theory by one axiom only. We repeat the procedure, now with 15 axioms. Again, it takes 8 splits and 16 runs of Paradox to shrink the theory by one axiom. The procedure is repeated 8 times, shrinking the theory by one axiom each time, until we have zoomed in on the 8 axioms with no finite model.



For a theory of n axioms, it takes in the worst case $n^2/2$ runs of Paradox to find the smallest finitely unsatisfiable subdomain.

The worst case is, however, very unlikely for large n. Typically, only a small subset of the theory makes the existence of a finite model impossible. Our tests (see chapter 5), revealed that many problems, often of over a thousand axioms, have finitely unsatisfiable subsets of just 2-3 axioms. The probability of these axioms ending up in the same subset after the first split is thus much higher, which in each step means a reduction by half of the number of axioms. Thus, a larger n tends to increase the probability of a logarithmic complexity.

We discuss alternative approaches to the implementation of zooming in chapter 6.

4.4 Summary

We have presented a naive algorithm that searches for *injective and non-surjective* or *non-injective and surjective* functions. We have shown how this algorithm can be further generalised, by the use of existential quantification, reflexive relations and limiting predicates. Often, these generalisations yield a large number of test-cases. We resolve this by the use of *zooming*, which significantly reduces the number of combinations to check. With the algorithms presented in this section, we are able to classify all of the example problems of section 3. In the next chapter, we shall evaluate these algorithms on some standard test problems for ATP systems.

Chapter 5

Experimental Results

Infinox has been tested and evaluated using a subset of problems from the *TPTP Problem Library* [18], which is a comprehensive collection of test problems for automated theorem proving systems. In this section, we describe how the test problems were selected, and compare the performance of the different methods.

5.1 The test problems

The test problems for evaluation of Infinox were selected from the categories of the problem library that are meaningful for our purpose. This excludes problems already identified as *unsatisfiable*. Since it has already been shown that these do not have models, it is unnecessary to show that they lack finite models. For the same reason, problems identified as *theorems* are excluded from the set of test problems. If a problem has theorem status, it means that a theorem prover has established that its axioms and negated conjecture are unsatisfiable, meaning that the problem has no countermodel. Thus, it is superfluous to show that they lack finite countermodels. We also left out problems identified as *satisfiable* with known finite models. Similarly, *countersatisfiable* problems with known finite countermodels were disregarded. After eliminating the problems of the above kinds, we were left with a total of 1272 problems to focus on. These problems were taken from the following categories, as of April 2008:

Open: The abstract problem has never been solved. (27 problems).

Unknown: The problem has never been solved by an ATP system. (1075 problems).

Satisfiable: Some interpretations are models of the axioms. We considered only problems with no known finite model. (122 problems)

Countersatisfiable: Some interpretations are models of the negation of the conjecture. We considered only problems with no known finite counter model. (48 problems)

5.2 The results

We tested on a 2x Dual Core processor operating at 1 GHz, with a time-out of 15 minutes per problem, and a time-out of two seconds for each call to E and Paradox.

In total, Infinox classified 390 out of the 1272 test problems as finitely unsatisfiable. Since the actual number of the test problems that are finitely unsatisfiable is unknown, this means that we have identified *at least* 30% of them. The number of successfully classified problems from each category are as follows:

Unknown: 366 (1075)

Open: 3 (27)

Satisfiable: 21 (122)

CounterSatisfiable: 0 (48)

It may seem like an astounding result that as many as three open problems were classified by Infinox. However, the complexity of the problems cannot be measured in the same way as for traditional theorem provers. Many of the most difficult problems trivially have no finite model. As an example, one of the classified open problems is Goldbachs conjecture, which is one of the oldest unsolved problems in number theory. The number-theoretical nature of this problem directly implies that any model must be infinite. Despite this, finite model finders have fruitlessly attempted to solve this problem (and many others), according to the TSTP (Thousands of Solutions from Theorem Provers) Solution Library [19]. With a tool like Infinox, they no longer need to.

5.3 Evaluation

Since there is no other tool similar to Infinox, it is not an easy task to evaluate the overall test results. Instead, we compare the performance of the different methods.

5.3.1 Abbreviations

For the purpose of readability, we introduce the following abbreviations of the methods:

R Search for injective and non-surjective terms, try all reflexive predicates found in the problem as equality relation, including "=". This method is introduced in section 3.3, and explained in detail in section 4.2.3.

RP Same as R, with the added use of limiting predicates. With the addition of a predicate P, such that $P(X)$ evaluates to true for all X, the method RP would subsume the method R. In order to evaluate the benefits of the use of pure limiting predicates, such a predicate was not added when performing these tests. This method is introduced in section 3.4, and explained in detail in section 4.2.4.

SNI Search for surjective and non-injective terms.

The letter **"Z"** appended to the above codes indicate the added use of zooming to select test terms an predicates.

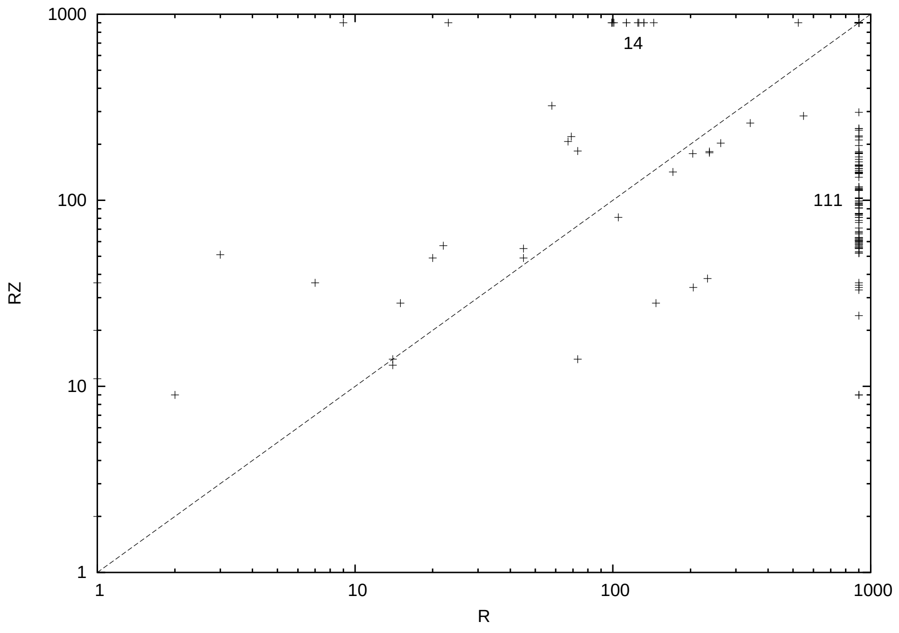
5.3.2 Scatterplots

When evaluating a method, we are not only interested in the total number of successful tests, but also in what the given method contributes to the overall result. If there are a large number of problems solvable by one method but not another, and vice versa, this indicates that the use of a combination of these methods would be beneficial. We are looking for the State Of The Art Contributors (SOTAC) [17]; the methods that were able to classify problems that no other method classified.

To facilitate a clear overview of the results and enable an easy comparison of the methods, we present the result data as scatterplots.

Each axis represents a method's running time in seconds, using a logarithmic scale. Each cross represents a problem. Crosses below the diagonal line are problems where the method represented by the vertical axis performed better, i.e. it solved the problem in the shortest time. Similarly, crosses above the diagonal line are problems for which the method represented by the horizontal line performed the best. Since we used a time-out of 900 seconds when testing, crosses located along the 900 mark are problems that the corresponding method was unable to solve. Consequently, since the majority of the problems were not solved, these are represented by a cross in the upper right corner, meaning that both methods either timed out or ran out of test terms.

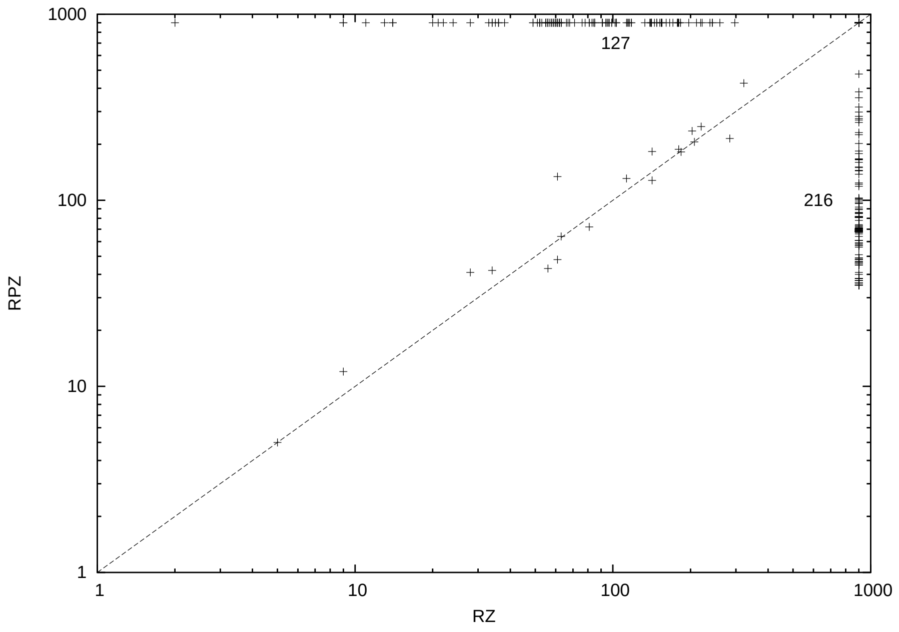
Zooming or no zooming? The scatterplot in figure 1 below shows that the use of zooming as a way to select test terms significantly increases the ratio of classified problems.



The horizontal axis represents the method R, while the vertical axis corresponds to its zoomed counterpart, RZ. We note that, while some dozen problems were solved by both methods, there are a total of 111 problems along the 900 mark of method R, which were solved by the zoomed version alone. Only 14 problems were solved by R alone.

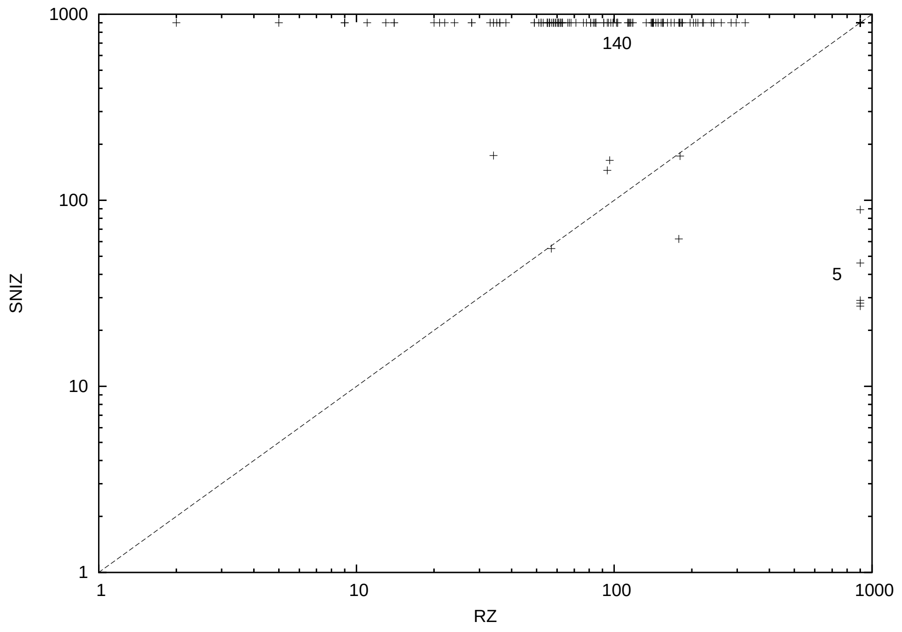
The results indicate that it is in most cases preferrable to use the zoom feature. Only in a few cases did the plain R method succeed when the zoomed version did not. Still, it may be of value try the plain R method when RZ has failed. The two methods together classified just over 44% of the total number of problems that were successfully classified by Infinox.

Limiting predicates or no limiting predicates? In the figure below, we see how the use of limiting predicates affects the result. The horizontal axis corresponds to the method RZ, while the vertical axis corresponds to RPZ.



A number of crosses are gathered around the diagonal line, indicating that the performances of both methods were very similar on the problems that both methods were able to solve. Even more interesting are the crosses gathered along the 900 second mark of both methods. We see that 127 problems were classified by RZ but not by RPZ, and 216 problems were classified by RPZ but not RZ. This shows that the two methods complement each other very well and should ideally be combined. RPZ performed somewhat better, with a total of 235 classified problems, compared to RZ, which classified 146. Together, the two methods account for almost 93% of the total number of classified problems.

Injective and non-surjective or non-injective and surjective? In the figure below, we compare the results of the method that searches for terms that are injective and non-surjective with the results of the method that searches for non-injecitve and surjective terms. We look at the zoomed version of both methods. The horizontal axis corresponds to RZ, while the vertical axis corresponds to SNIZ.



We see that very few problems were solved by SNIZ. One reason for this is that in RZ, we consider all reflexive predicates as equality relation, while in SNIZ we are limited to using standard equality ("="). With fewer test cases, we are less likely to come across a term that fits our description. Another reason may be that the test problems are initially constructed by people. It may be more intuitive to deal with a certain kind of functions (injective and non-surjective) rather than the reverse (non-injective and surjective), and thus these functions occur more frequently.

We conclude by the poor overall results that the SNIZ method is not practical to use. However, since SNIZ classifies 5 problems for which all of the other methods fail, it may be worthwhile to investigate ways in which to improve this method.

Reflexive predicates The added use of reflexive predicates as equality relation, as explained in sections 3.3. and 4.2.3, has shown to be a useful generalisation. It accounts for 41 out of the 146 problems classified by RZ, and 13 out of the 235 problems classifed by RPZ. However, since regular equality performed better, it should ideally be tested before any other predicates.

Existentially quantified variables Using existential quantification to generate terms and predicates has proved to be of major significance to the results. In 104 out of the 146 problems classified by RZ, and 219 out of the 235 problems classified by RPZ, the identified terms and/or predicates include variables that are existentially quantified.

Term depth Out of all of the 390 classified problems, 100% of the identified terms and predicates had depth 1. This does not exclude the possibility of there being terms or predicates of larger depths that possess the desired properties. These may exist, however they are much more difficult to find.

5.3.3 Time-outs

Infinox has three different time-out settings. The settings that are most suitable highly depend on the nature of the problem and the global time-out.

The global time-out setting The global time-out should ideally be as long as possible. In our test results, the majority of the identified terms were found in the time interval of 100 and 500 seconds, with an E limit of 2 seconds. Thus, a time-out of at least 10 minutes per method is recommended. This should of course be adjusted according to the time-out settings of E and Paradox.

E time-out setting Naturally, a longer time-out for E means that it is more likely to be able to prove our conjectures. However, it may also mean that there won't be enough time to go through all of the tests. The E time-out should be set in accordance with the global time-out and the size of the problem.

Paradox time-out setting Using the zoom feature requires a time-out setting for Paradox. In our tests, we used a time-out of 2 seconds, which our results indicate is a reasonable limit. This is also the time limit used as default. With a longer limit, we can decrease the risks of the zoomed problem having a finite model which was not found by Paradox in the given time limit. However, with a longer limit, there is the risk of global time-out before the zooming has finished and any terms have been tested.

5.3.4 The less successful methods

The four methods R, RZ, RPZ and SNIZ together classify all of the in total 390 problems classified in our tests. A number of variations on these methods were tested, but these did not add to the overall result. These include the use of generation of terms and predicates up to a given depth. Our tests indicate that if there exist terms with the desired properties, these are often syntactically present in the problem. The generation of terms also yields a large number of test cases, often too many to process before the time-out.

Another less successful attempt was to combine the limiting predicates by conjunction, disjunction and negation. For example, the predicates $p(X)$and $q(X)$ would yield the new predicates $p \wedge q$, $p \vee q$,$\neg p$ and $\neg q$. These would then be used in addition to the plain limiting predicates.

There are infinitely many ways to generate terms and predicates to test. If we haven't found a term with the desired properties at the lower term-depths, the chances of generating "the right term" - if it exists - are just too slim.

5.4 Testing

During the implementation phase, Infinox was continuously tested on 737 problems known to be finitely satisfiable to ensure that the program does not wrongly infer the non-existence of finite models for these problems. While the correctness of the methods has been proved, the testing helped to locate bugs in the code, such as misplaced parentheses that changed the intended quantification scope of variables.

Chapter 6

Future work

Since we are dealing with a semi-decidable problem, the perfect algorithm for proving finite unsatisfiability will never be invented. On the other hand, this means that there are infinitely many opportunities to improve the results, either by enhancing the methods we have, or by inventing new ones.

6.1 Refine the methods

Perhaps the simplest way to improve our methods is to refine the techniques of term/predicate selection. The more terms we have to test, the more likely it is that one of them has the desired property. Since the running time is generally bounded with a time-out, it is equally important to rule out the terms that for various reasons are unnecessary to test. The relationship between the size of the problem and the required global and local time-outs is something that should be analysed in order to achieve the best possible results.

6.1.1 Selection of terms and predicates

The selection of test terms and predicates is of great importance to the result. It is thus important to decide what terms are meaningful to test and what limitations we should make to reduce the search space.

What to include At present, terms and predicates are selected based on those syntactically present in the problem. This has proved to work well for many problems, however it may also be of value to introduce new terms. By the use of skolemization, existential quantifiers are replaced with skolem functions, which may be used as new test terms. If we can find a function f and a relation r, such that

$$\forall Y : \exists X : r(f(X), Y)$$

i.e. f is surjective with respect to r, then we can add the axiom

$$\forall Y : r(f(g(Y), Y)$$

to the theory, for a new function symbol g, and use the method that searches for injective and non-surjective functions on g.

One may also consider new ways of combining terms and predicates, and analyse what combinations are meaningful to test. The generation of new limiting predicates by the use of conjunction, disjunction and negation has been tested without major success. It would be interesting to look into other ways in which to create limiting predicates. For example, one may introduce a predicate for each function we test, that is true for an element iff the function is injective in that element.

What to exclude When considering a problem of larger size, there may not be enough time to test all the combinations of terms and predicates. The use of zooming has helped in many of these cases, by focusing on a smaller part of the problem where a finite model cannot be found. But, as we have seen, zooming does not work for all problems. In fact, it never will, since this would imply the decidability of a semi-decidable problem. Still, further refinement of the zooming procedure could lead to even better results. One possible way to improve the performance of zooming is to remember what axioms cannot be removed without making the model finite. For example, if we start with the axiom set $\{A, B, C, D\}$, and find that $\{B, C, D\}$ has a finite model, then we do not need to remove A in future steps.

In order to reduce test terms further, it would also be of great value to investigate what kind of terms are meaningful to test, in terms of term depth and other attributes. As an example, it is meaningful to test the term $f(g(X), h(X))$ for injectivity only if both $g(X)$ and $h(X)$ are injective.

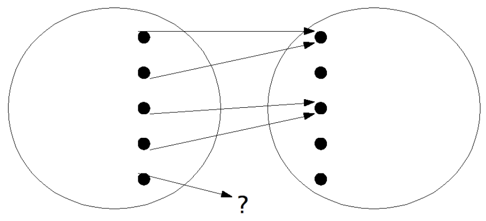
6.1.2 Find the perfect time-out

A very important question is how to balance the global and local time-outs. Ideally, the global time-out would be unlimited, and the local time-outs set to as long as possible. Since this is generally not practical, we must adjust the local time-outs to match the size of the problem and the time we are willing to wait for a result. Finding the best balance requires a lot of experimentation, but significant improvements to performance can be made by tuning the parameters correctly.

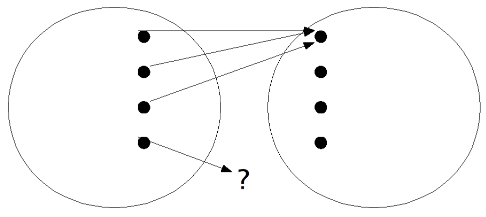
6.1.3 Other variatons

We have seen how we can weaken the constraints on the terms by the use of limiting predicates. Another alternative is to consider terms that are injective in all but a limited number of elements, and adjust the constraint of non-surjectivity accordingly.

If a function f maps any n elements to k elements and is injective in all other elements, where n >= k, and at least (n-k) + 1 elements in the codomain are uncovered by the function, then the domain must be infinite, as illustrated by the pictures below.



With four elements that are mapped to two elements, and three elements not covered by the function, we must introduce a fifth element to the domain, which must be mapped to a sixth elemement, and so on, given that the function is injective in all other elements.



Here, three elements are mapped to a single element. With three elements not covered by the function, we must introduce a fourth element to the domain, which in turn must be mapped to a fifth element, given that all elements except the first three map to different elements.

6.2 Develop new methods

There are many properties that a problem may possess which imply infinity of any of its models. By exploring such properties and finding ways to express them in a way that can be understood by a first-order theorem prover, we can extend Infinox with new methods.

An example of a property that can be used to show infinity of models is this: *"if there for any model of a problem exists a smaller model of the same problem, then all models must be infinite"*.

The proof is simple. If the problem has finite models, then there must exist a smallest model. For this model, the statement is contradictory.

To be able to express this property in first-order logic, we need to use some tricks to avoid quantification over models. A non-surjective function can be used to "shrink" a model, by applying it to each entry in each function table.

This particular method has been implemented and evaluated without major success, which is why it is not yet a part of Infinox. It may, however be worth looking into alternative ways to implement this method.

Another example is the existence of a relation r, that is *irreflexive*, *transitive* and *serial,* i.e.

1. $\forall X : \neg r(X, X)$
2. $\forall X, Y : r(X, Y) \wedge r(Y, Z) \Longrightarrow r(X, Z)$
3. $\forall X : \exists Y : r(X, Y)$

Given that the domain contains at least one element, say a_1, these axioms yield an infinite sequence of unique elements:

By axiom 3, there exists an element, a_2, such that $r(a_1, a_2)$. In the same way, there exists an element $a_{(i+1)}$ for every element a_i, such that $r(a_i, a_{(i+1)})$. By axiom 2, we get $r(a_i, a_j)$, which by axiom 1 implies $a_i \neq a_j$ for all i, j such that $i < j$. This sequence of elements can thus not be finite: Suppose there is a last element, a_k. By axiom 3, there exists an element $a_{(k+1)}$ such that $r(a_k, a_{(k+1)})$, and $a_i \neq a_{(k+1)}$ for all a_i such that $1 \leq i \leq k$. Thus, every element of the sequence gives rise to a new, unique element.

This method has been implemented, but not yet evaluated. Evaluation and investigation of possible generalisations of this method is left as future work.

6.3 Evaluation

A thorough evaluation of the performance of the different methods and time-out settings is crucial to be able to create improved future versions of Infinox, both to adjust parameters, and to get an insight in what can be improved upon. A useful improvement that could be made with this knowledge is to automatically base the local time-outs on the given global time-out and problem size. The aim is to let Infinox automatically decide the best approach for a given problem, and thus save the user having to go through each method one by one.

6.4 Other uses

To simplify the addition of new properties to Infinox is something that should be considered in future implementations. This could be further generalised to allow the user to search for terms or predicates with any type of property. One way to do this could be to provide a specific language to let the users program these properties themselves.

Chapter 7

Conclusions

We have introduced Infinox, a new tool that can be used to disprove the existence of finite models of first-order theories. Infinox is especially well suited as a complement to finite model finders; by proving that a finite model cannot exist, it is no longer necessary to search for one.

Disproving the existence of finite models We have demonstrated how terms that are *injective and non-surjective,* or *non-injective and surjective* imply the non-existence of finite models. Infinox automates the process of finding such terms. In the search for terms with the desired properties, we focus on terms that are syntactically present in the theory.

Existential quantification to create test terms With the use of *existential quantification,* variables are instantiated by a constant in order to create new test terms of the desired arity. This addition has improved the results considerably. A majority of the identified terms in our test problems are existentially quantified.

Zooming to reduce test terms By the use of *zooming*, we focus on a smaller part of the problem where a finite model cannot be found, and thus reduce the number of terms to test. Zooming works very well for a lot of problems, while it fails miserably for some others. A major reason for this is that there is no guarantee that we zoom in to the "right" part of the problem. It is possible that we zoom in to a subproblem for which none of our methods work, while another part of the problem holds a function of the kind we are looking for. Another possibility is that we zoom in to a subproblem that does in fact have a finite model. The *zooming* feature relies on the assumption that if no finite model has been found within a set time limit, then no finite model exists.

Generalising the properties By generalising the properties that we look for, we are able to detect more terms that imply the non-existence of a finite model.

Reflexive relations can be used as a complement to regular equality when defining the property of injectivity and non-surjectivity. For surjectivity and non-injectivity we need stronger constraints on the relation, which is why only equality is used for this property at present.

With *limiting predicates,* we can identify terms for which the given property is valid on an (infinite) subset of the domain, as defined by the predicate. In this way we can identify terms that do not have the desired property when the full domain is considered.

Pros and cons With the above generalisations, we get an increased number of tests to perform; one for each combination of terms, reflexive predicates and limiting predicates. More test cases increases the likelihood of a term with the desired property being among them. When using a global time-out, this can also be a disadvantage, since there might not be enough time to go through all combinations. Our results indicate that the use of limiting predicates should be used as a complement to the general method, since they are both able to classify some problems that the other method cannot.

The results Infinox has classified over 30% of the standard first-order logic test problems as finitely unsatisfiable. Many of these problems have never before been solved (nor classified) by an automated system. Since this classification has never been done automatically before, the new problem status definition “FinitelyUnsatisfiable” has been added to the TPTP libary of standard problems for automated theorem provers.

Bibliography

[1] J. Bruggeman, J. Kamps, M. Masuch, B. O'Nuallain, *Automated Reasoning for Theory-Building in the Social Sciences,* CCSOM Working Paper 96-144, 1996.

[2] A. Church, *A Note on the Entscheidungsproblem,* Journal of Symbolic Logic, 1, pp. 40-41, 1936.

[3] K. Claessen, N. Sörensson, *New Techniques that Improve MACE-style Finite Model Finding,* In Proc. of Workshop on Model Computation (MODEL), 2003.

[4] S. Cook, *The Complexity of Theorem Proving Procedures,* Proc. 3rd ACM Symp. on the Theory of Computing, ACM, pp. 151-158, 1971.

[5] N. Eén, N. Sörensson, *The MiniSat Page, http://www.minisat.se.*

[6] B. Han, S. Lee, H. Yang, *A Model-Based Diagnosis System for Identifying Faulty Components In Digital Circuits,* Appl. Intell., 10(1), pp. 37-52, 1999.

[7] M. Kaufmann, J. Strother Moore, *Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification,* Spanish Royal Academy of Science (RACSAM), 98(1), pp. 181-196, 2004.

[8] P. Lucas, *The Representation of Medical Reasoning Models in Resolution-Based Theorem Provers,* Artificial Intelligence in Medicine, 5(5), pp. 395-414, 1993.

[9] W. McCune, *Prover9 and Mace, http://www.cs.unm.edu/~mccune/mace4*

[10] W. McCune, *Solution of the Robbins Problem,* Journal of Automated Reasoning, 19(3), pp. 263-276, 1997.

[11] P. Norvig, S. Russell, *Artificial Intelligence - A Modern Approach,* Prentice Hall, 2003.

[12] A. Riazanov, A. Voronkov, *Vampire,* Proc. CADE-16, LNAI 1632, 1999.

[13] A. Robinson, A. Voronkov, *Handbook of Automated Reasoning,* MIT Press, 2001.

[14] J.A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle,* Journal of the ACM (JACM), 12(1), pp. 23-41, 1965.

[15] S. Schulz, *The E Equational Theorem Prover, http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover*

[16] R. M. Smullyan, *First-Order Logic,* Courier Dover Publications, 1995.

[17] G. Sutcliffe, C. Suttner, *Evaluating General Purpose Automated Theorem Systems,* Artificial Intelligence, 131:39-54, 2001.

[18] G. Sutcliffe, C. Suttner, *The TPTP Problem Library for Automated Theorem Proving, http://www.cs.miami.edu/~tptp*

[19] G. Sutcliffe, C. Suttner, *The TSTP Solution Library for Automated Theorem Proving, http://www.cs.miami.edu/~tptp/TSTP*

[20] G. Sutcliffe, C. Suttner, *TPTP Format for Problems, http://www.cs.miami.edu/~tptp/TPTP/QuickGuide/Problems*

[21] J. Vännänen, *A Short Course on Finite Model Theory,* Department of Mathematics, University of Helsinki, 1994.

[22] C. Weidenbach et al., *The Spass Homepage,* Max-Planck-Institut Informatik, *http://www.spass-prover.org,* 2008.

[23] H. Zhang, J. Zhang, *SEM: a System for Enumerating Models,* In Proc. of International Joint Conference on Artificial Intelligence (IJCAI'95), 1995.

Appendix A

Using Infinox

Requirements Infinox requires installations of E [15] and Paradox [3]. These must either be located in the working directory, or added to the PATH variable.

The problems must be expressed in the TPTP-format [20].

Temporary files During execution, Infinox creates temporary files, which are stored in the *temp* directory. The temp directory is not removed after execution, but all of the temporary files created by that program run are deleted, given that the program terminates normally.

Example use The following command will run Infinox on the file *problem.tptp,* using reflexive relations and limiting predicates, and 4 seconds time-out for E and Paradox, and a global time-out of 10 minutes.

```
infinox problem.tptp -r -p -elimit=4 -plimit=4 -timeout=600
```

Output In the case of time-out, or if Infinox runs out of terms to test,

"# RESULT: GaveUp"

is printed to stdout. If a term of the desired property is found,

"#RESULT: FinitelyUnsatisfiable"

is printed, together with the found term, and relation and predicate, where applicable.

Flags

-v print details of the progress of the program to stdout

-filelist=<filename> run program on all problems listed in the given file

-dir=<directoryname> run program on all problems found in the given directory

-o=<filename> print results to the given file

-elimit=<n> time limit in seconds for each call to E (default=2)

-plimit=<p> time limit in seconds for each call to paradox (default=2)

-timeout=<n> timeout in seconds (default=3600)

-r use all reflexive predicates as equality relation

-p use limiting predicates, can be used in combination with:

- **-and** use pairwise conjunction of limiting predicates
- **-or** use pairwise disjunction of limiting predicates
- **-not** use negation of limiting predicates

-g=<n> generate all terms of depth n from all function symbols present in the problem

-zoom use zoom feature to select terms

-ino search for injective and non-surjective terms (default)

-oni search for non-injective and surjective terms. NOTE: invalidates the -r flag

-m2 find a smaller model by applying a non-surjective function

-**sr** search for a relation that is serial, irreflexive and transitive

-term=<t> test the the term t only