Uninterpreted Functions modulo Associativity, Commutativity and Inverse Functions

Raphael Spörk

Uninterpreted Functions modulo Associativity, Commutativity and Inverse Functions

Master's Thesis

at

Graz University of Technology

submitted by

Raphael Spörk

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

25 November 2013

© Copyright 2013 by Raphael Spörk

Advisor: Prof. Roderick Bloem

Nicht Interpretierte Funktionen modulo Assoziativität, Kommutativität und Inverse Funktionen

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Raphael Spörk

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie (IAIK),
Technische Universität Graz
A-8010 Graz

25. November 2013

© Copyright 2012, Raphael Spörk

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Prof. Roderick Bloem

Abstract

Equalities are ubiquitous in mathematics and science. One question that arises in that context is whether or not an equality logically follows from a set of equalities containing uninterpreted functions. An uninterpreted function is a function that only has a name and an arity associated with it. Congruence closure algorithms can be used to solve this problem. This work proposes three extensions to congruence closure algorithms, namely congruence closure modulo associativity and commutativity, congruence closure modulo inverse functions, as well as congruence closure modulo associativity, commutativity and inverse functions.

Standard congruence closure algorithms only enforce one property on the occurring uninterpreted functions symbols, the so called functional consistency property. This means that two applications of a function with congruent parameters have to have the same result. We extend the congruence closure algorithm inside Z3 with the existing theory of uninterpreted functions modulo associativity and commutativity and we propose an extension to the congruence closure algorithm that allows to handle inverse uninterpreted functions as well. In the case of associative and commutative uninterpreted functions, one has to extend the notion of congruence between two terms, because here the associative and commutative law can be used to exchange the order of the arguments. Therefore we extend the term matching of the congruence closure algorithm inside Z3 to take the associative and commutative property of functions into account. This is done via term flattening. Furthermore, in the presence of associative and commutative functions, it is very important that one uses a reduction ordering on terms that is capable of handling associative and commutative functions, here we use an existing ordering for associative and commutative uninterpreted functions. We also extend the deduction rule of the congruence closure to learn new equalities that follow from equalities with associative and commutative functions. In the case of inverse uninterpreted functions the standard congruence meaning can be used. However, the procedure has to be extended to recognize and remove applications of inverse functions to each other, in order to simplify terms. Also the deduction rule of the congruence closure algorithm has to be extended to deduce new equalities that are a consequence of equalities containing inverse uninterpreted functions. Finally, we propose a way of combining the two previous algorithms to get a congruence closure algorithm that works with both associative and commutative uninterpreted functions and uninterpreted functions that are inverse to each other. We also present an evaluation of the algorithms using our implementation inside the Z3 theorem prover.

Kurzfassung

Gleichungen sind allgegenwärtig in der Mathematik und der Wissenschaft. Eines der Probleme, das sich mit Gleichungen ergibt, ist die Frage, ob eine Gleichung logisch aus einer Menge von Gleichungen mit nicht interpretierten Funktionen folgt. Eine nicht interpretierte Funktion ist eine Funktion, welche nur ein Funktionssymbol und eine Arität besitzt. Dieses Problem lässt sich mittels eines Congruence Closure Algorithmus lösen. In dieser Arbeit stellen wir drei Erweiterungen des Congruence Closure Algorithmus vor, welche diesen um assoziative und kommutative Funktionen, zueinander inversen Funktionen und der Kombination von beiden erweitern.

Normalerweise fordern Congruence Closure Algorithmen nur eine Eigenschaft von den nicht interpretierten Funktionen, nämlich die funktionale Konsistenz. Hierbei handelt es sich um die Eigenschaft, dass zwei Instanzen der selben Funktion gleich sein müssen, wenn die Parameter gleichwertig sind. Wir erweitern den Congruence Closure Algorithmus des Z3 Theorem Provers mit der existierenden Theorie der nicht interpretierten Funktionen modulo Assoziativität und Kommutativität. Des Weiteren stellen wir einen Congruence Closure Algorithmus vor, welcher inverse nicht interpretierte Funktionen handhaben kann. Im Falle von assoziativen und kommutativen nicht interpretierten Funktionen muss der Begriff der funktionalen Konsistenz erweitert werden, da mittels Assoziativität und Kommutativität die Reihenfolge der Parameter einer Funktion verändert werden kann. Deshalb erweitern wir den Algorithmus insofern, als dass beim Vergleichen von Termen deren Assoziativität und Kommutativität berücksichtigt wird. Des Weiteren wird eine spezielle Ordnung für Terme benötigt, welche auch mit assoziativen und kommutativen Funktionen funktioniert. Eine weitere wichtige Erweiterung ist die Lernfunktion des Congruence Closure Algorithmus, denn auch hier müssen die zusätzlichen Eigenschaften der Funktionen berücksichtigt werden. Im Falle von nicht interpretierten Funktionen, die invers zueinander sind, kann die normale Definition der funktionalen Konsistenz verwendet werden. In diesem Fall betreffen die Erweiterungen die Anwendung von inversen Funktionen aufeinander, welche nicht nur erkannt, sondern auch entfernt werden muss, da ansonsten nicht die einfachste Form eines Terms gefunden wird. Außerdem muss die Lernfunktion des Algorithmus erneut angepasst werden, um Gleichungen, die aus der Präsenz von inversen Funktionen folgen, auch erkennen zu können. Weiters stellen wir eine Möglichkeit der Kombination dieser beiden Algorithmen vor, um einen Congruence Closure Algorithmus zu erhalten, der sowohl assoziative und kommutative als auch zueinander inverse Funktionen handhaben kann. Schlussendlich präsentieren wir noch die Ergebnisse unserer Evaluierung, welche auf unserer Implementierung der Algorithmen im Framework des Z3 Theorem Prover basieren.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Listings

Acknowledgements

There are a lot of people who I like to thank for their support during the creation of this thesis. First of all I want to thank my supervisor Roderick Bloem for his support and the fact that he awoke my interest in the fields of formal methods for verification and synthesis. Without his ideas and theoretical background knowledge, I would have never been able to finish this thesis.

Secondly, I want to thank Bettina Könighofer, Georg Hofferek and Robert Könighofer for their feedback and ideas during the creation of this work and their help with the theoretical background of this thesis. I also want to thank Leonardo de Moura for his help regarding the Z3 theorem prover.

Special thanks goes to my friends and study colleagues for their support and the countless hours we spent studying and working for the university during my study program. Last but not least, I want to thank my parents for their moral and financial support during my time at the university.

Raphael Spörk
Graz, Austria, November 2013

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [1].
- Conchon et al. [12] kindly allowed us to use their benchmarks for the performance evaluation of our implementation.

Chapter 1

Introduction

1.1 Background and Motivation

Software becomes more important in our lives everyday, but software often contains numerous flaws. These flaws not only cost the companies a lot of money, sometimes they can also endanger the lives of human beings. Therefore the development of software systems that are bug-free is very important, especially for systems where the health of humans is at stake. There have been numerous examples where bugs in software lead to huge costs or even the loss of human life. For example, in 1996 an Ariane-5 rocket launched by the European Space Agency was self-destructed due to a bug in the software [20]. The incorrect handling of a data conversion exception was the reason for the self-destruction and the estimated costs of this bug were 370 million US dollar. However, an even more severe bug occurred in 1992. Here a bug in the software system of the MIM-104 patriot system caused the clock of the software system to drift [39]. The MIM-104 patriot system is used to intercept enemy missiles. After about 100 hours of operation the system clock of the intercept system was wrong by about $\frac{1}{3}$ of a second, which caused the intercepting rocket to miss its target by about 600 meters. An Iraqi Scud rocket went on to hit a barrack of the US military and 28 American soldiers were killed in the incident. More examples for severe software bugs can easily be found [21].

A big part of the development of modern software systems is dedicated to guarantee the correctness of the developed software. There exist a lot of methods which try to ensure the quality of a given program. These methods range from informal methods like testing and simulation [34] to formal methods like synthesis or proofs of correctness for software systems. However, the problem with testing is that it is impossible to cover all possibilities due to the large input space. Then again the problem with formal methods most often is that they are either not available or don't scale for real world applications and can therefore not be used [10]. Ongoing research in the area of formal methods is currently changing this fact, today there exist tools that help with the verification of software or tools that are capable of synthesising software systems.

1.1.1 Satisfiability and Satisfiability Modulo Theories

As mentioned above, the construction of correct software systems is as important as never before and one technology that is often used in this context is satisfiability modulo theories (SMT) [29, 33]. SMT is an extension of the classical satisfiability (SAT) problem, which is one of the most fundamental problems of theoretical computer science [23] since it belongs to the set of the NP-complete problems. The classical SAT problem is the question whether or not there exists an assignment of true or false values to the literals of a propositional formula such that the overall formula becomes true. A propositional formula is satisfiable if there exists an assignment of true and false values to its literals that makes the whole formula true. If such an assignment does not exist the formula is unsatisfiable. For example the formula

$\phi = a \vee b$ is satisfiable, because the overall formula is true when both a and b are assigned to true.

In the case of SMT one considers the satisfiability of first-order logic formulas regarding a theory. The theory is used to restrict the meaning of certain predicates and function symbols in the first-order logic formula. For example in the theory of linear integer arithmetic the functions symbols $+$ and $-$ represent addition and subtraction for integers and the predicates $<$ and $=$ also have the usual mathematical meaning associated with them. As mentioned earlier, SMT formula can be used for program verification [24] for example by checking whether certain assertions hold inside a program or not. In this context one is often not interested whether or not the assertion is true under any interpretation of the function $+$, but wants to know whether the formula is satisfiable with the mathematical interpretation of the function $+$. This is what background theories do, they fix the meaning of certain functions and predicates. Manifold theories exist today, some of the more prominent theories cover linear integer arithmetic, uninterpreted functions, array, and bit vectors.

1.1.2 Uninterpreted Functions and Congruence Closure

As previously mentioned, one important theory in the context of satisfiability modulo theories is the theory of uninterpreted functions. In this theory function symbols do not have any predefined meaning and the only predicate that is defined is $=$ which has the usual mathematical interpretation [33]. A congruence closure algorithm can be used to solve the satisfiability problem of a first-order logic formula containing equalities and inequalities over uninterpreted functions. Famous congruence closure algorithms were developed by Nelson and Oppen [35], Shostak [43], and Downey et al. [19]. More recent algorithms include the one developed by Nieuwenhuis and Oliveras [37].

The theory of uninterpreted functions is interesting because it can be used as an abstraction in various verification and synthesis applications. Burch and Dill [9] use uninterpreted functions as an abstraction in their verification tool for the control logic of pipelined microprocessors. They use uninterpreted functions to abstract the commands executed by the CPU like reading or writing to a register. Since some of the abstracted functions fulfil the associative and commutative property they have stated that the availability of a theory for uninterpreted functions modulo associative and commutative functions would be great. The correctness of the processor is then checked by comparing the implementation description with the specification of the processor. For this purpose both the implementation description and the specification are described by transition systems. Based on the idea of Burch and Dill [9] Velev and Bryant [44] created a tool that is capable of handling memory and functional units that have an undetermined latency. Gulwani and Tiwari [24] have used a combination of linear integer arithmetic and uninterpreted functions to verify assertions inside programs.

We developed a lock-set synthesis tool, which automatically synthesis lock-sets for simple C programs. Inside our tool we also use uninterpreted functions as abstraction for arithmetical operations, since we are not interested in the exact result, but only that the program behaves the same as the sequential program would. However since arithmetical operators like $+$ and $\cdot$ are both associative and commutative we also need uninterpreted functions that are associative and commutative.

Satisfiability in the Context of Congruence Closure

Since we will use SMT solvers to solve the problem of equality with uninterpreted functions, we have to clarify the meaning of satisfiability in this context. As stated earlier, the congruence closure algorithm solves the problem of deciding whether or not an equality follows logically from a set of equations or not. However, we will consider the problem whether or not a set of equations and inequalities containing uninterpreted functions can be satisfied. This means the algorithm will search for values for all of the occurring constants and function applications such that all the equalities and inequalities are fulfilled. If such values can be found we call these values a model for the set of equalities and inequalities and the set is satisfiable. Note that every satisfiable set has a model and if we can find a model for a set the set must be satisfiable. If no model for a set exists then we call the set unsatisfiable. In the case of unsatisfiable sets we are interested in the contradiction. Therefore, it is important that the algorithms are capable of producing proofs for the unsatisfiability of the given set of equalities and inequalities.

1.2 Problems Addressed in this Thesis

The problem of equality in the context of uninterpreted functions has been addressed in the literature before [35, 43, 19, 3, 12, 30, 37], see Section 2 for a detailed discussion of the of similarities and differences. The congruence closure algorithm can be used to solve the problem whether a given equality follows from a set of equalities over uninterpreted functions. The only constraint on the uninterpreted function is that if $x = y$ then $f(x) = f(y)$, the so called functional consistency property, no other properties of f are known or assumed by the algorithm. In our work we focus on three extensions of the standard algorithm. First congruence closure modulo associativity and commutativity. Here some of the uninterpreted functions are assumed to fulfil the associative and commutative property. The second extension is congruence closure modulo inverse functions, here two uninterpreted function symbols can be inverse to each other, which means that the following property holds for the functions f and g if they are inverse to each other $f(g(x)) = g(f(x)) = x$. The third extension we consider is the combination of the two theories mentioned above, a congruence closure algorithm modulo associativity, commutativity and inverse functions. The theory of congruence closure modulo associativity and commutativity was addressed in the literature before [30, 12, 4], however only Conchon et al. [12] have actually implemented their algorithm inside a theorem prover. Congruence closure modulo inverse functions was not addressed before to our knowledge.

1.3 Outline of the Solution

In this thesis we will present three extensions of the standard congruence closure algorithm [35, 43, 19], namely the extensions covering associativity and commutativity [4, 12, 30] and inverse functions, as well as the combination of the two extensions. We have implemented our algorithms inside the Z3 theorem prover [16] from Microsoft. Z3 is an state of the art theorem prover, which was developed with software verification and analysis in mind. For this purpose it supports manifold theories. Moreover, Z3 is also able to handle quantifiers as well. It has since participated in the SMT competition, where it has achieved several top finishes within different categories. Z3 is freely available at `http://z3.codeplex.com/`.

1.3.1 Congruence Closure Modulo Associativity and Commutativity

As mentioned earlier, congruence closure modulo associativity and commutativity is an extension to the standard congruence closure algorithm where some functions have the associative and commutative property in addition to the functional consistency property. The theory of congruence closure modulo associativity and commutativity was addressed in the literature before [30, 12, 4]. While the basic algorithm stays the same in presence of associativity and commutativity, there are some changes that need to be made for the congruence closure algorithm to respect the associativity and commutativity of functions. These changes deal with term matching, which now has to be made with respect to associativity and commutativity. This means that the terms $f(a, f(b, c))$ and $f(b, f(a, c))$ are the same if f is associative and commutative, the definition of associativity and commutativity can be found in Section 4.2. The other important change is that all the congruence closure algorithms use a deduction step to learn new equalities from the equalities given as input to the procedure. Therefore, the deduction step of the congruence closure algorithm has to be adapted to be compatible with associative and commutative uninterpreted functions. The deduction step of the congruence closure algorithm is used to learn equalities that logically follow from other equalities. Another change that needs to be made is the introduction of a reduction ordering for terms that can handle associativity and commutativity, such an ordering was presented by Rubio and Nieuwenhuis [40]. Given two terms a reduction ordering decides which of the two terms is smaller. This is needed in the term replacement step of the algorithm.

1.3.2 Congruence Closure Modulo Inverse Functions

As in the case with associativity and commutativity when directly handling inverse functions inside the congruence closure algorithm, we have to add additional steps to the procedure. In the case of inverse functions these steps have to capture the following two problems. First a step that allows the system to eliminate the application of inverse functions to a term, i.e. $f(g(x)) = x$, is inserted. The second step is a deduction step that is used to learn new equations in the context of inverse functions. Like in the associative and commutative case we need a more dedicated deduction step in the context of inverse functions. To our knowledge the problem of congruence closure modulo inverse functions was not addressed in the literature before.

1.3.3 Congruence Closure Modulo Associativity, Commutativity and Inverse Functions

The last congruence closure algorithm we propose in our work is the combination of the two algorithms mentioned before. In the case of congruence closure modulo associativity, commutativity and inverse functions the equalities and inequalities can contain both associative and commutative as well as inverse functions. Since associativity and commutativity are defined for binary functions and the inverse function property is defined for unary function, an uninterpreted function cannot have all three properties. This fact makes the combination somewhat easier. We combined the two algorithms by extending the associative and commutative congruence closure algorithm with the necessary steps to handle inverse functions inside a congruence closure algorithm. This means that associative and commutative functions are handled in the exact same way as in the pure congruence closure algorithm modulo associativity and commutativity. However, the algorithm is also capable of removing the application of inverse functions and to learn equalities that are a logical consequence of equalities with inverse functions in them.

1.4 Structure of this Thesis

The rest of this document is split into six chapters. In Chapter 3 we introduce the necessary notation and background for the rest of this thesis. Nothing new is presented in this chapter.

In Chapter 4 we explain the various congruence closure algorithms that already exist, as well as the extensions for associativity and commutativity, inverse functions, and the combination of the two theories. All the algorithms in this chapter are explained with the help of rewrite systems.

The implementation of the algorithms is presented in Chapter 5. We will use various examples to explain the differences between the theoretical algorithms and their implementation inside the Z3 theorem prover.

Chapter 6 contains the evaluation of the presented algorithms. All the algorithms are benchmarked against the Z3 theorem prover using axioms for the additional properties of uninterpreted functions.

We then present related work of our algorithms in Chapter 2. Here we will discuss similar work and point out the differences to our work.

Finally, in Chapter 7 we conclude our work with a summary and a discussion of the important points made in this thesis. We also propose future work to extend this thesis.

Chapter 2

Related Work

2.1 Satisfiability Modulo Theories

Recent advances in the field of SAT and SMT solving have led to an explosion of applications for the technology, especially in the field of verification.

Bozzano et al. [7] proposed a new way of combining two arbitrary theories for SMT solving. In their work no theory solver for the combined theory is required. Instead, they only rely on solvers for the two theories. Normally the Nelson-Oppen integration schema [36] is used for the combination of two disjoint theories, which is explained in section 3.1.2. Their approach does not combine the two arbitrary theories but uses the respective theory solvers in isolation from each other. The mutual consistency of the formula is ensured by enumerating all possible equalities of interface variables from the purified formula. The advantage of this method is that no dedicated solver for the combined theory is needed which makes the combination of more than two theories easier.

2.2 Uninterpreted Functions

The logic of uninterpreted functions and equality has been used in the literature for various verification tasks. Burch adn Dill [9] use in their verification tool for the control logic of pipelined microprocessors. Because most of the bugs in a microprocessor design are in the control logic, they only verify the control logic. In addition, there are other methods to prove the correctness of the data path of processors. To prove the correctness Burch and Dill [9] convert both the behavioural and the implementation description into transition functions. The resulting transition functions are then used to determine whether the implementation is in accordance with the behavioural description. Before a state of the implementation is compared with the specification state, all partial processed instructions in the pipeline need to be finished (the pipeline is flushed). This is ensured in a way that does not change the user visible state. Although this method cannot be used to verify modern processor architectures, it allows the verification of pipelined processors, something that was not possible before.

Velev and Bryant [44] extended the idea of Burch and Dill [9] and used uninterpreted functions for the verification of processors where both functional units and memory may have indeterminate latency. In addition, they showed a way to model exceptions and branch prediction.

Gulwani and Tiwari [24] present an algorithm for assertion checking of programs. They use the combined logic of uninterpreted functions and linear integer arithmetic for program abstraction before they verify the assertions of the program. A surprising result of their work is that the problem is co-NP hard, even for loop-free programs.

2.3 Congruence Closure

Numerous congruence closure algorithms have been published in the literature so far. On the one hand there are the three classical congruence closure algorithms by Nelson and Oppen [35], Shostak [43], and Downey et al. [19]. A common property of all these algorithms is that they are formulated over graphs. On the other hand there is the incremental algorithm by Nieuwenhuis and Oliveras [37], which is also capable of producing explanations for obtained equalities. In contrast to the classical algorithms this algorithm is not formulated on graphs. The runtime benefits from the incremental property of this algorithm, since this algorithm is called a large number of times by DPLL procedures.

While none of these congruence closure algorithms is able to handle associative and commutative function symbols directly, all can be extended to handle associative and commutative function symbols.

2.4 Congruence Closure Modulo Associativity and Commutativity

The idea of congruence closure modulo associativity and commutativity has been addressed before. Marché [30] considered the problem of ground AC-completion. They showed that a canonical rewrite system for a set of equalities, where some of the used uninterpreted functions are associative and commutative must be finite. Furthermore they have proven, that the completion procedure in this case always terminates with the canonical system as a result.

Jouannaud and Marché [28, 27] considered the problem of completion modulo associativity, commutativity and identity. The addition of identity allows them to associate neutral elements with uninterpreted functions. For example if x is the neutral element of the uninterpreted function f then the following property holds: $\forall y. f(x, y) = y$. The introduction of identity for uninterpreted functions leads to the usage of constrained rewrite systems instead of standard rewrite systems. In a constraint rewrite system, rewrite rules have constraints associated with them. These constraints are used to ensure the termination of the procedure in the presence of neutral elements.

Bachmair et al. [4] presented an abstract congruence closure algorithm modulo associativity and commutativity. The difference between an abstract congruence closure and the standard congruence closure is that in the abstract setting the original signature is extended with new constants. These constants are then used to abstract terms of the original equalities. This makes the rules smaller, rules only have a maximum of one uninterpreted function application on each side. However this leads to a larger number of rules in the rewrite system.

Conchon et al. [12] extended the ground AC-completion algorithm with Shostak theories. This allows them to combine the ground AC-completion with other theories (like linear integer arithmetic).

Although all of these papers present rewrite systems to solve the problem of ground AC-completion, Conchon et al. [12] were actually the only ones to implemented their algorithm inside a modern theorem prover, their tool is named *Alt-Ergo*. The implementation of the algorithm within *Z3* is the main contribution of our work. Furthermore, none of the other authors have considered the problem of uninterpreted function modulo inverse functions.

Chapter 3

Preliminaries

In this chapter we establish the needed notation as well as the needed background for the rest of this thesis. We start by taking a look at satisfiability modulo theories, to be more precise, we start by recalling first-order logic. Then we will focus on theories and finally we will explain how to solve satisfiability modulo theory problems. The second part of this chapter is dealing with the theory and notation of rewrite systems. We will use rewrite systems to explain all the congruence closure algorithms in this thesis.

3.1 Satisfiability Modulo Theories

Satisfiability modulo theories is an instance of a constraint satisfaction problem, such constraint satisfaction problems arise in numerous applications, like software or hardware verification [15]. The most well-known instance of a constraint satisfaction problem is the classical propositional satisfiability problem (SAT). In the classical SAT problem the question is whether an assignment of true and false to variables occurring in a Boolean, in such a way that the overall formula becomes true, exists. If such an assignment can be found, the formula is said to be satisfiable and the corresponding assignment is called a model for the formula, otherwise the formula is unsatisfiable [32]. A program that can decide this problem is called a SAT solver. The classical SAT problem is NP-complete [23]. There are a lot of problems where a more expressive language, like arithmetic, is needed. These problems can be expressed as first-order logic formulas where certain functions and predicates have a defined meaning. This meaning is usually defined by a background theory. Examples for background theories are linear integer arithmetic or the theory of arrays. The problem whether or not such formulas are satisfiable is called the SMT problem and a procedure that can solve it is called an SMT solver.

3.1.1 First-Order Logic

Before we explain how theories can be used to extend first order logic formula and how the resulting formulas can be solved, we want to briefly recall the principles of first-order logic. We will start by defining the syntax of first-order logic, then we will look at terms and formulas in first-order logic, and we will conclude by defining free variables, sentences and quantifier free formulas.

Syntax

The signature Σ of first-order logic formulas consists of three sets, the set of variables ($\mathbb{V}$), the set of functions ($\mathbb{F}$), and the set of predicates ($\mathbb{P}$) [26], where all function symbols in $\mathbb{F}$ and all predicate symbols in $\mathbb{P}$ have an associated arity.

Terms

The set of terms π in first-order logic is defined as follows:

- if $v \in \mathbb{V}$ then $v \in \pi$,
- if $f \in \mathbb{F}$ with arity 0 then $f \in \pi$, and
- if $f \in \mathbb{F}$ with arity n and $t_1, \dots, t_n \in \pi$ then $f(t_1, \dots, t_n) \in \pi$

Formulas

From terms we can define the meaning of formulas in first-order logic:

- if $p \in \mathbb{P}$ with arity n and $t_1, \dots, t_n \in \pi$ then $p(t_1, \dots, t_n)$ is a formula.
- if $t_1, t_2 \in \pi$ then $t_1 = t_2$ is a formula.
- if ϕ and ψ are formulas then $\phi \vee \psi$, $\phi \wedge \psi$, $\neg\phi$, and $\phi \rightarrow \psi$ are formulas as well.
- if ϕ is a formula and $x \in \mathbb{V}$ then $\forall x.\phi$ and $\exists x.\phi$ are formulas as well.

Free Variables, Sentences and Quantifier Free Formulas

A variable x is a *free variable* in a formula ϕ if it is not bound by any quantifier. For example, if we consider the formula $\forall y.p(x, y)$ then the variable x is a free variable while the variable y is not, because it is bound by the $\forall$ quantifier.

Further, a *sentence* denotes a formula which does not contain any free variables. So for example $\forall x.\exists y.p(x, y)$ is a sentence but $\forall y.p(x, y)$ is not.

Finally, a formula which does not contain any quantifiers is called a *quantifier-free formula*. Thus $p(x, y)$ is a quantifier free formula, while $\forall y.p(x, y)$ is not.

Semantics

After we have established the necessary notation for first-order logic we will take a brief look at the semantics of first-order logic. The evaluation of first-order logic differs from the evaluation of propositional formulas in some ways [26]. First, we need to know what the meaning of our functions and predicates is, for example the function f can stand for the addition of integer numbers and the predicate p can stand for the smaller relation an integer numbers. Second, in first-order logic we have variables that can stand for many different things like students, animals, . . . , our universe of values. Third, quantifier change the way that formulas are evaluated, for example $\forall x.p(x)$ means that for every possible instance of x in our universe of values the predicate p has to hold.

A *model* $\mathbb{M}$ of the pair $(\mathbb{F}, \mathbb{P})$, where $\mathbb{F}$ is the set of function symbols and $\mathbb{P}$ is the set of predicates, consists of the following data [26]:

- A non-empty set A, which is the universe of concrete values.
- for each $f \in \mathbb{F}$ with arity 0, a concrete element $f^{\mathbb{M}}$ of A.
- or each $f \in \mathbb{F}$ with arity $n > 0$, a concrete function $f^{\mathbb{M}} : A^n \rightarrow A$.
- for each $p \in \mathbb{P}$ with arity $n > 0$, a subset $p^{\mathbb{M}} \subseteq A^n$.

A *look-up table* or *environment* for a universe A is a function $l : \mathbb{V} \rightarrow A$ [26]. By $l[x \mapsto a]$ we denote the look-up table where the variable x is mapped to the value a, any other variable y is mapped to $l(y)$ respectively.

With the definition of models and look-up tables we are now able to give a semantic to formulas of first-order logic. Given a model $\mathbb{M}$ and a look-up table l, we define the *satisfaction relation* $\mathbb{M} \models_l \phi$, where ϕ is a formula in first-order logic, by structural induction on ϕ [26]:

- p: If ϕ is of the form $p(t_1, \ldots, t_n)$, then we interpret the terms $t_1, \ldots, t_n$ in our set A by replacing all variables with their values according to the look-up table l, we also interpret all function symbols $f \in \mathbb{F}$ by $f^{\mathbb{M}}$. This computes concrete values $a_1, \ldots, a_n$ of A for each of the terms. Now $\mathbb{M} \models_l p(t_1, \ldots, t_n)$ holds if and only if $(a_1, \ldots, a_n)$ is in the set $p^{\mathbb{M}}$.
- $\forall x$: The relation $\mathbb{M} \models_l \forall x \phi$ holds if and only if $\mathbb{M} \models_{l[x \mapsto a]} \forall x \phi$ holds for all $a \in A$.
- $\exists x$: The relation $\mathbb{M} \models_l \exists x \phi$ holds if and only if $\mathbb{M} \models_{l[x \mapsto a]} \exists x \phi$ holds for some $a \in A$.
- $\neg$: The relation $\mathbb{M} \models_l \neg \phi$ holds if and only if it is not the case that $\mathbb{M} \models_l \phi$ holds.
- $\wedge$: The relation $\mathbb{M} \models_l \phi_1 \wedge \phi_2$ holds if and only if $\mathbb{M} \models_l \phi_1$ and $\mathbb{M} \models_l \phi 2$ hold.
- $\vee$: The relation $\mathbb{M} \models_l \phi_1 \vee \phi_2$ holds if and only if $\mathbb{M} \models_l \phi_1$ or $\mathbb{M} \models_l \phi 2$ hold.

From this we can define the notion of satisfiability for formulas in first-order logic. A formula ϕ is satisfiable if and only if there is some model $\mathbb{M}$ and some environment l that $\mathbb{M} \models_l \phi$ holds [26], otherwise the formula is unsatisfiable. For a more detailed description of the semantics of first-order logic we refer to [26].

3.1.2 Theories

After we have established the notation of first-order logic, we will now define theories in the context of satisfiability modulo theories. The definition is as follows [33]: A theory is a collection of sentences over a signature Σ. We say that a formula ϕ is satisfiable modulo a theory T if $T \cup \phi$ is satisfiable. For example, consider the signature Σ consisting the symbols 0, 1, $+$, $-$, and $<$. Let $\mathbb{Z}$ be the structure that interprets these symbols. If $\mathbb{Z}$ interprets these symbols in the usual way over the integers, then the set of first-order sentences that are true in $\mathbb{Z}$ form the theory of linear integer arithmetic. A satisfiability problem for a theory is said to be decidable if there exists a decision procedure for the quantifier free case.

Common Theories

There exist a lot of different theories, but some theories have gained more attention than others. Some of the more prominent theories are [33]:

- *Linear Arithmetic:* The signature of the linear integer arithmetic looks as follows $\Sigma = \{0, 1, +, -, <, \leq, =\}$. Where all the predicates and functions have their usual mathematical meaning.
- *Difference Arithmetic:* $\Sigma = \{0, 1, -, \leq\}$, where $-$ and $\leq$ again have their usual mathematical meaning. A basic formula in difference arithmetic has the form $x - y \leq c$, where x and y are variables and c is a numerical constant. The overall formula then contains only logical connections between these basic formulas.

- *Bit-Vectors:* $\Sigma = \{+, -, <, \leq, =, bvor, bvand, bvnot, bvxor\}$. The theory of bit-vectors is similar to the linear integer arithmetic. Instead of using mathematical integers for numbers the theory of bit-vectors uses bit-vectors to represent numbers. This form of representation for numbers is the same as on computers, where integers also have a fixed size (e.g. 64-bit). The mathematical operations $+$ and $-$ are calculated modulo the maximal number that can be represented with the used bit-vector size. In addition to the arithmetical operators, bit-wise operators, like *and* or *or*, can be used in the formulas for this theory.

- *Uninterpreted functions:* Is also called the theory of free functions. Many decision procedures for other theories can be reduced to this one. For example, the theory of arrays is reduced to the theory of uninterpreted functions. Function symbols occurring in formulas do not have a predefined meaning in this theory and the only applicable relational operator is $=$. The decision procedure for this theory is the construction of a congruence closure which can then be used to decide whether or not an equality logically follows from a set of equalities, this is called the word problem for a set of equalities.

Procedures that are capable of deciding the satisfiability of such theories are called theory solvers [33].

Combination of Theories

In some cases it is not enough to use a single theory for solving. Instead multiple theories have to be combined to solve a formula. In this case, some fundamental questions arise, for example is the combination of two solvable theories still solvable or how does one get a decision procedure for the combined theory [33]. Given two theories T_1 and T_2, we use $T_1 \oplus T_2$ to denote the combined theory that is the union of the sentences of T_1 and T_2.

First we consider two *strongly disjoint theories* T_1 and T_2 over the signatures Σ_1 and Σ_2 respectively. Two theories are strongly disjoint if Σ_1 and Σ_2 have no common sort symbols and thus also do not share any function or predicate symbols [33]. Sorts in the context of SMT are like data-types in programming languages, for example the sort integer is defined by the linear integer arithmetic. A decision procedure in this case is easy, since the problem can be split up into two parts S_1 and S_2, where S_1 only contains literals of Σ_1 and S_2 only contains literals of Σ_2.

The other case we can consider is if the theories T_1 and T_2 are *disjoint*. Two theories are disjoint if they do not share any function or predicate symbols, they may however share sort symbols [33]. For example the theory of linear integer arithmetic and the theory of arrays are disjoint because they don't share any function or predicate symbols but they both have the sort integer. Here the Nelson-Oppen procedure [36] can be used to get a decision procedure for $T_1 \oplus T_2$. The basic idea of this combination is that the formula is transformed into a equisatisfiable pure formula. A formula over the signature $(\Sigma_1 \cup \Sigma_2)$ is pure if every literal is a Σ_i literal for $i = 1, 2$. All quantifier free $(\Sigma_1 \cup \Sigma_2)$ formulas can be transformed into a equisatisfiable pure formula [33]. This is done by using the following satisfiability preserving transformation:

$F[t] \rightsquigarrow F[u] \land u = t$ where u is a fresh variable.

Consider the following example which uses the theory of linear integer arithmetic and the theory of arrays: $\phi = load(y + 2, array) = x \land load(y - 5, array) = z$, where $load(pos, array)$ retrieves the value stored at the position pos from the array $array$. Then the equisatisfiable pure formula looks as follows, $\phi_p = (u_1 = y + 2 \land u_2 = x \land u_3 = y - 5 \land u_4 = z) \land (u_2 = load(u_1, array) \land u_4 = load(u_3, array))$. The resulting formula can then be checked for satisfiability. For more detailed information about theories and the combination we refer to [33, 36].

3.1.3 SMT Solving

Now that we have established the notion of satisfiability and defined what theories in the context of SMT are, we can tackle the problem of solving SMT formulas. Today there are two basic methods that are used to combine SAT solving with theory solvers, the *eager* and the *lazy* approach [38]. In the *eager* approach the problem is transformed into an equisatisfiable propositional formula and a SAT solver is then used to solve the SMT problem. A problem with this approach is that all theory information must be encoded in the propositional formula. The *lazy* approach also converts the problem into a propositional formula, but does not include theory information from the start. Thus, the SAT solver tries to find a model for the propositional structure of the formula in the first step, with all terms and predicates from the theory treated as propositional atoms. If the SAT solver is able to find such a model, the theory solver is asked whether this model is consistent with the theory. If it is not, an additional clause is added to the propositional formula which represents the conflict with the theory. The resulting formula is then again checked by the SAT solver. If the SAT solver again finds a model this model is checked by the theory solver. This cycle repeats until the SAT solver returns *unsatisfiable* or the theory solver determines that the model is consistent with the used theory. From now on we only consider variable free and thus also quantifier free instances of the SMT problem. An example for the necessary changes to the explained algorithms in this section can be found in [31].

DPLL

Before we introduce an algorithm to solve SMT formulas, we take a look at the Davis-Putnam-Logemann-Loveland (DPLL) procedure [14, 13] for propositional satisfiability. DPLL can be described as transition system with five simple rules [38], where the procedure is either in the *FailState* or in a state of the form $M \parallel F$ where M is a sequence containing the (partial) assignment and F is a formula in conjunctive normal form (CNF). One important property of the sequence M is that it never contains both a literal and its negation. Moreover with each literal in M there is an additional information associated whether this literal is a decision literal or not. Decision literals in M are denoted by l^d. A clause C is said to be conflicting in a state $M \parallel F, C$ if $M \models \neg C$, where all literal occurring in C are defined in M. Now we can state the five transition rules that make up the classical DPLL algorithm:

- *UnitPropagate:*
  $$M \parallel F, C \vee l \Longrightarrow Ml \parallel F, C \vee l \text{ if } \begin{cases} M \models \neg C, and \\ l \text{ is undefined in } M \end{cases}$$
  In order for any CNF formula to be satisfied all the clauses have to be true. If there is a clause in the formula where all but one literal are assigned false and the remaining literal is unassigned, we can assign a value to this literal to make this clause true.

- *PureLiteral:*
  $$M \parallel F \Longrightarrow Ml \parallel F \text{ if } \begin{cases} l \text{ occurs in some clause of } F, \\ \neg l \text{ does not occur in some clause of } F \text{, and} \\ l \text{ is undefined in M} \end{cases}$$
  A *pure* literal in F can always be assigned such that it is true. A literal is *pure* if it occurs in F but its negation does not occur in F.

- *Decide:*
  $$M \parallel F \Longrightarrow Ml^d \parallel F \text{ if } \begin{cases} l \text{ or } \neg l \text{ occurs in a clause in } F \text{, and} \\ l \text{ is undefined in } M \end{cases}$$
  This rule captures a split case situation, if we extend M with l but do not find a model for the formula F. Then we still have to consider adding $\neg l$. Thus, the literal is marked as a decision

literal when adding it to the current partial model M. If we cannot find a model we use the *Backtrack* rule to check if we can find a model with the negated literal from this decision.

- *Fail:*
  $M \parallel F, C \Longrightarrow FailState$ if $\begin{cases} M \models \neg C, \text{and} \\ M \text{ contains no decision literals} \end{cases}$
  If M does not contain a decision literal but a clause C which evaluates to false, we have found a conflict and know that the formula is unsatisfiable. We report this by changing into the $FailState$.

- *Backtrack:*
  $Ml^dN \parallel F, C \Longrightarrow M\neg l \parallel F, C$ if $\begin{cases} Ml^dN \models \neg C, \text{and} \\ N \text{ contais no decision literal} \end{cases}$
  If a conflict clause C is detected but decision literals still remain, the *Fail* rule does not apply. In this case the *Backtrack* rule is used to backtrack one decision level. The last decision made is replaced with its negation and all subsequent assignments which followed the particular decision are discarded. Note that *Backtrack* does not mark the literal as decision literal since the other possibility has already been explored without success.

The procedure is finished when it reaches a final state, a state is final if their is no rule applicable. If the final state is the $FailState$ then the formula is unsatisfiable, otherwise the formula is satisfiable and M contains the assignment for all literals occurring in the formula F. For the proof of this we refer to [38].

The following example shows a run of the DPLL algorithm. It was taken from [38]:

$\emptyset$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*Decide*
a^d	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*UnitPropagate*
$a^d \neg b$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*UnitPropagate*
$a^d \neg bc$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*UnitPropagate*
$a^d \neg bcd$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*Backtrack*
$\neg a$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*UnitPropagate*
$\neg ad$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*Decide*
$\neg ad \neg c^d$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	$\Longrightarrow$*UnitPropagate*
$\neg ad \neg c^d b$	$\parallel \neg a \vee \neg b, b \vee c, \neg a \vee \neg c \vee d, b \vee \neg c \vee \neg d, a \vee d$	

The last state of the system is the final state of the derivation and M is the model that satisfies the formula.

However, modern versions of the DPLL procedure do not implement the system as stated above. They use several improvements, for example the *PureLiteral* rule is used as a preprocessing step rather than a rule and the chronological backtracking is replaced with a more powerful backtracking mechanism for efficiency reasons [38]. Learning conflict clauses has also been shown to improve the performance of the algorithm and is usually done in practice [38]. We now take a look at a DPLL system with learning and the more sophisticated backjump mechanism which consists of six rules: The rules *Decide*, *Fail*, and *UnitPropagate* are taken from the classical DPLL shown above and the following rules are added:

- *Backjump:*
  $Ml^dN \parallel F, C \Longrightarrow Ml' \parallel F, C$ if $\begin{cases} Ml^dN \models C \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ F, C \models C' \vee l' \text{ and } M \models \neg C', \\ l' \text{ is undefined in } M, \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } Ml^dN \end{cases}$

In contrast to backtracking, backjumping does not undo the last decision made but a decision which is responsible for the conflict. Consider the following example to show the strength and the application of the backjump rule [38]:

$\emptyset$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*Decide*
a^d	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*UnitPropagate*
$a^d b$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*Decide*
$a^d b c^d$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*UnitPropagate*
$a^d b c^d d$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*Decide*
$a^d b c^d d e^d$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*UnitPropagate*
$a^d b c^d d e^d \neg f$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	$\Longrightarrow$*Backtjump*
$a^d b \neg e$	$\| \neg a \vee b, \neg c \vee d, \neg e \vee \neg f, f \vee \neg e \vee \neg b$	

The clause $f \vee \neg e \vee \neg b$ is conflicting before the application of the backtracking step. This conflict is a result of the decisions a^d and e^d together with their respective unit propagations, therefore we can infer that the decisions are incompatible with each other. Moreover, the given clause set entails $\neg a \vee \neg e$ and $\neg b \vee \neg e$. Such clauses are called backjump clauses, since their presence would have allowed a unit propagation at an earlier step [38]. Backjump now goes back to that level and adds the unit propagation of that clause to the model. For the example if we use $\neg b \vee \neg e$ as the backjump clause we end up in a state with the partial model $a^d b \neg e$, when using a backjump step instead of the backtracking step.

- *Learn:*

  $M \parallel F \Longrightarrow M \parallel F, C$ if $\begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \text{and} \\ F \to C \end{cases}$

  In modern implementations one can make additional use of the backjump clauses, by adding them to the formula as learned clauses. This is called conflict driven learning and helps to avoid reaching similar conflict states repeatedly. However, the rule stated here is more general and allows the addition of any clause that is entailed by the formula. This represents a more general learning approach then conflict driven learning.

- *Forget:*

  $M \parallel F, C \Longrightarrow M \parallel F$ if $\begin{cases} F \models C \end{cases}$

  The *Forget* rule can be used to remove any clause C from F that is entailed by the remainder of F. This includes clauses learned via the application of the *Learn* rule.

DPLL(T)

The DPLL algorithm from the previous section can be adapted to solve the SMT problem. One important difference in this case is that instead of propositional literals we have to consider first-order terms and predicates. This however does not change the rules *Decide*, *Fail*, and *UnitPropagate* since they still regard all literals as syntactical units just like in the propositional case [38]. The only rules adapted are *Learn*, *Forget*, and *Backjump*; here, entailment in F becomes entailment in the theory T. In the following rules, $\models$ denotes the propositional notion of satisfiability and $\models_T$ the first order notion of entailment modulo the theory T.

- *T-Learn:*

  $M \parallel F \Longrightarrow M \parallel F, C$ if $\begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or } M \\ F \models_T C \end{cases}$

- *T-Forget:*
  $M \parallel F, C \Longrightarrow M \parallel F$ if $\begin{cases} F \models_T C \end{cases}$
- *T-Backjump:*
  $$Ml^dN \parallel F, C \Longrightarrow Ml' \parallel F, C \text{ if } \begin{cases} Ml^dN \models \neg C \text{ and there is some clause } C' \vee l' \text{ such that:} \\ F, C \models_T C' \vee l' \text{ and } M \models \neg C', l' \text{ is undefined in } M\text{, and} \\ l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } Ml^dN \end{cases}$$

With these rules we can now model the naive lazy approach for SMT solving [38]. Whenever we reach a state that is final with respect to the application of the rules *Decide*, *Fail*, *UnitPropagate*, and *T-Backjump*, the model M can be consistent with the theory T or not. In the case of consistency it is a model for T. If it is inconsistent with the theory, then there exists a subset $l_1, \ldots, l_n$ of M for which $\emptyset \models_T \neg l_1 \vee \cdots \vee l_n$. This clause can then be learned via a *T-Learn* step and the procedure can be restarted. During this restart the clauses learned from the theory solver are kept as part of the formula. For the proof of correctness for the DPLL(T) procedure we refer the interested reader to [38].

In practice there exist various improvements of this naive approach [38]. The difference of those approaches to the naive one is that the clause learning is tighter incorporated in the solving process, i.e., the detection of inconsistencies with the theory is done before a final model is available. This has the advantage that those conflicts are detected before the whole model is built and thus the procedure can be restarted at an earlier point which yields a performance increase.

3.2 Rewrite Systems

Equations are an important part of mathematics and other sciences. Sometimes we want to solve equations, other times we want to check whether an other equality follows from a set of equalities. Rewrite systems are directed equations, which can be used to replace subterms of an expression until the simplest form of the term is obtained [18]. Rewrite systems are like non-deterministic Markov algorithms over terms and have the full power of Turing machines. Essentially, rewriting is the theory of normal forms and is related to Church's Lambda Calculus and Curry's Combinatory Logic [18].

Rewrite system are used to solve a plethora of problems. Buchberger and Loos [8] use term rewriting for algebraic simplification. The goal of algebraic simplification is to obtain simpler but equivalent objects as well as to compute unique representatives for a class of equivalent objects. Baeten and Weijland [5] show in their work that logic programs can be given semantics via term rewriting systems. They also show that the addition of a priority ordering on the rewrite rules gives a procedural semantic for the depth-first search rule inside Prolog. Rewrite systems can also be used directly for computation. Derschowitz [17] proposed a programming language similar to Prolog based on rewrite rules. The main difference to Prolog is that rewrite rules are equivalences and not implications in Horn-clause form. Hsiang [25] presented a first-order logic theorem prover based on a rewrite system. They developed a canonical term-rewriting system for Boolean algebra, which allows them to translate first-order predicate calculus into a form of equational logic.

Before we define the syntax of rewrite systems we consider the following example from [18]. This is the "Coffee Can Problem", where we have a can containing black and white beans arranged in some order. We will represent the content of the can as a sequence of bean colors.

$$white\ white\ black\ black\ white\ white\ black\ black$$

The rewrite rules for our example are:

$$\begin{aligned} black\ white &\to black \\ white\ black &\to black \\ black\ black &\to white \end{aligned}$$

The goal of the game is that we end up with the fewest possible number of beans left in the can. For our example a possible sequence of derivations is:

$$\begin{aligned}
&white\ white\ black\ \underline{black\ white}\ white\ black\ black\\
&white\ white\ \underline{black\ black}\ white\ black\ black\\
&white\ white\ white\ \underline{white\ black}\ black\\
&white\ white\ \underline{white\ black}\ black\\
&white\ \underline{white\ black}\ black\\
&\underline{white\ black}\ black\\
&\underline{black\ black}\\
&white
\end{aligned}$$

3.2.1 Syntax

Terms

Terms in the case of rewrite systems are the same as in the first-order logic case, see Section 3.1.1. A term is called a *ground* term if it does not contain a variable. We will call the set of ground terms $\mathbb{G}$ from now on. We will use $\mathbb{F}_i$ to denote the set of all functions with arity i.

Any term in π can be visualized as a finite ordered tree, where the leaves are either variables from $\mathbb{V}$ or constants from $\mathbb{F}_0$ [18]. The internal nodes of the tree consist of function symbols from $\mathbb{F}_1 \cup \cdots \cup \mathbb{F}_n$ where $n \geq 1$ and have a outdegree which is equal to the arity of the function. The position of a subterm within another term can be represented in Dewey decimal notion, describing the way from the root to the subterm. We write $t\ |_p$ for the subterm of t at the position p. Consider the following example: if $t = f(c, f(a, b))$ then $t\ |_{2.1}$ denotes the first subterm of t's second subterm, which is the term a. Furthermore $t[s]$ means that s is a subterm occurring in t. Finally a subterm is a *proper* subterm of t if it is distinct from t. If we replace a subterm of t at the position p with a term s we will write this as $t[s]_p$.

A substitution σ is a replacement operation that maps variables to terms, written as $\{x_1 \rightarrow t_1, \ldots, x_m \rightarrow t_m\}$ [18]. Note that only finitely many variables x_i are not mapped to themselves. Formally a substitution is a function from $\mathbb{V}$ to π extended to a function from π to itself in a way that $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for each function in $\mathbb{F}$ and for every term in π. We say that two terms s and t unify when there is a substitution σ such that $s\mu = t\mu$, with the substitution σ being the *unifier*. We call the unifier σ the *most general unifier (MGU)* if, for all other unifiers τ the following property holds: $\forall s \in \pi : s\tau = (s\sigma)\rho$ where ρ is an substitution.

Rewrite Rules

The difference between equations and rewrite rules is that equations are an unordered pair, but rewrite rules are an ordered pair $\langle l, r\rangle$ over a set of terms π [18]. Rewrite rules are written as $l \rightarrow r$. Because rewriting rules are ordered, they are only used to replace instances of the left-hand side with instances of the right-hand side, unlike equations which can be used in both directions. Furthermore, a set of rewriting rules R over π is called a term-rewriting system.

We say that a term $s \in \pi$ rewrites to a term $t \in \pi$ with respect to R, written as $s \rightarrow_R t$, if $s\ |_p = l\sigma$ and $t = s[r\sigma]_p$ for a rule $l \rightarrow r \in R$, p being the position in s and σ being a substitution [18]. This is equivalent to saying that $t = u[l\sigma]_p$ and $s = u[r\sigma]_p$, for some position p in u. A term s is said to be *irreducible* or in *normal form* if there is no term t in π such that $s \rightarrow_R t$.

Properties of Rewrite Systems

A term-rewriting system can have some of the following properties [18]:

- *Ground*: A rewriting system is a ground rewriting system if all rules are ground, thus when all rules are elements of $\mathbb{G} \times \mathbb{G}$.
- *Normalizing*: A rewriting system is normalizing if every term has at least one normal form.
- *Unique normalization:* If every term has exactly one normal form the rewriting system has the unique normalization property.
- *Reduced:* Let R be a rewriting system, if for every rule $l \rightarrow r \in R$ the right-hand side r of a rule is irreducible under R and if for every term s that is smaller than l, s is also irreducible then the rewrite system is said to be *reduced.*
- *Convergent:* If every possible sequence of rewriting leads to the same unique normal form for a term the rewriting system is convergent, this property is sometimes also called *canonical* in the literature. However we will use a different definition for *canonical.*
- *Canonical:* If a rewrite system is both *reduced* and *convergent* then it is called a *canonical* rewrite system. In this thesis we use this definition when we refer to a *canonical* rewrite system.

In our work we will only consider ground and canonical rewrite systems, since only these can be used to decide the problem of congruence closure. It is important that the system is canonical because otherwise it cannot be used as a decision procedure, for example if the system is not canonical then there exist terms which have more then one normal form. Since these normal forms are used to detect equality of terms in the case of congruence closure, it is very important that the normal form of a term is unique. The need for ground rewrite systems arise out of the fact that the completion procedures described in Chapter 4 are only guaranteed to terminate if the set of equations is ground [30]. With a ground set of equations the completion procedures always yield a ground and canonical rewrite system as result, see Chapter 4 for more details.

3.2.2 Congruence

Replacing terms leads to the important notion of *congruence* [18]. An equivalence relation $\approx$ is a congruence on a set of terms if $f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)$ whenever $s_i \approx t_i$ for $i = 1, \ldots, n$. This results in the fact that the reflexive-transitive closure $\leftrightarrow^*$ of any rewrite relation $\rightarrow$ is a congruence.

For terms s and t we write $s \leftrightarrow_E t$ whenever $s = u[l\sigma]_p$ and $t = u[r\sigma]_p$ for some position p in u, equation $l = r$, and a substitution σ [18]. Informally, $s \leftrightarrow_E t$ denotes that s has a subterm which can be replaced with the other side of an equation in E that makes it equal to t. The relation $\leftrightarrow_E^*$ is the congruence closure of E, i.e. it is the smallest congruence over τ such that $l\sigma \leftrightarrow_E^* r\sigma$ for $l = r \in E$ and all substitutions σ over π. We are interested in congruences because they allow us to learn new equalities between terms, which are consequences of the given set of equations.

3.2.3 Completion

Before we discuss completion procedures for rewrite systems, we have to establish the notion of *confluence* [2]. A term rewriting system R is *confluent*, if for all s,t,$t' \in \pi$, whenever $s \rightarrow_R^* t$ and $s \rightarrow_R^* t'$, then there exists a $u \in \pi$ such that $t \rightarrow_R^* u$ and $t' \rightarrow_R^* u$. The *confluence* property means that one can diverge from a common ancestor but there will always be a way to arrive at a common descendent. This is important for rewrite systems because we want convergent rewrite systems, where every term has exactly one normal form that should always be found independent of the order in which possible rewrite rules are applied.

When talking about confluence we also have to consider *critical pairs* of rewrite rules. Let $l \rightarrow r$ and $s \rightarrow t$ be two rewrite rules with distinct variables, p be the position of a nonvariable subterm of s, and μ the most general unifier of $s\mid_p$ and l. Then the equation $t\mu = s\mu[r\mu]_p$ is a *critical pair* formed by the two rules [18]. The presence of a critical pairs implies that there exist two possible rewritings for some terms.

Confluence of finite terminating rewrite system can be decided [18]. However, if a rewriting system fails the test, this is because a critical pair does not have a common normal form, then one needs a *completion* procedure, which adds new rules such that the critical pair has a common normal form. This, however, may generate new critical pairs which do not have a common normal form. Therefore, the procedure must also generate rewrite rules for them, which leads to the fact that the procedure must not arrive at a *convergent* system. In a *convergent* system, all critical pairs have a common normal form. Although the completion procedure does not have to succeed in the unrestrained case [18], it does for more constrained problems, see Chapter 4 for more details about the correctness of the used algorithms.

Before we explain a generic completion procedure for rewrite systems we have to establish what a *reduction ordering* is. A relation $\succ$ is a reduction ordering on the terms of a rewrite system if the following properties hold [42, 41]:

- for all $s \in \pi$ $s \succ s$ does not hold.
- for all $s, t \in \pi$ if $s \succ t$ does hold then $t \succ s$ does not.
- for all $s, t, u \in \pi$ if $s \succ t$ and $t \succ u$ hold then $s \succ u$ also holds.
- the set of terms π is well ordered with respect to $\succ$, that is, all nonempty subsets contain their least element
- if $t_i \succ t'_i$ then also $f(t_1, \ldots, t_i, \ldots, t_n) \succ f(t_1, \ldots, t'_i, \ldots, t_n)$ for all functions f.
- if $s \succ t$ then also $s\sigma \succ t\sigma$ for all substitutions σ

A completion procedure works as follows [18]: Given a set of equations and a reduction ordering ($\succ$) on the terms a completion procedure tries to find a *canonical* system. First the equations are used to generate the initial rewrite rules for the rewrite system. Here the reduction ordering on terms is used to decide the orientation of the rewrite rules that follow from an equation. For example, the equation $l = r$ generates the rewrite rule $l \rightarrow r$ if $l \succ r$ or the rewrite rule $r \rightarrow l$ if $r \succ l$. The rewrite rules generated by the procedure can always be used to simplify terms occurring in the remaining equations or in already generated rules. After the initial rewrite system from the set of equations is finished, the completion procedure generates critical pairs and orients them as well. Consider the critical pair $s = t$. Suppose it can be simplified to $u = v$, where u and v are not identical. Then a rule $u \rightarrow v$ or $v \rightarrow u$ is added to provide a rewrite proof for $s = t$. The orientation of the new rule is again decided with the help of the reduction ordering on terms. This new rule is then used to form new critical pairs. The procedure described here can have three different outcomes: a *canonical* system was found (success), nothing was found (failure), or the procedure can loop forever (it generates an infinite *canonical* system).

Chapter 4

Algorithms

After we have established the notion of rewrite systems and SMT in the previous chapter, we will look at various congruence closure algorithms in this chapter. We will present the three algorithms we have implemented inside *Z3*, namely congruence closure modulo associativity and commutativity, congruence closure modulo inverse functions and the combination of the two. We will also describe the standard congruence closure algorithm which neither takes associativity and commutativity nor inverse function into account. We will state the three classical congruence closure algorithms from Shostak [43], Downey et al. [19], and Nelson and Oppen [35]. Modern versions of the congruence closure algorithm use certain optimization to be better suited for their specific application. For example the congruence closure algorithm from Nieuwenhuis and Oliveras [37] was tailor-made to be used in modern SMT solvers, since it was extended that an explanation for the result can be produced without increasing the runtime of the algorithm.

The chapter is organized as follows. The first part is about the standard congruence closure as rewrite system. After that we will explain how this algorithm can be modified to handle associativity and commutativity. The last part of this chapter is about the modifications needed to cover uninterpreted functions modulo inverse functions for both the standard congruence closure as well as the congruence closure algorithm modulo associativity and commutativity.

4.1 Congruence Closure

In the following we give the completion procedure for obtaining a rewrite system for the standard congruence closure algorithm. The state of the completion procedure is expressed by tuples $(E; R)$, where E is a set of ground equations over a signature Σ and R is the current set of rewriting rules. The procedure starts in the state $(E; \emptyset)$. The following six inference rules, with a suitable reduction ordering $(\succ)$ on terms, are used to build a rewrite system in the standard congruence closure case [18]. The reduction ordering is used to orient equations from E when they are turned into rewrite rules, see *Orient* rule. As noted in Section 3.2 equations in E are not orientated, while rewrite rules in R are always orientated.

- *Delete:*
  $(E \cup \{s = s\}; R) \vdash (E; R)$
  The *delete* rule is used to remove a trivial equation $s = s$ from the set of equations.

- *Compose:*
  $(E; R \cup \{s \to t\}) \vdash (E; R \cup \{s \to u\})$ if $t \to_R u$
  The *compose* rule is used to rewrite the right-hand side of a rule, whenever possible. This rule is used to ensure that the resulting rewrite system is reduced, which is one property a canonical rewrite system must fulfil.

- *Simplify:*
  $(E \cup \{s = t\}; R) \vdash (E \cup \{s = u\}; R)$ if $t \rightarrow_R u$
  $(E \cup \{s = t\}; R) \vdash (E \cup \{u = t\}; R)$ if $s \rightarrow_R u$
  The *simplify* rule is used to rewrite the terms occurring on either side of an equation, whenever this is possible.

- *Orient:*
  $(E \cup \{s = t\}; R) \vdash (E; R \cup \{s \rightarrow t\})$ if $s \succ t$
  Orient is used to turn an equation into a rewrite rule. Since rewrite rules are only used in one direction the equation has to be oriented before the rewrite rule may be added. This is done according to the defined ordering of terms. If $s \succ t$ then the rule $s \rightarrow t$ is added, however if $t \succ s$ then the rule $t \rightarrow s$ must be added.

- *Collapse:*
  $(E; R \cup \{s \rightarrow t\}) \vdash (E \cup \{u = t\}; R)$ if $s \rightarrow_R u$ using the rule $l \rightarrow r \in R$ with $(s \succ l) \vee (s = l \wedge s \succ r)$
  The *collapse* rule is used to rewrite the left-hand side of a rule $s \rightarrow t$ by a rule $l \rightarrow r$. Whenever the left-hand side of a rule is changed, the rule is removed from the set of rules and reinserted into the set of equations. This step is necessary because after the left-hand side of a rule was changed the property that it is smaller than the right-hand side of that rule can no longer be guaranteed.

- *Deduce:*
  $(E; R) \vdash (E \cup \{s = t\}; R)$ if $s = t$ is a critical pair of the set of rules R
  This rule is used to add critical pairs to the set of equations. As mentioned in Chapter 3 it is important that all critical pairs have a rewrite rule, otherwise not all terms have a unique normal form.

The procedure is finished if none of the rules described above can be applied. The result of the procedure is a ground and canonical rewrite system. A proof idea for the correctness of the procedure can be found in [3].

After we have established the completion procedure, we want to look at an example. Consider the following set of equations $E = \{f(a, b) = a, b = c, f(c, c) = f(a, d), f(a, b) = d\}$. For term ordering we use the following reduction ordering $f \succ d \succ c \succ b \succ a$. The steps of the completion procedure can be seen in Table 4.1.

The resulting rewrite system can be used to show that the equation $f(b, b) = f(f(a, c), f(d, b))$ logically follows from the given set of equations. This can be seen in Table 4.2 where the reduction of the two sides of the equation with the generated rewrite rules is shown. So the congruence closure procedure can be used to solve the word problem for a set of equations, where we want to know whether an equality logically follows from a set of equations.

4.1.1 Important Strategies

After we have established the inference rules for a congruence closure completion procedure in the previous section, we now want to look at the most important congruence closure algorithms developed by Shostak [43], Downey et al. [19], and Nelson and Oppen [35]. These algorithms can be seen as different strategies how to use the inference rules from the completion procedure for congruence closure. The key difference between the three algorithms is the order in which the inference rules are applied to get a congruence closure for a given input. Another difference between the algorithms is the input representation.

Step	E	R	Transition Rule
1	$\{f(a,b) = a, b = c, f(c,c) = f(a,d), f(a,b) = d\}$	$\emptyset$	Orient $f(a,b) = a \Rightarrow f(a,b) \to a$
2	$\{b = c, f(c,c) = f(a,d), f(a,b) = d\}$	$\{f(a,b) \to a\}$	Simplify $f(a,b) = d$ by $f(a,b) \to a$
3	$\{b = c, f(c,c) = f(a,d), a = d\}$	$\{f(a,b) \to a\}$	Orient $b = c \Rightarrow c \to b$
4	$\{f(c,c) = f(a,d), a = d\}$	$\{f(a,b) \to a, c \to b\}$	Simplify $f(c,c) = f(a,d)$ by $c \to b$
5	$\{f(b,b) = f(a,d), a = d\}$	$\{f(a,b) \to a, c \to b\}$	Orient $f(b,b) = f(a,d) \Rightarrow f(a,d) \to f(b,b)$
6	$\{a = d\}$	$\{f(a,b) \to a, c \to b, f(a,d) \to f(b,b)\}$	Orient $d = a \Rightarrow d \to a$
7	$\emptyset$	$\{f(a,b) \to a, c \to b, f(a,d) \to f(b,b), d \to a\}$	Collapse $f(a,d) \to f(b,b)$ by $d \to a$
8	$\{f(a,a) = f(b,b)\}$	$\{f(a,b) \to a, c \to b, d \to a\}$	Orient $f(a,a) = f(b,b) \Rightarrow f(b,b) \to f(a,a)$
9	$\emptyset$	$\{f(a,b) \to a, c \to b, d \to a, f(b,b) \to f(a,a)\}$	

Table 4.1: The deduction steps for the congruence closure completion procedure for our example.

Step	Left-Hand Side	Right-Hand Side	Rewrite Rule
1	$f(b,b)$	$f(f(a,c), f(d,b))$	$f(b,b) \to f(a,a)$
2	$f(a,a)$	$f(f(a,c), f(d,b))$	$c \to b$
3	$f(a,a)$	$f(f(a,b), f(d,b))$	$d \to a$
4	$f(a,a)$	$f(f(a,b), f(a,b))$	$f(a,b) \to a$
5	$f(a,a)$	$f(a,a)$	

Table 4.2: The necessary deduction steps to show that the equation $f(b,b) = f(f(a,c), f(d,b))$ logically follows from the set of equations, using the rewrite system built by the congruence closure completion procedure.

Step	E	R	Inference Rule
1	$\{a = b, f(f(a)) = f(b)\}$	$\emptyset$	Orient $a = b$
2	$\{f(f(a)) = f(b)\}$	$\{b \to a\}$	Simplify $f(f(a)) = f(b)$ by $b \to a$
3	$\{f(f(a)) = f(a)\}$	$\{b \to a\}$	Orient $f(f(a)) = f(a)$
4	$\emptyset$	$\{b \to a, f(f(a)) \to f(a)\}$	

Table 4.3: An example showing the steps of the completion procedure from Shostak's congruence closure algorithm.

Shostak's Method

The basic implementation of the Shostak algorithm, without any performance boosts, is as follows [3]. The algorithm starts by picking an equation $s = t$ from the set of equations E. This equation is then simplified until no more simplification steps on the terms s and t are possible. If s and t simplify to the same term, the equality is trivial and thus deleted from the set of equations using the *deletion* rule. If s and t are different terms after the simplification steps the *orientation* rule is used to generate a rewrite rule from the equation. After this rewrite rule is added to the set of rules, the algorithm performs all possible collapse steps. After each of these collapse steps the algorithm performs all *deduction* steps that arise out of the performed collapse step. When all collapse and deduction steps have been carried out, the algorithm starts again by picking another equation from the set of equations. This cycle repeats until the set of equations E is empty.

Shostak's algorithm [43] can be described by the following combination of rules [3], where the Simplification* means that the *simplification* rule is used until it cannot be applied anymore. By Simplification$\circ$ Deletion we denote that first the *simplification* rule is used, followed by an application of the *delete* rule. Shostak $=$ (Simplification* $\circ$ (Deletion $\cup$ Orientation) $\circ$ (Collapse $\circ$ Deduction*)*)*

An important property of Shostak's method is that it is a dynamic congruence closure algorithm. This allows the algorithm to accept new equations after some equations have already been processed, which enables the procedure to work incrementally [3].

Now we want to look at an example how Shostak's algorithm works, therefore we consider the following set of equations $E = \{a = b, f(f(a)) = f(b)\}$ [3]. The steps of the completion procedure using Shostak's strategy can be seen in Table 4.3.

Downey-Sethi-Tarjan

In this section we take a look at the congruence closure algorithm from Downey, Sethi and Tarjan [19]. The algorithm expects the input to be a directed acyclic graph (DAG), representing the terms used in the equations, with an equivalence relation specified on its nodes [3], which represent the input equations. Both the DAG and the equivalences on the nodes are specified as rewrite rules. For our running example this means that the input to the procedure will be $(\emptyset, D_1 \cup C_1)$ where D_1 is the DAG of the input terms and C_1 represents the input equivalences over the nodes of the DAG. For the example from the previous section we have the following sets, $D_1 = \{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4\}$ and $C_1 = \{c_0 \to c_1, c_3 \to c_4\}$.

The algorithm works as follows. Since the input is already a partially finished rewrite system, some rules are available at the start of the procedure, namely the input DAG and the equivalences over the DAG. So the algorithm starts by using a possible *collapse* rule, if the use of the collapse rule enables a *deduction* step the step is carried out as well. This is done until no more collapse steps are possible. After this the equations that were moved to the E component via collapsing of rules and deduction are simplified and deleted if they are trivial after the simplification. All non-trivial equations are converted into rewrite rules via the *orientation* step of the completion procedure. If all equations in the E component have

Step	E	R	Inference Rule
1	$\emptyset$	$\{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4\} \cup \{c_0 \to c_1, c_3 \to c_4\}$	Collapse $f(c_0) \to c_2$ by $c_0 \to c_1$
2	$f(c_1) = c_2$	$\{a \to c_0, b \to c_1, f(c_2) \to c_3, f(c_1) \to c_4\} \cup \{c_0 \to c_1, c_3 \to c_4\}$	Simplify $f(c_1) = c_2$ by $f(c_1) \to c_4$
3	$c_4 = c_2$	$\{a \to c_0, b \to c_1, f(c_2) \to c_3, f(c_1) \to c_4\} \cup \{c_0 \to c_1, c_3 \to c_4\}$	Orient $c_4 = c_2$
4	$\emptyset$	$\{a \to c_0, b \to c_1, f(c_2) \to c_3, f(c_1) \to c_4\} \cup \{c_0 \to c_1, c_3 \to c_4, c_4 \to c_2\}$	

Table 4.4: The steps of the completion procedure for congruence closure from Downey, Sethi, and Tarjan.

been processed, the procedure starts again by trying to collapse rules and deduce new equations. The procedure continues until no more collapse and deduction steps are possible and the set of equations E is empty.

The Downey, Sethi, and Tarjan algorithm can be described by the following combination of inference rules.
$DST = ((\mathit{Collapse} \circ (\mathit{Deduction} \cup \{\epsilon\}))^* \circ (\mathit{Simplification}^* \circ (\mathit{Deletion} \cup \mathit{Orientation}))^*)^*$
where ϵ is the null inference rule which does not change the state of the procedure.

To get a better understanding of the algorithm we consider the example from the previous section. This time we are using the congruence closure algorithm from Downey, Sethi, and Tarjan. The steps of the completion procedure can be seen in Table 4.4.

Nelson-Oppen

In this section we introduce the Nelson-Oppen congruence closure strategy [3]. The Nelson-Oppen procedure is quite different from the two strategies described above, it does not use the normal *deduction* rule described above. Instead, it uses the following modified version of the deduction inference rule:

- *NODeduction:*
  $(E; R) \vdash (E \cup \{s = t\}; R)$ if there are two rules $f(c_1, \ldots, c_n) \to s$ and $f(d_1, \ldots, d_n) \to t \in R$ such that $c_i \to u$ and $d_i \to u$ for $i = 1 \ldots n$
  This deduction steps adds an equality between the right-hand side of two rewrite rules, if the left-hand side of the two rewrite rules are congruent to each other.

The input for the Nelson-Oppen method is $(E; D)$, where D is the input dag, like in the Downey, Sethi, and Tarjan algorithm. The E component is initialized with the equivalences specified on the input dag. For the running example the two sets look like this: $D = \{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4\}$ and $E = \{c_0 = c_1, c_3 = c_4\}$.

The procedure starts by selecting an equation from the set of equations E. This equation is then simplified until no more *simplification* steps are possible. After the simplification, the equation is either deleted if it is trivial or orientated if it is not trivial. After this the *NODedcution* rule is applied until no more non-trivial equations can be learned from it. The procedure continues until the E component is empty.

1	$\{c_0 = c_1, c_3 = c_4\}$	$\{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4\}$	Orient $c_0 = c_1$
2	$\{c_3 = c_4\}$	$\{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4, c_0 \to c_1\}$	NODeduction $c_2 = c_4$ from $f(c_0) \to c_2$ and $f(c_1) \to c_4$ using the rule $c_0 \to c_1$ for establishing congruence between the left sides
3	$\{c_2 = c_4, c_3 = c_4\}$	$\{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4, c_0 \to c_1\}$	Orient $c_2 = c_4$
4	$\{c_3 = c_4\}$	$\{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4, c_0 \to c_1, c_2 \to c_4\}$	Orient $c_3 = c_4$
5	$\emptyset$	$\{a \to c_0, b \to c_1, f(c_0) \to c_2, f(c_2) \to c_3, f(c_1) \to c_4, c_0 \to c_1, c_2 \to c_4, c_3 \to c_4\}$	

Table 4.5: The steps of the completion procedure for congruence closure from Nelson and Oppen.

Now we can state the Nelson-Oppen strategy for congruence closure [3]:
$NO = (Simplification^* \circ (Orientation \cup Deletion) \circ NODeduction^*)^*$

Again using the same set of equations as in the previous two cases we want to illustrate how the Nelson-Oppen method works. The steps of the procedure can be seen in Table 4.5.

4.2 Congruence Closure Modulo Associativity and Commutativity

In the previous section we established the completion procedure for the standard congruence closure, now we will extend this completion procedure to take associativity and commutativity into account. In order to achieve this the rules of the completion procedure have to be slightly modified. The modified version of the completion procedure looks as follows [12]. In addition to the changes to the completion procedure, the reduction ordering of terms must also be adjusted to handle the presence of associative, commutative functions. There has been a lot of research in the area of term ordering in the presence of associative and commutative functions. We will use the ordering introduced in [40].

Before we state the necessary changes to the completion procedure we give some important definitions. A binary function f is commutative if the following property holds: $\forall x, y. f(x, y) = f(y, x)$. A binary function f is associative if the following property holds: $\forall x, y, z. f(f(x, y), z) = f(x, f(y, z))$. The flattened form of a term $flat(t)$ is obtained by using the following rules:

- $flat(g(t_1, \dots, t_n)) = g(flat(t_1), \dots, flat(t_n))$, where g is a function that is not associative and commutative.
- $flat(f(s, t)) = f(s_1, \dots, s_n, t_1, \dots, t_n$, where f is an associative and commutative function and $flat(s) = f(s_1, \dots, s_n)$ and $flat(t) = f(t_1, \dots, t_n)$

Flattening removes nested applications of an associative and commutative function and transforms them into one call with a variable number of arguments. For example the flattened form of the term $t = f(a, f(b, c))$ is $flat(t) = f(a, b, c)$. We use $=_{AC}$ to express that two terms are equal with respect to associativity and commutativity, for example $f(f(a, b), c)$ is equal to $f(b, f(a, c))$ with respect to associativity and commutativity. One way to determine whether or not two terms are equal with respect to associativity and commutativity is to compare their flattened forms. When comparing commutative functions one has to consider all permutations of the parameters to decide whether or not the two functions are equal. By $s \rightarrow_{AC\backslash R} t$ we denote that s can be rewritten to t using the rewrite rules in R, where term matching is done by the $=_{AC}$ operator. For example the term $t = f(a, f(b, c))$ can be rewritten using the rule $f(b, f(a, c)) \rightarrow a$, since $f(a, f(b, c)) =_{AC} f(b, f(a, c))$. Not only term matching has to be done in respect to the associativity and commutativity of functions also the reduction ordering ($\succ$) has to be adapted to work in the presence of associative and commutative functions, a suitable reduction ordering in presence of associative and commutative functions can be found in [40].

- *Delete:*
  $(E \cup \{s = t\}); R) \vdash (E; R)$ if $s =_{AC} t$
  Like in the standard congruence closure, this rule is used to delete trivial equations from the context. But in the case of associativity and commutativity the triviality of an equation is extended to cover associativity and commutativity, this means that an equation of the form $f(a, f(b, c)) = f(f(b, a), c)$ is considered trivial when f is an associative, commutative function.

- *Compose:*
  $(E; R \cup \{s \rightarrow t\}) \vdash (E; R \cup \{s \rightarrow u\})$ if $t \rightarrow_{AC\backslash R} u$
  The *compose* rule is simply extended by taking associativity and commutativity into account when rewriting the right-hand side of a rewrite rule. For example the rule $f(x, f(y, z)) \rightarrow f(b, a)$ can be composed by the rule $f(a, b) \rightarrow c$ if f is an associative commutative function.

- *Simplify:*
  $(E \cup \{s = t\}; R) \vdash (E \cup \{s = u\}; R)$ if $t \rightarrow_{AC\backslash R} u$
  $(E \cup \{s = t\}; R) \vdash (E \cup \{u = t\}; R)$ if $s \rightarrow_{AC\backslash R} u$
  The *simplify* rule is again used to rewrite the left- and right-hand side of equations in E. When rewriting terms the associativity and commutativity of functions is considered.

- *Orient:*
  $(E \cup \{s = t\}; R) \vdash (E; R \cup \{s \rightarrow t\}$ if $s \succ t$
  When turning an equation into a rewrite rule no changes are needed in the case of associative and commutative functions.

- *Collapse:*
  $(E; R \cup \{s \rightarrow t\}) \vdash (E \cup \{u = t\}; R$ if $s \rightarrow_{AC\backslash R} u$ by the rule $l \rightarrow r \in R$ with $s \rightarrow t \succeq l \rightarrow r$
  The *collapse* rule also is only extended in the form that associativity and commutativity are used when a rule's left-hand side is rewritten, the rule still has to be reinserted into the set of equations in order to obtain the correct orientation, after simplifying the left-hand side.

- *Deduce:*
  $(E; R) \vdash (E \cup \{s = t\}; R)$ if $s = t \in \mathit{headCP}(R)$ where $\mathit{headCP}(R) = \{f(b, r') = f(b', r) | l \rightarrow r \in R, l' \rightarrow r' \in R \exists a^{\mu} : l =_A Cf(a^{\mu}, b) \wedge l' =_A Cf(a^{\mu}, b')\}$
  Deduce is again used to add equations for critical pairs to the rewrite system. In the case of associative, commutative functions critical pairs are overlaps between left-hand side of rules with regard to associativity and commutativity. The left-hand sides of two rules overlap whenever there exists an a^{μ}, which is maximal, that is part of both left-hand sides. Suppose we have the following two rewrite rules $f(a, b) \rightarrow c$ and $f(a, d) \rightarrow e$. Then $a^{\mu} = a$ and deduction yields the following equation: $f(c, d) = f(e, b)$. This can be seen by considering the following reductions, the term $f(f(a, b), d)$ can be rewritten using $f(a, b) \rightarrow c$ which yields the term $f(c, d)$ or it can be rewritten using $f(a, d) \rightarrow e$ which results in the term $f(e, b)$. Thus the term $f(c, d)$ and the term $f(e, b)$ must be equal and we need a rewrite rule to capture this critical pair.

Again the procedure is finished if none of the above rules can be applied anymore. The result is a ground and canonical rewrite system for the congruence closure of the given set of equalities. The proof for the correctness of the procedure can be found in [3].

The completion procedure stated above can be used to generate a congruence closure modulo associativity and commutativity for the example taken from [12], where the set of equations is $E = \{f(a_1, a_4) = a_1, f(a_3, a_6) = f(a_5, a_5), a_5 = a_4, a_6 = a_2\}$ and we use $f \succ a_6 \succ \cdots \succ a_1$ as the reduction ordering. The steps to generate the rewrite system for this set of equations can be seen in Table 4.6.

The resulting rewrite system can then be used to check whether the equality $a_1 = f(a_1, f(a_6, a_3))$ logically follows from the set of equations. The necessary deduction steps for the proof are shown in table 4.7

The application of the rewrite system to the equation $a_1 = f(a_1, f(a_6, a_3))$ shows that this equation logically follows from the set of equations, because both terms have the same normal form in regard to the rewrite system.

4.3 Congruence Closure Modulo Inverse Functions

After we have established the completion procedure for congruence closure and congruence closure modulo associativity and commutativity in the previous sections, this section is going to deal with a new extension of the congruence closure algorithm, namely the congruence closure modulo inverse functions. Two functions f and g are inverse to each other if the following property holds $\forall x : f(g(x)) = g(f(x)) = x$. We will denote this by $f = g^{-1}$.

A completion procedure that handles the presence of inverse functions directly needs the following two additional rules. It is important to state that both the standard congruence closure completion procedure or the associative, commutative congruence closure completion procedure can be extended by adding the two rules to handle inverse functions as well.

We use $s =_{f=g^{-1}} t$ to denote that the terms t and s are equal with respect to functions f and g being inverse to each other. For example the terms $s = f(g(a))$ and $t = g(f(a))$ are equal to each other, since both can be simplified to a. By $s \rightarrow_{f=g^{-1}} t$ we denote that $s =_{f=g^{-1}} t$, where the occurrence of $f(g(x))$, x being an arbitrary term, inside the term s has been replaced with x. For example $h(f(g(a))) \rightarrow_{f=g^{-1}} h(a)$.

- *Application:*
  $(E \cup \{s = t\}; R) \vdash (E \cup \{u = t\}; R)$ if $f = g^{-1}$ and $s \rightarrow_{f=g^{-1}} u$
  $(E \cup \{s = t\}; R) \vdash (E \cup \{s = u\}; R)$ if $f = g^{-1}$ and $t \rightarrow_{f=g^{-1}} u$
  This rule is used to eliminate the occurrence of any inverse function application. By inverse function application we denote the occurrence of the term $f(g(x))$ or the term $g(f(x))$ where $f = g^{-1}$ anywhere within a given term. In this case the term can immediately be replaced with x. This rule can be used on both sides of an equation.

- *IFDeduction:*
  $(E; R \cup \{f(s) \rightarrow t\}) \vdash (E \cup \{s = g(t)\}; R \cup \{f(s) \rightarrow t\})$ if $f = g^{-1}$
  $(E; R \cup \{t \rightarrow f(s)\}) \vdash (E \cup \{s = g(t)\}; R \cup \{t \rightarrow f(s)\})$ if $f = g^{-1}$
  This rule is used to add equations to the context that logically follow from the presence of inverse functions. Consider the following example: From the equation $f(s) = t$ the equation $s = g(t)$ logically follows when $f = g^{-1}$. Note that rewrite rules that follow from equalities learned via IFDeduction do not need to be considered for further IFDeduction steps, since the learned equality would be trivial. For our example from $g(t) \rightarrow s$, the rewrite rule for the equality $s = g(t)$, the following equality follows $f(s) = t$, which is the equality justifying the rewrite rule $f(s) \rightarrow t$ and thus the equality is trivial.

Step	E	R	Inference Rule
1	$\{f(a_1, a_4) = a_1, f(a_3, a_6) = f(a_5, a_5), a_5 = a_4, a_6 = a_2\}$	$\emptyset$	Orient $f(a_1, a_4) = a_1$
2	$\{f(a_3, a_6) = f(a_5, a_5), a_5 = a_4, a_6 = a_2\}$	$\{f(a_1, a_4) \to a_1\}$	Orient $f(a_3, a_6) = f(a_5, a_5)$
3	$\{a_5 = a_4, a_6 = a_2\}$	$\{f(a_1, a_4) \to a_1, f(a_3, a_6) \to f(a_5, a_5)\}$	Orient $a_5 = a_4$
4	$\{a_6 = a_2\}$	$\{f(a_1, a_4) \to a_1, f(a_3, a_6) \to f(a_5, a_5), a_5 \to a_4\}$	Compose $f(a_3, a_6 \to f(a_5, a_5)$ by $a_5 \to a_4$
5	$\{a_6 = a_2\}$	$\{f(a_1, a_4) \to a_1, f(a_3, a_6) \to f(a_4, a_4), a_5 \to a_4\}$	Orient $a_6 = a_2$
6	$\emptyset$	$\{f(a_1, a_4) \to a_1, f(a_3, a_6) \to f(a_4, a_4), a_5 \to a_4, a_6 \to a_2\}$	Collapse $f(a_3, a_6) \to f(a_4, a_4)$ by $a_6 \to a_2$
7	$\{f(a_3, a_2) = f(a_4, a_4)\}$	$\{f(a_1, a_4) \to a_1, a_5 \to a_4, a_6 \to a_2\}$	Orient $f(a_3, a_2) = f(a_4, a_4)$
8	$\emptyset$	$\{f(a_1, a_4) \to a_1, a_5 \to a_4, a_6 \to a_2, f(a_3, a_2) \to f(a_4, a_4)\}$	Deduce $f(a_1, a_4) = f(a_1, f(a_3, a_2))$ from $f(a1_1, a_4) \to a_1$ and $f(a_4, a_4) \to f(a_3, a2)$
9	$\{f(a_1, a_4) = f(a_1, f(a_3, a_2))\}$	$\{f(a_1, a_4) \to a_1, a_5 \to a_4, a_6 \to a_2, f(a_3, a_2) \to f(a_4, a_4)\}$	Simplify $f(a_1, a_4) = f(a_1, f(a_3, a2))$ by $f(a_1, a_4) \to a_1$
10	$\{a_1 = f(a_1, f(a_3, a_2))$	$\{f(a_1, a_4) \to a_1, a_5 \to a_4, a_6 \to a_2, f(a_3, a_2) \to f(a_4, a_4)\}$	Orient $a_1 = f(a_1, f(a_3, a_2))$
11	$\emptyset$	$\{f(a_1, a_4) \to a_1, a_5 \to a_4, a_6 \to a_2, f(a_3, a_2) \to f(a_4, a_4), f(a_1, f(a_3, a_2)) \to a_1\}$	

Table 4.6: The steps of the associative commutative congruence closure completion procedure for the example.

Step	Left-Hand Side	Right-Hand Side	Rewrite Rule
1	a_1	$f(a_1, f(a_6, a_3))$	$a_6 \to a_2$
2	a_1	$f(a_1, f(a_6, a_3))$	$f(a_1, f(a_3, a_2)) \to a_1$
3	a_1	a_1	

Table 4.7: Shows the deduction steps for the equation $a_1 = f(a_1, f(a_6, a_3))$ when using the rewrite system generated by the associative commutative congruence closure completion procedure.

The procedure is finished if no more rules can be applied. The result is a ground and canonical rewrite system. Any congruence closure algorithm that terminates and returns a ground and canonical rewrite system, can be extended with this rules to handle inverse functions and it will still terminate with a ground and canonical rewrite system. For a proof idea of this we consider the two rules which are added. The first rule removes applications of inverse functions to each other. Since there can only be finitely many such application, the rule can only be applied finitely often. Thus the procedure must still terminate if the underlying procedure terminates. As the rule does not change the R component of the state or introduces variables, the generated rewrite system must still be ground and canonical if the underlying procedure generates such a system. The second rule generates at most two equality for each rule in the rewrite system. If the underlying procedure terminates, it can only produce finitely many rules, thus only finitely many new equations are added. Rules added from these equations do not need to be considered for further IFDeductions, because the equations learned from those deductions are trivial. Thus the procedure still terminates if the underlying procedure terminates. Again the rule does not change the R component or introduces variables. Therefore, the generated rewrite system will still be ground and canonical if the underlying procedure generates a ground and canonical system.

After the necessary extensions to the completion procedure have been explained we are going to look at two examples. The first example illustrates using the standard congruence closure procedure extended with inverse functions, while the second one uses the associative, commutative congruence closure completion procedure extended with inverse functions.

For the first example consider the following set of equations $E = \{f(g(e)) = a, g(c) = b, u(g(c), c) = u(a, u(b, c))\}$ where $f = g^{-1}$, with the reduction ordering $u \succ g \succ f \succ e \succ d \succ c \succ b \succ a$. The necessary deduction steps for the congruence closure completion procedure with inverse functions can be seen in Table 4.8.

This example shows that real implementations of this algorithm will perform one simple optimization, namely there should be no *IFDeductions* from equations that where added via *IFDeductions*.

The resulting rewrite system can then be used to check whether $u(b, c) = u(e, u(g(c), f(b)))$ logically follows from the given equations. The deduction steps for this proof can be seen in Table 4.9. Note that before using the rewrite system to check whether or not an equality logically follows, one has to able the *application* rule to the equality which should be checked until this is not possible anymore.

The generated rewrite system shows that the equation $u(b, c) = u(e, u(g(c), f(b)))$ is a logical consequence from the given set of equations.

Now we consider the associative, commutative congruence closure completion procedure extended with inverse functions. For this purpose we use the following set of equations $E = \{u(a, b) = c, u(f(g(d)), a) = e, f(a) = b\}$, where u is an associative commutative function and $f = g^{-1}$, with the reduction ordering $u \succ g \succ f \succ e \succ d \succ c \succ b \succ a$. The steps of the completion procedure can be found in Table 4.10.

In this example we applied the optimization that no *IFDeduction* steps are performed on equations that were learned via an *IFDeduction* step. This was done to keep the example shorter.

This set of rewrite rules can be used to show that the equation $u(u(g(b), b), d) = u(e, f(a))$ is indeed a logical consequence of the set of equations given above. Table 4.11 contains the proof of this fact.

Step	E	R	Inference Rule
1	$\{f(g(e)) = a,\ g(c) = b,\ u(g(c), c) = u(a, u(b, c))\}$	$\emptyset$	Application $f(g(e)) = a$
2	$\{e = a,\ g(c) = b,\ u(g(c), c) = u(a, u(b, c))\}$	$\emptyset$	Orient $e = a$
3	$\{g(c) = b,\ u(g(c), c) = u(a, u(b, c))\}$	$\{e \to a\}$	Orient $g(c) = b$
4	$\{u(g(c), c) = u(a, u(b, c))\}$	$\{e \to a, g(c) \to b\}$	Simplify $u(g(c), c) = u(a, u(b, c))$ by $g(c) \to b$
5	$\{u(b, c) = u(a, u(b, c))\}$	$\{e \to a, g(c) \to b\}$	IFDeduce $f(b) = c$ from $g(c) \to b$
6	$\{u(b, c) = u(a, u(b, c)),\ f(b) = c\}$	$\{e \to a, g(c) \to b\}$	Orient $u(b, c) = u(a, u(b, c))$
7	$\{f(b) = c\}$	$\{e \to a,\ g(c) \to b,\ u(a, u(b, c)) \to u(b, c)\}$	Orient $f(b) = c$
8	$\emptyset$	$\{e \to a,\ g(c) \to b,\ u(a, u(b, c)) \to u(b, c),\ f(b) \to c\}$	IFDeduce $g(c) = b$ from $f(b) \to c$
9	$\{g(c) = b\}$	$\{e \to a,\ g(c) \to b,\ u(a, u(b, c)) \to u(b, c),\ f(b) \to c\}$	Simplify $g(c) = b$ by $g(c) \to b$
10	$\{b = b\}$	$\{e \to a,\ g(c) \to b,\ u(a, u(b, c)) \to u(b, c),\ f(b) \to c\}$	Delete $b = b$
11	$\emptyset$	$\{e \to a,\ g(c) \to b,\ u(a, u(b, c)) \to u(b, c),\ f(b) \to c\}$	

Table 4.8: The steps performed by the congruence closure completion procedure extended with inverse functions.

Step	Left-Hand Side	Right-Hand Side	Rewrite Rule
1	$u(b, c)$	$u(e, u(g(c), f(b)))$	$e \to a$
2	$u(b, c)$	$u(a, u(g(c), f(b)))$	$g(c) \to b$
3	$u(b, c)$	$u(a, u(b, f(b)))$	$f(b) \to c$
4	$u(b, c)$	$u(a, u(b, c))$	$u(a, u(b, c)) \to u(b, c)$
5	$u(b, c)$	$u(b, c)$	

Table 4.9: The necessary proof steps to show that the equation $u(b, c) = u(e, u(g(c), f(b)))$ logically follows from the given set of equations. The used rewrite system was generated with the congruence closure completion procedure with inverse functions.

Step	E	R	Inference Rule
1	$\{u(a,b) = c, u(f(g(d)),a) = e, f(a) = b\}$	$\emptyset$	Orient $u(a,b) = c$
2	$\{u(f(g(d)),a) = e, f(a) = b\}$	$\{u(a,b) \to c\}$	Application $u(f(g(d)),a) = e$
3	$\{u(d,a) = e, f(a) = b\}$	$\{u(a,b) \to c\}$	Orient $u(d,a) = e$
4	$\{f(a) = b\}$	$\{u(a,b) \to c, u(d,a) \to e\}$	Deduce $u(c,d) = u(e,b)$ from $u(a,b) \to c$ and $u(d,a) \to e$
5	$\{f(a) = b, u(c,d) = u(e,b)\}$	$\{u(a,b) \to c, u(d,a) \to e\}$	Orient $f(a) = b$
6	$\{u(c,d) = u(e,b)\}$	$\{u(a,b) \to c, u(d,a) \to e, f(a) \to b\}$	IFDeduce $g(b) = a$ from $f(a) \to b$
7	$\{u(c,d) = u(e,b), g(b) = a\}$	$\{u(a,b) \to c, u(d,a) \to e, f(a) \to b\}$	Orient $u(c,d) = u(e,b)$
8	$\{g(b) = a\}$	$\{u(a,b) \to c, u(d,a) \to e, f(a) \to b, u(e,b) \to u(c,d)\}$	Orient $g(b) = a$
9	$\emptyset$	$\{u(a,b) \to c, u(d,a) \to e, f(a) \to b, u(e,b) \to u(c,d), g(b) \to a\}$	

Table 4.10: The steps performed by the congruence closure modulo associativity and commutativity completion procedure extended with inverse functions.

Step	Left-Hand Side	Right-Hand Side	Rewrite Rule
1	$u(u(g(b),b),d)$	$u(e,f(a))$	$g(b) \to a$
2	$u(u(a,b),d)$	$u(e,f(a))$	$u(a,b) \to c$
3	$u(c,d)$	$u(e,f(a))$	$f(a) \to b$
4	$u(c,d)$	$u(e,b)$	$u(e,b) \to u(c,d)$
5	$u(c,d)$	$u(c,d)$	

Table 4.11: Application of the rewrite system generated by the congruence closure modulo associativity and commutativity completion procedure with inverse functions to the equation $u(u(g(b),b),d) = u(e,f(a))$. This shows that the equation is a logically consequence of the given set of equations.

Chapter 5

Implementation

In the previous chapters we established SMT, rewrite systems, as well as the various congruence closure algorithms. This chapter presents the prototype implementation of the congruence closure modulo associativity and commutativity and the congruence closure modulo inverse functions algorithms as well as their combination. All three algorithms were implemented inside the Z3 theorem prover [16]. The Z3 theorem prover is written in C++ and is publicly available at `http://z3.codeplex.com/`. Z3 provides a textual user interface and supports a lot of different background theories, including linear integer arithmetic, uninterpreted functions with equality, the theory of arrays, real numbers, quantifiers. Z3 is also capable of producing models for satisfiable formulas, as well as producing a proof of unsatisfiability for formulas which are not satisfiable.

5.1 Congruence Closure in Z3

The previous chapters have dealt with the theoretical background of congruence closure algorithms. This section introduces the congruence closure algorithm as it is implemented inside the Z3 theorem prover. It is important to note that in practice congruence closure algorithms are not implemented as pure rewrite systems, but employ several enhancements in order to be more efficient. A modern variant of a congruence closure algorithm was introduced by Nieuwenhuis and Oliveras.[37].

5.1.1 Definitions

Before we state the congruence closure algorithm that is used by Z3, we declare a few terms and data structures that we will use throughout the explanations of the various implemented congruence closure algorithms.

The *congruence class* of an element a is the set containing all elements that are equal to a, including a itself. In our setting each congruence class has an representative element associated with it. The *representative element* is an element of the congruence class and it is used instead of all other elements of the congruence class during the computation of the congruence closure algorithm. The *context* of the Z3 theorem prover is the set of all congruence classes for the elements occurring in the formula.

Z3 uses *proof trees* to represent the various congruence classes. A proof tree is a tree with directed edges for which the following properties hold. The representative element of the class is the root of the tree. All elements of the proof tree have a path to the representative element of the congruence class. Each edge in the tree has a justification for the equality of the elements it connects associated with it. There are two types of justifications inside the Z3 theorem prover, `EQUATION` and `CONGRUENCE`. `EQUATION` is used for elements that are equal because there is an equality in the input between them, and `CONGRUENCE` is used for equalities that are not in the input but learned during the procedure. The advantage of using proof trees is that they can be easily used to generate proofs for the equality of two elements contained in the congruence class. For this, one simply has to search a common ancestor of the two elements one wants to prove to be equal. A common ancestor is an element that is reachable from both elements in the proof tree. Such an element has to exist, since all elements in the proof tree can reach the root of the congruence class. In order to prove the equality of the elements one has to prove that both are equal to the common ancestor, this can be done using the justifications associated to the edges of the proof tree.

The `enode` data structure is used by Z3 to represent all terms and equalities that occur inside a formula. An `enode` stores the following information:

- *Root*: The representative element of the `enodes` congruence class.
- *Class Size*: The number of elements in this `enodes` congruence class.
- *Parents*: The set of `enodes` which have this `enode` as a parameter. For example the `enode` $f(a, b)$ is a parent of the `enodes` a and b.
- *Next Node*: The pointer to the next `enode` in the proof tree. A justification for the equality between this `enode` and its next `enode` is associated with this pointer.

A *congruence table* is a hash-map that is used to recognize congruences between elements that have the same top-level function symbol. For each function symbol with arity > 0, one congruence table is created inside the Z3 theorem prover. Each element with a function symbol at the top is inserted into the congruence table of the top-level function symbol. The hash value of an element is based on its parameters, for example the element $f(a, b)$ uses the terms a and b to calculate its hash value.

5.1.2 Congruence Closure Algorithm

After we have explained the terms and the data structures, we now can state the congruence closure algorithm used by the Z3 theorem prover. We will start by giving a basic description of the necessary steps to perform a congruence closure for a given set of equalities and inequalities. After that, we will explain how the steps are carried out inside the Z3 theorem prover.

At the start of the procedure all terms are in their own congruence class. The procedure then starts to add equalities from the input formula to the context. Whenever an equality is added, the congruence classes of the two elements have to be merged, the merge of congruence classes is an union of the two sets. During the merge a new representative element has to be chosen for the merged congruence class. After that we have to check if the added equality allows us to learn a new equality due to the functional consistency property. For example, if we add the equality $a = b$ then we learn the equality $f(a, b) = f(a, a)$. If the two sides of a learned equality are not in the same congruence class, the congruence classes of the two elements have to merged. If at one point after a merge the resulting congruence class contains two elements which should not be equal, then we can stop the procedure and return that the given set of equalities and inequalities is unsatisfiable. If all equalities from the input and all learned equalities can be processed without detecting a contradiction the input is satisfiable.

Now we want to explain how these basic steps are performed inside the Z3 theorem prover. At the start of the solving process all equalities from the input are added to the *equalities to propagate* queue, which holds all equalities that still need to be processed, and all `enodes` with a function symbol at the top-level are added to their respective congruence table. After that the first equality from the equalities to propagate queue is added to the context. There are four `enodes` involved in this process, n_1 the side of the equation with the smaller number of elements in the congruence class, n_2 side of the equation with the larger number of elements in the congruence class, r_1 the root (representative) element of n_1, and r_2 the root element of n_2. If both sides have the same number of elements in their congruence class then

n_1 is the left-hand side of the equation and n_2 is the right-hand side. An equation is considered trivial if $r_1 = r_2$ and trivial equation are not added to the context.

In the basic algorithm the merging of congruence classes is a simple union of the two sets, however, since the Z3 theorem prover uses proof trees to store the congruence classes rather than sets we have to explain how two proof trees can be merged. The first step during this merge is that the path from n_1 to r_1 in the tree is inverted. This can be done since all edges are equalities and thus can be used in both directions. After this step n_1 is the new root of the proof tree for the congruence class of n_1. This step is necessary since the next step adds a directed edge from n_1 to n_2, which merges the two proof trees to one common proof tree. In order to show that the resulting proof tree is indeed a proof tree for the merged congruence class we consider the following two cases. First, all elements of the congruence class of r_2 are still in the tree and have a path to r_2. Second, all elements of the congruence class of r_1 are also still in the tree and after inverting the path from n_1 to r_1, all elements have a path to n_1. Since n_1 is connected with n_2 and n_2 has a path to r_2 all elements in the congruence class of r_1 have a path to r_2, thus the tree is a proof tree for the merged congruence class with r_2 being the representative element.

The deduction step from the basic algorithm is done with the help of the congruence table inside the Z3 theorem prover. As mentioned earlier, each `enode` uses the root of its parameters to calculate its hash value. Whenever two congruence classes are merged the parents of `enodes` in the congruence class of n_1 have to update their hash value, since elements in this congruence class now use r_2 instead of r_1 as their root element. This recalculation can lead to the fact that two `enodes` have the same hash value, for example if c is the root element of b, then the `enodes` $f(a, b)$ and $f(a, c)$ have the same hash value. Two `enodes` that have the same hash value, have to be equal to each other, thus if they are not already in the same congruence class then an equality between the two elements is added to the equalities to propagate queue.

The procedure continues until either the equalities to propagate queue is empty and no contradiction was found, then the input is satisfiable or till a contradiction is detected after a merge of two congruence classes. A contradiction is detected, if the left-hand side and the right-hand side of an inequality are in the same congruence class. This check is carried out in the same phase as the hash update, since inequalities are also among the parents of `enodes`.

Listing 5.1 shows the congruence closure algorithm explained above as pseudocode. The function `Z3_congruence_closure` is the main function that processes the equalities to propagate queue, named `eq_queue` here, and calls the function `add_equality` to merge the congruence classes. `add_equality` merges the two congruence classes as explained above. Note that for the sake of simplicity the functions `update_hash`, `report_conflict`, `merge`, and `conflict_reported` are not stated in detail, as they were either explained before or their exact implementation is not necessary for the understanding of the algorithm.

```
Z3_congruence_closure() {
  while(not conflict_reported() and eq_queue.has_element) {
    e = eq_queue.top;
    add_equality(e.lhs, e.rhs, e.js);
  }
  if(conflict_reported())
    return unsat;

  return sat;
}

add_equality(lhs : enode, rhs : enode, js : justification) {
  if(lhs.class_size > rhs.class_size) {
    n1 = rhs;
    r1 = rhs.root;
    n2 = lhs;
    r2 = lhs.root;
  } else {
    n1 = lhs;
    r1 = lhs.root;
    n2 = rhs;
    r2 = rhs.root;
  }

  if(r1 == r2)
    return;

  foreach e in r1.class {
    e.root = r2;
  }

  foreach node in r1.class {
    foreach e in node.parents {
      e.update_hash();
      if(e.is_dis_eq() and e.lhs.root == e.rhs.root)
          report_conflict(e, n1, n2);
    }
  }

  r2.proof_tree = merge(r1.proof_tree, r2.proof_tree, n1, n2);
}
```

Listing 5.1: The pseudocode representation of the Z3 congruence closure algorithm.

5.1.3 Examples

Example 1

We will illustrate the congruence closure algorithm inside Z3 with the help of an example, which is shown in Listing 5.2. The example uses the SMT-LIBv2 input language [11]. It starts by declaring the logic that is used by the formula `(set-logic QF_UF)`, which is the theory of uninterpreted functions with equality (EUF). After the declaration of the background theory, a sort named `I` is declared, with the `(declare-sort I)` command. A sort in SMT is comparable to a data-type in programming languages, e.g., the background theory of linear integer arithmetic provides the predefined sort `Int`, which represents (mathematical) integers. It is important to note that we only declare the sort, which means we are not restricting the possible values that exist in this sort. After the sort declaration the function symbols used in the formula are declared. The first function symbol that is declared is the function `f` which takes two parameters of the type `I` and returns a value of the type `I`, as seen in the `(declare-fun f (I I) I)` command. Then we declare the unary function `g`, which takes a parameter of the sort `I` and returns a value of the type `I`, this is done in Line 6. In the following lines, three functions `a`, `b`, and `c` are declared, they act as constants in the examples since they do not take any parameters. This completes the definition of the signature of the formula and we are now ready to state the formula itself, which is done with one or more `assert` statements. In this case one `assert` statement which asserts three equalities ($\{f(a,b) = a, g(b) = c, f(g(b), b) = c\}$) as well as three inequalities ($\{\neg(b = c), \neg(a = b), \neg(a = c)\}$) is used. Form now on we will us $a \neq b$ as abbreviation for $\not(a = b)$. The `(check-sat)` command is used to start the solving process of the formula. The last command invoked in the example is the `(exit)` command, which causes the SMT solver to close. This is necessary because some SMT solvers like Z3 support incremental solving. Additional equations can be added to the already solved formula, where the SMT solver does not start a new but extends the current solution.

This can potentially result in a decreased runtime for certain applications, where incremental solving can be used.

```
(set-logic QF_UF)

(declare-sort I)

(declare-fun f (I I) I)
(declare-fun g (I) I)
(declare-fun a () I)
(declare-fun b () I)
(declare-fun c () I)

(assert
  (and
    (= (f a b) a)
    (= (g b) c)
    (= (f (g b) b) c)
    (not (= b c))
    (not (= a b))
    (not (= a c))
  )
)

(check-sat)
(exit)
```

Listing 5.2: Example for the congruence closure algorithm in Z3. This example is written in the SMT-LIBv2 input format. It declares five functions and asserts three equalities as well as three inequalities.

As stated earlier each term and each equality and inequality is represented as an `enode`. For the example the following `enodes` are constructed: $\{a, b, c, f(a,b), g(b), f(g(b),b), f(a,b) = a, g(b) = c, f(g(b),b) = c, b \neq c, a \neq b, a \neq c\}$. At the start of the procedure, each `enode` is in its own congruence class and therefore each `enode` is its own *root* as well. Thus, the size of each congruence class is 1. The initial congruence classes have the following members: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}\}$, $\{\underline{f(a,b)}\}$, $\{\underline{g(b)}\}$, $\{\underline{f(g(b),b)}\}$, where the underlined elements are the *root* elements of the respective congruence class. At the start of the procedure, the *next* pointers for the proof tree start with `NULL`, since all congruence classes have only one element.

After all the `enodes` have been constructed and inserted into their congruence tables, the procedure starts to add equations to the current context, starting with the equations from the asserted formula. The equations are added in the order they appear in the original formula. The first equation added in our example is $f(a,b) = a$. Adding an equation to the context leads to the merging of the two corresponding congruence classes. In this example, both the congruence class of $f(a,b)$ and the one of a have the size 1, which leads to the merging of the congruence class of $f(a,b)$ into the congruence class of a, this means that $n_1 = f(a,b)$ and $n_2 = a$. Since both n_1 and n_2 are the *root* elements of their congruence classes, r_1 and r_2 get the following values $r_1 = n_1$ and $r_2 = n_2$. Now that the four `enodes` involved have their value we can check whether or not the equation is redundant. An equation is redundant when $r_1 = r_2$ since the two sides of the equation are already in the same congruence class, which is not the case in the example. The first step in the merging process is that all elements in the congruence class of r_1 get a new *root* element r_2. Since this *root* change causes the hash value of any `enode` that is a parent of an `enode` in the congruence class of r_1 to change, these `enodes` have to recalculate their hash values. After updating the hash values, the proof trees of the two congruence classes have to be merged. This finishes the merging process of the two congruence classes and after the merge the congruence classes have the following elements: $\{\underline{a},\ f(a,b)\}$, $\{\underline{b}\}$, $\{\underline{c}\}$, $\{\underline{g(b)}\}$, $\{\underline{f(g(b),b)}\}$. The corresponding proof trees for the congruence classes can be seen in Figure 5.1. Note that only the proof trees for congruence classes with more than one element are shown.

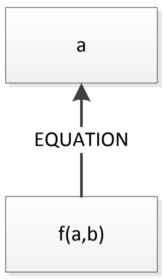

Figure 5.1: The proof trees after the equation $f(a,b) = a$ was added to the context. Only the proof trees for congruence classes with more than one element are shown. The edges are labelled with the justification for the equality of the two corresponding elements.

The next equality processed by Z3 is the equality $g(b) = c$ from the asserted formula. Since again both congruence classes have only one member, the congruence class of $g(b)$ is merged into the congruence class of c. The four `enodes` are assigned the following values $n_1 = r_1 = g(b)$ and $n_2 = r_2 = c$. Here the *root* change of elements in the congruence class of r_1 leads to a recalculated hash value for an `enode`. The `enode` $f(g(b),b)$ has to recalculate its hash value, since the new *root* of $g(b)$ is c, $f(g(b),b)$ uses the arguments c and b for the calculation of its hash value. But since no other `enode` uses the parameters c and b for its hash value no new equations are learned from this *root* change. The congruence classes after the merge have the following elements: $\{\underline{a},\ f(a,b)\}$, $\{\underline{b}\}$, $\{\underline{c},\ g(b)\}$, $\{\underline{f(g(b),b)}\}$ and the corresponding proof trees for the congruence classes can be seen in Figure 5.2.

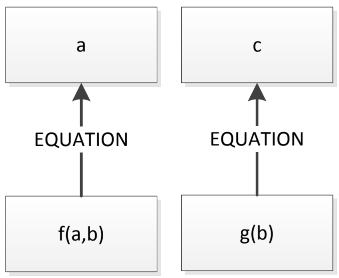

Figure 5.2: The proof trees after the equation $g(b) = c$ was added to the context. Only the proof trees for congruence classes with more than one element are shown. The edges are labelled with the justification for the equality of the two corresponding elements.

The next step adds the last equality from the asserted formula to the context, $f(g(b),b) = c$. Here the congruence class of $f(g(b),b)$ is merged into the congruence class of c, since the congruence class of c is bigger then the congruence class of $f(g(b),b)$. Thus, the four `enodes` have the following values $n_1 = r_1 = f(g(b),b)$ and $n_2 = r_2 = c$. The merge of the two congruence classes leads to the following congruence classes: $\{\underline{a},\ f(a,b)\}$, $\{\underline{b}\}$, $\{\underline{c},\ g(b),\ f(g(b),b)\}$. The corresponding proof trees for the congruence classes can be found in Figure 5.3.

This finishes the congruence closure procedure for the example, since all equations from the asserted formula were processed and no new equations were detected during the solving process. The example

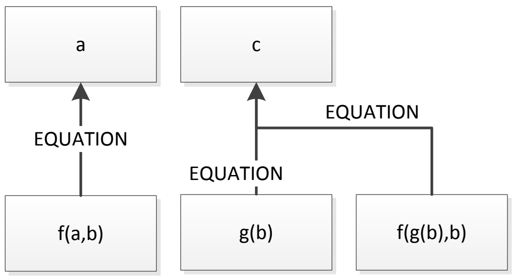

Figure 5.3: The proof trees after the equation $f(g(b), b) = c$ was added to the context. Only the proof trees of congruence classes with more than one element are shown. The edges are labelled with the justification for the equality of the two elements.

is satisfiable, since at the end of the procedure the `enodes` a, b, and c are all in separate congruence classes.

If we make a slight change to our example, we get a model for our formula. Listing 5.3 shows the modified example. The change is that we inserted the `(get-model)` command at the end, in order to tell the Z3 SMT solver that we want a model for our formula.

```
(set-logic QF_UF)

(declare-sort I)

(declare-fun f (I I) I)
(declare-fun g (I) I)
(declare-fun a () I)
(declare-fun b () I)
(declare-fun c () I)

(assert
  (and
    (= (f a b) a)
    (= (g b) c)
    (= (f (g b) b) c)
    (not (= b c))
    (not (= a b))
    (not (= a c))
  )
)

(check-sat)
(get-model)
(exit)
```

Listing 5.3: The modified example with the get-model command to retrieve a model for the formula.

The insertion of the `get-model` command does not change the way our formula gets solved, it just adds an additional step at the end of the chain. After the propagation of our equalities is finished, all terms occurring in the formula are assigned a value. For this purpose, Z3 iterates all the `enodes` and assigns them values according to the congruence class they are in. First, `enodes` that take no parameter are assigned and from those values the various functions get defined. A model for the formula is shown in Listing 5.4.

```
(model
  ;; universe for I:
  ;;   I!val!1 I!val!2 I!val!0
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun I!val!1 () I)
  (declare-fun I!val!2 () I)
  (declare-fun I!val!0 () I)
  ;; cardinality constraint:
  (forall ((x I)) (or (= x I!val!1) (= x I!val!2) (= x I!val!0)))
  ;; -----------
  (define-fun b () I
    I!val!1)
  (define-fun a () I
    I!val!0)
  (define-fun c () I
    I!val!2)
  (define-fun f ((x!1 I) (x!2 I)) I
    (ite (and (= x!1 I!val!0) (= x!2 I!val!1)) I!val!0
    (ite (and (= x!1 I!val!2) (= x!2 I!val!1)) I!val!2
      I!val!0)))
  (define-fun g ((x!1 I)) I
    (ite (= x!1 I!val!1) I!val!2
      I!val!2))
)
```

Listing 5.4: Shows the model generated by Z3 for the example. It gives values to all the constants and provides a table for the used functions.

The model declaration starts by defining the values that exist in our sort `I`. For the example, we have three different values for elements of the sort type `I`. These are the values `I!val!0`, `I!val!1`, and `I!val!2`, the `forall` application assures that elements of the type `I` only take one of the three values. After the values that exist in the formula are established, the model gives values for the term occurring in the formula, starting with the three constants `a`, `b`, and `c`. The last declaration tells which values the functions take for specific input values. As we can see in the function declaration only the instances which occur as terms in the formula are explicitly considered in the model for the function, all other instances are abbreviated in the else branch.

Example 2

We can also consider the following change to the example from Listing 5.2, to illustrate how Z3 handles formulas that are unsatisfiable and how proofs of unsatisfiability are generated by the Z3 theorem prover. The modified example is shown in Listing 5.5.

```
(set-logic QF_UF)
(set-option :produce-proofs true)

(declare-sort I)

(declare-fun f (I I) I)
(declare-fun g (I) I)
(declare-fun a () I)
(declare-fun b () I)
```

```
(declare-fun c () I)

(assert
  (and
    (= (f a b) a)
    (= (g b) c)
    (= (f (g b) b) c)
    (= (f c b) b)
    (not (= b c))
    (not (= a b))
    (not (= a c))
  )
)

(check-sat)
(get-proof)
(exit)
```

Listing 5.5: The example was extended with an additional equation to make it unsatisfiable. Additionally the commands to provide a proof of unsatisfiability were added

We can look at the needed changes to make the example unsatisfiable as well as to generate a proof of unsatisfiability. In order to make the example unsatisfiable we added an additional equation to our formula, namely $f(c, b) = b$. To get a proof of unsatisfiability we added the following two commands to the input file, `(set-option :produce-proofs true)`. This commands tells the Z3 SMT solver that it should generate a proof if the formula is unsatisfiable. With the `(get-proof)` command the proof of unsatisfiability can be retrieved from the Z3 theorem prover.

The solving process for this formula again starts by adding the equality $f(a, b) = a$ to the context. The addition of this equality leads to same result as in the previous case, except that we have an additional congruence class for the `enode` $f(c, b)$. However, when we add the second equation to the context ($g(b) = c$) we have stated that the `enode` $f(g(b), b)$ has to recalculate its hash value. As stated above, it uses the arguments c and b to calculate its new hash value since c is the *root* element of the congruence class that $g(b)$ is in. But in this example we have a second `enode` ($f(c, b)$) that uses the arguments c and b to calculate its hash value. Thus, we learn the following equation from our hash-table $f(g(b), b) = f(c, b)$. This equation gets added to the equations to propagate queue with the justification `CONGRUENCE`. The congruence classes after this merging contain the following elements: $\{\underline{a}, f(a, b)\}$, $\{\underline{b}\}$, $\{\underline{c}, g(b)\}$, $\{\underline{f(g(b), b)}\}$, $\{\underline{f(c, b)}\}$, the corresponding proof trees for the congruence classes stay the same as shown in Figure 5.2.

The addition of the third equation from the asserted formula $f(g(b), b) = c$ to the context. Here again all the things stated above also hold true in this case and the congruence classes after the merge look like this: $\{\underline{a}, f(a, b)\}$, $\{\underline{b}\}$, $\{\underline{c}, g(b), f(g(b), b)\}$, $\{\underline{f(c, b)}\}$. Figure 5.3 shows the proof trees for the congruence classes that contain more than one element.

After that, we add the last equation from the asserted formula to the context ($f(c, b) = b$). In this case we choose to merge the congruence class of $f(c, b)$ into the congruence class of b and thus, the four `enodes` have the following values $n_1 = r_1 = f(c, b)$ and $n_2 = r_2 = b$. The *root* change for elements in the congruence class of r_1 does not lead to the recalculation of any hash values. The congruence classes contain the following elements after all equations from the formula have been added: $\{\underline{a}, f(a, b)\}$, $\{\underline{b}, f(c, b)\}$, $\{\underline{c}, g(b), f(g(b), b)\}$. The proof trees for the congruence classes can be seen in Figure 5.4.

Now all the equations contained in the asserted formula have been propagated, but since a new equation was learned during the propagation the procedure is not finished in this case. The equality $f(g(b), b) = f(c, b)$ still needs to be added to the context. When this equality is added to the context the congruence class of $f(c, b)$ is merged into the congruence class of $f(g(b), b)$ due to the number of

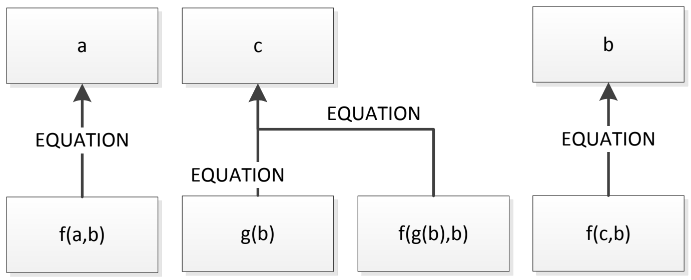

Figure 5.4: The proofs trees after the equation $f(c, b) = b$ has been added to the context. Only the proof trees for congruence classes with more than one element in them are shown. All edges are labelled with the justification for the equality of the two associated elements.

elements in the congruence classes. For our example the four `enodes` are assigned as follows $n_1 = f(c, b)$, $r_1 = b$, $n_2 = f(g(b), b)$, and $r_2 = c$. The change of the *root* element for all elements in the congruence class of r_1 leads to re-computation of the hash values of the following `enodes`: $f(a, b)$, $g(b)$, $f(g(b), b)$, $f(c, b)$ but no new equalities are learned from these changes. The `enode` b has two inequalities $a \neq b$ and $b \neq c$ as parents. Since the *root* of b changed the inequalities have to be checked for contradictions. The inequality $a \neq b$ is still fulfilled but the second inequality $b \neq c$ is no longer true since b and c are in the same congruence class, which leads to the fact that the formula is unsatisfiable. Since we now know that the formula is unsatisfiable the procedure is finished, regardless of the fact whether all equations have been propagated or not. The final congruence classes contain the following `enodes`, $\{\underline{a}, f(a, b)\}$, $\{\underline{c}, g(b), f(g(b), b), b, f(c, b)\}$. The corresponding proof trees for the congruence classes are shown in Figure 5.5. Please note that $f(c, b)$ no longer points to b but that the edge was reversed. This was done while merging the corresponding proof trees.

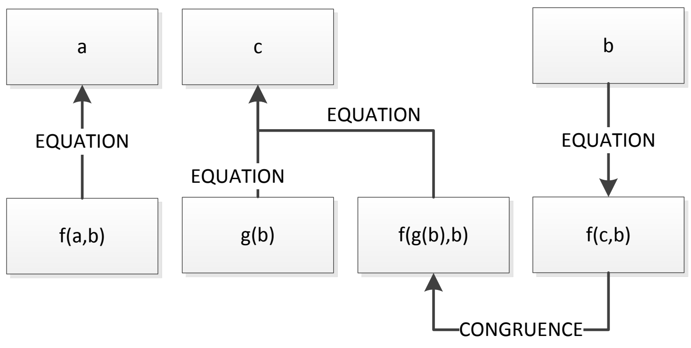

Figure 5.5: The proof trees at the end of the procedure for the modified example. Only the proof trees for congruence classes with more than one element are shown. The edges are labelled with the justification for the equality of the elements.

In Listing 5.6 we can see the proof of unsatisfiability provided by the Z3 SMT solver.

```
(get-proof)
((set-logic QF_UF)
(proof
(let (($x16 (= b c)))
(let ((?x9 (g b)))
(let ((?x12 (f ?x9 b)))
(let (($x13 (= ?x12 c)))
(let (($x20 (= a c)))
(let (($x21 (not $x20)))
(let (($x18 (= a b)))
(let (($x19 (not $x18)))
(let (($x17 (not $x16)))
(let ((?x14 (f c b)))
(let (($x15 (= ?x14 b)))
(let (($x11 (= ?x9 c)))
(let ((?x7 (f a b)))
(let (($x8 (= ?x7 a)))
(let (($x22 (and $x8 $x11 $x13 $x15 $x17 $x19 $x21)))
(let ((@x43 (asserted $x22)))
(let ((@x48 (|and-elim| @x43 $x13)))
(let ((@x58 (trans (symm (|and-elim| @x43 $x15) (= b ?x14)) (monotonicity
    (symm
(|and-elim| @x43 $x11) (= c ?x9)) (= ?x14 ?x12)) (= b ?x12))))
(let ((@x50 (|and-elim| @x43 $x17)))
(|unit-resolution| @x50 (trans @x58 @x48 $x16) false))))))))))))))))))))
   )
```

Listing 5.6: Proof of unsatisfiability for the modified example provided by the Z3 SMT solver.

The proof of unsatisfiability shown in Listing 5.6 is built with the help of the proof tree shown in figure 5.5 in the following way. The inequality that leads to the contradiction is $b \neq c$. The proof trees contains all the information needed to construct the proof, one just has to follow the way from b to c in order to construct the proof. The first step in the proof chain is the proof of the equality $b = f(b, c)$ which can be retrieved via an and elimination on the asserted formula. The second step is the proof for the congruence of $f(c, b)$ and $f(g(b), b)$. For this proof one needs to prove the following two equalities $c = g(b)$ and $b = b$. In order to prove that $c = g(b)$ holds, the proof tree contains the information that $g(b)$ and c are asserted to be equal and therefore the proof is again an and elimination on the original formula. The second proof is trivial, since b is always equal to b. After constructing the two necessary proofs, the proof for $f(g(b), b) = f(c, b)$ can be constructed, since $g(b)$ is proven to be equal to c and b is equal to b so due to the functional consistency of uninterpreted functions $f(g(b), b)$ must be equal to $f(c, b)$. The last step of the proof is the equality between $f(g(b), b)$ and c, which again is asserted in the original formula. The contradiction can then be constructed from the previously constructed proof that $b = c$ and the asserted inequality between b and c.

5.2 Congruence Closure Modulo Associativity and Commutativity

In the previous section we have explained how the standard congruence closure algorithm in Z3 [16] works. Now we want to take a look at the extensions we made to handle associative and commutative functions as well. We introduced the necessary rewrite rules to handle associative commutative functions in Section 4.2.

5.2.1 Congruence Closure Algorithm

In this section we will explain the changes we made to the data structures as well as the algorithm to handle associative and commutative uninterpreted functions. We will start by briefly describing the changes to the data structures defined in 5.1.1. Then we will explain how we modified the congruence closure algorithm to handle the theory of uninterpreted functions with equality modulo associativity and commutativity.

In the context of associative and commutative functions, each `enode` of an associative and commutative function symbol has a list of current arguments associated with it. The list of *current arguments* stores the flattened list of arguments the function currently has. Note that the list of current arguments is always sorted and always contains at least two elements. For example, if the function f is associative and commutative the current arguments of the term $f(b, f(a, c))$ are $\{a, b, c\}$. This list of arguments is used to compare whether or not two terms are equal considering the associativity and commutativity of the function, since two terms that are equal have the same list of current arguments. This list is also used to calculate the hash value of an `enode`.

We also added a *current set of rules* to the context of the Z3 theorem prover. The current set of rules stores all the rewrite rules that are generated during the run of the procedure.

While the basic congruence closure procedure stays the same, some changes are necessary to handle associative and commutative functions. A reduction ordering is now used to decide the assignment of the four `enodes`. n_1 now is the side of the equation which has the bigger root element according to the reduction ordering, while n_2 is the side of the equation that has the smaller root element according to the reduction ordering. r_1 is still the root element of n_1 and r_2 is the root element of n_2. We use the reduction ordering introduced by Rubio and Nieuwenhuis [40]. If the two sides of the equality are the same according to the reduction ordering n_1 is the left-hand side of the equality and n_2 is the right-hand side of the equality.

The merging process of two proof trees stays the same as in the non associative and commutative case, however, before the hash values of `enodes` are recalculated due to the root change, a rewrite rule for the equality is generated, if the justification for the equality is `CONGRUENCE`, then no rewrite rule will be generated, since this rule would be trivial. The generated rewrite rule is always of the form $r_1 \to r_2$. After the rewrite rule is generated and added to the current set of rules, all possible collapse and compose steps are carried out for the current set of rules. After that the current set of rules is used to carry out all possible simplification steps on the terms occurring in the formula. During this simplification, the current arguments of terms occurring in the formula change to reflect the changed argument list of an function application. For example, the rewrite rule $f(a, b) \to a$ can be used to simplify the current arguments of $f(b, f(a, c))$ to $\{a, c\}$. However, the rule cannot be used to simplify the current arguments of the term $f(a, b)$, since a term must always have at least two elements in the list of current arguments, this simplification is reflected in the changed root for the term $f(a, b)$. After this step the hash values of the `enodes` is recalculated. As in the standard algorithm, if two elements have the same hash, they have to be equal and an equality between them is added to the equalities to propagate queue.

After all equalities from the equalities to propagate queue were processed, an additional step is carried out. This step is the associative and commutative deduction step described in Section 4.2. Equalities that are recognized during this step are added to the equalities to propagate queue. These equalities have the newly introduced justification of `SUPERPOSITION`, since the associative and commutative deduction step is sometimes also called AC-Superposition [4]. During this deduction step new `enodes` are created for terms that do not occur in the formula but are discovered to be equal to other terms.

The algorithm is finished if a contradiction is detected or if the equalities to propagate queue is empty and no new equalities are learned using the associative and commutative deduction step. If a contradiction is detected, the input is unsatisfiable, otherwise it is satisfiable.

Listing 5.7 shows the modified congruence closure algorithm as pseudo-code. The changes are the introduction of the reduction ordering (`ac_greater`) and the use of the functions `add_rule`, `update_rules`, and `simplify_enodes` inside the merge function. These functions are used to generate rules, to update the rules and to update the current arguments of the `enodes` during the merging as described above. Note that the deduction step is not shown here, because it only adds equations for critical pairs to the equalities to propagate queue.

```
ac_congruence_closure() {
  while(not conflict_reported() and eq_queue.has_element) {
    e = eq_queue.top;
    add_equality(e.lhs, e.rhs, e.js);
    if(not eq_queue.has_element)
      deduce_new_equalities();
  }
  if(conflict_reported())
    return unsat;

  return sat;
}

add_equality(lhs : enode, rhs : enode, js : justification) {
  if(ac_greater(lhs.root, rhs.root)) {
    n1 = rhs;
    r1 = rhs.root;
    n2 = lhs;
    r2 = lhs.root;
  } else {
    n1 = lhs;
    r1 = lhs.root;
    n2 = rhs;
    r2 = rhs.root;
  }

  if(r1 == r2)
    return;

  add_rule(r1, r2);
  update_rules();
  simplify_enodes();

  foreach e in r1.class {
    e.root = r2;
  }

  foreach node in r1.class {
    foreach e in node.parents {
      e.update_hash();
      if(e.is_dis_eq() and e.lhs.root == e.rhs.root)
          report_conflict(e, n1, n2);
    }
  }

  r2.proof_tree = merge(r1.proof_tree, r2.proof_tree, n1, n2);
}
```

Listing 5.7: The pseudocode representation of the congruence closure algorithm modulo associativity and commutativity.

Another important change is the fact that the dedicated congruence closure modulo associativity and commutativity does not create a complete model for satisfiable examples. Not complete in this case means that only the instances needed during the computation of the input are shown in the model. This is a design decision, since the creation of a complete model would mean that all possible instances of the associative and commutative law have to be generated, which would lead to a significant time overhead.

5.2.2 Examples

Example 1

In the example shown in Listing 5.8 we define one associative and commutative function `f` that takes two parameters. Associativity and commutativity are denoted by the `:ac` addition in the function declaration. We also declare three constants a, b, and c. Our set of equations is $E = \{f(a, f(b, c)) = f(a, b), f(b, f(a, c)) = c, f(a, b) = a\}$ with one inequality $\{b \neq c\}$.

```
(set-logic QF_ACUF)

(declare-sort I)

(declare-fun f (I I) I :ac)
(declare-fun a () I)
(declare-fun b () I)
(declare-fun c () I)

(assert
  (and
    (= (f a (f b c)) (f a b))
    (= (f b (f a c)) c)
    (= (f a b) a)
    (not (= b c))
  )
)

(check-sat)
(exit)
```

Listing 5.8: The example with associative commutative functions. This example is satisfiable.

We have the following `enodes` in the example: $\{a$, b, c, $f(a, b)$, $f(a, c)$, $f(b, c)$, $f(a, f(b, c))$, $f(b, f(a, c))$, $f(a, f(b, c)) = f(a, b)$, $f(b, f(a, c)) = c$, $f(a, b) = a$, $b \neq c\}$, when these `enodes` are created and inserted into their congruence table, we immediately discover a new congruence between the `enodes` $f(a, f(b, c))$ and $f(b, f(a, c))$. Both `enodes` have the same *current arguments*, namely $\{a$, b, $c\}$. This learned equality is appended to the list of equalities which have to be propagated. At the start of the procedure the list contains all the equations from the asserted formula. The congruence classes at the beginning of the procedure look like this: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}\}$, $\{\underline{f(a, b)}\}$, $\{\underline{f(a, c)}\}$, $\{\underline{f(a, f(b, c))}\}$, $\{\underline{f(b, f(a, c))}\}$, $\{\underline{f(b, c)}\}$. Again, the underlined elements are the root elements of their corresponding congruence class.

After the `enodes` have been created and inserted into their congruence tables, the algorithm starts to propagate equalities, again starting with the equations asserted in the formula. So the first equality added to the context is $f(a, f(b, c)) = f(a, b)$. As mentioned earlier, in the context of associative and commutative uninterpreted functions one has to use a reduction ordering to decide which congruence class should get merged into the other. Here the decision is made that the congruence class of $f(a, f(b, c))$ should get merged into the congruence class of $f(a, b)$. The four `enodes` are assigned the following values $n_1 = r_1 = f(a, f(b, c))$ and $n_2 = r_2 = f(a, b)$. The rewrite rule created for this equation is $f(a, f(b, c)) \rightarrow f(a, b)$. These rewrite rules are also used to simplify the *current arguments* of associative and commutative functions. In this case however the rule cannot be used to simplify any *current arguments*, since there is no `enode` which has $f(a, f(b, c))$ as a strict subterm. The next step consists of updating the *root* element of all elements in the congruence class of r_1. This leads to the re-computation of hash values for non associative and commutative `enodes` as explained in Section 5.1.2. The merging of the proof trees is also performed as described earlier. The congruence classes contain the following elements after the merge: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}\}$, $\{\underline{f(a,b)}, f(a,f(b,c))\}$, $\{\underline{f(a,c)}\}$, $\{\underline{f(b,f(a,c))}\}$, $\{\underline{f(b,c)}\}$. The corresponding proof trees are illustrated in Figure 5.6.

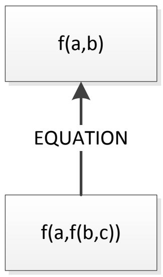

Figure 5.6: The proof trees after the equation $f(a,f(b,c)) = f(a,b)$ was added to the context by the congruence closure modulo associativity and commutativity algorithm. The figure only contains proof trees for congruence classes that have more than one element. The edges are labelled with the justification for the equality of the two corresponding elements.

The next equation that gets added to the context is $f(b,f(a,c)) = c$. Both `enodes` are the *root* of their respective congruence classes and therefore the reduction ordering decides that the congruence class of $f(b,f(a,c))$ gets merged into the congruence class of c. Thus, the four `enodes` get assigned the following values, $n_1 = r_1 = f(b,f(a,c))$ and $n_2 = r_2 = c$. The corresponding rewrite rule that gets added to the context is $f(b,f(a,c)) \to c$. The addition of this rewrite rule allows a collapse step of the first rule, the first rule thus changes to become $f(a,b) \to c$. Note that during rule collapsing, the rules get reorientated immediately instead of re-adding them to the set of equations. The changed first rule, however, can then be used to collapse the second rule to become $f(c,c) \to c$. Next these rules are used to simplify the current arguments of the terms occurring in the formula. In this example the two `enodes` $f(a,f(b,c))$ and $f(b,f(a,c))$ are simplified to have $\{c,c\}$ as current arguments. After that, the *root* element of all `enodes` in the congruence class of r_1 are updated to become r_2, no new equations are discovered during this step. Finally, the proof trees for the two congruence classes get merged. This leads to the following congruence classes: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}, f(b,f(a,c))\}$, $\{\underline{f(a,b)}, f(a,f(b,c))\}$, $\{\underline{f(a,c)}\}$, $\{\underline{f(b,c)}\}$ with their corresponding proof trees displayed in Figure 5.7. Note that we just write the original form of the `enodes` in the congruence classes and proof trees and not the form with their *current arguments*.

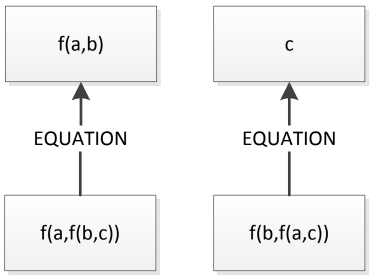

Figure 5.7: The proof trees after the equation $f(b,f(a,c)) = c$ has been added to the context. The figure only contains the proof trees for congruence classes that contain more than one element. The edges are labelled with the justification for the equality of the two elements associated with them.

After this, the last asserted equation ($f(a, b) = a$) gets added to the context. Again, both enodes are the *root* of their respective congruence classes. Therefore the reduction ordering chooses to merge the congruence class of $f(a, b)$ into the congruence class of a. Therefore, we have $n_1 = r_1 = f(a, b)$ and $n_2 = r_2 = a$ and the rule $f(a, b) \to a$ gets added to the context. The addition of this rule allows a collapse step on the rule $f(a, b) \to c$ which changes to become $c \to a$. This rule can then be used to compose and collapse the rule $f(c, c) \to c$ to become $f(a, a) \to a$. The current set of rules is $\{c \to a$, $f(a, a) \to a$, $f(a, b) \to a\}$. With these rules the simplification of the associative and commutative uninterpreted functions enodes leads to the discovery of the following congruences. First applying $c \to a$ to $f(b, c)$ and $f(a, b)$ leads to the discovery of the equation $f(b, c) = f(a, b)$. The second congruence we discover is the congruence between $f(a, f(b, c))$ and $f(a, c)$ since using the rules $c \to a$ and $f(a, b) \to a$ makes both sides equal. Note we do not consider the congruence of $f(b, f(a, c))$ and $f(a, c)$ because $f(b, f(a, c))$ is congruent to $f(a, f(b, c))$. After the merging, the congruence classes contain the following elements: $\{\underline{a}$, $f(a, b)$, $f(a, f(b, c))\}$, $\{\underline{b}\}$, $\{\underline{c}$, $f(b, f(a, c))\}$, $\{\underline{f(a, c)}\}$, $\{\underline{f(b, c)}\}$. The corresponding proof trees for the congruence classes can be seen in Figure 5.8.

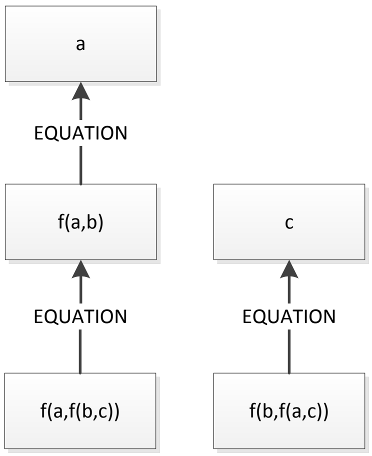

Figure 5.8: The proof trees for the congruence classes after the equation $f(a, b) \to a$ has been added to the context.

After adding all asserted equations to the context, the equations that were learned during the procedure are added to the context as well. The first equation learned in this example was the congruence of $f(a, f(b, c))$ and $f(b, f(a, c))$, which was discovered during the enodes creation. The respective *root* elements are a and c. The reduction ordering decides that the congruence class of $f(b, f(a, c))$ gets merged into the congruence class of $f(a, f(b, c))$, thus $n_1 = f(b, f(a, c))$, $r_1 = c$, $n_2 = f(a, f(b, c))$, and $r_2 = a$. However, this equality is justified by a congruence and thus no rule is created, as those rules would be trivial rules. Although the rules remain unchanged, the *root* elements of enodes in the congruence class of r_1 get changed. The corresponding proof trees also get merged. So after the merge the congruence classes contain the following elements, $\{\underline{a}$, $f(a, b)$, $f(a, f(b, c))$, c, $f(b, f(a, c))\}$, $\{\underline{b}\}$, $\{\underline{f(a, c)}\}$, $\{\underline{f(b, c)}\}$. The proof trees for the congruence classes are displayed in Figure 5.9.

The next congruence that was learned was $f(b, c) = f(a, b)$. The corresponding *root* elements are $f(b, c)$ and a. Thus the reduction ordering chooses to merge the congruence class of $f(b, c)$ into the congruence class of a which leads to the following assignment of the four enodes, $n_1 = r_1 = f(b, c)$, $n_2 = f(a, b)$, and $r_2 = a$. Since this equation is also justified by a congruence again no rule is changed. So after the merging the congruence classes contain the elements $\{\underline{a}$, $f(a, b)$, $f(a, f(b, c))$, c, $f(b, f(a, c))$, $f(b, c)\}$, $\{\underline{b}\}$, $\{\underline{f(a, c)}\}$. Their corresponding proof trees can be seen in Figure 5.10.

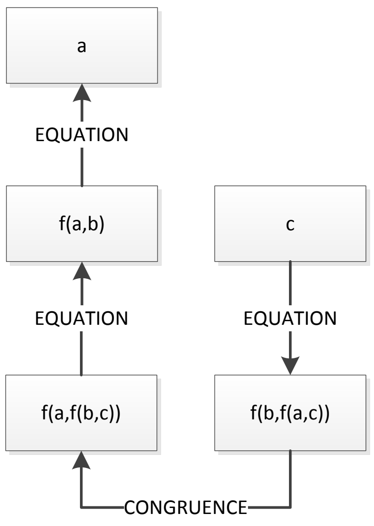

Figure 5.9: The proof trees for the congruence classes after the equation $f(a, f(b, c)) = f(b, f(a, c))$ was added to the context by the congruence closure modulo associativity and commutativity algorithm.

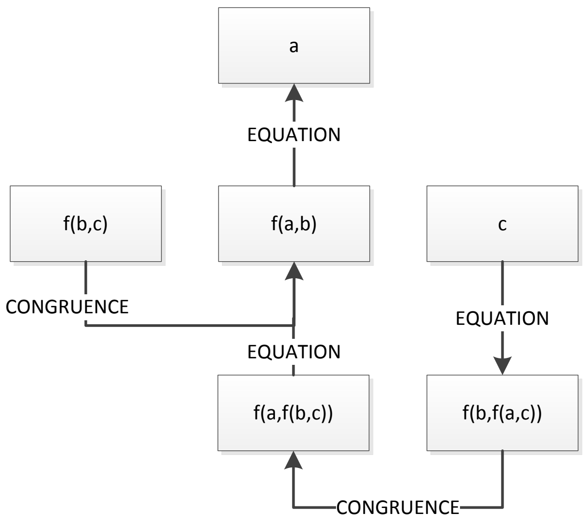

Figure 5.10: The proof trees for the congruence classes after the congruence $f(b, c) = f(a, b)$ has been added to the context by the associative and commutative congruence closure algorithm.

The last congruence that was learned during the propagation of the asserted equations was $f(a, c) = f(a, f(b, c))$ with $f(a, c)$ and a as *root* elements of the congruence classes. The reduction ordering is used to decide that the congruence class of $f(a, c)$ must be merged into the congruence class of a. No rule is created for these equations since it's a congruence and no new congruences are learned during the merge process. After the merging we have the following congruence classes: $\{\underline{a}$, $f(a, b)$, $f(a, f(b, c))$, c, $f(b, f(a, c))$, $f(b, c)$, $f(a, c)\}$, $\{\underline{b}\}$ with their corresponding proof trees in Figure 5.11.

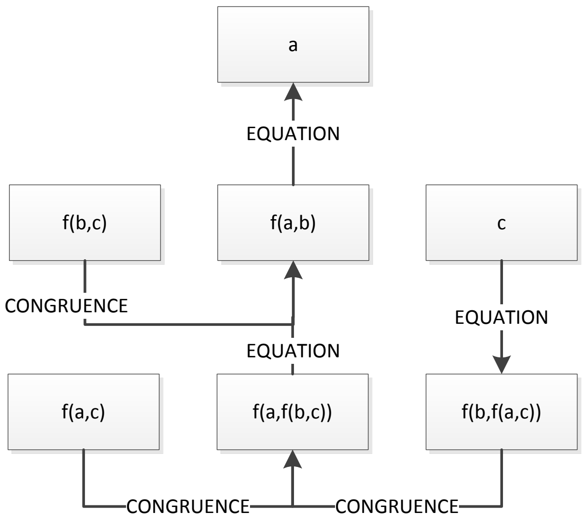

Figure 5.11: The proof trees after the congruence $f(a, c) = f(a, f(b, c))$ has been added to the context by the congruence closure modulo associativity and commutativity algorithm.

Now that all equations are propagated, the algorithm tries to deduce new equations using the deduction rule introduced in Section 4.2. The current rules for the example are $\{c \to a,\ f(a, a) \to a,\ f(a, b) \to a\}$, thus the overlap between the two rules $f(a, a) \to a$ and $f(a, b) \to a$ has to be considered. The AC superposition is calculated as follows:

$$\begin{gathered} f(a, a) \to a \text{ and } f(a, b) \to a \\ f(f(a, a), b) =_{AC} f(f(a, b), a) \\ f(a, b) = f(a, a) \end{gathered}$$

The equation learned via AC superposition is $f(a, b) = f(a, a)$. Since there is no `enode` for the term $f(a, a)$ in the asserted formula, the term is now created. Like all other `enodes` that are created, it starts in its own congruence class as *root* of this class. However, when inserting it into the congruence-table for the function `f`, a new congruence between $f(a, a)$ and $f(a, f(b, c))$ is learned. This congruence is added to the equalities to propagate queue. After the creation of the `enode` $f(a, a)$ is finished, the equation $f(a, b) = f(a, a)$ with the justification `SUPERPOSITION` is added to the equalities to propagate queue.

After the deduction step the equalities to propagate queue is non-empty, thus the procedure still has to add equations to the context. The next equation in the equalities to propagate queue is the congruence between $f(a, a) = f(a, f(b, c))$, with the *root* elements $f(a, a)$ and a. Here the congruence class of $f(a, a)$ gets merged into the congruence class of a and the resulting congruence classes after the merge are: $\{\underline{a},\ f(a, b),\ f(a, f(b, c)),\ c,\ f(b, f(a, c)),\ f(b, c),\ f(a, c),\ f(a, a)\}$, $\{\underline{b}\}$. The corresponding proof trees can be seen in Figure 5.12.

The last remaining equality in the equalities to propagate queue is $f(a, b) = f(a, a)$, which was the result of the AC superposition step. However, since both `enodes` are already in the same congruence class, there is nothing left to do and the procedure is finished.

When we now ask the Z3 theorem prover to give us a model for the above example, we get the output shown in Listing 5.9.

```
(model
```

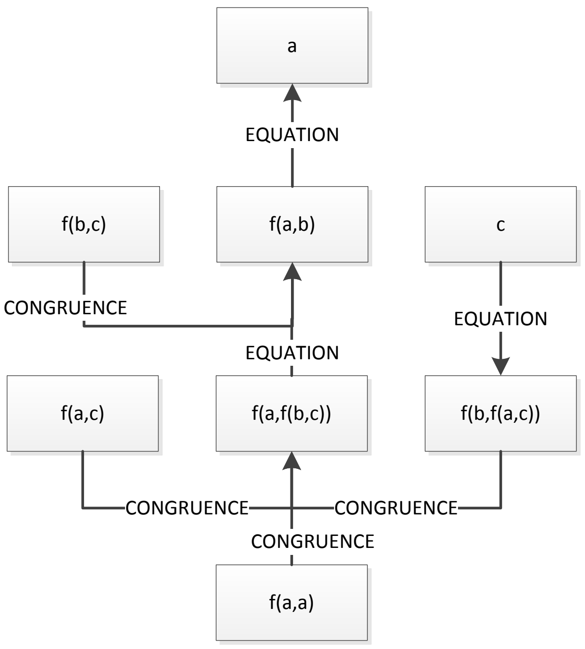

Figure 5.12: The proof trees after the congruence $f(a,a) = f(a, f(b,c))$ has been added to the context by the congruence closure algorithm modulo associativity and commutativity.

```
  ;; universe for I:
  ;;   I!val!1 I!val!0
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun I!val!1 () I)
  (declare-fun I!val!0 () I)
  ;; cardinality constraint:
  (forall ((x I)) (or (= x I!val!1) (= x I!val!0)))
  ;; -----------
  (define-fun b () I
    I!val!1)
  (define-fun a () I
    I!val!0)
  (define-fun c () I
    I!val!0)
  (define-fun f ((x!1 I) (x!2 I)) I :ac
    (ite (and (= x!1 I!val!1) (= x!2 I!val!0)) I!val!0
    (ite (and (= x!1 I!val!0) (= x!2 I!val!0)) I!val!0
      I!val!0)))
)
```

Listing 5.9: The model returned by the Z3 SMT solver for the example using associative commutative functions.

We can see that the Z3 solver has assigned the value `I!val!0` to the first congruence class and the value `I!val!1` to the second congruence class, which only contains the term b. The first thing to note about models for associative commutative functions is, the model contains only values for arguments that where needed during the solving process. This means that the model is not complete for associative commutative functions but contains all the instances that are needed to assign values to the terms occurring in the formula. Complete models are not generated because it takes to long to create all the instances of the associative and commutative law and thus would have a negative impact on the performance of the solver. The second thing to note is that only one entry is created for two arguments. This means that the same arguments just switched in place are not in the table, because the functions are commutative and thus the value is the same. This is done to keep the model small. Thus, if the formula contains both $f(1,2)$ and $f(2,1)$, assuming f is an associative and commutative function, only the first occurring term will be in the model the other term has the same result and therefore does not need an own entry.

Example 2

The second example we want to consider is an unsatisfiable example in order to show how proofs of unsatisfiability are created. Therefore, we consider the formula shown in Listing 5.10.

```
(set-logic QF_ACUF)
(set-option :produce-proofs true)

(declare-sort I)

(declare-fun f (I I) I :ac)
(declare-fun a () I)
(declare-fun b () I)
(declare-fun c () I)

(assert
  (and
    (= (f a b) a)
    (= (f a c) b)
    (not (= (f a c) (f b b)))
  )
)

(check-sat)
(get-proof)
(exit)
```

Listing 5.10: Example of an unsatisfiable SMT formula containing associative commutative functions. The commands to retrieve the proof are also included in the example.

The solving process for this example follows the same principles as before and we therefore are not going into details here. When adding the equations from the formula, two rules are created, $f(a,b) \to a$ and $f(a,c) \to b$. The superposition of these two rules leads to the equation $f(a,c) = f(b,b)$, which contradicts the assertion. The corresponding proof from the Z3 solver is shown in Listing 5.11.

```
((set-logic QF_ACUF)
(proof
(let ((?x12 (f b b)))
(let ((?x10 (f a c)))
(let (($x13 (= ?x10 ?x12)))
(let (($x11 (= ?x10 b)))
(let (($x14 (not $x13)))
(let ((?x7 (f a b)))
(let (($x8 (= ?x7 a)))
(let (($x15 (and $x8 $x11 $x14)))
(let ((@x36 (asserted $x15)))
(let ((@x40 (|and-elim| @x36 $x11)))
(let ((@x39 (|and—elim| @x36 $x8)))
(let ((@x41 (|and—elim| @x36 $x14)))
(|unit—resolution| @x41 (symm (|AC—superposition| @x39 @x40 (= ?x12 ?x10)
   ) $x13)
 false))))))))))))))
```

Listing 5.11: The corresponding proof of unsatisfibility for the example shown in Listing 5.10.

Proofs of unsatisfiability for formulas containing associative and commutative functions are created in the same way as in the standard congruence closure case. The only extension necessary is that two additional proof rules are needed. The first rule is a rule that captures the equality of terms with regard to associativity and commutativity, an example of this proof step is:

```
(|associativity-commutativity| (= (f b (f a c)) (f a (f b c))))
```

The second addition is the rule to capture the superposition between two rules, as shown in Listing 5.11. All the other steps for the proof generation are done in the same way as explained during the standard congruence closure algorithm again utilizing the constructed proof tree.

5.3 Congruence Closure Modulo Inverse Functions

After presenting the implementation of the congruence closure algorithm inside the Z3 theorem prover as well as the necessary extension to handle associative and commutative functions in the previous two sections, we now want to explain the changes we made to the congruence closure algorithm to handle inverse functions. The necessary theory for a congruence closure modulo inverse functions was established in Section 4.3.

5.3.1 Congruence Closure Algorithm

In this section we explain changes we made to the standard congruence closure algorithm inside the Z3 theorem prover to handle inverse functions. For an explanation of the used terms and data structures see Section 5.1.1. The only addition to the data structures is the introduction of two new justifications for equalities. One justification for the Application rule and one justification for the IFDeduce rule.

The basic algorithm stays the same as described in Section 5.1.2 and thus we are not going to restate it here. However, the following changes were made to handle inverse functions inside the congruence closure algorithms. Whenever an `enode` is created for a term it is checked whether the Application rule can be used. It is important to note that the check and the use of the Application rule are only made for the two top most function symbols. For example, the `enode` $f(g(x))$, where $f = g^{-1}$, is simplified using the Application rule, while the term $h(g(f(x)))$ is not, since this simplification was already added during the creation of $g(f(x))$. Another change to the theoretical Application rule is that instead of directly rewriting the term, an equality between the left-hand side and the right-hand side of the used rewrite rule is added to the equalities to propagate queue. For example, instead of directly using the rewrite rule $f(g(x)) \to x$, where $f = g^{-1}$, as done by the Application rule, the equality $f(g(x)) = x$ is added to the equalities to propagate queue.

The IFDeduce rule was added during the adding step of equalities to the context. After the merging of the congruence classes is finished and the hash values of `enodes` were updated, the deduction step for inverse functions is carried as an additional step. At this, the added equality implicitly generates the rewrite rule $r1 \to r2$, we do not actually create or add rewrite rules in the congruence closure modulo inverse functions algorithm, for which then all possible IFDeduction steps are carried out. For example, if we add the equality $f(x) = a$, where $f = g^{-1}$, to the context the following equality is generated by the IFDeduce rule, $g(a) = x$.

Listing 5.12 shows the congruence closure algorithm modulo inverse functions as pseudocode. The only change in the pseudocode compared to the original Z3 congruence closure algorithm is the addition of the deduction step at the end of the merging function `add_equality`. Note that the additional step of adding equalities for `enodes` containing inverse function applications at the top is not shown in the pseudo-code, since this step is carried out during the creation of the respective `enodes`.

```
inv_congruence_closure() {
  while(not conflict_reported() and eq_queue.has_element) {
    e = eq_queue.top;
    add_equality(e.lhs, e.rhs, e.js);
  }
  if(conflict_reported())
    return unsat;

  return sat;
}

add_equality(lhs : enode, rhs : enode, js : justification) {
  if(lhs.class_size > rhs.class_size) {
    n1 = rhs;
    r1 = rhs.root;
    n2 = lhs;
    r2 = lhs.root;
  } else {
    n1 = lhs;
    r1 = lhs.root;
    n2 = rhs;
    r2 = rhs.root;
  }

  if(r1 == r2)
    return;

  foreach e in r1.class {
    e.root = r2;
  }

  foreach node in r1.class {
    foreach e in node.parents {
      e.update_hash();
      if(e.is_dis_eq() and e.lhs.root == e.rhs.root)
        report_conflict(e, n1, n2);
    }
  }

  r2.proof_tree = merge(r1.proof_tree, r2.proof_tree, n1, n2);

  deduce_new_equalities_with_inv_functions(r1, r2);
}
```

Listing 5.12: The pseudocode representation of the congruence closure algorithm modulo inverse function.

It is important to note that the congruence closure algorithm modulo inverse functions also does not create a complete model for satisfiable examples, as in the congruence closure modulo associativity and commutativity case only the instances needed during the solving process are in the model. This decision was made to keep the time overhead for model generation as small as possible.

5.3.2 Example

The following example will be used to illustrate how the congruence closure algorithm modulo inverse functions works:

```
(set-logic QF_IUF)
(set-option :produce-proofs true)

(declare-sort I)

(declare-fun f (I) I)
(declare-fun g (I) I :INV f)
(declare-fun h (I) I)
(declare-fun i (I) I :inv h)
(declare-fun x () I)
(declare-fun y () I)
(declare-fun a () I)

(assert
    (and
      (= (g (f x)) a)
      (= a (i (h (i (h y)))))
      (not (= x y))
    )
)

(check-sat)
(get-proof)
(exit)
```

Listing 5.13: The example for the congruence closure algorithm modulo inverse functions inside Z3. This example is unsatisifiable and contains the commands to retrieve the proof of unsatisfiability.

The example declares four unary functions: f, g, h, and i with the following properties $g = f^{-1}$ and $i = h^{-1}$. It also uses three constant symbols x, y, and a. For this example the following `enodes` are created by the Z3 theorem prover: $\{x$, a, $f(x)$, $g(f(x)$, $g(f(x)) = a$, y, $h(y)$, $i(h(y))$, $h(i(h(y)))$, $i(h(i(h(y))))$, $a = i(h(i(h(y))))$, $x \neq y\}$.

When these `enodes` are created, the following inverse function application equalities are learned: $g(f(x)) = x$, $i(h(y)) = y$, $h(i(h(y))) = h(y)$, and $i(h(i(h(y)))) = i(h(y))$. These equalities are added to the equalities to propagate queue with the justification `INVERSE FUNCTION APPLICATION`. The initial congruence classes look like this: $\{\underline{x}\}$, $\{\underline{a}\}$, $\{\underline{f(x)}\}$, $\{\underline{g(f(x))}\}$, $\{\underline{y}\}$, $\{\underline{h(y)}\}$, $\{\underline{i(h(y))}\}$, $\{\underline{h(i(h(y)))}\}$, $\{\underline{i(h(i(h(y))))}\}$.

The first equality that gets added to the context, like always, is the first equation from the asserted formula, $g(f(x)) = a$. Since no associative and commutative uninterpreted functions are used, the decision which class gets merged into which is again based on the size of the classes, or in this case which class is on the right-hand side of the added equation if they have the same size. The four `enodes` get the following values: $n_1 = r_1 = g(f(x))$ and $n_2 = r_2 = a$. The *root* changes cause no new equation to be detected since there are no parents of $g(f(x))$ that are not equalities. However $f(a) = f(x)$ can be deduced from the equality using the deduction rule for inverse functions. After the merging the congruence classes have the following elements: $\{\underline{x}\}$, $\{\underline{a}, g(f(x))\}$, $\{\underline{f(x)}\}$, $\{\underline{y}\}$, $\{\underline{h(y)}\}$, $\{\underline{i(h(y))}\}$, $\{\underline{h(i(h(y)))}\}$, $\{\underline{i(h(i(h(y))))}\}$. The corresponding proof trees can be seen in Figure 5.13.

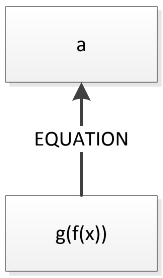

Figure 5.13: The proof trees after the equation $g(f(x)) = a$ has been added to the context by the congruence closure algorithm modulo inverse functions. Only proof trees for congruence classes with more than one element are shown. All the edges are labelled with the justification of the equality for the two associated elements.

Next, the equation $a = i(h(i(h(y))))$ is added to the context. Here the congruence class of $i(h(i(h(y))))$ gets merged into the congruence class of a since the congruence class of a has a higher cardinality. The four `enodes` inside Z3 have the following values $n_1 = r_1 = i(h(i(h(y))))$ and $n_2 = r_2 = a$. The *root* changes again cause no new equations to be learned, however a new equation can be learned via the inverse function deduction rule, $h(a) = h(i(h(y)))$. The congruence classes after the merge have the following members $\{\underline{x}\}$, $\{\underline{a}, g(f(x)), i(h(i(h(y))))\}$, $\{\underline{f(x)}\}$, $\{\underline{y}\}$, $\{\underline{h(y)}\}$, $\{\underline{i(h(y))}\}$, $\{\underline{h(i(h(y)))}\}$ and the corresponding proof trees can be seen in Figure 5.14.

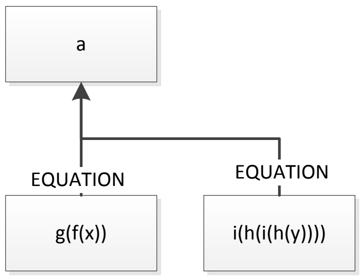

Figure 5.14: The proof trees for the congruence classes after the equation $a = i(h(i(h(y))))$ has been added to the context by the congruence closure algorithm modulo inverse functions.

After all equations from the asserted formula were propagated, the learned equations must be propagated by the congruence closure algorithm. The first equation that was learned during the procedure was $g(f(x)) = x$. Here the congruence class of x gets merged into the congruence class of $g(f(x))$, since the congruence class of $g(f(x))$ contains more elements. The four `enodes` are assigned as follows $n_1 = r_1 = x$, $n_2 = g(f(x))$, and $r_2 = a$. Here no new equalities of any kind are learned and thus the congruence classes after the merge look like this: $\{\underline{a}$, $g(f(x))$, $i(h(i(h(y))))$, $x\}$, $\{\underline{f(x)}\}$, $\{\underline{y}\}$, $\{\underline{h(y)}\}$, $\{\underline{i(h(y))}\}$, $\{\underline{h(i(h(y)))}\}$, with the proof trees illustrated in Figure 5.15.

The next equality that gets added is $i(h(y)) = y$. At this the congruence class of $i(h(y))$ is merged into the one of y because both have the size one. No equations are learned during the merging of the class and thus the congruence classes after the merge have the following elements, $\{\underline{a}$, $g(f(x))$, $i(h(i(h(y))))$, $x\}$, $\{\underline{f(x)}\}$, $\{\underline{y}$, $i(h(y))\}$, $\{\underline{h(y)}\}$, $\{\underline{h(i(h(y)))}\}$. The associated proof trees can be seen in Figure 5.16.

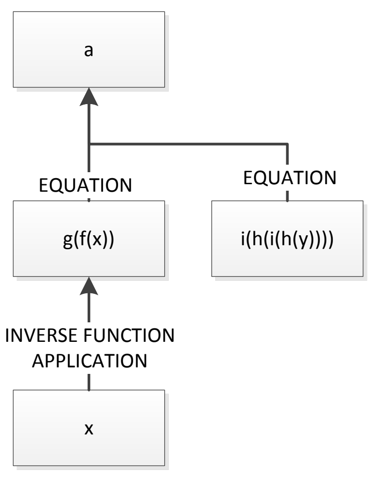

Figure 5.15: The proof trees for the congruence classes after the equation $g(f(x)) = x$ has been added to the context by the congruence closure algorithm modulo inverse functions.

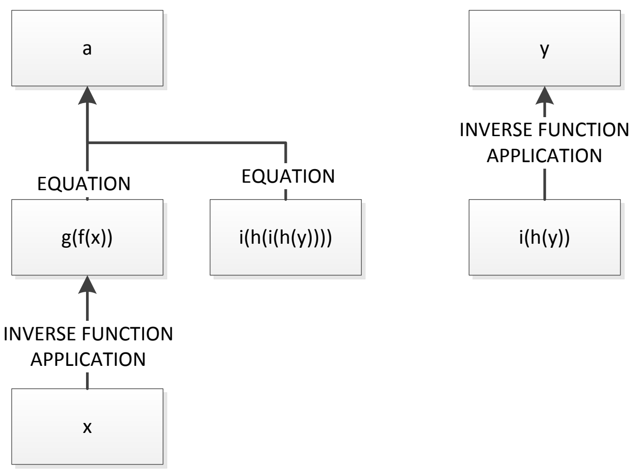

Figure 5.16: The proof trees for the congruence classes after the equation $i(h(y)) = y$ has been added to the context by the congruence closure algorithm modulo inverse functions.

The next equation that was learned during the creation of the `enodes` was $h(i(h(y))) = h(y)$. Since both congruence classes have the same size the congruence class of $h(i(h(y)))$ is merged into the congruence class of $h(y)$. During the merging process the following equation can be learned with the inverse function deduction rule, $i(h(i(h(y)))) = y$. After the merging we have the following congruence classes: $\{\underline{a}, g(f(x)), i(h(i(h(y)))), x\}$, $\{\underline{f(x)}\}$, $\{\underline{y}, i(h(y))\}$, $\{\underline{h(y)}, h(i(h(y)))\}$ with the corresponding proof trees shown in Figure 5.17.

The following topmost element of the equalities to propagate queue is $i(h(i(h(y)))) = i(h(y))$. Here the congruence class of $i(h(y))$ gets merged into the congruence class of $i(h(i(h(y))))$. The four `enodes` are assigned the following values $n_1 = i(h(y))$, $r_1 = y$, $n_2 = i(h(i(h(y))))$, and $r_2 = a$. During the merge of the classes the context gets unsatisfiable since the two `enodes` y and x are in the same congruence class, which is in violation with the inequality stating that x and y can't have the same value. As mentioned earlier at this point the algorithm stops, since a contradiction was detected and

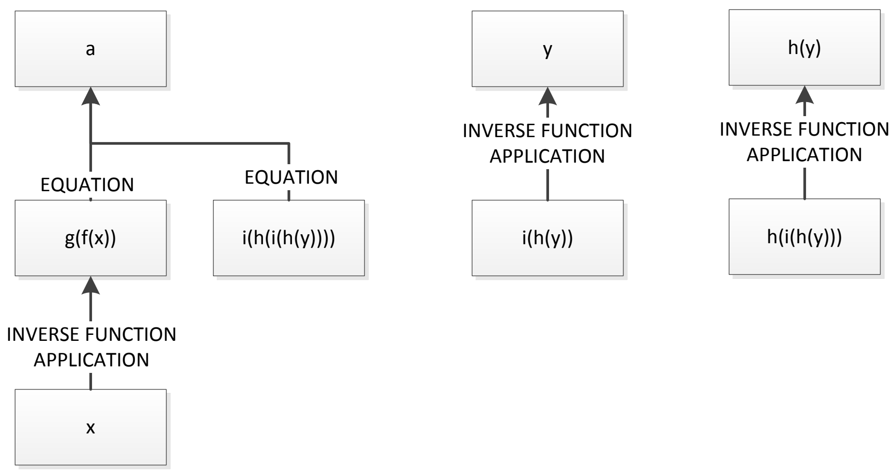
a
y
h(y)
INVERSE FUNCTION APPLICATION
INVERSE FUNCTION APPLICATION
EQUATION
EQUATION
g(f(x))
i(h(i(h(y))))
i(h(y))
h(i(h(y)))
INVERSE FUNCTION APPLICATION
x

Figure 5.17: The proof trees for the congruence classes after the equation $h(i(h(y))) = h(y)$ has been added to the context by the congruence closure algorithm modulo inverse functions.

thus the formula is unsatisfiable. Listing 5.14 shows the corresponding proof of unsatisfiability for the example. Figure 5.18 illustrates the proof trees that were used for the generation of the proof.

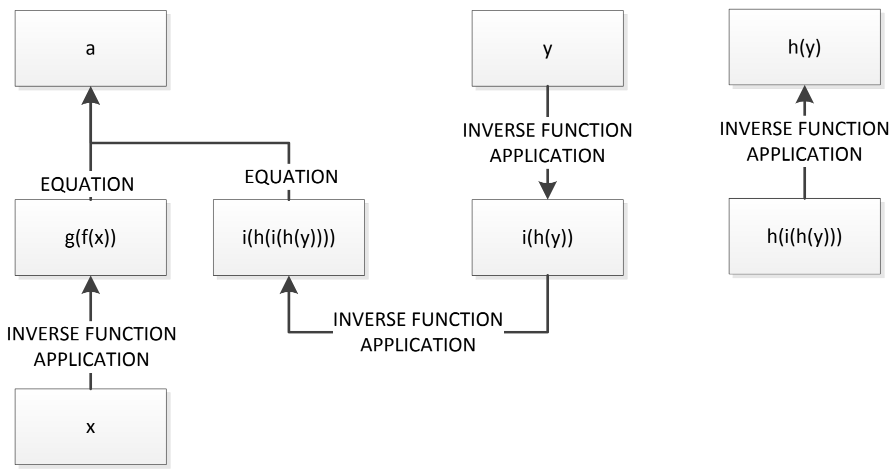
a
y
h(y)
INVERSE FUNCTION APPLICATION
INVERSE FUNCTION APPLICATION
EQUATION
EQUATION
g(f(x))
i(h(i(h(y))))
i(h(y))
h(i(h(y)))
INVERSE FUNCTION APPLICATION
INVERSE FUNCTION APPLICATION
x

Figure 5.18: The proof trees for the congruence classes after the equation $i(h(i(h(y)))) = i(h(y))$ has been added to the context by the congruence closure algorithm modulo inverse functions.

```
((set-logic QF_IUF)
(proof
(let (($x16 (= x y)))
(let ((@x56 (symm (|inverse function application| (= y (i (h y)))) (= (i
    (h y))
y))))
(let ((@x54 (symm (|inverse function application| (= (i (h y)) (i (h (i (
    h y))))
)) (= (i (h (i (h y)))) (i (h y))))))
(let ((?x11 (h y)))
(let ((?x12 (i ?x11)))
(let ((?x13 (h ?x12)))
(let ((?x14 (i ?x13)))
(let (($x15 (= a ?x14)))
(let (($x17 (not $x16)))
(let ((?x6 (f x)))
(let ((?x7 (g ?x6)))
(let (($x9 (= ?x7 a)))
(let (($x18 (and $x9 $x15 $x17)))
(let ((@x39 (asserted $x18)))
(let ((@x43 (|and-elim| @x39 $x15)))
(let ((@x42 (|and-elim| @x39 $x9)))
(let ((@x60 (trans (trans (|inverse function application| (= x ?x7)) @x42
     (= x a
)) @x43 (= x ?x14))))
(let ((@x44 (|and-elim| @x39 $x17)))
(|unit-resolution| @x44 (trans (trans @x60 @x54 (= x ?x12)) @x56 $x16)
    false))))
))))))))))))))))
```

Listing 5.14: Shows the proof of unsatisfiability for the example with inverse functions.

The proof in Listing 5.14 shows the necessary steps to proof that both x and y are equal to a and thus must be equal to each other. Like in the associative and commutative case, two additional proof rules were added: one rule that allows the removal of inverse function applications, seen in Listing 5.14, and a proof rule for equations learned via the inverse function deduction rule.

5.4 Congruence Closure Modulo Associativity, Commutativity and Inverse Functions

The previous section showed how we extended the standard congruence closure algorithm inside the Z3 theorem prover to handle inverse functions. In this section we show how the congruence closure algorithm modulo associativity and commutativity introduced in Section 5.2 can be extended to handle inverse functions. The resulting algorithm is a congruence closure algorithm modulo associativity, commutativity and inverse functions.

5.4.1 Congruence Closure Algorithm

In this section we explain the needed changes to the congruence closure algorithm to get a congruence closure algorithm modulo associativity, commutativity and inverse functions. We explained the extensions to cover associativity and commutativity in Section 5.2.1 and we will use this algorithm as starting point for our congruence closure algorithm modulo associativity, commutativity and inverse functions.

We extended the congruence closure algorithm modulo associativity and commutativity, as explained in Section 5.2.1, in the same way as we extended the standard congruence closure algorithm in order to handle inverse function. During the creation of `enodes` for the terms occurring in the formula the Application rule, introduced Section 4.3, is inserted in the exact same way as described in Section 5.3.1. We also added the IFDeduce rule in the exact same way as described in Section 5.3.1. The only difference with the IFDeduce rule to the standard case is that the rewrite rule is added to the current set of rewrite rules as described for the congruence closure algorithm modulo associativity and commutativity, see Section 5.2.1.

Listing 5.15 shows the pseudocode of the congruence closure algorithm modulo associativity, commutativity, and inverse functions. Again the additional step during the creation of the `enodes`, which adds equalities for inverse functions, is not shown.

```
aci_congruence_closure() {
  while(not conflict_reported() and eq_queue.has_element) {
    e = eq_queue.top;
    add_equality(e.lhs, e.rhs, e.js);
    if(not eq_queue.has_element)
      deduce_new_equalities();
  }
  if(conflict_reported())
    return unsat;

  return sat;
}

add_equality(lhs : enode, rhs : enode, js : justification) {
  if(ac_greater(lhs.root, rhs.root)) {
    n1 = rhs;
    r1 = rhs.root;
    n2 = lhs;
    r2 = lhs.root;
  } else {
    n1 = lhs;
    r1 = lhs.root;
    n2 = rhs;
    r2 = rhs.root;
  }

  if(r1 == r2)
    return;

  add_rule(r1, r2);
  update_rules();
  simplify_enodes();

  foreach e in r1.class {
    e.root = r2;
  }

   foreach node in r1.class {
    foreach e in node.parents {
      e.update_hash();
      if(e.is_dis_eq() and e.lhs.root == e.rhs.root)
        report_conflict(e, n1, n2);
    }
  }

  r2.proof_tree = merge(r1.proof_tree, r2.proof_tree, n1, n2);

  deduce_new_equalities_with_inv_functions(r1, r2);
}
```

Listing 5.15: The pseudo-code of the congruence closure algorithm modulo associativity, commutativity, and inverse functions.

It is important to note that we do not generate complete models with this algorithm, since the time needed to generate all the instances of the associative and commutative law or of inverse function instances for a complete model has a negative impact on the overall performance of the algorithm.

5.4.2 Example

Listing 5.16 shows the example we will use to illustrate how the congruence closure algorithm modulo associativity, commutativity and inverse functions works. The formula consists of the associative and commutative uninterpreted function `f` and the two unary functions `g` and `h`, which are inverse to each other. The following three equations $f(a,b) = c$, $f(f(a,a), f(b,b)) = d$, $f(g(h(h(g(c))))), c) = e$ as well as the inequation $d \neq c$ are asserted. During the creation of the of the `enodes`, in this example, the following two equalities were learned, $h(g(c)) = c$ and $g(h(h(g(c)))) = h(g(c))$. Both equations were learned via the inverse function application rule. At the beginning of the procedure we have the following congruence classes: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{f(a,b)}\}$, $\{\underline{c}\}$, $\{\underline{f(a,a)}\}$, $\{\underline{f(b,b)}\}$, $\{\underline{f(f(a,a), f(b,b))}\}$, $\{\underline{d}\}$, $\{\underline{g(c)}\}$, $\{\underline{h(g(c))}\}$, $\{\underline{h(h(g(c)))}\}$, $\{\underline{g(h(h(g(c))))}\}$, $\{\underline{f(g(h(h(g(c)))), c)}\}$, $\{\underline{e}\}$.

```
(set-logic QF_ACIUF)
(set-option :produce-proofs true)

(declare-sort I)

(declare-fun f (I I) I :ac)
(declare-fun g (I) I)
(declare-fun h (I) I :inv g)
(declare-fun a () I)
(declare-fun b () I)
(declare-fun c () I)
(declare-fun d () I)
(declare-fun e () I)

(assert
  (and
    (= (f a b) c)
    (= (f (f a a) (f b b)) d)
    (= (f (g (h (h (g c)))) c) e)
    (not (= d e))
  )
)

(check-sat)
(get-proof)
(exit)
```

Listing 5.16: An example containing associative commutative functions as well as inverse function applications. This example is unsatisfiable.

The first equation that gets added by the procedure is $f(a,b) = c$. Since the theory of uninterpreted functions with associative and commutative functions is used, a reduction ordering is used to decide which congruence class gets merged into the other. The reduction ordering chooses to merge the congruence class of $f(a,b)$ into the congruence class of c. The resulting four `enodes` are $n_1 = r_1 = f(a,b)$ and $n_2 = r_2 = c$, while the generated rule is $r_1 \rightarrow r_2$, which can be used to simplify the term $f(f(a,a), f(b,b))$ to use $\{c, c\}$ as current arguments. The merge process does not lead to the discovery of further equations. The congruence classes after the merge have the following elements, $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}, f(a,b)\}$, $\{\underline{f(a,a)}\}$, $\{\underline{f(b,b)}\}$, $\{\underline{f(f(a,a), f(b,b))}\}$, $\{\underline{d}\}$, $\{\underline{g(c)}\}$, $\{\underline{h(g(c))}\}$, $\{\underline{h(h(g(c)))}\}$,

$\{\underline{g(h(h(g(c))))}\}$, $\{\underline{f(g(h(h(g(c)))), c)}\}$, $\{\underline{e}\}$. The proof trees for the congruence classes are visualized in Figure 5.19.

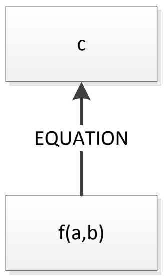

Figure 5.19: The proof trees for the congruence classes after the equation $f(a, b) = c$ has been added to the context by the congruence closure algorithm modulo associativity, commutativity and inverse functions.

$f(f(a, a), f(b, b)) = d$ is the topmost equation from the equalities to propagate queue and thus gets added to the context. Here the reduction ordering decides that the congruence class of $f(f(a, a), f(b, b))$ should be merged into the congruence class of d. During the merge process no rules can be changed and no new equations are learned. The congruence classes after the merging contain the following elements: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}, f(a, b)\}$, $\{\underline{f(a, a)}\}$, $\{\underline{f(b, b)}\}$, $\{\underline{d}, f(f(a, a), f(b, b))\}$, $\{\underline{g(c)}\}$, $\{\underline{h(g(c))}\}$, $\{\underline{h(h(g(c)))}\}$, $\{\underline{g(h(h(g(c))))}\}$, $\{\underline{f(g(h(h(g(c)))), c)}\}$, $\{\underline{e}\}$. Figure 5.20 shows the corresponding proof trees for the congruence classes.

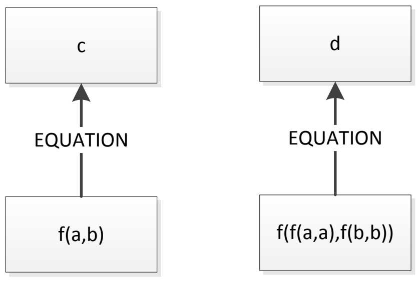

Figure 5.20: The proof trees for the congruence classes after the equation $f(f(a, a), f(b, b)) = d$ has been added to the context by the congruence closure algorithm modulo associativity, commutativity and inverse functions.

The last equation asserted in the formula that gets added to the context is $f(g(h(h(g(c)))), c) = e$. Here the congruence class of $f(g(h(h(g(c)))), c)$ gets merged into the congruence class of e. During the merging process no rules are changed and no new equations are learned. The congruence classes after the merging are $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}, f(a, b)\}$, $\{\underline{f(a, a)}\}$, $\{\underline{f(b, b)}\}$, $\{\underline{d}, f(f(a, a), f(b, b))\}$, $\{\underline{g(c)}\}$, $\{\underline{h(g(c))}\}$, $\{\underline{h(h(g(c)))}\}$, $\{\underline{g(h(h(g(c))))}\}$, $\{\underline{f(g(h(h(g(c)))), c)}\}$, $\{\underline{e}, f(g(h(h(g(c)))), c)\}$ with their corresponding proof trees in Figure 5.21.

After all equations from the original formula have been added to the context, newly learned equations are added to the context. The first equation learned was $h(g(c)) = c$. Here the congruence class of $h(g(c))$ is merged into the congruence class of c. The rule created for this equality does not change the current rules and also does not allow to learn any new equations. The congruence classes after the merge have the following elements: $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}, f(a, b), h(g(c))\}$, $\{\underline{f(a, a)}\}$, $\{\underline{f(b, b)}\}$, $\{\underline{d},$

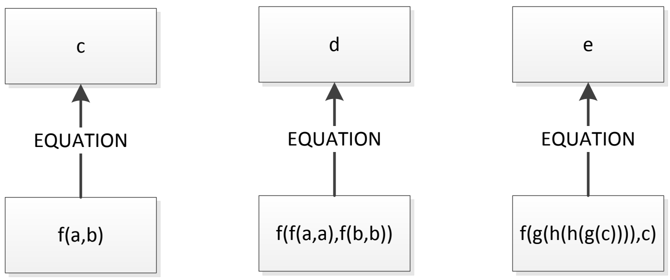

Figure 5.21: The proof trees for the congruence classes after the equation $f(g(h(h(g(c)))), c) = e$ has been added to the context by the congruence closure algorithm modulo associativity, commutativity and inverse functions.

$f(f(a,a), f(b,b))\}$, $\{\underline{g(c)}\}$, $\{\underline{h(h(g(c)))}\}$, $\{\underline{g(h(h(g(c))))}\}$, $\{\underline{f(g(h(h(g(c)))), c)}\}$, $\{\underline{e}, f(g(h(h(g(c)))), c)\}$, with the proof trees displayed in Figure 5.22.

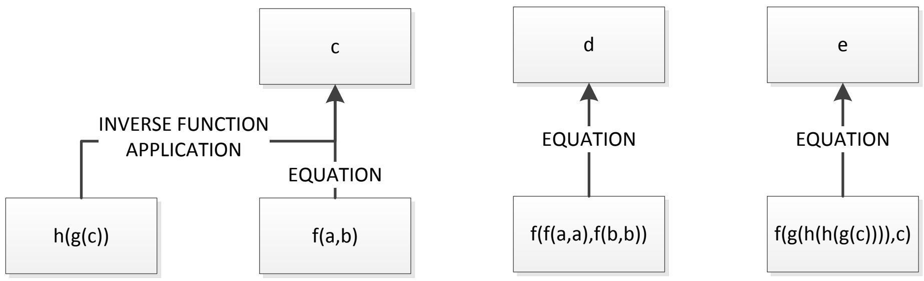

Figure 5.22: The proof trees for the congruence classes after the equation $h(g(c)) = c$ has been added to the context by the congruence closure algorithm modulo associativity, commutativity and inverse functions.

The next equation that was learned during the creation of the `enodes` was $g(h(h(g(c)))) = h(g(c))$. The four `enodes` for this equation are $n_1 = r_1 = g(h(h(g(c))))$, $n_2 = h(g(c))$, and $r_2 = c$. During the merge process the following congruence is learned $f(f(a,a), f(b,b)) = f(g(h(h(g(c)))), c)$, which also allows to collapse the first learned rule $f(f(a,a), f(b,b)) \rightarrow d$ by the rule $f(g(h(h(g(c)))), c) \rightarrow e$ since both rules use the current arguments $\{c,\ c\}$ and thus are equal under associativity and commutativity for the current context. The first rule becomes $d \rightarrow e$, due to the hash values which are used in the comparison of d and e. In addition, the following equation can be deduced using the inverse function deduction rule: $h(c) = h(h(g(c)))$. The congruence classes after the merging are, $\{\underline{a}\}$, $\{\underline{b}\}$, $\{\underline{c}$, $f(a,b)$, $h(g(c))$, $g(h(h(g(c))))\}$, $\{\underline{f(a,a)}\}$, $\{\underline{f(b,b)}\}$, $\{\underline{d}$, $f(f(a,a), f(b,b))\}$, $\{\underline{g(c)}\}$, $\{\underline{h(h(g(c)))}\}$, $\{\underline{f(g(h(h(g(c)))), c)}\}$, $\{\underline{e}$, $f(g(h(h(g(c)))), c)\}$. Figure 5.23 shows the proof trees for the congruence classes.

The last equation added to the context is the learned congruence of $f(f(a,a), f(b,b))$ and $f(g(h(h(g(c)))), c)$, with the *root* elements d and e. Here the congruence class of d is merged into the congruence class of e. However, this makes the current context unsatisfiable, which causes the procedure to stop. The proof of unsatisfiability provided by the Z3 theorem prover can be found in Listing 5.17. The proof was generated using the proof tree shown in Figure 5.24. In the construction of the proof both the rules for associative and commutative uninterpreted functions as well as for inverse uninterpreted functions were used. The proof shows that the given set of equations causes the terms $f(f(a,a), f(b,b))$ and $f(g(h(h(g(c)))), c)$

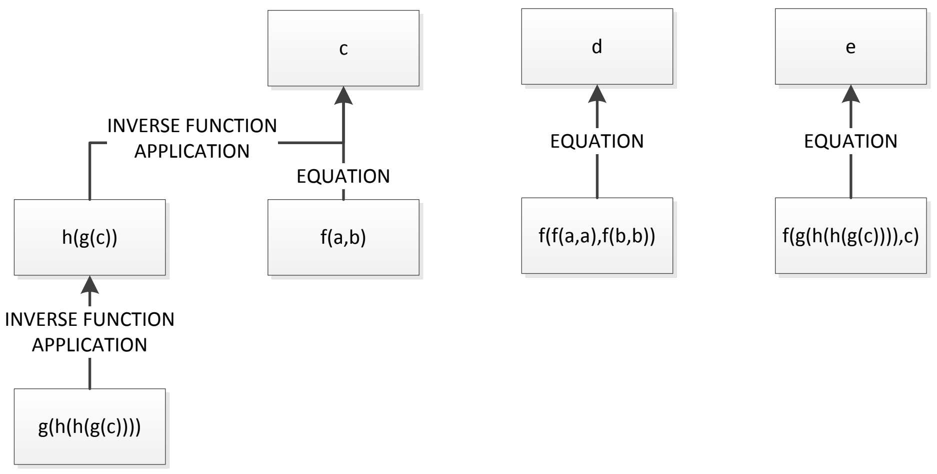

Figure 5.23: The proof trees for the congruence classes after the equation $g(h(h(g(c)))) = h(g(c))$ has been added to the context by the congruence closure algorithm modulo associativity, commutativity and inverse functions.

to be equal, which in turn leads to the equality of d and e. However, this is a contradiction to the asserted inequality of d and e.

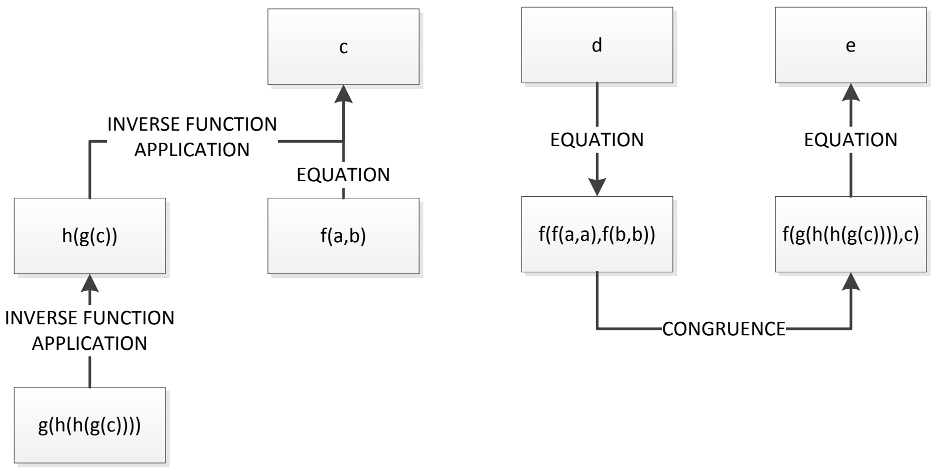

Figure 5.24: The proof trees for the congruence classes after the equation $f(f(a,a),f(b,b)) = f(g(h(h(g(c)))),c)$ has been added to the context by the congruence closure algorithm modulo associativity, commutativity and inverse functions.

```
((set-logic QF_ACIUF)
(proof
(let (($x22 (= d e)))
(let ((?x15 (g c)))
(let ((?x16 (h ?x15)))
(let ((?x17 (h ?x16)))
(let ((?x18 (g ?x17)))
(let ((?x19 (f ?x18 c)))
(let (($x21 (= ?x19 e)))
(let (($x23 (not $x22)))
(let ((?x11 (f b b)))
(let ((?x10 (f a a)))
(let ((?x12 (f ?x10 ?x11)))
(let (($x14 (= ?x12 d)))
(let ((?x7 (f a b)))
(let (($x9 (= ?x7 c)))
(let (($x24 (and $x9 $x14 $x21 $x23)))
(let ((@x45 (asserted $x24)))
(let ((@x50 (|and-elim| @x45 $x21)))
(let ((@x56 (trans (|inverse function application| (= ?x18 ?x16)) (|
    inverse func
tion application| (= ?x16 c)) (= ?x18 c))))
(let ((@x65 (symm (monotonicity @x56 (= ?x19 (f c c))) (= (f c c) ?x19)))
    )
(let ((@x67 (trans (monotonicity (|and-elim| @x45 $x9) (= ?x12 (f c c)))
    @x65 (=
 ?x12 ?x19))))
(let ((@x70 (trans (trans (symm (|and-elim| @x45 $x14) (= d ?x12)) @x67
    (= d ?x1
9)) @x50 $x22)))
(let ((@x51 (|and-elim| @x45 $x23)))
(|unit-resolution| @x51 @x70 false))))))))))))))))))))))
```

Listing 5.17: The corresponding proof of unsatisfiability for the example with both associative commutative and inverse functions.

Chapter 6

Evaluation

In the previous chapter we have presented the implementation of the algorithms for congruence closure modulo associativity and commutativity, congruence closure modulo inverse functions, and the combination of both. This chapter contains the evaluation of the performance of these algorithms. The first part of the chapter focuses on the performance of the associative commutative congruence closure algorithm. The second part of this evaluation deals with the congruence closure modulo inverse functions. The last part of the chapter concentrates on the performance of the combined theory of associativity and commutativity with inverse functions.

All experiments in this chapter were performed on an Intel Core i7 CPU with 3.5 GHz and 16 GB RAM. The runtimes were determined by averaging over 10 runs that were conducted for each test-file. For all test-files a timeout of 30 minutes was used.

6.1 Congruence Closure Modulo Associativity and Commutativity

In order to evaluate the performance of the congruence closure algorithm modulo associativity and commutativity we have decided to compare our algorithm against Z3 with associativity and commutativity as axioms.

The axioms for associativity and commutativity in SMT-LIBv2 format are given in Listing 6.1:

```
(forall ((x I) (y I)) (= (f x y) (f y  x)))
(forall ((x I) (y I) (z I)) (= (f x (f y z)) (f (f x y) z)))
```

Listing 6.1: The axioms for associativity and commutativity in SMT-LIBv2 format.

Here f is the function that should have the associative and commutative property and I is the sort type of our terms in the example. The first line states that the functions is commutative, i.e., the order of the arguments does not matter, and the second line states that the associative property, i.e., the bracketing also does not have any influence on the semantics of a given expression.

Table 6.1 shows the average run time of the congruence closure algorithm modulo associativity and commutativity and the congruence closure algorithm with axioms for examples containing associative commutative uninterpreted functions. An average runtime of TIMEOUT means that all 10 runs that were conducted timed out for the example.

All examples named qf_acuf_XX.smt2 are our own test files while all files named test__eqs-XX__depth-YY.smt2 are from [12], and used with their friendly permission. All files with the name pattern test__eqs-XX__depth-YY_U.smt2 are just the negated versions of the file with the same name pattern. Since all examples from [12] represent valid formulas, their originals are satisfiable and the corresponding negated version (suffix _U) are unsatisfiable.

Testfilename	Runtime CC mod AC	Runtime Z3 with Axioms	verdict
qf_acuf_01.smt2	13.3	13.2	unsat
qf_acuf_02.smt2	13.4	12.7	unsat
qf_acuf_03.smt2	13.7	13	unsat
qf_acuf_04.smt2	23.5	TIMEOUT	sat
qf_acuf_05.smt2	13.4	12.8	unsat
qf_acuf_06.smt2	14.1	107.6	sat
qf_acuf_07.smt2	14.2	TIMEOUT	sat
qf_acuf_08.smt2	13.9	13.8	unsat
qf_acuf_09.smt2	72.6	TIMEOUT	sat
qf_acuf_10.smt2	14.2	12.8	unsat
qf_acuf_11.smt2	14.3	13.7	unsat
qf_acuf_12.smt2	14	14.4	unsat
qf_acuf_13.smt2	21.3	20567	sat
qf_acuf_14.smt2	13.4	13.1	unsat
qf_acuf_15.smt2	13.6	12.5	unsat
qf_acuf_16.smt2	52.3	TIMEOUT	sat
qf_acuf_17.smt2	13.9	12.3	unsat
qf_acuf_18.smt2	14.9	12.6	unsat
qf_acuf_19.smt2	15.1	OUT OF MEMORY	sat
qf_acuf_20.smt2	17.6	TIMEOUT	sat
qf_acuf_21.smt2	14.3	14.3	sat
qf_acuf_22.smt2	14	12.5	unsat
qf_acuf_23.smt2	13.8	12.6	unsat
qf_acuf_24.smt2	13.5	13.4	unsat
qf_acuf_25.smt2	13.9	13	unsat
qf_acuf_26.smt2	13.8	12.6	unsat
qf_acuf_27.smt2	14	12.7	unsat
test__eqs-12__depth-3.smt2	27.5	167984.4	sat
test__eqs-12__depth-3_U.smt2	33657.5	224.2	unsat
test__eqs-3__depth-12.smt2	40	OUT OF MEMORY	sat
test__eqs-3__depth-12_U.smt2	39.4	OUT OF MEMORY	unsat
test__eqs-3__depth-3.smt2	20.5	TIMEOUT	sat
test__eqs-3__depth-3_U.smt2	20.1	24.8	unsat
test__eqs-3__depth-6.smt2	25.5	TIMEOUT	sat
test__eqs-3__depth-6_U.smt2	25.2	9304.8	unsat
test__eqs-6__depth-12.smt2	26.2	OUT OF MEMORY	sat
test__eqs-6__depth-12_U.smt2	1613.2	OUT OF MEMORY	unsat
test__eqs-6__depth-3.smt2	18.5	59.1	sat
test__eqs-6__depth-3_U.smt2	449.9	62.6	unsat
test__eqs-6__depth-6.smt2	20.4	TIMEOUT	sat
test__eqs-6__depth-6_U.smt2	762.1	49075.8	unsat

Table 6.1: The average runtime of the associative commutative congruence closure compared with Z3 with axioms. All runtimes are in ms and were averaged over 10 runs.

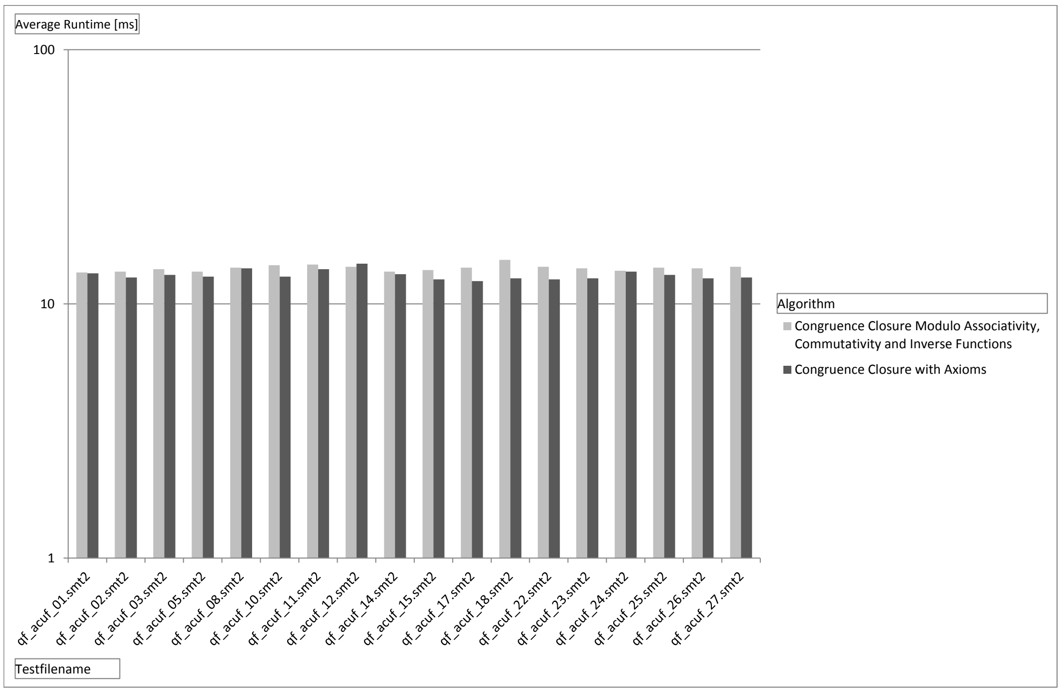

Figure 6.1: The comparison between the congruence closure algorithm modulo associativity and commutativity and the congruence closure algorithm where the associativity and commutativity are axiomatized. All examples contained in this graphic are unsatisfiable.

When comparing the runtime of the congruence closure modulo associativity and commutativity with Z3 using axioms we see that the strength of the dedicated congruence closure algorithm lies in the solving of examples that are satisfiable. In the following paragraphs we will analyse the performance for different groups of examples in more detail.

Figure 6.1 compares the standard congruence closure algorithm, where the associative and commutative property of functions has been added via axioms, with the congruence closure algorithm modulo associativity and commutativity described in Section 5.2. This plot only contains examples that are unsatisfiable out of the group qf_acuf_XX.smt2. As we can see both algorithms are able to handle these examples quite well. The reason for this result is the model based quantifier instantiation (MBQI) done by Z3 for the axioms, which is quickly able to identify the needed instances of the associative and commutative law to show that the given formula is unsatisfiable. MBQI is a technique used by Z3 when solving formulas containing quantifiers. The MBQI is used to decide which instances of quantifiers should be generated first, the decision is based on the currently build model inside the Z3 theorem prover, for more details about the MBQI we refer to [22].

In Figure 6.2 the comparison between the two algorithms is shown for unsatisfiable formulas that have a more complex structure, the examples out of the group test__eqs-XX__depth-YY_U.smt2. Here we can see that for two test files (test__eqs-12__depth-3_U.smt2 and test__eqs-6__depth-3_U.smt2) our implementation takes longer to find the conflict than the model based quantifier instantiation of the Z3 solver with standard congruence closure. But for all other examples our approach is either as fast as the model based quantifier instantiation or faster. For two examples (test__eqs-3__depth-12_U.smt2 and test__eqs-6__depth-12_U.smt2) the Z3 solver with standard congruence closure and axioms did not find a solution, because the solver ran out of memory during the solving process.

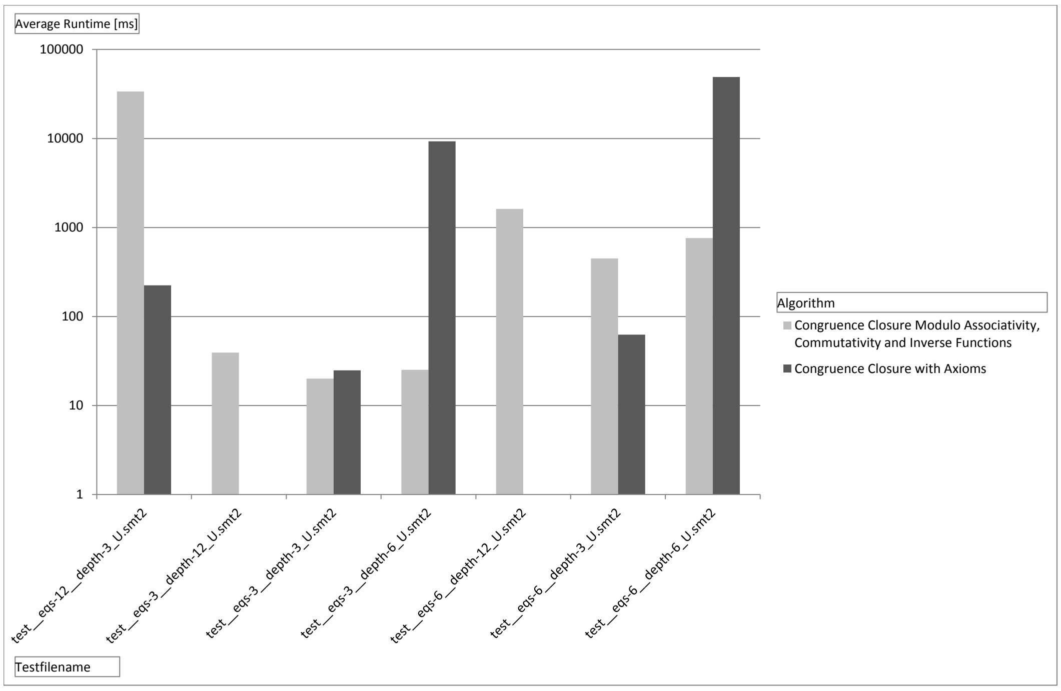

Figure 6.2: Comparison between congruence closure modulo associativity and commutativity and congruence closure with axioms for unsatisfiable examples from Conchon et al.

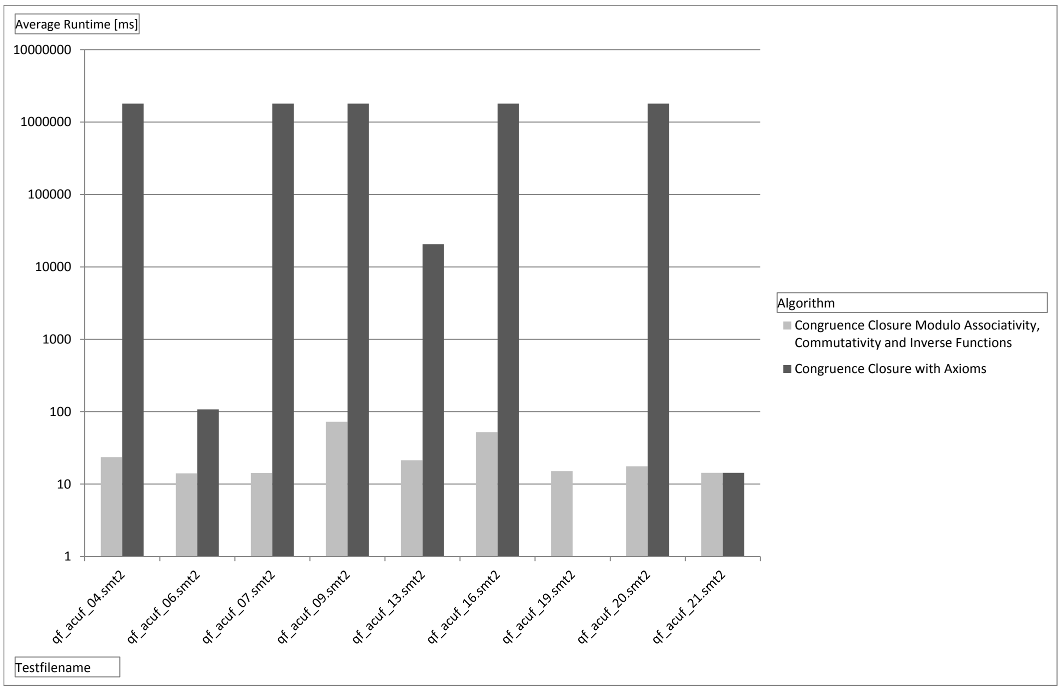

Figure 6.3: The runtime comparison for the two congruence closure algorithms on satisfiable examples.

The picture, however, drastically changes when we compare the performance of the two algorithms on satisfiable examples. For the nine examples, out of the group qf_acuf_XX.smt2, that are satisfiable the standard congruence closure is only able to calculate the result for three of them, while our implementation is able to solve all nine of them, where the slowest example has an average runtime of 72.6ms. For five examples Z3 runs into a timeout using axioms and for one example the solver runs out of memory during the solving process. This really shows the improvement that is achievable when integrating associativity and commutativity inside the congruence closure algorithm. The problem here for the standard algorithm with MBQI is, that the MBQI does not help solving satisfiable examples, since the order in which the instances of the quantifiers are generated does not lower the number of instances that have to be generated. Figure 6.3 shows the runtime comparison for the satisfiable examples out of the group qf_acuf_XX.smt2.

The same results can be observed when we evaluate the implementation using the examples from [12]. The standard congruence closure algorithm with axioms is able to calculate the result only for two of the seven benchmark files. For three files the timeout is reached before the algorithm was finished and for two files the Z3 solver ran out of memory before the process was finished. The runtime comparison for these examples can be seen in Figure 6.4.

6.2 Congruence Closure Modulo Inverse Functions

In this section we take a look at the performance of the congruence closure algorithm modulo inverse functions. Like in the previous section, we will benchmark our implementation against the Z3 solver using the standard congruence closure algorithm with axioms for the inverse function property. Listing

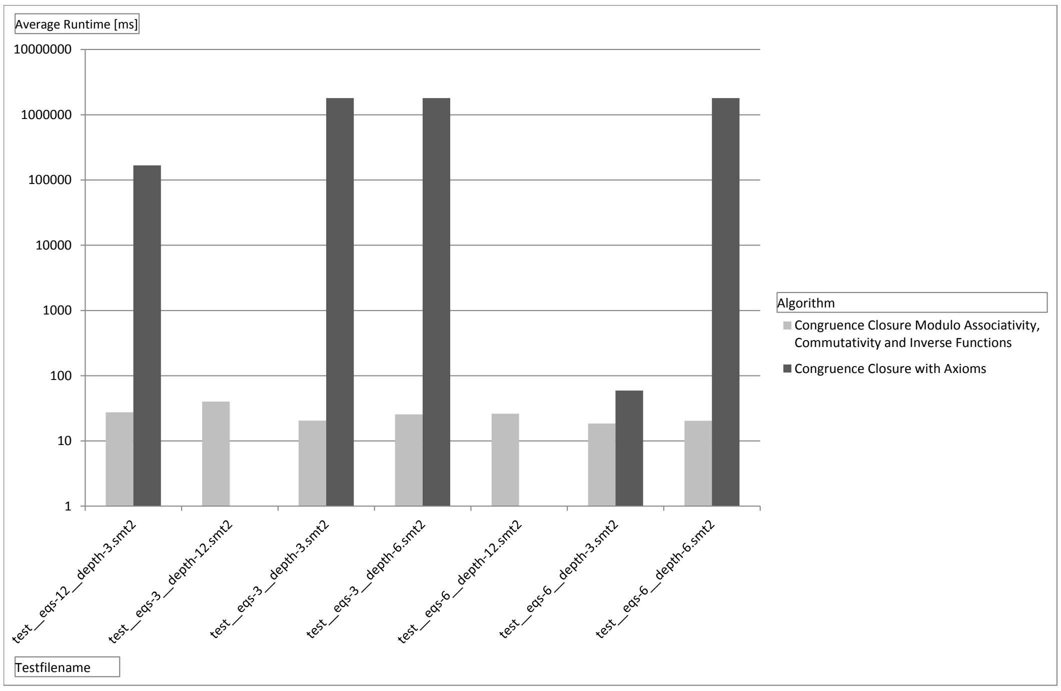

Figure 6.4: The runtime comparison between the congruence closure algorithm modulo associativity and commutativity and the congruence closure algorithm with axioms using the satisfiable examples from Conchon et al.

Testfilename	Runtime for CC Modulo Inverse Functions in ms	Runtime for CC with Axioms in ms	Verdict
`qf_iuf_01.smt2`	16.6	12.4	unsat
`qf_iuf_02.smt2`	13.3	12.1	unsat
`qf_iuf_03.smt2`	13.6	12.4	unsat
`qf_iuf_04.smt2`	14	TIMEOUT	sat
`qf_iuf_05.smt2`	15.2	TIMEOUT	sat
`qf_iuf_06.smt2`	13.2	12.4	unsat

Table 6.2: The runtime comparison between the standard congruence closure with axioms and congruence closure modulo inverse functions.

6.2 shows the axioms for inverse functions, where $f = g^{-1}$.

```
(forall ((x Int)) (= f(g(x)) x))
(forall ((x Int)) (= g(f(x)) x))
```

Listing 6.2: The axioms for inverse functions in the SMT-LIB v2 format.

Table 6.2 shows the average runtimes for the congruence closure algorithm modulo inverse functions and the congruence closure algorithm with axioms for test-files containing uninterpreted inverse functions.

Figure 6.5 shows the runtime comparison for the six examples with inverse functions. Two of the six examples were actually satisfiable. For both of them the congruence closure algorithm with axioms for the inverse functions was not able to calculate a solution, due to running into the timeout of 30 minutes. For the other four examples both algorithms were able to calculate the result in a similar time period. This again shows that the quantifier based model instantiation inside Z3 is able to quickly identify the needed instances of the axioms in order to arrive at a contradiction, but struggles with examples that are satisfiable.

6.3 Congruence Closure Modulo Associativity, Commutativity and Inverse Functions

In the previous two sections we looked at the performance of the congruence closure algorithm modulo associativity and commutativity and the congruence closure algorithm modulo inverse function against the standard algorithm, where the additional information has to be added as an axiom. We found out that especially in the satisfiable case the extended algorithms achieve a much better performance than the congruence closure algorithm with axioms. In this section we will evaluate the combination we described in Section 5.4.

The axioms for associativity and commutativity can be found in Listing 6.1 and the axioms for inverse functions are shown in Listing 6.2. Table 6.3 shows the average runtimes for the congruence closure algorithm modulo associativity, commutativity and inverse functions and the congruence closure algorithm with axioms for examples containing both uninterpreted functions which are associative and commutative and inverse uninterpreted functions.

As we can see, the trend regarding the results from the previous two evaluations continuous with the congruence closure modulo associativity, commutativity and inverse functions. Both the congruence closure algorithm modulo associativity, commutativity and inverse functions and the congruence closure algorithm with axioms are able to solve all unsatisfiable examples. Here there do not exist any significant

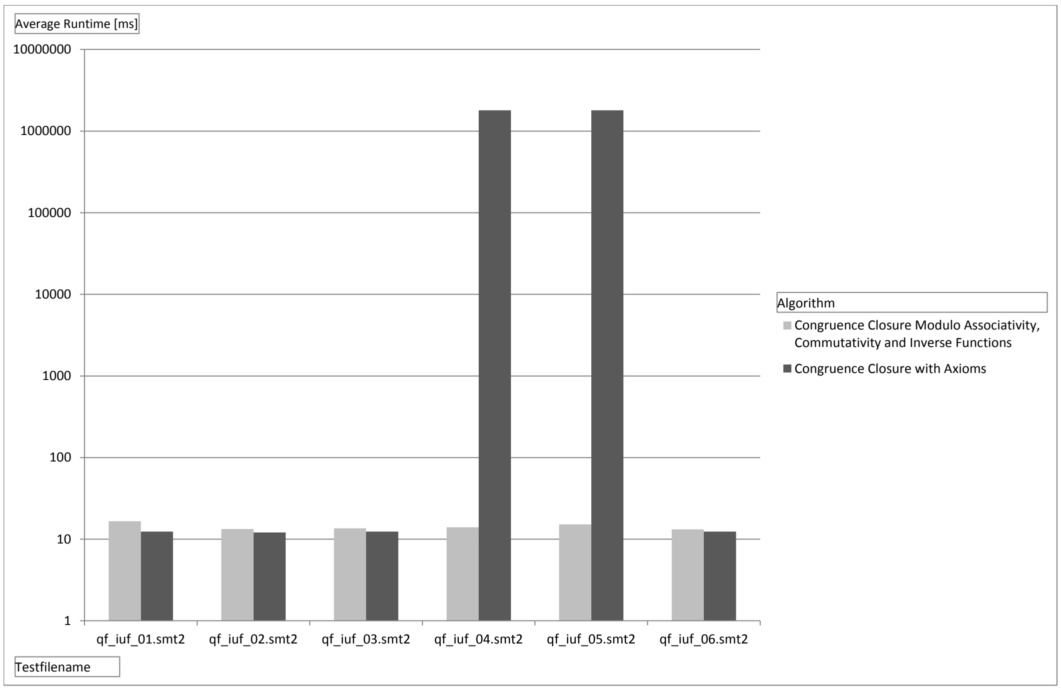

Figure 6.5: The runtime comparison between congruence closure modulo inverse functions and the congruence closure with axioms.

Testfilename	Runtime for CC Modulo Associativity, Commutativity and Inverse Functions in ms	Runtime for CC with Axioms in ms	Verdict
qf_aciuf_01.smt2	13.3	12.6	unsat
qf_aciuf_02.smt2	14.2	12.7	unsat
qf_aciuf_03.smt2	14	13	unsat
qf_aciuf_04.smt2	14	12.8	unsat
qf_aciuf_05.smt2	14.5	12.7	unsat
qf_aciuf_06.smt2	15.4	12.7	unsat
qf_aciuf_07.smt2	16.3	TIMEOUT	sat
qf_aciuf_08.smt2	14.7	OUT OF MEMORY	sat
qf_aciuf_09.smt2	41.5	OUT OF MEMORY	sat
qf_aciuf_10.smt2	20.3	OUT OF MEMORY	sat

Table 6.3: The average runtimes for the combined theory of associativity, commutativity and inverse functions.

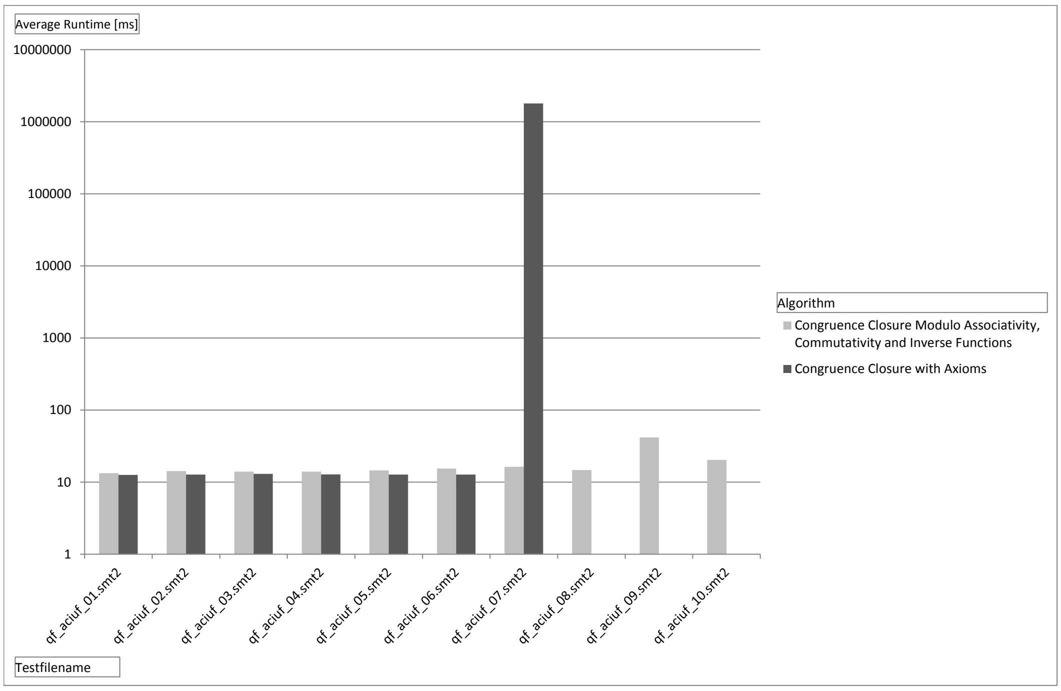

Figure 6.6: Shows the average runtimes for the congruence closure algorithm modulo associativity, commutative and inverse functions and the congruence closure algorithm with axioms.

differences in the runtime. This can easily be seen in Figure 6.6, which illustrates the runtime comparison between the two algorithms. However, if we consider the examples that are satisfiable, the congruence closure algorithm with axioms is not capable of calculating the result for a single one. From the four examples that were actually satisfiable the congruence closure algorithm with axioms runs into the time-out once and exceeds the memory limit in the other three cases. This again shows that the model based quantifier instantiation is able to handle unsatisfiable formulas but struggles with satisfiable examples.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this work we presented our implementation of the congruence closure algorithm modulo associativity and commutativity [4, 12, 30] inside the Z3 theorem prover [16]. Although the theory of congruence closure modulo associativity and commutativity is not new, only the Alt-Ergo theorem prover has built in support for it. We also presented an algorithm for congruence closure modulo inverse functions, as well as the implementation of the algorithm, something that was not done before to our knowledge. Further we proposed a way how the two algorithms can be combined to get a congruence closure algorithm modulo associativity, commutativity, and inverse functions. Finally, we presented the experimental results of our implementation.

7.1.1 Congruence Closure Modulo Associativity and Commutativity

Although some work was done in the field of congruence closure modulo associativity and commutativity [12, 4, 30] before only the Alt-ergo theorem prover has built-in support for the theory of uninterpreted functions with associativity and commutativity [6]. We extended the congruence closure algorithm inside Z3 theorem prover to handle these axioms directly in the congruence closure algorihtm rather than adding them as additional axioms to the formula. As the results of our evaluation have shown, dedicated congruence closure algorithms outperform the standard congruence closure algorithm with axioms. This can easily be seen for satisfiable examples, where the Z3 theorem prover struggles to find models for the formulas, when using axioms for associativity and commutativity. For the case of unsatisfiable examples both the dedicated algorithm and the standard algorithm have nearly the same performance.

The key difference between the standard congruence closure algorithm and the congruence closure algorithm modulo associativity and commutativity is the use of a dedicated reduction ordering, which must be capable of handling associative and commutative functions. Such a reduction ordering was introduced in [40]. Another key difference is the matching of subterms during the algorithm. In the presence of associative and commutative functions this matching becomes more difficult. Since one has to match terms considering the associative and commutative property of the functions symbols, for this purpose we consider the flattened form of associative and commutative functions during this matching process. A very important step in the completion procedure for congruence closure modulo associativity and commutativity is the deduction step. The equalities learned during this step are very important for the correctness of the algorithm, as these equalities cover critical pairs.

7.1.2 Congruence Closure Modulo Inverse Functions

In Section 4.3 we introduced a congruence closure algorithm modulo inverse functions, something that was not done before to our knowledge. Like the congruence closure algorithm modulo associativity and commutativity, the congruence closure algorithm modulo inverse functions is an extension of the standard congruence closure algorithm. In order to handle inverse functions directly in the congruence closure algorithm we added two rules to the completion procedure. One rule allows the completion procedure to remove the application of inverse functions, i.e. $g(f(x)) = x$ if $g = f^{-1}$. The other rule is a dedicated deduction rule for inverse functions.

As in the associative and commutative case, the experimental results of our work show that the extension of the congruence closure algorithm allows to solve more examples compared to using axioms to cover the additional properties of the uninterpreted functions. Again, we were able to solve examples that are satisfiable with our algorithm whereas the standard algorithm with axioms struggles to solve such formulas.

7.1.3 Congruence Closure Modulo Associativity, Commutativity and Inverse Functions

We also presented a way to combine the theories of equality with uninterpreted functions modulo associativity and commutativity and equality with uninterpreted functions and inverse functions. An important thing to note about the combination is that the two theories are disjoint, since associativity and commutativity are defined for binary functions, while inverse functions are unary. This means that there are no functions that are associative, commutative and an inverse of another function at the same time. Thus, the completion procedure for the combined theory is an extension of the completion procedure for the associative and commutative case. As with the completion procedure for congruence closure modulo inverse functions, the completion procedure for congruence closure modulo associativity and commutativity was extended with the two rules needed to handle inverse functions.

The experiments again show that the dedicated congruence closure algorithm for the two combined theories beats the standard congruence closure algorithm with axioms, especially in the case of satisfiable examples. Again, this result can be explained by the fact that the dedicated algorithm only deduces critical pairs during the deduction step. Whereas the standard algorithm with axioms tries to build a complete model for which all the possible instances of the associative and commutative law as well as all applications of inverse functions have to be created.

7.2 Discussion

7.2.1 Evaluation

Our evaluation has shown that the strength of the dedicated congruence closure algorithms lies in the solving process for satisfiable formulas. A considerable performance increase is brought by the fact that the dedicated procedures only generate critical pairs instead of all instances of the laws for associativity, commutativity, and inverse functions. This is evident in the fact that already small satisfiable examples are unsolvable with axioms due to the high number of instances generated from the axioms. However, when considering unsatisfiable examples no performance increase was achieved. This shows that the model based quantifier instantiation done by Z3 for these formulas performs very good at detecting the instances of the mathematical laws that are needed to proof unsatisfiability of the given formula.

The evaluation in our opinion clearly shows that dedicated congruence closure algorithms for associativity, commutativity and inverse functions are needed. Without explicit support, the solving of satisfiable formulas is not feasible, either because of the long runtime or the fact that the solver runs out of memory while generating instances of the mathematical laws.

7.2.2 Proofs of Unsatisfiability and Models

Two important things in the context of SMT are the proof of unsatisfiability and the model for satisfiable examples. The proof of unsatisfiability shows the deduction rules required to infer the conflict from the formula. In the context of congruence closure modulo associativity and commutativity, two new proof rules were added to the Z3 theorem prover. The two rules are used for equality modulo associativity and commutativity and for equalities that are results of the associative and commutative deduction step in the completion procedure. In the case of inverse functions, another two rules were added to the proof framework. Again, these rules govern the equality of terms regarding inverse functions and the equalities learned via inverse function deduction steps. Our algorithm is capable of producing proofs of unsatisfiability for formulas containing associative, commutative and inverse uninterpreted functions.

As we have stated in the description of our implemented algorithms (Chapter 5) we have not implemented a complete model generation. Instead of all possible instances, only the needed instances of the function are in the model. This decision was made since the generation of a complete model containing all the instances would lead to a significant increase of the runtime and thus is simply not desirable.

If the need for a complete model arises this can easily be implement inside the SMT solver or the partial model taken from the SMT solver can be extended by the application to fill out the gaps in the model. However, from our point of view a model containing all the critical pairs and the instances present in the formula is enough for the most applications regarding the theory of uninterpreted functions modulo associativity, commutativity and inverse functions.

7.3 Future Work

Although we have defined a useful framework for handling uninterpreted functions with associativity, commutativity and inverse functions, there is still a lot of work to be done.

7.3.1 Evaluation

We presented a first evaluation of the algorithms described in Chapter 5 and this evaluation showed the performance gain that can be achieved when using a dedicated congruence closure over the standard congruence closure algorithm with axioms for additional properties. During our evaluation we used our own examples as well as examples provided by Conchon et al. [12]. However, we think it would be beneficial to have a common, publicly available pool of examples for the theory of uninterpreted functions with associativity, commutativity and inverse functions.

7.3.2 Combination with Model Based Quantifier Instantiation (MBQI)

Our preliminary evaluation showed that the model based quantifier instantiation might not be able to deal well with satisfiable examples, but handles unsatisfiable examples really well. Thus, it might be possible to use the MBQI to enhance the deduction steps of the dedicated algorithms. The same heuristic the MBQI uses to decide which instances of the axioms should be instantiated could be used to decide which of the possible deduction steps should be carried out by the dedicated algorithm. This could yield in a performance increase for the dedicated algorithms in the case of unsatisfiable examples. However, this does not change the performance of the dedicated algorithms in the case of satisfiable examples, since here all of the deduction steps have to be carried out anyway and the order of them does not matter.

7.3.3 Combination with other Theories

A further interesting topic is the combination of the theory of uninterpreted functions modulo associativity, commutativity and inverse functions with other powerful theories, like linear integer arithmetic. This combination would allow to solve more complicated problems, as well as the development of new encodings for existing problems, due to the availability of a dedicated logic.

The list of potentially useful combinations is of course long, thus we are only going to pick a few representative logics. For example, the combination with the theory of arrays would allow to use arrays in formulas containing associative, commutative and inverse uninterpreted functions. Another interesting theory might be the theory of linear integer arithmetic. Moreover, the addition of quantifiers could help to increase the number of problems that can be solved.

Besides the combination of the presented theories with others, the addition of more extensions to the congruence closure algorithm is worth considering. For example, the theory of uninterpreted functions modulo idempotent functions or a combination of associative, commutative and idempotent uninterpreted functions could be very interesting. Another extension where there already exists a theoretical foundation would be the completion modulo associativity, commutativity and identity [27, 28]. In this extension, the uninterpreted functions have neutral elements, an element e is a neutral element if the following property holds $\forall a. f(e, a) = a$, which can be defined in the formula.

Finally, we can conclude that dedicated congruence closure algorithms are needed to efficiently handle the theories of uninterpreted functions with equality modulo associativity and commutativity, uninterpreted functions modulo inverse functions and the combination of the two. In this work we presented our implementation of the congruence closure modulo associativity and commutativity algorithm, as well as introduced a congruence closure modulo inverse functions algorithm. Furthermore, we proposed a way how the two algorithms can be combined and we presented the results of our evaluation of the various algorithms, but there is still a lot of work to be done in the future.

Bibliography

[1] Keith Andrews. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. Oct. 22, 2012. http://ftp.iicm.edu/pub/keith/thesis/.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. New York, NY, USA: Cambridge University Press, 1998. ISBN 0-521-45520-0.

[3] Leo Bachmair and Ashish Tiwari. "Abstract Congruence Closure and Specializations". In: *Automated Deduction - CADE-17*. Ed. by David McAllester. Vol. 1831. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 64–78. ISBN 978-3-540-67664-5. doi:10.1007/10721959_5. http://dx.doi.org/10.1007/10721959_5.

[4] L. Bachmair et al. "Congruence Closure Modulo Associativity and Commutativity". In: *Frontiers of Combining Systems*. Ed. by Hélène Kirchner and Christophe Ringeissen. Vol. 1794. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 245–259. ISBN 978-3-540-67281-4. doi:10.1007/10720084_16. http://dx.doi.org/10.1007/10720084_16.

[5] J.C.M. Baeten and W.P. Weijland. "Semantics for Prolog via term rewrite systems". In: *Conditional Term Rewriting Systems*. Ed. by S. Kaplan and J.-P. Jouannaud. Vol. 308. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1988, pp. 3–14. ISBN 978-3-540-19242-8. doi:10.1007/3-540-19242-5_1. http://dx.doi.org/10.1007/3-540-19242-5_1.

[6] François Bobot et al. "The Alt-Ergo automated theorem prover (2008)". In: *URL http://alt-ergo.lri.fr* (2008).

[7] Marco Bozzano et al. "Efficient satisfiability modulo theories via delayed theory combination". In: *In Proc. CAV 2005, volume 3576 of LNCS*. Springer, 2005, pp. 335–349.

[8] B. Buchberger and R. Loos. "Algebraic Simplification". English. In: *Computer Algebra*. Ed. by Bruno Buchberger et al. Vol. 4. Computing Supplementa. Springer Vienna, 1983, pp. 11–43. ISBN 978-3-211-81776-6. doi:10.1007/978-3-7091-7551-4_2. http://dx.doi.org/10.1007/978-3-7091-7551-4_2.

[9] Jerry Burch and David Dill. "Automatic Verification of Pipelined Microprocessor Control". In: Springer-Verlag, 1994, pp. 68–80.

[10] Edmund M. Clarke and Jeannette M. Wing. "Formal Methods: State of the Art and Future Directions". In: *ACM Comput. Surv.* 28.4 (Dec. 1996), pp. 626–643. ISSN 0360-0300. doi:10.1145/242223.242257. http://doi.acm.org/10.1145/242223.242257.

[11] David R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial*. http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf. Accessed: 2013-08-23. Mar. 18, 2013.

[12] Sylvain Conchon, Evelyne Contejean, and Mohamed Iguernelala. "Canonized Rewriting and Ground AC Completion Modulo Shostak Theories". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by ParoshAziz Abdulla and K.RustanM. Leino. Vol. 6605. Lecture Notes

in Computer Science. Springer Berlin Heidelberg, 2011, pp. 45–59. ISBN 978-3-642-19834-2. doi:10.1007/978-3-642-19835-9_6. http://dx.doi.org/10.1007/978-3-642-19835-9_6.

[13] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN 0001-0782. doi:10.1145/368273.368557. http://doi.acm.org/10.1145/368273.368557.

[14] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN 0004-5411. doi:10.1145/321033.321034. http://doi.acm.org/10.1145/321033.321034.

[15] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN 0001-0782. doi:10.1145/1995376.1995394. http://doi.acm.org/10.1145/1995376.1995394.

[16] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN 3-540-78799-2, 978-3-540-78799-0. http://dl.acm.org/citation.cfm?id=1792734.1792766.

[17] Nachum Dershowitz. "Computing with rewrite systems". In: *Information and Control* 65.2–3 (1985), pp. 122–157. ISSN 0019-9958. doi:http://dx.doi.org/10.1016/S0019-9958(85)80003-6. http://www.sciencedirect.com/science/article/pii/S0019995885800036.

[18] Nachum Dershowitz and Jean-Pierre Jouannaud. "Handbook of Theoretical Computer Science (vol. B)". In: ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990. Chap. Rewrite systems, pp. 243–320. ISBN 0-444-88074-7. http://dl.acm.org/citation.cfm?id=114891.114897.

[19] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. "Variations on the Common Subexpression Problem". In: *J. ACM* 27.4 (Oct. 1980), pp. 758–771. ISSN 0004-5411. doi:10.1145/322217.322228. http://doi.acm.org/10.1145/322217.322228.

[20] Mark Dowson. "The Ariane 5 software failure". In: *SIGSOFT Softw. Eng. Notes* 22.2 (Mar. 1997), pp. 84–. ISSN 0163-5948. doi:10.1145/251880.251992. http://doi.acm.org/10.1145/251880.251992.

[21] Simson Garfinkel. *History's Worst Software Bugs*. http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all. Accessed: 2013-09-03. Nov. 8, 2005.

[22] Yeting Ge and Leonardo Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 306–320. ISBN 978-3-642-02657-7. doi:10.1007/978-3-642-02658-4_25. http://dx.doi.org/10.1007/978-3-642-02658-4_25.

[23] Jun Gu. "Local Search for Satisfiability (SAT) Problem". In: *Systems, Man and Cybernetics, IEEE Transactions on* 23.4 (1993), pp. 1108–1129. ISSN 0018-9472. doi:10.1109/21.247892.

[24] Sumit Gulwani and Ashish Tiwari. "Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions". In: *Proceedings of the 15th European conference on Programming Languages and Systems*. ESOP'06. Vienna, Austria: Springer-Verlag, 2006, pp. 279–293. ISBN 3-540-33095-X, 978-3-540-33095-0. doi:10.1007/11693024_19. http://dx.doi.org/10.1007/11693024_19.

[25] Jieh Hsiang. “Refutational theorem proving using term-rewriting systems”. In: *Artificial Intelligence* 25.3 (1985), pp. 255–300. ISSN 0004-3702. doi:http://dx.doi.org/10.1016/0004-3702(85)90074-8. http://www.sciencedirect.com/science/article/pii/0004370285900748.

[26] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. New York, NY, USA: Cambridge University Press, 2004. ISBN 052154310X.

[27] Jean-Pierre Jouannaud and Claude Marché. “Completion modulo associativity, commutativity and identity (AC1)”. In: *Design and Implementation of Symbolic Computation Systems*. Ed. by Alfonso Miola. Vol. 429. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1990, pp. 111–120. ISBN 978-3-540-52531-8. doi:10.1007/3-540-52531-9_130. http://dx.doi.org/10.1007/3-540-52531-9_130.

[28] Jean-pierre Jouannaud and Claude Marché. “Termination and Completion modulo Associativity, Commutativity and Identity”. In: *Theoretical Computer Science* 104 (1992), pp. 29–51.

[29] Shuvendu Lahiri and Shaz Qadeer. “Back to the Future: Revisiting Precise Program Verification using SMT Solvers”. In: *SIGPLAN Not.* 43.1 (Jan. 2008), pp. 171–182. ISSN 0362-1340. doi:10.1145/1328897.1328461. http://doi.acm.org/10.1145/1328897.1328461.

[30] Claude Marché. “On ground AC-completion”. In: *Rewriting Techniques and Applications*. Ed. by RonaldV. Book. Vol. 488. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 411–422. ISBN 978-3-540-53904-9. doi:10.1007/3-540-53904-2_114. http://dx.doi.org/10.1007/3-540-53904-2_114.

[31] Leonardo de Moura and Nikolaj Bjørner. “Efficient E-Matching for SMT Solvers”. In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 183–198. ISBN 978-3-540-73594-6. doi:10.1007/978-3-540-73595-3_13. http://dx.doi.org/10.1007/978-3-540-73595-3_13.

[32] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. “A Tutorial on Satisfiability Modulo Theories”. In: *Proceedings of the 19th international conference on Computer aided verification*. CAV’07. Berlin, Germany: Springer-Verlag, 2007, pp. 20–36. ISBN 978-3-540-73367-6. http://dl.acm.org/citation.cfm?id=1770351.1770358.

[33] Leonardo Moura and Nikolaj Bjørner. “Formal Methods: Foundations and Applications”. In: ed. by Marcel Vinícius Oliveira and Jim Woodcock. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. Satisfiability Modulo Theories: An Appetizer, pp. 23–36. ISBN 978-3-642-10451-0. doi:10.1007/978-3-642-10452-7_3. http://dx.doi.org/10.1007/978-3-642-10452-7_3.

[34] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.

[35] Greg Nelson and Derek C. Oppen. “Fast Decision Procedures Based on Congruence Closure”. In: *J. ACM* 27.2 (Apr. 1980), pp. 356–364. ISSN 0004-5411. doi:10.1145/322186.322198. http://doi.acm.org/10.1145/322186.322198.

[36] Greg Nelson and Derek C. Oppen. “Simplification by Cooperating Decision Procedures”. In: *ACM Trans. Program. Lang. Syst.* 1.2 (Oct. 1979), pp. 245–257. ISSN 0164-0925. doi:10.1145/357073.357079. http://doi.acm.org/10.1145/357073.357079.

[37] Robert Nieuwenhuis and Albert Oliveras. *Fast Congruence Closure and Extensions*. 2006.

[38] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)”. In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN 0004-5411. doi:10.1145/1217856.1217859. http://doi.acm.org/10.1145/1217856.1217859.

[39] *Patriot Missile Defense Software Problem Led to System Failure at Dhahran, Saudi Arabia*. `http://www.gao.gov/products/IMTEC-92-26`. Accessed: 2013-08-19. Feb. 4, 1992.

[40] Albert Rubio and Robert Nieuwenhuis. "A total AC-compatible ordering based on RPO". In: *Theoretical Computer Science* 142 (1995), pp. 209–227.

[41] Alex Sakharov. *Reduction Order From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein*. Accessed: 2013-11-16. `%5Curl%7Bhttp://mathworld.wolfram.com/ReductionOrder.html%7D`.

[42] Alex Sakharov. *Strict Order From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein*. Accessed: 2013-11-16. `%5Curl%7Bhttp://mathworld.wolfram.com/StrictOrder.html%7D`.

[43] Robert E. Shostak. "An Algorithm for Reasoning about Equality". In: *Commun. ACM* 21.7 (July 1978), pp. 583–585. ISSN 0001-0782. doi:10.1145/359545.359570. `http://doi.acm.org/10.1145/359545.359570`.

[44] Miroslav N. Velev and Randal E. Bryant. "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction". In: 2000, pp. 112–117.