Universität Karlsruhe (TH)
Research University · founded 1825

Institut für Theoretische Informatik

Automated Theorem Proving in Interactive Proof Construction

Diplomarbeit by

Christoph Sticksel

Supervisors:

Dr. habil. Peter Baumgartner (National ICT Australia)

Prof. Dr. P. H. Schmitt (Universität Karlsruhe)

13th Juli 2007

Abstract

The main contribution of this thesis is the application of the first-order theorem prover DARWIN, the implementation of the Model Evolution calculus, to software verification problems. It is attempted to embed the theorem prover as a decision procedure in the KEY system for formal specification and verification.

As a true first-order calculus, Model Evolution does not have to rely on ground instantiations, giving it an advantage in reasoning with quantifiers and uninterpreted function symbols that is required for the class of proof obligations that are examined.

This work is also a first approach towards satisfiability modulo theories in Model Evolution. It gives a heuristic implementation that is shown to be successful for a number of examples and discusses alternative possibilities to lift ground procedures of satisfiability modulo theories to the first-order calculus.

Deutsche Zusammenfassung

(German Abstract)

Diese Arbeit verfolgte das Ziel, den Theorembeweiser DARWIN als Entscheidungsprozedur im KEY-System zur Softwareverifikation einzusetzen. Dort werden die in einem integrierten Ansatz von objektorientierter Softwareentwicklung und -spezifikation gewonnenen Beweisverpflichtungen in einem taktisch geführten Ansatz bearbeitet. Dabei ist an bestimmten Stellen Interaktion mit den Benutzern nötig, um die nächsten Schritte auszuwählen.

Um diese Interaktion zu eliminieren, wurden vom KEY-System ausgehend Ansätze untersucht, Beweisverpflichtungen an automatische Theorembeweiser zu delegieren. Diese Arbeit konzentriert sich darauf, DARWIN, die Implementierung des Kalküls der Modellevolution, auf seine Eignung als Entscheidungsprozedur für Probleme der Softwareverifikation, generiert von KEY, zu untersuchen.

Die Schwierigkeiten dieses Ansatzes sind hauptsächlich auf der Seite von DARWIN zu finden, jedoch kann dessen Einsatz Erfolg für KEY versprechen. Modellevolution ist ein instanzbasiertes Verfahren für Logik erster Stufe, das den im aussagenlogischen Fall äußerst erfolgreichen DPLL-Algorithmus liftet. In beiden Ansätzen wird inkrementell eine Interpretation der Klauselmenge konstruiert bis ein Widerspruch auftritt oder ein Modell gefunden wurde. Die direkte Verfügbarkeit eines Modells könnte es KEY erlauben, dieses als zusätzliche Information über einen fehlgeschlagenen Beweis zu verwenden, oder einen Beweis zu zertifizieren. Im Gegensatz zu anderen instanzbasierten Methoden für Logik erster Stufe, die Grundinstanzen annehmen, steht der Modellevolution die Termstruktur zur Verfügung, so dass ein echt mächtigeres Schließen möglich ist.

Beweisverpflichtungen, insbesondere im Kontext der Softwareverifikation, treten häufig in Verbindung mit einer Hintergrundtheorie auf, die die Semantik der Programmumgebung formalisiert. Erfüllbarkeit modulo einer Theorie ist für instanzbasierte Methoden ein aktueller Forschungsbereich. In dieser Arbeit wird ein erster Ansatz verfolgt, mit Modellevolution modulo einer Theorie zu schließen und den Kalkül somit einem wichtigen Anwendungsgebiet zu öffnen.

Als Hintergrundtheorie für Beweisverpflichtungen wählt KEY die Vereinigung der Theorie der Arrays und der linearen Ganzzahlarithmetik (*AUFLIA*). Quantoren sind in dieser Theorie ebenso erlaubt wie uninterpretierte Funktionssymbole, so dass der Kalkül der Modellevolution als Verfahren für Logik erster Stufe seine Stärken hier besonders ausspielen kann.

Die Verbindung zwischen KEY und DARWIN wird von der SMT-LIB-Initiative geschaffen, deren Ziel eine Bibliothek von Benchmarks für Erfüll-

barkeit modulo Theorien ist. Der Export in deren syntaktischer Spezifikation ist in KEY bereits vorhanden, in einem ersten Schritt wurde ein Import in diesem Format für DARWIN implementiert.

Die Theorie der Gleichheit liegt fast jeder Theorie zugrunde, insbesondere auch der *AUFLIA*-Theorie. Gleichheitsschließen in Modellevolution wurde zwar formuliert, war zum Zeitpunkt dieser Arbeit jedoch noch nicht in DARWIN verfügbar. Um laufende Arbeiten daran nicht zu duplizieren, wurde eine Approximation von Gleichheitsschließen mit Ersetzungsregeln verwendet, die rein operational definiert wurde, ohne den Kalkül zu verändern, so daß trotzdem Ergebnisse erzielt werden konnten, für die mit der Integration des Gleichheitsschließens in den Kalkül der Modellevolution eine starke Verbeserung zu erwarten ist.

Aus einer Betrachtung verschiedener Verfahren für Schließen modulo Theorien im aussagenlogischen DPLL-Kalkül ergaben sich keine im Rahmen dieser Arbeit umsetzbare Ansätze, diese in den Kalkül der Modellevolution zu liften, wie Modellevolution selbst ein Lifting des DPLL-Verfahrens ist.

Es wurde dann ein heuristischer Ansatz verfolgt, der im Allgemeinen unvollständig ist und wiederum rein operational ohne Veränderung des eigentlichen Kalküls implementiert ist. Die Klauselmenge wird in eine kanonische Form transformiert und mit einer Menge von passend formulierten Theorieaxiomen, in diesem Fall der *AUFLIA*-Theorie, vereinigt.

Mit diesem Ansatz konnten verschiedene Beispiele aus dem Bereich der Softwareverifikation, die von KEY oder der SMT-LIB stammten, erfolgreich bearbeitet werden.

Declaration / Erklärung

I hereby declare that this thesis is entirely written by myself. All sources of information are listed in the bibliography.

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Christoph Sticksel
Karlsruhe, 13th July 2007

Contents

Chapter 1

Introduction

The ultimate goal set for this work is to use the theorem prover DARWIN as a decision procedure for the verification of software within the KEY system that is used in an integrated approach to object-oriented software development and verification. It generates proof obligations from a specification and uses its own prover to follow them. At certain points in the tactically guided proof construction, interaction with the user is required to choose the next derivation steps.

To minimize interactivity and involvement of the user, possibilities have been explored to delegate proof obligations to back-end provers to be embedded in the KEY system. This work focusses on the implementation of the Model Evolution calculus DARWIN, which is an automated theorem prover, as a decision procedure for such proof obligations.

With the challenges of the approach mostly on DARWIN's side, it can offer some rewards to the KEY system. The Model Evolution calculus is an instance-based method on clauses in first-order logic with a proper first-order reasoning. The idea behind instance-based methods is to build up an interpretation of the clause set until contradiction or saturation. With the direct availability of an interpretation, the generated proof could be certified or a model could be returned, making it possible to rely on the proof or use information from the interpretation in the KEY system. In contrast to other instance-based methods that use ground instances, Model Evolution retains the first-order structure of the clauses and can exploit the advantage of still having the term structure at its hand making the reasoning more powerful.

Proof obligations especially in a software verification context most often imply reasoning modulo a background theory that formalizes the semantics of the programming environment. For instance-based methods that appeared only recently, satisfiability modulo theories is an active area of current research and no prior results exist for Model Evolution modulo the-

ories. This work takes a first step towards making the calculus applicable to problems from the important domain of software verification.

Outline of the Work

Setting the stage for our endeavors we need to introduce several systems that will be described for the rest of this chapter.

As the KEY system is already equipped to export its proof obligations in the SMT-LIB format, it was only necessary to make DARWIN literate in that format. Chapter 2 will discuss issues of our implementation.

The following chapter 3 is concerned with the background theory satisfiability has to be checked modulo. KEY sets this theory to be the so-called *AUFLIA* theory of arrays, uninterpreted functions and linear integer arithmetic.

Equational reasoning is at the heart of almost any background theory and also plays an important role in the *AUFLIA* theory that is used throughout this work. Though the calculus of Model Evolution with equality has been defined, there is no implementation yet. In chapter 4 we describe our approximation of rewrite-based equational reasoning that was conceived to be easily implementable without duplicating efforts for and interfering with a soon available version of DARWIN with equational reasoning.

Chapter 5 finally contains thoughts towards theory reasoning in the Model Evolution calculus and a first heuristic approach. We have evaluated it with proof obligations from the KEY system as well as with benchmarks from SMT-LIB and discuss the performance of our approach.

We conclude with chapter 6 by summarizing our results and pointing out some open ends that could be subjects of further research.

System descriptions

The already mentioned KEY system is the starting point and source of proof obligations. The implementation of the Model Evolution calculus DARWIN is to be used as a decision procedure within KEY. We will briefly discuss the main features of the calculus and its roots in the DPLL procedure. The link between KEY and DARWIN is provided by the SMT-LIB initiative that created a library of benchmarks to compare solvers for satisfiability modulo theories and established a syntactical specification for those benchmarks that will be used a means of exchange here.

KEY

The KEY system is a joint project at the Universities of Karlsruhe and Koblenz-Landau and the Chalmers University in Göteborg, see [BHS07] for a detailed description.

The KEY approach to software verification tries to integrate formal specification and verification as seamlessly as possible into the process of designing software. This is achieved by embedding the KEY tools into computer-aided software engineering (CASE) tools, currently there are plug-ins for the environments Borland Together and Eclipse IDE available.

The target language is JAVA CARD which is a subset of the JAVA language that basically allows sequential, i. e. non-threaded, JAVA programs without dynamic class loading and floating point types. JAVA CARD is supposed to be used on smart cards and similar devices where the full features of JAVA are not needed, but applications are often security-critical so that verifying the correctness of an implementation is more than desirable.

In Borland Together the software is modelled using the Unified Modelling Language (UML) and verification conditions are specified in the Object Constraint Language (OCL) which is part of UML. The Eclipse IDE chose the Java Modelling Language (JML) for both tasks.

In both environments the user can apply standard design patterns that have been prepared to make it easier for them to write verification conditions, they are further supported by other means to make formal specification really accessible to a user with an intermediate literacy in formal methods.

The formal specification that has been created in a CASE tool is then translated to a dynamic logic and the interactive prover runs a sequent calculus on it. This part of the KEY system can be used as a stand-alone tool as well. The derivation is governed by the tactlet mechanism that applies to the sequent a set of find-and-replace rules that consists of the derivation rules of the calculus as well as of more heuristic and theory specific rules. It is by choosing a tactlet to be applied where the user can and sometimes has to guide the derivation.

To decrease interactivity, many approaches have been considered, one of them being the translation to SMT-LIB syntax of an open proof goal to be closed by an external prover that can be called at any time in the derivation.

DARWIN

The Model Evolution calculus was conceived as a first-order lifting of the Davis-Putnam-Logemann-Loveland procedure (DPLL), named after its authors who described it in [DP60] and [DLL62]. Though DPLL was originally intended as a decision procedure for satisfiability of first-order clause sets as well, it handles quantifiers quite inefficiently and is nowadays al-

most exclusively but with great success used only on ground propositional clauses.

This ground propositional DPLL procedure can be described quite elegantly with a sequent-style calculus as in [Tin02]. It starts with a set of clauses Φ and an empty context Λ that in each derivation step is a set of ground literals that must be true for the clause set Φ to be satisfiable. This is meant by the sequent $\Lambda \vdash \Phi$ that is maintained throughout the derivation. Λ is incrementally filled from the clause set Φ until it becomes contradictory or saturated. In the latter case the context Λ is a Herbrand model for the clause set Φ.

On a short glance at the calculus as described by [Tin02], the context Λ is grown by the rule

$$\frac{\Lambda \vdash \Phi, \{l\}}{\Lambda, l \vdash \Phi, \{l\}} \qquad \text{if } l \notin \Lambda \text{ and } \neg l \notin \Lambda \tag{1.1}$$

that puts the literal l into the context – unless the literal l or its opposite $\neg l$ has already been asserted – so that the unit clause $\{l\}$ is satisfied.

The second rule that affects the context guesses the truth value of a literal p and splits the derivation tree in two branches for asserting p and $\neg p$, respectively

$$\frac{\Lambda \vdash \Phi, \{C \vee p\}}{\Lambda, p \vdash \Phi, \{C \vee p\} \qquad \Lambda, \neg p \vdash \Phi, \{C \vee p\}} \qquad \text{if } p \notin \Lambda \text{ and } \neg p \notin \Lambda \tag{1.2}$$

In a derivation branch the context always grows, literals are never removed from it.

There are two rules left that modify the clause set but not the context. The subsume rule removes clauses that are already satisfied by the context while the resolve rule detects a contradiction between a literal in the context and a literal in a clause. It tries to repair the situation by removing the literal from the clause so that another literal has to satisfy the clause. If this is not possible, the current branch of the derivation tree is closed and if there is no choice point left to backtrack to, the clause set is unsatisfiable.

If the calculus reaches a sequent $\Lambda \vdash \varnothing$ with an right-hand side that has become empty by subsumption, the clause set is satisfiable and the current context Λ is a model for it.

The Model Evolution calculus is a direct lifting of this DPLL procedure as described by [BT03]. The context is now no longer a set of ground literals but a finite representation of a first-order interpretation. The rules from the DPLL calculus are changed to handle non-ground literals in the clause set and two additional rules are needed.

But still, the underlying idea stays the same: asserting truth values from unit clauses or splitting on guessed truth values for literals from non-unit clauses, removing subsumed literals and detecting contradictions between

the context and literals in clauses. If all branches in the derivation tree are closed, the clause set is unsatisfiable, if the right-hand side of the sequent is empty, the context is a model for the clause set which has thus been proven to be satisfiable.

The version of DARWIN that is current at the time of writing implements the Model Evolution calculus as introduced here, a detailed description is [BFT04]. The Model Evolution calculus with equality in [BT05] has not been implemented yet and only this work is a first approach towards satisfiability modulo theories with Model Evolution.

SMT-LIB

The goal of the SMT-LIB initiative, as stated by its coordinators, is to "produce an on-line library of benchmarks for Satisfiability Modulo Theories" [RT06]. A benchmark is to be understood as a single formula that may underlie a set of assumptions whose satisfiability is to be checked with respect to a specified background theory.

While the SMT-LIB format is not explicitly intended to be used as a means of exchange between systems, it will do very well for this purpose. A proof obligation or a goal in KEY is nothing but a formula to be checked for satisfiability in a certain background theory. As will be described below, SMT-LIB defines background theories separately from the formula to be checked, thus minimizing the information that has to be passed from KEY to DARWIN and allowing the latter to employ its own rules of inference on the theory.

The current version of the SMT-LIB at time of writing was 1.2, released on 30 August 2006, see [RT06]. As the SMT-LIB initiative is gaining more and more support from the community it is expected to evolve with input from the community and become a common format. Yet it is at the current state completely sufficient for the purposes of this work and there exist many benchmarks from the domain of software verification.

Chapter 2

SMT-LIB

DARWIN in its current version cannot read SMT-LIB files. As the Model Evolution calculus works on sets of clauses in clausal normal form (CNF), DARWIN has to be presented its input in that form in the sort-free first-order predicate logic without equality. On the other hand, SMT-LIB was designed to ease the creation of benchmarks thus allowing many convenience notations and not constraining the input to a normal form. Its logic is many-sorted and contains equality, making the translation of SMT-LIB input to DARWIN not as trivial as it may sound and requiring some successive steps that will be detailed in the following sections.

2.1 Overview of the specification

The SMT-LIB format as in [RT06] was kept modular to be extensible and adaptable to many applications. The specification distinguishes between the underlying logic, the background theory and the formula language.

The underlying logic is fixed to a basic many-sorted first-order logic with equality. Only sorts and sorted symbols can be defined, there are no subsorts or quantifiers over sorts keeping the logic semantically equivalent to an unsorted first-order logic with equality. In fact, [RT06] define a translation of formulas in the many-sorted logic of SMT-LIB to unsorted first-order logic.

Symbols, i. e. predicate and function symbols, must be declared with their argument sorts and in the case of functions also with their value sort. Overloading of function and predicate symbols is permitted if the sort of the function's value is the same for all its versions with the same arity. Thus each functional subterm has a unique sort that can be determined locally from the declaration of the function symbol.

SMT-LIB requires only the signature to be formally, i. e. machine-readably, described, while the theory itself may well remain only informal in a natural language. There will only be a limited number of interesting

theories, and solvers will make use of their own specialized axiomatizations for popular theories, therefore not requiring to be given one. The theories of integer arithmetic, real arithmetic or extensional arrays are examples of frequent background theories.

The formula language is the fragment of the logic that the benchmark uses, it is an expansion of the signature of the background theory. Again, the current SMT-LIB specification does not require the definition of the formula language to be machine-readable, it is rather to be seen as a hint for the user as to what formulas to expect. A set of benchmarks could e. g. restrict its language to a quantifier-free fragment of first-order logic and a solver then take advantage of this fact. The combination of a theory and a formula language is called sublogic, again there will be a certain number of interesting ones. Each year there is a competition between SMT solvers that features about ten different sublogics as divisions in the competition[1].

Technically there are three file types defined, corresponding to the distinguished aspects above. The underlying logic is fundamental and cannot be changed. A theory file defines the background theory, a logic file refers to exactly one theory and additionally specifies the formula language for the benchmark. Distinct theories or sublogics cannot be combined and used together in one sublogic or benchmark, respectively. A combination of theories like the integer and array theory has to be formulated as a new theory.

The benchmark file finally refers to one logic and contains exactly one formula to be checked for satisfiability. It may give more formulas as assumptions and use free sort, predicate and function symbols that have to be declared.

2.2 Sorts

In an SMT-LIB formula sorts are only referenced in quantifiers, but there they are compulsory. Bound variables always quantify over elements of one sort, e. g. s in these two formulas:

$$\forall x : s \quad \Phi$$

$$\exists x : s \quad \Psi$$

The universally quantified formula Φ holds for all elements x of sort s and there is an element x of sort s for which Ψ holds. According to the translation semantics given in [RT06] these formulas are to become

$$\forall x \quad s_s(x) \to \Phi$$

$$\exists x \quad s_s(x) \wedge \Psi$$

---

[1]See `www.smtcomp.org` for the current SMT-COMP'07 in July 2007

in the unsorted first-order logic where $s_s(x)$ is a predicate that is true exactly for elements of sort s. This predicate is also used in clauses that represent the sort constraints of the declarations of functions and constants. For each constant c with sort s there has to be a clause

$$s_s(c)$$

and for each function $f(x_1, \ldots, x_n)$ whose arguments are of the sorts s_1 to s_n, respectively, and whose value is of sort s, this is to be constrained by the clause

$$s_{s_1}(x_1) \wedge \cdots \wedge s_{s_n}(x) \rightarrow s_s(f(x_1, \ldots, x_n))$$

where again s_s and s_{s_1} to s_{s_n} are predicates that are true for all elements of their respective sort.

This translation gives an exact semantic of the many-sorted first-order logic of SMT-LIB in the usual unsorted first-order logic. However, we chose to implement the translation for DARWIN in a different way. We observed that the introduction of sort predicates slowed down the proof search considerably (see section 5.2.3 for an example) due to the nature of an instance-based method. Especially theories mostly contain quantified implications. If each bound variable there is guarded by a predicate constraining its sort, the Model Evolution calculus does not only have to assert all literals in the conclusion of the implication, but also the sort predicates in the premiss. Depending on the depth of the terms the quantified variables are instantiated with, this can considerably lengthen the derivation. Take a simple axiom for commutativity of a function plus as an example:

$$\forall t : \mathrm{Int} \ \forall x : \mathrm{Int} \ \forall y : \mathrm{Int} \quad t = \mathrm{plus}(x, y) \rightarrow t = \mathrm{plus}(y, x)$$

According to the translation semantics it would become

$$\forall t \ \forall x \ \forall y \quad s_{\mathrm{Int}}(t) \wedge s_{\mathrm{Int}}(x) \wedge s_{\mathrm{Int}}(y) \rightarrow (t = \mathrm{plus}(x, y) \rightarrow t = \mathrm{plus}(y, x))$$

which is already in CNF and can thus be written as

$$\{ \ t = \mathrm{plus}(y, x), \ \neg t = \mathrm{plus}(x, y), \ \neg s_{\mathrm{Int}}(t), \ \neg s_{\mathrm{Int}}(x), \ \neg s_{\mathrm{Int}}(y) \ \}$$

in the usual clause syntax. To assert e. g.

$$P(\mathrm{plus}(\mathrm{plus}(c, d), \mathrm{plus}(a, b)))$$

from

$$P(\mathrm{plus}(\mathrm{plus}(a, b), \mathrm{plus}(c, d))),$$

with P being an arbitrary predicate symbol, the calculus would have to assert

$$s_{\mathrm{Int}}(\mathrm{plus}(a, b))$$

and

$$s_{\mathrm{Int}}(\mathrm{plus}(c,d))$$

as well which would in turn involve the sort constraints from the definitions of plus

$$\forall x\,\forall y\quad s_{\mathrm{Int}}(x)\wedge s_{\mathrm{Int}}(y)\rightarrow s_{\mathrm{Int}}(\mathrm{plus}(x,y))$$

and the sort constraints

$$s_{\mathrm{Int}}(a)$$

etc. for the definitions of the constants first.

For that reason the sort constraints were dismissed, or formally speaking, assumed to be true for all elements. This makes the formula to prove weaker, but if the weaker formula is found to be unsatisfiable, a more constrained formula certainly is as well.

On the other hand, we found that our equational reasoning discussed in section 4.3 profits from an explicit inclusion of sorts, again because of the way an instance-based method like Model Evolution works. Without sort constraints the calculus is not aware of the existence of different sorts and may choose to assert an equation between terms of distinct sorts. This will, of course, lead to a greater search space in the derivation. For that reason we chose to use a ternary predicate eqs instead of the usual equality predicate, adding the sort of the terms in the first argument of the predicate. That way, equality does not admit to assertions of equations between distinct sorts.

The availability of an implementation of the Model Evolution calculus with equality ([BT05]) will obsolete the need for the ternary predicate eqs. Equations will then only be asserted from equations that are in the context and as long as the clause set only contains well-sorted equations between terms of the same sort, ill-sorted equations can never occur in the derivation. See section 4.3 for details on the implementation of equality.

2.3   Formula language

Formulas can occur as an axiom in the background theory or in the benchmark either as an assumption or as the formula itself. In all cases they have the same syntax that is an extension of the usual syntax for first-order logic. The concrete syntax can be found in [RT06], in the following only the extensions are discussed and how they are transformed for DARWIN. They are introduced mostly for the convenience of writing benchmarks and do not enhance the expressiveness of the logic.

if-then-else for formulas For formulas Φ_0, Φ_1 and Φ_2 there is an if-then-else connective

$$\text{if } \Phi_0 \text{ then } \Phi_1 \text{ else } \Phi_2$$

provided that is syntactically equivalent to

$$\Phi_0 \rightarrow \Phi_1 \wedge \neg\Phi_0 \rightarrow \Phi_2.$$

It is replaced with the equivalent during recursive parsing, note that this grows the length of the formula exponentially in the number of occurrences of if-then-else formulas in the if part.

if-then-else for terms A different form of if-then-else chooses one of two given terms t_1 and t_2 depending on the validity of the formula Φ:

$$\text{ite}(\Phi, t_1, t_2).$$

Observe that this is a term that has a formula and two terms as arguments. The term's value depends on the validity of the embedded formula. The approach followed here introduces a new constant symbol v that is substituted for the if-then-else term. The formula

$$(\Phi \rightarrow v = t_1) \wedge (\neg\Phi \rightarrow v = t_2)$$

is then conjunctively added to the atom that contains the if-then-else term. The new variable v will be free in the formula, however, its sort can be deduced at each of its three occurrences. In both equalities, the sort of the right hand side will be accessible and identical to the sort of v. Where v replaces the original if-then-else term, it will occur as the argument of either a function or a predicate and both will be declared with the sorts of their arguments.

As this translation has to add a formula further up in the parse tree, it cannot be applied while terms are recursively parsed. Therefore the if-then-else term will only be eliminated after the whole formula has been parsed.

E. g. the formula

$$\neg P(f(\text{ite}(\Phi, t_1, t_2)))$$

becomes

$$\neg(P(f(v)) \wedge (\Phi \rightarrow v = t_1) \wedge (\neg\Phi \rightarrow v = t_2)).$$

Assignments A formula can contain formula or term variables that stand for a formula or a term, respectively. Those variables are always bound by an assignment and allow to use abbreviations for frequently occurring subformulas or subterms.

A formula or term variable is always expanded to the formula or term it has been assigned during parsing. Though this information might help building DARWIN's term database that stores each term only once, it is discarded. It would be too much an effort to keep the assignment information while it is not guaranteed that after putting the formula in CNF the assigned term or formula will still be present unchanged.

n-**ary connectives**   SMT-LIB allows most connectives to be of arbitrary arity. When building the CNF, only conjunctions and disjunctions are allowed to be greater than binary. If any other connective (e. g. xor or iff) appears with an arity greater than two, it is reduced to a left-associative chain of binary connectives.

Equalities are allowed to be greater than binary in SMT-LIB as well. They are reduced to a chain of binary equations. The distinct predicate that states the pairwise disjointness of its terms analogously becomes a chain of pairwise inequalities.

Numbers   DARWIN cannot deal with arithmetic, thus all numbers, be they rational or integer, are treated as uninterpreted functions. Each number is prefixed with *n* and the decimal point is replaced by an underscore to conform to the syntax of identifiers.

Annotations   All annotations are removed regardless of their content. Currently there is no formal specification of any annotation so there is no loss of information. However, it is considered to define certain annotations in future versions of the SMT-LIB that may e. g. denote a predicate as commutative or associative.

Overloading   Overloading of functions is not supported thus violating the SMT-LIB specification that allows this to a limited extent that is called "ad hoc" overloading in [RT06]. Two formulas with the same arity must not have a different sort, the sort of their arguments may differ. In any case the sort of a term can be determined by only considering the term itself. As each formula is closed, there are no free variables and the sort of a variable can be read from the quantifier, constants and functions have to be declared.

Not supporting overloading does not lose any expressive power as the overloaded functions can always be renamed.

2.4   Building the CNF

To create the clausal normal form of a formula, the existing approach in DARWIN to read input from the TPTP benchmark library[2] is reused. For TPTP formulas not in CNF the embedded E prover[3] is called to return another TPTP file in CNF.

Thus the set of formulas read from an SMT-LIB benchmark is written in TPTP format and passed to E whose output is then read by DARWIN. Though DARWIN's logic does not use sorts and we further chose to drop the sort predicates, we still need to know the sorts of constants and functions for our equational reasoning that will be introduced in chapter 4.

The TPTP format does not deal with sorts, however, by referring to the declarations from the SMT-LIB input this information can be preserved – except for the Skolem constants introduced by the E prover. In the worst case a formula like

$$\exists x : s_x \, \exists y : s_y \quad x = y$$

with s_x and s_y being the sorts of x and y, respectively, is skolemized to

$$c_x = c_y$$

where the sorts of c_x and c_y cannot be determined. Both are fresh constants that have not been declared in the SMT-LIB benchmark. Equations are always between terms of the same sort, however in this case, the sort of neither side of the equation is known.

In the formula

$$\exists x : t \quad P(x, a)$$

the bound variable x of sort t will be skolemized, i. e. the formula is of type δ in Smullyan's uniform notation, see e. g. [Smu95]. We then replace the atom $P(x, a)$ by a conjunction of the atom and a predicate for the sort of x, yielding in this case

$$\exists x \quad P(x, a) \wedge s_t(x).$$

If $s_t(x)$ evaluates to true for any x, the satisfiability of the formula is preserved. After skolemization we get

$$P(c_x, a) \wedge s_t(c_x)$$

and the subformula $s_t(c_x)$ tells us that c_x is of sort t which we add to our declarations of functions and constants. We then replace the conjunction by the original, now skolemized atom

$$P(c_x, a)$$

---

[2]See [SS98]
[3]See [Sch04]

still preserving satisfiability as again this is equivalent to having $s_t(x)$ evaluate to true for any x.

This mechanism to find the sorts of skolem symbols is applied after the described removal of if-then-else terms in section 2.3. The connectives iff and xor have both polarities and thus can be both δ and γ formulas. Their subformulas are treated as δ formulas, because skolemization will happen for existentially quantified variables in their subterms.

Removing the sort predicates after skolemization, but before the CNF is built, is not possible, as calling the E prover is one atomic step. Thus the predicates have to be removed from the returned clauses. We have to evaluate the sort predicates to true for any argument, therefore every positive occurrence of a sort predicate in a clause will trivially satisfy it and we remove that clause. By a similar argument we remove negative occurrences of a sort predicate from every clause as that literal can never be satisfied.

The formula

$$\exists x : s_1 \; \exists y : s_2 \quad P(x) \operatorname{xor} Q(y)$$

is replaced by

$$\exists x \; \exists y \quad (P(x) \wedge s_1(x)) \operatorname{xor}(Q(y) \wedge s_2(y))$$

which is skolemized and transformed to the clause set

$$\{\, P(c_x),\, Q(c_y) \,\}$$
$$\{\, P(c_x),\, s_1(c_x) \,\}$$
$$\{\, Q(c_y),\, s_2(c_y) \,\}$$
$$\{\, \neg P(c_x),\, \neg Q(c_y),\, \neg s_1(c_x),\, \neg s_2(c_y) \,\}.$$

The sort predicates $s_1(c_x)$ and $s_2(c_y)$ tell the sort of the Skolem constants and can now be removed. Evaluating $s_1(x)$ and $s_2(x)$ to true for any x, this is equisatisfiable to the clause set

$$\{\, P(c_x),\, Q(c_y) \,\}$$
$$\{\, \neg P(c_x),\, \neg Q(c_y) \,\}$$

which is exactly the CNF of the original formula without the added sort predicates.

The correctness of the transformations follows quite easily if one assumes that the introduced sort predicate $s(x)$ is true for every x. Then for every formula Φ satisfiable by a model $\mathcal{M}$, it is also a model for Φ with added sort predicates:

$$\mathcal{M} \vDash \Phi \quad \Leftrightarrow \quad \mathcal{M} \vDash \Phi \wedge s(x)$$

Satisfiability is thus preserved. Of course, it does not matter when the sort predicate is eliminated and this may well happen after building the CNF. By this argument, the E prover may use the definitory CNF instead of simple CNF as it does for larger formulas. Inserting and removing the sort predicates does not change the satisfiability of the formula in that case either.

2.5 SMT-LIB to DARWIN

Putting together the steps described above, we are presenting input in SMT-LIB syntax to DARWIN using some intermediate transformations.

The parser for SMT-LIB files was initially taken from [RT06] and happened to be written in OCaml, the programming language of DARWIN. It was slightly adapted to handle version 1.2 of the SMT-LIB specification and incorporated into DARWIN.

The goal of parsing is to convert the input to the TPTP format in order to use the E prover to clausify it and pass it on to DARWIN's proof procedure. In that translation the different types of formulas from the benchmark and the theory are mapped to the according formula roles in TPTP. Axioms from the theory take the role `axiom`, an assumption in the benchmark becomes a `hypothesis` and the formula is the `conjecture`. When processing these formulas, the E prover negates the `conjecture` and returns the clauses as a `negated_conjecture`. To preserve the polarity of the original formula, it is negated prior to passing it to the E prover to neutralize the negation done there.

The elements of SMT-LIB that are not defined in TPTP, viz. if-then-else subformulas, assignments, *n*-ary connectives, numbers and annotations are removed during recursive parsing as described in section 2.3. If-then-else subterms are only eliminated after parsing, overloading of function symbols is not supported.

To be able to take advantage of the sortedness of the SMT-LIB input for theory reasoning that will be described later, the declarations of predicates, functions and sorts are kept and the formulas to be clausified are prepared as described in section 2.4 to be able to recover the sort of the introduced skolem constants.

The formula is presented to and interpreted by DARWIN without sorts as the Model Evolution calculus does not support this. Following section 2.2, sort predicates from the translation semantics are not used, but as we will discuss later, our equational reasoning takes advantage of the sort information.

Chapter 3

Sublogic *AUFLIA*

Most of this work focusses on one specific sublogic, named *AUFLIA* that KEY delivers its proof obligations exported to SMT-LIB syntax in. It is short for "*Arrays, uninterpreted function symbols and linear integer arithmetic*" and in the files provided by [RT06] the logic refers to the theory *Int_ArraysEx*. The latter is the union of the integer theory and the theory of arrays with extensionality that have integer indices and values. A theory in SMT-LIB only defines predicate and function symbols and their properties, the fragment of the underlying logic benchmark formulas are drawn from is not defined but in the sublogic. The *AUFLIA* sublogic is concerned with closed, quantified formulas over an expansion of the theory signature with free sort, function and predicate symbols where the function symbol for multiplication must not occur nested. Thus it is always possible to see a multiplication as a linear sum which explains the term linear arithmetic used in *AUFLIA*'s name.

   The Model Evolution calculus is especially suited to problems in the *AUFLIA* logic as it is a true first-order calculus and as such can handle the possible term structures built up by quantifiers and uninterpreted functions quite well compared to approaches that use ground instantiations.

   The following sections describe those properties and issues using the *AUFLIA* logic in some detail.

3.1   Integer theory

The theory defines one single sort, *Int*, so that all constants and functions are of that sort as are the arguments of functions and predicates.

   Further defined are the two constants 0 and 1 and the functions unary minus $neg(x)$, binary subtraction $minus(x, y)$, binary addition $plus(x, y)$ and binary multiplication $times(x, y)$ that have the usual meaning in integer arithmetic. Both addition and multiplication are associative, commutative and have respectively 0 or 1 as their unit, binary subtraction is only

associative. In the linear arithmetic fragment there is no division and no distributivity of $\text{plus}(x, y)$ and $\text{times}(x, y)$ as this would violate the linearity constraint.

The relation predicates $\leq$ and $\geq$ are both reflexive, transitive and anti-symmetric, their strict counterparts $<$ and $>$ are transitive and irreflexive.

In this formulation the theory cannot be recursively axiomatized in SMT-LIB's underlying logic, the sorted first-order logic with equality. Being at least as expressive as Peano arithmetic, it is undecidable, too.

However, this is not as unfortunate as it may sound, because the sublogic *AUFLIA* will restrict its formula language on the background theory to a more tractable fragment of a combination of integer and array logic.

3.2   Array theory

The general theory of arrays with extensionality (*ArraysEx*) is modelled by two functions select and store:

$$\text{select} : Array \times Index \rightarrow Element$$

$$\text{store} : Array \times Index \times Element \rightarrow Array.$$

The semantic is given by the axioms

$$\forall a : Array \ \forall i : Index \ \forall e : Element \ \text{select}(\text{store}(a, i, e)) = e \qquad (3.1)$$

$$\forall a : Array \ \forall i : Index \ \forall j : Index \ \forall e : Element$$
$$(i = j) \vee (\text{select}(\text{store}(a, i, e), j) = \text{select}(a, j)) \qquad (3.2)$$

$$\forall a : Array \ \forall b : Array \ ((\forall i : Index \ \text{select}(a, i) = \text{select}(b, i)) \rightarrow a = b).$$
$$(3.3)$$

The first quite obvious axiom (3.1) states that an element *e* stored into an array *a* at index *i* can be read from that position. The second axiom (3.2) ensures that writing an element *e* to index *i* does not affect any element at a position *j* unless *j* is equal to *i*. The extensionality axiom (3.3) defines two arrays as equal if they have the same elements stored at the same indices, which would be called value identity in programming languages. If one leaves out the extensionality axiom (3.3), the equality on arrays becomes what would be called structural identity. This theory has been named *Arrays* in SMT-LIB in contrast to *ArraysEx* for the formulation including extensionality.

Axioms (3.1) and (3.2) can be combined in the read-over-write axiom, that is also called "McCarty axiom" after [MP67] where it first appeared:

$$\forall a : Array \; \forall i : Index \; \forall j : Index \; \forall e : Element$$
$$i = j \rightarrow \text{select}(\text{store}(a,i,e),j) = e) \wedge \tag{3.4}$$
$$i \neq j \rightarrow \text{select}(\text{store}(a,i,e),j) = \text{select}(a,i)).$$

We will use two axioms (3.1) and (3.2) instead, their CNF is shorter than (3.4). See [SBDL01] for a survey of array theories and decision procedures on them.

3.3 Combining Integer and Array theory

The *Int_ArraysEx* and *Int_Arrays* theories do not use distinct sorts for indices and elements but instead integers for both, the array functions become

$$\text{select} : Array \times Int \rightarrow Int \tag{3.5}$$

$$\text{store} : Array \times Int \times Int \rightarrow Array. \tag{3.6}$$

where the integer sort has been substituted for both the index and element sort in the axioms.

The *Int_ArraysEx* theory consisting of two separate theories may admit to using procedures for the combination of theories like the Nelson-Oppen or the Shostak methods (see [RRT05] for a survey). However, we will not now explore this possibility in our axiomatization.

3.4 *AUFLIA* formula language

The sublogic *AUFLIA* finally defines a formula language that is an extension of the *Int_ArraysEx* signature with free sort, function and predicate symbols. The formulas may use quantifiers arbitrarily, the only requirement is that the formula be closed. Thus there are no free variables and the sort of each term can easily be inferred. A term is either a variable, in that case the quantifier defines its sort. Otherwise it is a function or a constant whose sort can be taken from its declaration.

Only linear integer arithmetic is allowed, thus multiplication must not occur arbitrarily but merely as an abbreviation for n additions or subtractions:

$$n \cdot t = \sum_{\nu=1}^{n} t$$

with positive coefficient n and

$$-n \cdot t = -\sum_{\nu=1}^{n} t$$

with negative coefficient $-n$.

With this definition it is not possible to have quadratic or higher terms, justifying the naming linear arithmetic.

3.5   KEY and *AUFLIA*

Proof obligations from the KEY prover have the form

```
¬(typeHierarchyFormula ∧ typePredFormula ∧
                    antecedentSequent → succedentSequent),
```

note that the whole formula is negated.

There is no direct mapping of JAVA types to SMT-LIB sorts as the latter does not support subsorts that would be required to model JAVA's inheritance. Adding to that, JAVA provides more types than *AUFLIA* making it necessary to add constraints to map the needed parts of JAVA's object model.

The `typeHierarchyFormula` represents JAVA's object and type hierarchy in the formula, it is a conjunction of implications of sort predicates, each representing an inheritance. The `typePredFormula` contains constraints for guarded variables in quantifiers. They are needed for sorts that are not available in SMT-LIB and are formulated as a conjunction of sort predicates.

The implication of `antecedentSequent` and `succedentSequent` that is last in the conjunction is the actual proof obligation that comes from the sequent calculus the KEY prover employs.

No advantage can be taken of the structure proof obligations are presented in. After the CNF has been built, `typeHierarchyFormula` and `typePredFormula` being a conjunction of disjunctive formulas have become a set of clauses independent of the implication of `antecedentSequent` and `succedentSequent` that has become another set of clauses.

Should the intended interpretation of a function or predicate symbol be richer than what *AUFLIA* supplies, it becomes uninterpreted and is only defined with its sorts in `typePredFormula`. This may happen as soon as a multiplication is not linear. Then, instead of the interpreted function $times(x, y)$, an uninterpreted function $f_{\text{non\_linear\_mult}}(x, y)$ will be used.

Chapter 4

Equational Reasoning

The equality predicate is at the very heart of most theories and virtually none can do without it. Thus there has been a lot of sophistication in this area and many mature calculi can directly reason with it. Unfortunately, in the current version of DARWIN, equality is only handled in a naive axiomatic way that does not scale well. [BT05] introduce the Model Evolution calculus with equality, however, an implementation in DARWIN is not yet available but due soon after this work. Thus to avoid duplication, no effort was put into this aspect and instead we used a simple approximation of a rewrite-based approach to get results without any changes to the calculus that would be obsoleted by the proper implementation. We expect the result to improve significantly then.

4.1   Axiomatic Approach

The naive approach to handling equations would be to add axioms to the set of clauses interpreting the equality predicate to have reflexivity, symmetry and transitivity of the equality predicate:

$$\forall x \quad x = x \tag{4.1}$$

$$\forall x \, \forall y \quad (x = y) \rightarrow (y = x) \tag{4.2}$$

$$\forall x \, \forall y \, \forall z \quad (x = y) \wedge (y = z) \rightarrow (x = z). \tag{4.3}$$

We would then need another axiom for each function and each argument for the functional substituitivity inside the function's arguments

$$\forall x \, \forall y \quad (x = y) \rightarrow (f(\dots, x, \dots) = f(\dots, y, \dots)) \tag{4.4}$$

and an analogous set of equations for each predicate symbol.

This is how equational reasoning was implemented in the current version of DARWIN. However, this approach does not scale at all. The substituitivity clause (4.4) allows arbitrary assertions and the context can be grown without bounds. This problem surfaces when for example a clause like

$$f(a) = a \qquad\qquad (4.5)$$

is present. One can then assert

$$f^n(a) = a$$

for any n and the calculus might not terminate the derivation.

This weakness in reasoning with the so-called congruence axioms (4.1)–(4.4) does not only occur in Model Evolution or more generally in instance-based methods, but also in most other first-order calculi. This led to the development of better methods that do not admit to that problem.

4.2   Rewrite-Based Equational Reasoning

The idea of rewrite-based equational reasoning is to break the symmetry and instead see equations as unidirectional rewrite rules on terms preventing the assertion of literals that are by some measurement greater than the literals they are inferred from. Equation (4.5) would then be replaced by a rewrite rule

$$f(a) \Rightarrow a$$

that does only allow $f(a)$ to be replaced by a, but not the other direction, avoiding the problem of producing terms of arbitrary depth.

Instead of dealing directly with a set of rewrite rules, calculi are defined with a so-called reduction ordering on terms. Following [BG98], this is a well-founded partial ordering $\succeq$ where $s \succeq t$ implies $s\sigma \succeq t\sigma$ for all substitutions σ and terms s and t. In a well-founded ordering every non-empty ordered subset has a least element. A reduction ordering induces a relation on a set of terms that is well-founded, if the reduction ordering is. It is easy to see that a well-founded reduction ordering $\succeq$ is essential for the termination of rewriting.

For each equation, rewriting is then allowed in the direction that agrees with the reduction ordering. See [BG98] or [NR01] for surveys of different approaches to rewrite-based equational reasoning.

Looking at the resolution calculus where this approach to equational reasoning is easy to demonstrate due to the simple rules of inference, the basic equational rule is the paramodulation. In other calculi, especially in the Model Evolution calculus of [BT05], it is quite similar:

$$\frac{C \vee s = t \qquad D \vee u[s'] = v}{(C \vee D \vee u[t] = v)\sigma} \qquad\qquad (4.6)$$

where σ is the most general unifier of s and s'. In the paramodulation resolvent the occurrence of s' in u is replaced by t. Without any side-conditions, this rule admits to exactly the same problem of arbitrarily growing terms as discussed above.

Therefore the ordered paramodulation imposes the constraint that $t\sigma \not\succeq s\sigma$ and s' is not a variable which is equal to applying a rewrite rule $s \Rightarrow t$. The superposition rule adds the constraints $v\sigma \not\succeq u\sigma$ and $(s = t)\sigma \not\succeq (u = v)\sigma$ to allow rewriting only on the greater side of the greater of the two equations.

The soundness of ordered paramodulation and superposition follows from the soundness of the paramodulation rule and if either of the rules is implemented properly by respecting certain fairness conditions, the resulting calculus will be complete. For details refer to [BG98], the Model Evolution calculus with equality by [BT05] uses ordered paramodulation.

Finding the right reduction ordering is crucial and can give quite an advantage. It is even possible to choose reduction orderings that are compatible with features like associativity and commutativity of function symbols and allow to lift these properties to the equational reasoning supporting a part of a theory already there, see [NR01] for further reading.

4.3 Approximating Approach

The approach followed here to improve the naive equational reasoning is operational without touching the calculus of Model Evolution. We only change some literals in the clauses set and add certain axioms. This preprocessing is sufficient to quite successfully deal with equations though not complete in every case.

In a first step the binary equality predicate is replaced by a ternary predicate that has the sort of its terms in its first argument. This can greatly reduce the search space by blocking unification of equations on terms of different sorts and thus restricting the Model Evolution calculus to reason only on equations of the same sort that it would otherwise not be aware of.

Then, all reasoning with equations is transferred to a ternary rewrite predicate rew that models the rewrite relation $\Rightarrow$ between two terms and keeps the sort of the terms in its first argument as the equality eqs does. The problem clauses do keep the occurrences of the equality predicate eqs, but the axioms of the background theory will be formulated using rewrite predicates instead of equality.

The axiom

$$\forall t \, \forall x \, \forall y \; \mathrm{eqs}(t,x,y) \rightarrow \mathrm{rew}(t,x,y) \vee \mathrm{rew}(t,y,x) \tag{4.7}$$

allows to assert the rewrite rule

$$x \Rightarrow y$$

or

$$y \Rightarrow x$$

from an equation and the calculus will choose whatever it seems apt in the situation while it may as well assert both rules. By the rationale of a reduction ordering, the rewriting should be restricted to a certain direction, especially in the presence of a not range-restricted clause like (4.5). In that case adding a unit clause

$$\neg \, \mathrm{rew}(Int, a, f(a)) \tag{4.8}$$

blocks asserting the rewrite rule

$$a \Rightarrow f(a).$$

However, this has to be done manually by inspection of the problem, there is currently no algorithm for it. Systematically generating these clauses would involve having a reduction ordering that is also needed for ordered paramodulation or superposition reasoning available. As this will be implemented in a more sophisticated way for the next version of DARWIN, it was chosen not to focus on this aspect and rather to leave the task to a user who does not need to add any restriction on the rewrite predicate.

The actual application of rewrite rules is governed by substituitivity clauses

$$\forall t \; \forall x \; \forall s \; P(\dots, x, \dots) \wedge \mathrm{rew}(t, x, s) \rightarrow P(\dots, s, \dots) \tag{4.9}$$

for each predicate P and each argument of it where t is the sort of x and s.

To support this approach, positive and negative occurrences of equations become different predicates eqs and neqs, respectively. This blocks reasoning with equality predicates of different sign and prevents assertions of rewrite predicates from negative equations. Clauses may look like

$$\{ \, \mathrm{eqs}(Int, \mathrm{plus}(n1, n1), n2) \, \}$$

and

$$\{ \, \mathrm{neqs}(Array, \mathrm{store}(a, n1, n0), \mathrm{store}(a, n1, n1)) \, \}.$$

The axiom

$$\forall t \; \forall x \; \forall y \; \neg(\mathrm{eqs}(t, x, y) \wedge \mathrm{neqs}(t, x, y)) \tag{4.10}$$

is needed to detect contradictions that would otherwise be evident due to the different signs of the equality predicates. The clauses

$$\forall t \; \forall x \; \mathrm{eqs}(t, x, x) \tag{4.11}$$

$$\forall t \; \forall x \; \neg \, \mathrm{neqs}(t, x, x). \tag{4.12}$$

make equality reflexive and unequality irreflexive. These unit clauses will mostly be used to close derivation branches.

4.3.1 Properties of Equality

From equations (4.7), (4.9), (4.10), (4.11) and (4.12), the three properties reflexivity, symmetry and transitivity of the eqs predicate can be inferred in the following ways.

Reflexivity This is already an axiom in (4.11).

Symmetry The premiss

$$\mathrm{eqs}(t, x, y) \tag{4.13}$$

holds, therefore after applying the axiom (4.7)

$$\mathrm{rew}(t, x, y) \tag{4.14}$$

or

$$\mathrm{rew}(t, y, x) \tag{4.15}$$

must be true. If one assumes that eqs is not symmetric, then

$$\mathrm{neqs}(t, y, x) \tag{4.16}$$

holds, hence using substituitivity (4.9) for neqs we get either

$$\mathrm{neqs}(t, y, y) \tag{4.17}$$

from (4.14) or

$$\mathrm{neqs}(y, x, x), \tag{4.18}$$

from (4.15) both contradicting irreflexivity of neqs (4.12). □

Transitivity We use a similar argument and start with the premiss of

$$\mathrm{eqs}(t, x, y) \tag{4.19}$$

and

$$\mathrm{eqs}(t, y, z). \tag{4.20}$$

From the second clause (4.20) the rewrite relations

$$\mathrm{rew}(t, y, z) \tag{4.21}$$

or

$$\mathrm{rew}(t, z, y) \tag{4.22}$$

can be asserted from (4.7). In the first case of (4.21) using substituitivity (4.9) of eqs we get the goal

$$\mathrm{eqs}(t, x, z). \tag{4.23}$$

In the second case (4.22), we can assert

$$\text{rew}(t, x, y) \tag{4.24}$$

or

$$\text{rew}(t, y, x) \tag{4.25}$$

from the first clause (4.19). In the latter case (4.25), substituitivity (4.9) of eqs applied to the second clause (4.20) results in our goal

$$\text{eqs}(t, x, z) \tag{4.26}$$

again. Returning to the first case (4.24), we assume the negation of the goal

$$\text{neqs}(t, x, z) \tag{4.27}$$

and substituitivity (4.9) of neqs yields

$$\text{neqs}(t, y, z) \tag{4.28}$$

being contrary to (4.20).   □

Functional Substituitivity   does not hold with this formulation. It has to be axiomatized for each function f and each predicate P with clauses like

$$\forall x \, \forall s \, P(\ldots, f(\ldots, x, \ldots), \ldots) \wedge \text{rew}(t, x, s) \rightarrow P(\ldots, f(\ldots, s, \ldots), \ldots) \tag{4.29}$$

where t is the sort of x and s. This would clearly yield too many additional clauses and functional substituitivity will be introduced by other means than these explicit clauses.

As it will be discussed in detail and examples later, axioms for arithmetic will be given so that assertions with rewrite rules can most often be made separately from the predicate that contains an arithmetic term. A final step will then eliminate the term inside the predicate by the substituitivity axiom (4.9).

For the array theory the select and store functions will be turned into flat functions and then relations thus assertions can be made using predicate substituitivity (4.9). An approach to equality by flattening terms was presented as early as [Bra75] and already proved sound there. We will discus this further in section 5.2.1.

4.3.2   Paramodulation and Superposition

Returning to the superposition and paramodulation rule (4.6) from section 4.2, our operational rewriting can partly approximate it via the substituitivity in (4.29). Each equation $s = t$ can be turned into a rewrite rule

rew(p, s, t) or rew(p, t, s) with p being the sort of s and t. In contrast to proper equational reasoning where both possible directions of rewriting are considered and the reduction ordering selects one, the direction of the rewrite rule has to be set by the user. Adding clauses restricting rewriting like (4.8), the effect of a reduction ordering $\succeq$ can be approximated up to a certain extent and the constraint $t\sigma \not\succeq s\sigma$ can be formulated within limits. There is, however, no possibility to enforce other constraints like ordering on the equation $u = v$ on the right side or ordering on both $s = t$ and $u = v$, making the calculus at most an ordered paramodulation.

4.3.3 Model Evolution and Equality

The Model Evolution calculus with equality by [BT05] uses ordered paramodulation and asserts new equations only from equations that are in the context. If only well-sorted equations occur in the clause set, no ill-sorted equation can enter the context and every inferred equation will be well-sorted. Using a ternary equality predicate carrying its sort as in the current approach is then not necessary anymore.

The performance of the equational reasoning will certainly increase compared to the approximation presented here as ordered paramodulation is not operationally defined but implemented in the calculus with a complete reduction ordering that applies to each equation and does not require the user to specify unwanted directions of rewriting.

Chapter 5

Theory Reasoning

As well as the naive approach to handling equality by adding axioms to the set of problem clauses was bound to fail, an as naive approach will not work for theories either. In this case it is even worse due to the well-known incompleteness and undecidability theorems about integer arithmetic.

The target background logic *AUFLIA* has only recently been introduced as a division in the SMT-COMP competition. The two solvers, Yices[1] and CVC3[2] that took part in it were both employing a first-order version of the ground propositional DPLL($\mathcal{T}$) procedure that is described below.

For the purposes here, again the Model Evolution calculus was not modified but instead theory axioms were added in an operational way to the set of problem clauses that were preprocessed to be able to reason heuristically in certain cases while being incomplete in general.

5.1   Ground Propositional Clause Sets

The DPLL($\mathcal{T}$) approaches that will be presented work on ground propositional clause sets and reason with ground propositional rules as DPLL does. There are approaches to first-order clause sets, but they deal with ground instantiations and still use ground propositional inferences. The Model Evolution calculus is a lifting of the ground propositional DPLL procedure to first-order logic, working not with ground instances, but with first-order inferences and models enabling it to make use of the term structure. It would seem plausible that ground propositional DPLL($\mathcal{T}$) procedures for satisfiability modulo theories could similarly be lifted to first-order and it looks like a promising and natural way to incorporate reasoning modulo theories to the Model Evolution calculus.

Two other ground propositional approaches were considered for possibilities of lifting them to first-order. The first, implemented in MATHSAT,

---

[1]see [DdM06]
[2]see [BNOT06]

is quite similar to DPLL($\mathcal{T}$) and features a very sophisticated incremental and layered handling of theories. The second approach is based on using interpolants to alternatively under- and over-approximate the problem to iteratively find a statement about satisfiability.

All three approaches are quite successful in the ground case and though they seemed sufficiently similar to Model Evolution, no promising start to lift any of them to first-order logic was found that would lead to a feasible implementation in the scope of this work.

5.1.1   DPLL($\mathcal{T}$) and DPLL(X)

DPLL($\mathcal{T}$) as it was first conceived by [Tin02] operates on quantifier-free ground clauses of first-order logic with $\mathcal{T}$ being a theory, as usual a satisfiable set of closed formulas. There exist quite some variants of the original algorithm and it has also been extended to work on quantified problems and implemented in Yices and CVC3, for example.

Incorporating theories into a DPLL procedure can be done in two extreme ways. The so-called eager approach tries to use information from the theory as often as possible. This was done in [Tin02] by changing the side-conditions of the DPLL calculus rules, recall section 1 for a short introduction. Each condition involving the membership of a literal in the context ($l \in \Lambda$) was replaced by a corresponding condition about entailment in the theory of that literal by the context ($\Lambda \vDash_{\mathcal{T}} l$). With the changed calculus rules, the theory solver guides the derivation in each step. On the downside, this eager approach precludes modularity as each theory gives rise to a new calculus. Worse yet, many optimizations in state-of-the-art DPLL solvers conflict with this approach and can thus not be used.

The lazy approach on the other hand first abstracts from the theory by treating literals with an interpretation in the theory as purely propositional symbols and searches for models of the abstracted clause set. Each model is then refined by reinterpreting the abstract literals and checked for satisfiability in the theory. If the model for the abstracted clause set is invalid in the theory, a lemma is added to the clause set precluding that model and the derivation is restarted to find a different model until either a model is satisfiable in the theory or no more models can be found. This approach is very flexible as the DPLL and theory solver are coupled only loosely, both working independently of each other and being able to make use of advanced approaches in their own domain. However, the DPLL solver cannot use information available from the theory to guide its derivation and requires a restart with clauses added after each model that is invalid in the theory.

There exist, of course, many flavors between the lazy and the eager approach, they are caught by the description of Abstract DPLL Modulo Theories in [NOT06].

The DPLL($\mathcal{T}$) approach as presented in [GHN$^+$04] combines the eager and lazy approaches and their respective advantages. It consists of a general DPLL engine DPLL(X) that is independent of a theory (X being a placeholder for any theory) but interacts with a solver for a specific theory $\mathcal{T}$ via an interface.

All literals and clauses are purely propositional to the DPLL(X) part, all theory inferences are executed by the theory solver. The DPLL(X) procedure abstracts from the theory just as the lazy approach does, but its derivation is guided by the theory solver as in the eager approach. The theory solver is running in parallel to the DPLL(X) procedure and, glossing over details, DPLL(X) notifies the theory solver when it has set an abstract literal to true in its context and notifies it again when an abstract literal is removed from the context due to backtracking. The theory solver can be requested to check the consistency of a context in the theory $\mathcal{T}$ and to provide literals that are entailed by the theory.

With this approach, the DPLL(X) procedure can employ sophisticated optimization techniques on the propositional structure of the problem, but asking for information from the theory solver, it can also get some guidance to exploit properties of the theory. Of course, it is at the discretion of the DPLL(X) solver, when to cooperate with the theory solver and when to reason propositionally. Thus there are some degrees of freedom given to the implementation.

Yices as described by [DdM06] is a Simplex-based solver that uses the DPLL(X) framework. [GBT07] show an approach to handle quantified formulas that is implemented in CVC3. A new calculus rule allows adding an instance of a quantified formula to the clause set if the quantified formula has been asserted in the context. Though completeness can be achieved by doing the instantiation in a fair way, it is not efficient and instead done by a strategy involving triggers that is successful in the Simplify prover [DNS05].

5.1.2 Lifting DPLL($\mathcal{T}$) Approaches

Considering a lifting of any of the presented DPLL($\mathcal{T}$) approaches to first-order logic, we found challenges that were well beyond the scope of this work and were thus not attempted.

The lazy approach would require the least amount of changes to the Model Evolution calculus. In fact, the derivation procedure does not have to be changed. It is first necessary to find an abstraction of the theory literals in the first-order clause set the Model Evolution calculus provides models for. Any such first-order model then has to be represented with the abstracted literals refined and passed to a solver capable of deciding validity of the model representation modulo the theory. In the case of invalidity,

clauses have to be added to preclude the model and the procedure has to be restarted.

In the abstraction, care has to be taken for quantification. A literal that contains a quantified variable cannot be abstracted to a propositional constant but must instead become a first-order predicate that retains the quantification. For instance, this can be done by replacing the theory predicate $\leq$ in

$$\forall x \ \mathrm{minus}(x, 1) \leq x$$

by a new predicate $P(x)$ where x is still quantified

$$\forall x \ P(\mathrm{minus}(x, 1), x).$$

The show-stopper for the lazy approach and other approaches requiring an abstraction as well, is the first-order model produced by the Model Evolution calculus and its representation. The lazy approach needs to have the validity of a model in the theory checked, the eager approach requires entailments in the theory from the context and the DPLL(X) procedure may need both.

First-order contexts describe Herbrand interpretations, as usual a set of ground atoms. The so-called atomic representation of models (ARM), see [GP98], is a finite set of not necessarily ground atoms whose ground instances are to be evaluated to true. Contexts in the Model Evolution calculus are equivalent to representing models as disjunctions of implicit generalizations (DIG), see [FP05]. Similar to the ARM, a DIG is a finite set $\mathcal{A}$ of atoms, but with each atom having a finite set $\mathcal{B}$ of atoms, its exceptions. A ground atom is then to be evaluated to true if it is an instance of an atom in $\mathcal{A}$ but not an instance of any of its exceptions $\mathcal{B}$. Models represented as a DIG are stronger than an ARM but there are still certain infinite interpretations both are unable to represent.

To use a theory solver efficiently, the context has to be represented in a way that is easily tractable without requiring powerful reasoning. A representation of a context as a first-order formula will not suffice as the theory solver would then need to support quantifiers which would in turn enable it to solve the input formula itself thus obsoleting the Model Evolution calculus in the approach.

For example, take a formula like

$$\begin{aligned}
&\forall x \ (0 \leq x) \wedge (x \leq n) \rightarrow (\mathrm{select}(a, x) = \mathrm{plus}(e, e)) \\
&\wedge \\
&\exists x \ (0 \leq x) \wedge (x \leq n) \rightarrow (\mathrm{select}(a, x) \neq \mathrm{times}(2, e))
\end{aligned} \tag{5.1}$$

that is apparently unsatisfiable as the array a cannot have an index where its value equals $2e$ when it has the value $e + e$ at each index. Abstracting

the theory predicate $\leq$ in the four literals where it occurs and skolemizing the second line leads to

$$\forall x\ P_1(x) \wedge P_2(x) \rightarrow \text{select}(a, x) = \text{plus}(e, e)$$
$$\wedge \tag{5.2}$$
$$P_3 \wedge P_4 \rightarrow \text{select}(a, c) \neq \text{times}(2, e)$$

being

$$\{\ \neg P_1(x),\ \neg P_2(x),\ \text{select}(a, x) = \text{plus}(e, e)\ \} \tag{5.3}$$

$$\{\ \neg P_3,\ \neg P_4,\ \neg \text{select}(a, c) = \text{times}(2, e)\ \} \tag{5.4}$$

in CNF that is satisfiable by a model that can be represented by the set of formulas

$$\forall x\ \text{select}(a, x) = \text{plus}(e, e) \tag{5.5}$$

$$\neg \text{select}(a, c) = \text{times}(2, e) \tag{5.6}$$

and is invalid in the theory of integers and arrays where

$$\text{plus}(e, e) = \text{times}(2, e)$$

holds.

This simple example already shows why the model representation by first-order formulas is unsuitable. It needs to specify the truth value of each atom and not only the positive literals in the context. Leaving out the second formula (5.6) in the model representation would give a valid model to the theory solver. Thus the model representation would actually also have to include the formulas

$$\forall x\ \neg P_1(x) \equiv \forall x\ \neg 0 \leq x \tag{5.7}$$

$$\forall x\ \neg P_2(x) \equiv \forall x\ \neg x \leq n \tag{5.8}$$

$$\neg P_3 \equiv \neg 0 \leq c \tag{5.9}$$

$$\neg P_4 \equiv \neg c \leq n \tag{5.10}$$

explicitly setting the truth value of the implicitly negative atoms and thus the truth value of each ground atom.

Though the set of formulas representing the model has a certain structure that might be exploited, it still needs a theory solver capable of handling quantifiers. In the presented example it seems just as hard for the

theory solver to check the validity of the model representation (5.5)-(5.10) as the satisfiability of the original problem (5.1).

But the difficulties do not cease then. After the model has been shown to be invalid by the theory solver, a set of formulas has to be generated and added to the clause set to preclude this model. Then the procedure would have to be restarted to find another model. Assuming that a suitable representation of formulas to preclude a certain model existed as well as a representation of contexts tractable for a theory solver, termination of this procedure is, of course, not guaranteed. There might, of course, be infinitely many models and it is quite certain that one would not get a decision procedure from this approach.

To avoid restarting after a model has been found to be invalid in the theory, one could try to modify the proof search in DARWIN to enumerate all models instead of terminating after a model has been found. The derivation would then backtrack to a recent choice point and continue searching for other models. But it is neither here guaranteed that the search would terminate eventually, as the number of models might be infinite. Adding to that, DARWIN employs quite sophisticated search techniques like backjumping that make a tighter coupling with the theory solver as implemented in DPLL(X) more complicated. This was already noted by [Tin02] saying that "the optimization strategies used in DPLL-based SAT solvers do not immediately lift to satisfiability modulo theories".

The eager as well as the DPLL(X) approach require changes to the calculus. For the eager approach in [Tin02], the DPLL assert rule (1.1)

$$\frac{\Lambda \vdash \Phi, \{l\}}{\Lambda, l \vdash \Phi, \{l\}} \qquad \text{if } l \notin \Lambda \text{ and } \neg l \notin \Lambda$$

becomes

$$\frac{\Lambda \vdash \Phi, \{l\}}{\Lambda, l \vdash \Phi, \{l\}} \qquad \text{if } \Lambda \nvDash_{\mathcal{T}} l \text{ and } \Lambda \nvDash_{\mathcal{T}} \neg l. \tag{5.11}$$

by replacing the side-condition about membership $\in$ of literals in the context by entailment from theory $\nvDash_{\mathcal{T}}$ of that literal by the context. The corresponding assert rule in the Model Evolution calculus from [BT03] is

$$\frac{\Lambda \vdash \Phi, \{l\}}{\Lambda, l \vdash \Phi, \{l\}} \qquad \text{if } \nexists k \in \Lambda : k \geq l \text{ and } l \text{ is not contradictory with } \Lambda.$$

$$\tag{5.12}$$

and would have to be changed to restrict asserting literals to entailments from the context Λ. This requires a representation of a context Λ that has to be tractable for a theory solver and we are facing the same problems here that already struck us in the discussion of representing a model for the lazy approach. Furthermore, we have to deal with a side-condition more complicated than in the ground DPLL procedure. We have a literal l being an instance of a literal k in the context and a more complicated notion of

contradiction with a context that can only be hinted at for the scope of this work.

The less eager approach of DPLL(X) with its more loose coupling with the theory solver will admit to a mixture of the problems from the lazy and the eager approach.

Having said that, this direction does certainly not look like a dead end and there might be an answer to these questions enabling a lifting of DPLL techniques for theory reasoning to first-order logic. The main focus of research on that field would have to be in finding model representations and having a theory solver working on them.

5.1.3 MATHSAT

A procedure very similar to DPLL($\mathcal{T}$) is presented in [BBC+05]. It is a decision procedure for Linear Arithmetic Logic that is defined there as consisting of "boolean combinations of propositional variables and linear constraints over numerical variables". The logic is quantifier-free and satisfiability is checked by a DPLL procedure. Theory predicates are abstracted in a lazy way as described above.

The approach is layered in that it calls solvers of different generality to check the consistency of a model in a theory. The theory of equality is more general than the theory of real arithmetic that is in turn more general than the theory of integer arithmetic. Thus a solver for equality is called first that sees predicates from other theories as propositional. If the model is valid in that theory, a solver for real arithmetic is called. An inconsistency on real numbers holds on integers as well, but is easier to detect. If the model is still valid with real arithmetic, a solver for integer arithmetic is called last in the chain.

The procedure being similar to DPLL($\mathcal{T}$) leads to similar problems to lift it to first-order logic. The layered approach to solving satisfiability modulo theories could, however, be quite easily transferable to our first-order case and would certainly increase the performance there.

5.1.4 Approximation

A different and yet possibly usable approach is shown in [LM05], it is applicable to quantifier-free Presburger arithmetic (QFP). This theory has the advantage of a small-model property, i. e. if a formula is satisfiable, it has a finite model in a domain determined by the formula itself with an upper bound for the size of the model that can be calculated from the formula. Taking advantage of that, a polynomial encoding of integer variables and arithmetic in a QFP formula Φ to a purely Boolean formula in a finite domain that does not contain theory terms is defined. A theory abstraction

Φ_b of Φ that replaces theory predicates by new predicates just as in the previous sections is used as well.

The algorithm then alternatively under- and overapproximates the formula Φ. If the underapproximation is satisfiable, then Φ is and analogously if the overapproximation is unsatisfiable, then Φ also is. Starting with a domain size smaller than the upper bound for the formula's domain, i. e. an underapproximation, Φ is encoded to a Boolean formula Φ_u and then checked for satisfiability by a SAT solver. If it is unsatisfiable, a Craig interpolant between the theory abstraction Φ_b and Φ_u that is also free from theory terms is computed containing only variables common to both. This overapproximation is then checked for satisfiability. If it is, a conflict clause generated from the model is added and another underapproximation for a greater domain size is generated. The algorithm stops and returns unsatisfiable when either the overapproximation is unsatisfiable or the domain size becomes greater than the upper bound for the size of the model determined from the formula. Satisfiability is returned when the underapproximation is found satisfiable.

Challenges in a lifting of this approach include the interpolation as Craig's interpolation theorem does not directly hold in first-order logic, see e. g. [Bor05]. A suitable alternative formulation has to be found from the existing ones in the literature. For the application of this work, the background theory is quite different from QFP and it does not enjoy the small model property, neither can a finite model be assumed. Both under- and overapproximation would have to be different and again a thorough inspection or implementation of this approach is well outside the scope of this work.

5.2   Incomplete theory reasoning

As no approach could be conceived that would easily enable the Model Evolution calculus to reason with theories, we resorted to incompleteness and heuristics. The clause set is transformed and joined by particularly formulated theory axioms that allow to prove certain problems. In general, our approach will be incomplete.

Our approximation of rewrite-based equational reasoning does not admit to functional substituitivity. Therefore, the array functions select and store are turned into relations to regain this property. We do not have functional substituitivity for arithmetic functions neg, plus and times thus preventing them from being substituted in functional terms. Instead, rewriting rules have to be asserted separately, where the approximation of the reduction ordering can control the terms being generated. Refutation then has to be done in a final step via predicate substituitivity.

We first describe the transformations on the clause set that turn it into

a canonical form enabling reasoning with equations and theories. The formulation of the theory is then discussed and some examples are presented to demonstrate the various effects and advantages making our heuristic approach successful in the considered cases.

5.2.1 Canonical preprocessing

[BFdNT07] describe transformations on clauses that reduce a clause set to function-free clause logic and then use the Model Evolution calculus to find finite models in that logic. The transformations were implemented in DARWIN and the implementation in this work builds on them with some modifications.

Definitions and abstractions   At certain points in the preprocessing, a term has to be replaced by a constant or a variable depending on if the term is ground or contains universal variables.

Ground terms are "defined" by a fresh constant whose meaning is introduced by an added unit clause. The ground term t to be defined is replaced by a fresh constant c_t (not necessarily at each occurrence in the clause set) and the clause

$$\{\, \text{eqs}(s, t, c_t) \,\}$$

is added to define the constant c_t as a synonym for t. The definition of t by c_t can, of course, be reused in any clause of the clause set if needed.

If the term to be replaced contains a universally quantified variable, it is "abstracted" to a fresh universally quantified variable that is defined by adding a new literal to the same clause. Again, the term t containing variables is replaced at not necessarily each occurrence now by a universally quantified variable u_t and the literal

$$\text{neqs}(s, t, u_t)$$

is added to the clause t has been abstracted in. Therefore abstractions are local to each clause and can not be reused anywhere else in the clause set.

Both operations do not change satisfiability of a clause set Φ.

Soundness of Definitions   To show that defining a term does not change the satisfiability of a clause set Φ, we show that the modified clause set Φ' has a model $\mathcal{M}'$ if the original clause set Φ has a model $\mathcal{M}$. Let $\mathcal{M}$ be the Herbrand model of Φ where every ground term is interpreted with itself. Introducing the fresh constant c_t extends the signature of Φ and adds to it the clause $\{\, \text{eqs}(s, t, c_t) \,\}$. This makes c_t and the ground term t synonymous in $\mathcal{M}'$. We can then undo the definitions of t by c_t, therefore the clause set Φ' has a model if the original clause set Φ has a model.

Completeness cannot be proven as our axioms for the eqs predicate only approximate a congruence relation. As our argument shows, definitions are only allowed for ground terms as it has to assume a ground Herbrand model. For non-ground terms, an abstraction has to be used.

Soundness of Abstractions   An abstraction of a clause Φ to a new clause Φ' does not modify the signature of the clause. We show that Φ' is valid in every model that satisfies Φ, not considering the whole clause set as in the previous proof but only the one modified clause. Both clauses are in skolemized negation normal form and in the second clause the occurrence of the term t in the quantifier free formula Ψ has been abstracted to the universally quantified variable u.

The original clause

$$\Phi \equiv \forall x_1 \ \ldots \ \forall x_n \ \Psi[t]$$

becomes abstracted to

$$\Phi' \equiv \ \forall u \ \forall x_1 \ \ldots \ \forall x_n \ \text{neqs}(s,u,t) \lor \Psi[u]$$

which can be seen as an implication

$$\forall u \ \forall x_1 \ \ldots \ \forall x_n \ \text{eqs}(s,u,t) \rightarrow \Psi[u].$$

Assuming again a Herbrand model $\mathcal{M}$ for Φ, by definition it is also a model for all ground instances of Ψ where the universally quantified variables u, x_1, ..., x_n have been instantiated by the ground terms u', t_1, ..., t_n.

$$\mathcal{M} \vDash \text{eqs}(s,u',t) \rightarrow \Psi[u][u \mapsto u', \ x_1 \mapsto t_1, \ \ldots, \ x_n \mapsto t_n]$$

With the premiss of $\text{eqs}(s,u',t)$ we replace the term u' that has been substituted for u by t

$$\mathcal{M} \vDash \Psi[t][x_1 \mapsto t_1, \ \ldots, \ x_n \mapsto t_n]$$

which is by definition valid if $\mathcal{M}$ is a model for Φ which has been our assumption. Thus we have shown that every model of a clause Φ is also a model of the clause Φ' where an occurrence of the term t has been abstracted to a universal variable. Abstraction as presented here is sound, but as well as definitions, it is not complete due to the incomplete implementation of equational reasoning.

Equalities   The first step in the transformation that uses these definitions and abstractions processes equalities. First, negative occurrences of either the positive equality predicate eqs or the negative equality predicate neqs are turned into positive occurrences of neqs or eqs, respectively. If one of

the terms in an equation contains an array function that is to become a flat relation, the equation is oriented so that the function term occurs on the left-hand side. If both terms are to be flattened, they are replaced by either a definition or an abstraction.

If e. g. f is an array function, the clause

$$\{ \neg \, \text{eqs}(t, f(a), f(x)) \}$$

with x being a universal variable becomes

$$\{ \, \text{neqs}(t, c_{f(a)}, u_{f(x)}), \, \text{neqs}(t, f(x), u_{f(x)}) \, \}$$

with a new universal variable $u_{f(x)}$ defined in the new second literal and a new constant $c_{f(a)}$ that is defined in the additional clause

$$\{ \, \text{eqs}(t, f(a), c_{f(a)}) \, \}.$$

After this step all equality predicates eqs and neqs only occur positively and in each equality there is at most one function term that is to be made relational and it is always on the left-hand side.

Satisfiability is obviously preserved when changing negative occurrences of equalities into positive ones and, as equality is a symmetric relation, certainly also when orienting them. Replacing terms by definitions or abstractions has been shown to be valid before.

Flattening of functions   In the next step all function terms for array functions are flattened, i. e. each occurrence of these function symbols is replaced by a definition or an abstraction if the term is ground or contains universal variables, respectively. An exception holds for positive or negative equalities eqs and neqs. If the left-hand side is a function term with an array function, after the equality transformations the right-hand side can not be another function term that is to be flattened. The equation is then kept as it has exactly the form of a definition or abstraction. All occurrences of array functions inside this term will, of course, be defined or abstracted recursively. Ultimately, there will be no occurrence of an array function as argument of a function or predicate except in equations for definitions or abstractions.

Thus, a clause

$$\{ \, \text{eqs}(t, f(x, f(a, b)), g(y)), \, \text{neqs}(t, g(f(a, b)), y) \, \}$$

with the universal variables x and y where f is an array function becomes

$$\{ \, \text{eqs}(t, f(x, c_{f(a,b)}), g(y)), \, \text{neqs}(t, g(c_{f(a,b)}), y) \, \}$$

and the definition clause

$$\{ \, \text{eqs}(t, f(a, b), c_{f(a,b)}) \, \}$$

that is used in both literals.

Again, defining or abstracting terms is valid and no other transformation is done, therefore flattening of functions still preserves the satisfiability of the original clause set.

Turning functions into relations Now all array functions only occur in terms at the top level and only in positive or negative equations eqs and neqs on the left-hand side. They can be made relational by replacing the equalities by a predicate whose arity is greater than the arity of the function by one. The last argument of the predicate is the right-hand side of the equation, the sort of the equality is lost in that transformation. Negative equalities become a negative literal, positive equalities a positive one.

The clauses

$$\{ \text{eqs}(t, f(x, c_{f(a,b)}), g(y)), \ \text{neqs}(t, g(c_{f(a,b)}), y) \}$$

$$\{ \text{eqs}(t, f(a, b), c_{f(a,b)} \}$$

now become

$$\{ \text{r\_f}(x, c_{f(a,b)}, g(y)), \ \text{neqs}(t, g(c_{f(a,b)}), y) \}$$

$$\{ \text{r\_f}(a, b, c_{f(a,b)} \}$$

if f is an array function.

Following [BFdNT07], turning functions into relations preserves satisfiability if there is an axiom to enforce left-totality of the relation:

$$\forall x_1 \ \ldots \ \forall x_n \exists y \ \text{r\_f}(x_1, \ldots, x_n, y).$$

This is skolemized to the clause

$$\{ \text{r\_f}(x_1, \ldots, x_n, c_{\text{r\_f}}(x_1, \ldots, x_n)) \}. \tag{5.13}$$

Totality of relations However, this formulation of the totality axiom admits to arbitrary instantiations of x_1, $\ldots$, x_n that may prevent the calculus from terminating. For instance, when the select array function is made relational, the axiom

$$\{ \text{r\_select}(a, i, \text{select\_val}(a, i)) \} \tag{5.14}$$

with the universal variables a and i of sort *Array* and *Int*, respectively, admits to infinitely many asserts when arbitrary *Int* values can be generated by an axiom. They then give rise to infinitely many new arrays from a finite initial number where different values are stored at different indices.

Assuming that there is a finite number of "interesting" indices that are relevant in the derivation and can be extracted from the initial problem, an

index predicate is introduced and the totality axiom is changed to enforce totality only at a relevant index:

$$\{ \text{ r\_select}(a, i, \text{select\_val}(a, i)), \neg\, \text{index}(i) \,\} \qquad (5.15)$$

Indices are considered to be relevant where arrays are read or written. Thus for each ground term t that occurs as the second argument in a select or store function, a unit clause

$$\{ \text{ index}(t) \,\}$$

is added, making select total only at those indices. The index predicate is only added for ground terms, not for universally quantified terms as then new index predicates could be asserted circumventing the restriction of totality to a finite number of arguments.

A similar approach is used by [BMS05] to build a decision procedure for a fragment of integer array theory. They restrict the formulas to array properties that are defined as implications of an expression about the indices of an array (the index guard) and an expression about the value of an array. The index guards may be formulated in Presburger arithmetic, but nested array reads in the value expression are not allowed, making their logic fragment much less expressive than the *AUFLIA* logic. However, they achieve a decision procedure by instantiating universally quantified variables only where they appear as an index of an array.

Unfortunately, they show that extending their logic fragment by allowing nested reads in an array or more general arithmetic expressions in the index guard make satisfiability undecidable. Therefore one can certainly expect that the *AUFLIA* logic with the totality axiom for array reads restricted to a finite number of interesting indices will be incomplete, but nevertheless our assumption is justified as a sensible one and indeed, it did hold in many examples that were tried.

Canonical transformations   To finally get the canonical form of the clause set, the relations $\geq$ and $>$ are eliminated by reducing them to their duals $\leq$ and $<$ and negative occurrences of them are then replaced by positive occurrences of $\not\leq$ and $\not<$, similar to the approach to handling negative equality predicates in the last chapter.

Further, every constant c in a negated literal in a clause is abstracted to a new universally quantified variable x_c that is defined by adding the literal $\text{neqs}(t, c, x_c)$ to the clause as described before. By means of an example this has the effect of reducing equality to unification: the clauses

$$\{ \, P(\text{plus}(1, 1)) \, \}$$

$$\{ \, \neg P(2) \, \}$$

are contradictory and this could be detected using equational reasoning. The assertion $\text{rew}(Int, \text{plus}(1, 1), 2)$ as well as the application of predicate substituitivity would be needed.

However, if the second clause is transformed like

$$\{ \neg P(2) \} \quad \rightsquigarrow \quad \{ \text{eqs}(Int, x, 2) \rightarrow \neg P(x) \}$$
$$\rightsquigarrow \quad \{ P(x) \rightarrow \text{neqs}(Int, x, 2) \}$$

that can directly be unified with the first clause, we need to refute

$$\text{neqs}(Int, \text{plus}(1, 1), 2).$$

The transformation is sound, it again does only use abstraction, which is sound for any term including ground terms and, of course, constants c. Replacing negated predicates and reducing $>$ to $<$ is obviously sound as well.

Depth bound heuristic   The presence of a distinguished goal formula in the set of axioms and assumptions is exploited to give the iterative deepening search of DARWIN a hint. Instead of starting with a small depth bound and increasing it, the search begins with a bound that equals the maximum term depth in the problem formula following the rationale that the unsatisfiability is founded in the formula and not in the axioms or the assumptions and a refutation will most likely need terms of the formula's maximal depth. The depth bound is calculated before the preprocessing including the flattening to allow assertions of totality up to that bound that may be necessary to refute literals that have been flattened.

5.2.2   Theory fragments

The theory was formulated as a combination of fragments that are collected in appendix A. The essential fragment A.1 contains the equality axioms as in section 4.3, following in A.2 are axioms for the relations $<$, $\leq$, $\not<$ and $\not\leq$. Arithmetic is split into A.3, where only subtraction with the constant 1 is axiomatized, and A.4 for the full arithmetic of plus and times. The array theory is contained in A.5.

For each problem treated in the following examples a set of theory fragments was chosen that was just as expressive as required to prove its unsatisfiability. This can in a way be compared to the approach of MATHSAT, described in section 5.1.3, where theories are layered and satisfiability is checked proceeding from general to more specific theories.

There is currently no automation for this step, however, one could be conceived in a way as described above. DARWIN is started with a limited set of theory axioms that is grown by axioms for a more specific theory when the formula has been found to be valid. If invalidity is proven, it

will hold for more specific theories as well. Unfortunately, this procedure is not guaranteed to terminate and with *AUFLIA* being a combination of array and integer theory there might not exist a total ordering of theory fragments. One could think of a situation where a formula is valid up to a certain accumulation of theory fragments. For the next step there might be two possible theory fragments where neither one is more general than the other and the formula will be invalid if one fragment is added while the proof will not terminate if the other fragment is added. The question arises, if there are ways to always choose the first theory fragment or if it is possible to cut the theory fragments in a way that avoids such situations.

The dilemma of choosing the right theory layer for the next step could also be avoided by separating the theories and using a method to reason with a combination of theories like the Nelson-Oppen or Shostak approaches, see [RRT05] for a recent survey on this topic.

5.2.3 Examples

The transformations of the clause set to a canonical form to deal with equality and theories were evaluated by looking into selected examples. They were mostly taken from the SMT-LIB benchmarks at [RT07] and from proof obligations generated by the KEY system.

A simple start: `decproc1`

A first, rather simple example was extracted from the KEY sequent

```
==>
    \forall int x;
      \forall int y;
        (   leq(0, x) & leq(x, y) & lt(y, a.length)
          -> leq(a[x], a[y]))
 -> \forall int x;
      (   lt(0, x) & lt(x, a.length)
        -> leq(a[sub(x, 1)], a[x]))
```

that is straightforwardly translated to a formula in the *AUFLIA* logic fragment as described in section 3.5. There is no type hierarchy involved, thus no type predicates have to be introduced, the formula is just the negated

sequent from above:

$$\neg$$
$$(\forall x : Int \; \forall y : Int \; (0 \leq x) \wedge (x \leq y) \wedge (y < \text{length}(a))$$
$$\quad \rightarrow (\text{select}(a, x) \leq \text{select}(a, y)))$$
$$\rightarrow$$
$$(\forall x : Int \; (0 < x) \wedge (x < \text{length}(a))$$
$$\quad \rightarrow (\text{select}(a, \text{minus}(x, 1)) \leq \text{select}(a, x))). \tag{5.16}$$

The formula needs the definition of two uninterpreted functions: a constant a of sort *Array* and a function

$$\text{length} : Array \rightarrow Int.$$

(5.16) is quite simple and unsatisfiable by an apparent argument: the elements of array a are monotonically increasing with its index and this naturally implies that the preceding element is always less than or equal to its successor.

After the preprocessing described in a previous section, the clause set in CNF is

$$\{ \text{r\_select}(a, \text{minus}(c_1, 1), c_2) \} \tag{5.17}$$

$$\{ \text{r\_select}(a, c_1, c_3) \} \tag{5.18}$$

$$\{ 0 < c_1 \} \tag{5.19}$$

$$\{ c_1 < \text{length}(a) \} \tag{5.20}$$

$$\{ c_2 \not\leq c_3 \} \tag{5.21}$$

$$\{ 0 \not\leq x, \; x \not\leq y, \; y \not< \text{length}(a), \; u_1 \leq u_2,$$
$$\quad \text{neqs}(Array, a, u_3), \; \text{neqs}(Array, a, u_4), \tag{5.22}$$
$$\quad \neg \text{r\_select}(u_3, x, u_1), \; \neg \text{r\_select}(u_4, y, u_2) \}$$

$$\{ \text{index}(\text{minus}(c_1, 1)) \} \tag{5.23}$$

$$\{ \text{index}(c_1) \} \tag{5.24}$$

The transformations are straightforward, as an example we will only elaborate on how we arrived at (5.22). It originates from the premiss of (5.16) which is

$$\{ \neg x \leq 0, \; \neg x \leq y, \; \neg y < \text{length}(a), \; \text{select}(a, x) \leq \text{select}(a, y))) \} \tag{5.25}$$

in CNF. It does not contain an equality, but the select terms that must be flattened. They occur in the last literal twice and both occurrences containing a universal variable are abstracted to new universal variables u_1 and u_2, respectively. They are defined in newly added literals

$$\text{neqs}(Int, u_1, \text{select}(a, x))$$

$$\text{neqs}(Int, u_2, \text{select}(a, y))$$

and the original literal now is

$$u_1 \leq u_2.$$

The abstractions are then turned into relations reading

$$\neg\, \text{r\_select}(a, x, u_1)$$

$$\neg\, \text{r\_select}(a, y, u_2).$$

The negative occurrences of $<$ and $\leq$ become positive occurrences of $\not<$ and $\not\leq$ and the occurrences of the ground term a in the negative literals above are abstracted to new universal variables u_3 and u_4 defined by

$$\text{neqs}(Array, u_3, a)$$

$$\text{neqs}(Array, u_4, a)$$

so that we finally have (5.22)

$$\{\; 0 \not\leq x,\; x \not\leq y,\; y \not< \text{length}(a),\; u_1 \leq u_2,$$
$$\text{neqs}(Array, a, u_3),\; \text{neqs}(Array, a, u_4),$$
$$\neg\, \text{r\_select}(u_3, x, u_1),\; \neg\, \text{r\_select}(u_4, y, u_2) \;\}.$$

Observe that this can be regarded as an implication of the definitions in the last two lines and the original clause (5.25) in the first line.

The index predicate is true for the ground terms $\text{minus}(c_1, 1)$ in (5.17) and c_1 in (5.18) where the array a is read. The clauses (5.23) and (5.24) have been added for that reason.

The depth bound was taken from the deepest subterm in the original formula $\text{select}(a, \text{minus}(x, 1))$ which equals 3.

Looking at the clause set, it seems enough to have the theory fragments for equality (A.1), relations (A.2), arithmetic with $(x - 1)$ (A.3) and arrays (A.5).

Unsatisfiability of the clause set can be shown in a very short derivation. Each literal in (5.22) is either directly contradicted by a unit clause from the problem or by a literal that can be asserted in only one step from the theory. The solution is thus quite shallow in the search space of DARWIN and one can expect that it will find it quite fast.

Indeed, DARWIN proves the unsatisfiability of the problem in very short time that is even competitive to CVC3 and Yices. Of course, this comparison is quite unfair, considering the amount of manual tuning that went into DARWIN's proof by picking just the right theory fragments and leaving out unnecessary ones. Adding to that, DARWIN is incomplete on the *AUFLIA* logic, while CVC3 and Yices claim to be complete there.

Symmetric matrix: `symm-array`

The next example was taken from [Ran06] and slightly modified. It has to be proven that a symmetric matrix remains symmetric if only elements along the diagonal are changed.

Unfortunately, the array functions select and store cannot be extended easily to model two-dimensional arrays in the *AUFLIA* sublogic. The obvious approach to nest the select function is not possible as it always returns an integer and not an array, making terms like $\text{select}(\text{select}(a, i), j)$ invalid. The index is defined to be an integer as well, ruling out the possibility to take a pair of integers instead. One could work around this by defining a pairing function and its inverse projecting a pair of integers into one and vice versa, but considering the weak implementation of equality, it was instead opted to define new functions for reading and writing two-dimensional arrays and to axiomatize their theory paralleling the theory of one-dimensional arrays.

The functions were

$$\text{read} : Matrix \times Int \times Int \rightarrow Int$$

$$\text{write} : Matrix \times Int \times Int \times Int \rightarrow Matrix$$

and all axioms in A.5 were changed to hold for both indices. The preprocessing steps to flatten and turn into relational predicates for the one-dimensional array functions were applied to these two-dimensional functions as well.

The problem defines two matrices a and a_{final}, four constants e_0 to e_3 to be written to the matrix and a constant n as the dimension of the problem that was set to 3. The matrix a is assumed to be symmetric

$$\forall i : Int \; \forall j : Int \; (0 \leq i) \wedge (i \leq n) \wedge (0 \leq j) \wedge (j \leq n)$$
$$\rightarrow (\text{read}(a, i, j) = \text{read}(a, j, i)) \quad (5.26)$$

and a second assumption states that a_{final} is changed by writing elements along the diagonal

$$a_{\text{final}} = \text{write}(\text{write}(\text{write}(\text{write}(a, 0, 0, e_0), 1, 1, e_1), 2, 2, e_2), 3, 3, e_3). \quad (5.27)$$

Finally, the formula to prove is that a_{final} is still symmetric

$$\forall i : Int \; \forall j : Int \; (0 \le i) \wedge (i \le n) \wedge (0 \le j) \wedge (j \le n)$$
$$\rightarrow (\text{read}(a_{\text{final}}, i, j) = \text{read}(a_{\text{final}}, j, i)). \quad (5.28)$$

This example is hard enough to evaluate the handling of sorts and aspects of formulating the theory with. All proofs were run on a Pentium 4 with 3.4GHz and 1GB of RAM, and using the transformations described so far, it took about 70 seconds to prove unsatisfiability.

CVC3 and Yices were unable to produce a result and both gave up after seconds. This is again quite an unfair comparison as the optimized handling of arrays in their proof procedures could not be applied and they were left with an axiomatic formulation of the two-dimensional array.

Returning to DARWIN, if the ternary equality and rewriting predicates eqs, neqs and rew are replaced by binary predicates without sorts, no proof is found even after hours. If the sorts are strengthened so that each clause containing a universal variable of sort array is added a constraining sort predicate like

$$\{ \text{eqs}(Array, \text{select}(x, 0), 0), \; \neg \, \text{array}(x) \}$$

adding the implication that the variable x is an array, a proof is found, but only after more than twice the time. This supports our approach of using only sorted equality predicates and no further constraint predicates as presented in section 2.2.

A second observation was made with positive and negative formulations of relation predicates, in this case only $\le$. A total ordering of the numbers 0 to 3 was given by the clauses

$$\{\, 0 \le 1 \,\} \quad \{\, 1 \le 2 \,\} \quad \{\, 2 \le 3 \,\}$$

and not much surprisingly a proof was found about ten percent faster if these clauses were not present. The total ordering is only a further constraint that does not influence the unsatisfiability of the problem. However, if the total ordering was formulated using the negative relation predicate $\not\le$ by

$$\{\, \neg 0 \not\le 1 \,\} \quad \{\, \neg 1 \not\le 2 \,\} \quad \{\, \neg 2 \not\le 3 \,\},$$

it took three times as long as with positive predicate $\le$.

This can justify using the positive versions of relation predicates in the theory axioms and not applying the canonical transformations to them. In that way, some duality is enforced between the positive relation predicates in the theory and the negative relation predicates in the problem by requiring an intermediate step to assert the negative of one of them to be able to unify them while this is immediately possible inside the theory. The derivation in the slower case consequently considers more than three times the assert candidates than when the predicates are separated.

Arrays and arithmetic: `piVC_0df790`

To examine a combination of array and integer theory, a benchmark from the SMT-LIB *AUFLIA* benchmark suite was chosen, namely `piVC_0df790`. It is considered to be easy and its core is the unsatisfiability

$$\forall x : Int \ (0 \le x) \wedge (x \le n) \rightarrow (\text{select}(a, x) = e)$$
$$\leftrightarrow \tag{5.29}$$
$$\exists x : Int \ (0 \le x) \wedge (x \le n) \wedge (\text{select}(a, x) \ne e)$$

that was already used as an example in section 5.1.2.

It can easily be proven by DARWIN with the theory fragments of equality (A.1), relations (A.2), arithmetic with $(x - 1)$ (A.3) and arrays (A.5).

If (5.29) is modified to include a calculation task

$$\forall x : Int \ (0 \le x) \wedge (x \le n) \rightarrow (\text{select}(a, x) = e + e)$$
$$\leftrightarrow \tag{5.30}$$
$$\exists x : Int \ (0 \le x) \wedge (x \le n) \wedge (\text{select}(a, x) \ne 2 \cdot e)$$

whose result is specified by either one of

$$\text{eqs}(Int, \text{plus}(x, x), \text{times}(n2, x)) \tag{5.31}$$

$$\text{rew}(Int, \text{times}(n2, x), \text{plus}(x, x)) \tag{5.32}$$

$$\text{rew}(Int, \text{plus}(x, x), \text{times}(n2, x)), \tag{5.33}$$

it can be just as easily solved. In that case, it does not make a difference if rewriting or the equality predicate is used as the equality is range-restricted and does not admit to repeated application.

Non-linear arithmetic: `binomial`

A proof obligation for the binomial formula

$$(a + b)^2 = a^2 + 2 \cdot a \cdot b + b^2$$

comes as a KEY sequent

```
==>
 \forall int a;
   \forall int b;
      mul(add(a, b), add(a, b))
    = add(add(mul(a, a), mul(mul(2, a), b)),
          mul(b, b))
```

where all `mul` terms except

`mul(2, a)`

are non-linear. The formula is thus not in the *AUFLIA* sublogic and the non-linear `mul` terms would be translated to an uninterpreted function $f_{\text{non\_linear\_mult}}$ as in

$$\texttt{mul}(\texttt{mul}(2,\texttt{a}),\texttt{b}) \quad \rightsquigarrow \quad f_{\text{non\_linear\_mult}}(\text{times}(2,a),b).$$

Not having axioms for that function, DARWIN would find the problem to be satisfiable. However, if the non-linear multiplication is instead translated to the interpreted multiplication that is then made distributive over addition

$$\text{eqs}(Int, \text{times}(x, \text{plus}(y, z)), \text{plus}(\text{times}(x, y), \text{times}(x, z))) \qquad (5.34)$$

and the multiplication by two is reduced to an addition as in (5.31), (5.32) or (5.33) in the last example, the problem is quickly found unsatisfiable.

In that instance, non-linear multiplication can be handled. However, it also becomes obvious that an inductive formulation of linear multiplication will not succeed. It would be natural to define

$$\text{rew}(Int, \text{times}(x, y), \text{plus}(\text{times}(\text{minus}(x, 1), y), y)) \qquad (5.35)$$

that would terminate with the already available axiom for the neutral element of multiplication (A.49)

$$\forall x \ \text{rew}(Int, \text{times}(n1, x), x).$$

With that, one could assert

$$\text{rew}(Int, \text{times}(n2, a), \text{plus}(\text{times}(\text{minus}(n2, 1), a), a)),$$

but as there is no functional substituitivity for the arithmetic functions plus, times and minus, the subterms of plus on the right-hand side of the rewrite rule could not be further rewritten. The availability of equality in the Model Evolution calculus will obsolete the rewriting predicate and thus make an inductive formulation of linear multiplication possible.

Bubblesort invariant: arr2

As a last example to elaborate on, a quite classical example in the software verification domain was chosen from the SMT-LIB benchmarks. It is the invariant of the bubblesort algorithm stating that in an array that is sorted except for two succeeding elements, exchanging these elements will make

the whole array sorted. The formulation in (5.36) includes a new sort *Queue* and the three uninterpreted functions

$$\text{key} : Int \rightarrow Int$$

$$\text{elems} : Queue \rightarrow Array$$

$$\text{size} : Array \rightarrow Int$$

that require functional substituitivity. Fortunately, they do not occur nested with each other and not as subterms inside other functions except array functions that become predicates after the preprocessing. Thus the functions only occur as direct subterms in predicates and the axioms

$$\text{neqs}(Int, \text{key}(x), y) \wedge \text{rew}(Int, x, s) \rightarrow \text{neqs}(Int, \text{key}(s), y)$$

$$\text{neqs}(Int, x, \text{key}(y)) \wedge \text{rew}(Int, y, s) \rightarrow \text{neqs}(Int, x, \text{key}(s))$$

$$\text{neqs}(Array, \text{elems}(x), y) \wedge \text{rew}(Queue, x, s) \rightarrow \text{neqs}(Array, \text{elems}(s), y)$$

$$\text{neqs}(Array, x, \text{elems}(y)) \wedge \text{rew}(Queue, y, s) \rightarrow \text{neqs}(Array, x, \text{elems}(s))$$

$$\text{neqs}(Int, \text{size}(x), y) \wedge \text{rew}(Array, x, s) \rightarrow \text{neqs}(Int, \text{size}(s), y)$$

$$\text{neqs}(Int, x, \text{size}(y)) \wedge \text{rew}(Array, y, s) \rightarrow \text{neqs}(Int, x, \text{size}(s))$$

are sufficient to account for functional substituitivity of those three functions.

The formula

$\neg \forall i : Int \; \forall pp : Queue$

$\quad ($

$\qquad i \geq 0$

$\qquad \wedge$

$\qquad (\forall j : Int \; \forall k : Int$

$\qquad\quad (j > 0) \wedge (j < size(pp)) \wedge (k \geq 0) \wedge (k < j) \wedge (i \neq j)$

$\qquad\quad \rightarrow$

$\qquad\quad (key(select(elems(pp), j)) \leq key(select(elems(pp), k)))$

$\qquad)$

$\qquad \wedge$

$\qquad \neg(key(select(elems(pp), i)) \leq key(select(elems(pp), i - 1)))$

$\qquad \wedge$

$\qquad (i > 0)$ $\hspace{4cm}$ (5.36)

$\qquad \wedge$

$\qquad (i < size(pp))$

$\quad)$

$\quad \rightarrow$

$\quad (\forall j : Int \; \forall k : Int \; \forall ee : Array$

$\qquad (j > 0) \wedge (j < size(pp)) \wedge (k \geq 0) \wedge (k < j) \wedge (j \neq i - 1)$

$\qquad \wedge$

$\qquad ee = store(store(elems(pp), i - 1, select(elems(pp), i)),$

$\qquad\qquad\qquad i, select(elems(pp), i - 1))$

$\qquad \rightarrow$

$\qquad (key(select(elems(ee), j)) \leq key(select(elems(ee), k)))$

$\quad)$

can then be proven by DARWIN in 4 seconds which is not competitive to CVC3 and Yices that both take less than tenths of seconds.

AUFLIA Burns benchmarks

Finally, DARWIN was run on a series of SMT-LIB *AUFLIA* benchmarks about the Burns' mutual exclusion protocol (see [Bur78]). The required theory fragments were equality (A.1), relations (A.2) and arrays (A.5) and exactly half of the 14 benchmarks could be proven, each after well less than one second.

To evaluate our approach to theory reasoning without the dependence on equational reasoning, we removed the approximation of the rewrite-

based equational reasoning and reintroduced the standard binary equality predicate, keeping the preprocessing for theory reasoning described in this chapter. The proof obligations were then given to EKRHYPER that accepts DARWIN's input format. Its proof procedure as described by [BPF07] is the Hyper Tableaux calculus with superposition as equational reasoning.

As an encouraging result, all remaining benchmarks could be proven in a matter of about 1.5 seconds each. Though the Hyper Tableaux calculus is rather different from Model Evolution, our approach to theory reasoning still worked there. In the case of the Burns' benchmarks our formulation of theory fragments is complete and the preprocessing does not harm completeness. We can thus look forward to much better performance in the Model Evolution calculus with equality in terms of speed as well as in the number of solvable problems.

Chapter 6

Conclusion

Having set out to integrate the Model Evolution calculus of the theorem prover DARWIN into the software verification system KEY, we found several obstacles on the way. Not all of them could be passed in this work, some still remain there. Nevertheless, we found a working approach to some of them and pointers on how to tackle others, leaving open ends for future research.

6.1 Results

A first rather technical achievement of this work is the implementation of an interface to the SMT-LIB benchmark suite for DARWIN that enables it to process input in that format. The relaxation of the many-sorted logic to the unsorted logic of DARWIN was done by slightly deviating from the translation semantics as defined in SMT-LIB, justified by a better fitness for the Model Evolution calculus. Though building the CNF of the benchmark is delegated to the E prover that is not aware of sorts, care is taken that the sorts of Skolem symbols can be still deduced.

With the current unavailability of an implementation of the Model Evolution calculus with equality, an approximation of rewrite-based equational reasoning was conceived that did not need to modify the calculus by being purely operational. The approximation takes advantage of the information about sorts that has been saved in the translation from the original SMT-LIB input.

A lifting of existing DPLL procedures for satisfiability modulo theories could not be found. As a first approach to have theory reasoning in the first-order Model Evolution calculus, it was attempted to transform the clause set and reason with the theory formalized as a set of axioms. We could successfully prove a number of benchmarks from the target domain of software verification.

6.2   Further Work

Next steps following up on this work certainly should build on the Model Evolution calculus with equality. A proper implementation of ordered paramodulation will obsolete our approximation of rewrite-based equational reasoning and lead to a much better performance. As we showed in the last chapter, this will enable DARWIN to prove more problems and to be faster at it.

The theory of equality will not be needed any more and there is no use in keeping the information about sorts that was still required for our approximation. Possibilities should be explored to apply a reduction ordering compatible with associativity and commutativity of functions defined in the theory. Thus the theory formalization could become lighter by lifting those properties to the calculus.

The incompleteness of the theory reasoning could be tackled in several ways. Choosing an axiomatization that is complete and tractable for the calculus is an obvious step which might not be feasible for a variety of reasons. Further, the monolithic theory could be separated to layers or independent theories. The latter approach would admit to employing procedures like Nelson-Oppen or Shostak for the combination of theories. Equality would play an important role there. The layering of theories, that is not precluded by the approach of combining theories, would stack theories of increasing strength and decreasing tractability to be successively applied.

Instead of striving for completeness, some effort could be put into detecting incompleteness on a particular problem. The procedure would then try to exhaust its theory formulation and finally pass a suitably formulated proof obligation on to another theory solver. Here, the advantages of the Model Evolution calculus being of true first-order could be exploited to break the problem down to one of a smaller size that could be admissible to a solver less powerful on first-order logic, but with a stronger emphasis on theories.

The most challenging effort is certainly a lifting of some of the presented ground propositional DPLL($\mathcal{T}$) procedures to first-order logic. However, it could be most rewarding as it would be as superior to DPLL($\mathcal{T}$) as Model Evolution is to DPLL.

A bigger part in a lifting of DPLL($\mathcal{T}$) would be taken by the search for a suitable representation of first-order contexts and models from the Model Evolution calculus. The approach requires collaboration with a theory solver working on an interpretation for the clause set that is only perceived abstractly by the high-level Model Evolution calculus. It is an important open question if there are other model representations than DIGs that would lead to an even more powerful instance-based method or if such

a method would enable concise model representations tractable for a theory reasoner.

If the Model Evolution calculus has been shown to be useful for problems in the software verification domain, we can return to the pursuit of integration with the KEY system. A successful proof could return information enabling the KEY system to verify the correctness of the proof without having to rely on the correctness of the called procedure. On the other hand could a failed proof provide a model that could in turn give hints for debugging the program or its specification.

Not regarding an embedded proof procedure as a black box and presupposing its correctness, but instead requiring witnesses to certify delegated proof obligations is a central issue for reliably using cooperating theorem provers. Instance-based methods are well prepared for that requirement as the calculus explicitly reasons with candidate models of the clause set. The domain of software verification, besides being maybe the most important area of practical application for automated theorem proving, is especially concerned with heterogeneous theories modelling semantics of programs and therefore it is quite natural to have different theorem provers cooperate on such problems. Considering that, we will certainly see advances towards successful integration of instance-based methods in software verification.

Appendix

Appendix A

Unified theory

A.1 Equality

Reflexivity of eqs

$$\forall t \ \forall x \ \text{eqs}(t, x, x) \tag{A.1}$$

Irreflexivity of neqs

$$\forall t \ \forall x \ \neg \text{neqs}(t, x, x) \tag{A.2}$$

Rewriting from equality: conjunctive variant

$$\forall t \ \forall x \ \forall y \ \text{eqs}(t, x, y) \rightarrow \text{rew}(t, x, y) \tag{A.3}$$

$$\forall t \ \forall x \ \forall y \ \text{eqs}(t, x, y) \rightarrow \text{rew}(t, y, x) \tag{A.4}$$

Rewriting from equality: disjunctive variant

$$\forall t \ \forall x \ \forall y \ \text{eqs}(t, x, y) \rightarrow \text{rew}(t, x, y) \vee \text{rew}(t, y, x) \tag{A.5}$$

Substituitivity for eqs

$$\forall t \ \forall x \ \forall y \ \forall s \ \text{eqs}(t, x, y) \wedge \text{rew}(t, x, s) \rightarrow \text{eqs}(t, s, y) \tag{A.6}$$

$$\forall t \ \forall x \ \forall y \ \forall s \ \text{eqs}(t, x, y) \wedge \text{rew}(t, y, s) \rightarrow \text{eqs}(t, x, s) \tag{A.7}$$

Substituitivity for neqs

$$\forall t \ \forall x \ \forall y \ \forall s \ \text{neqs}(t, x, y) \wedge \text{rew}(t, x, s) \rightarrow \text{neqs}(t, s, y) \tag{A.8}$$

$$\forall t \ \forall x \ \forall y \ \forall s \ \text{neqs}(t, x, y) \wedge \text{rew}(t, y, s) \rightarrow \text{neqs}(t, x, s) \tag{A.9}$$

A.2   Relations

Reflexivity of $\leq$

$$\forall x \; (x \leq x) \tag{A.10}$$

Antisymmetry of $\leq$

$$\forall x \; \forall y \; (x \leq y) \wedge (y \leq x) \rightarrow \text{eqs}(Int, x, y) \tag{A.11}$$

Transitivity of $\leq$

$$\forall x \; \forall y \; \forall z \; (x \leq y) \wedge (y \leq z) \rightarrow (x \leq z) \tag{A.12}$$

Transitivity of $<$

$$\forall x \; \forall y \; \forall z \; (x < y) \wedge (y < z) \rightarrow (x < z) \tag{A.13}$$

Transitivity of $\leq$ and $<$

$$\forall x \; \forall y \; \forall z \; (x < y) \wedge (y \leq z) \rightarrow (x < z) \tag{A.14}$$

$$\forall x \; \forall y \; \forall z \; (x \leq y) \wedge (y < z) \rightarrow (x < z) \tag{A.15}$$

$<$ is stronger than $\leq$

$$\forall x \; \forall y \; (x < y) \rightarrow (x \leq y) \tag{A.16}$$

$$\forall x \; \forall y \; (x \leq y) \rightarrow (x < y) \vee \text{eqs}(Int, x, y) \tag{A.17}$$

Definition of $\nleq$

$$\forall x \; \forall y \; (x \nleq y) \wedge (x \leq y) \rightarrow \bot \tag{A.18}$$

Irreflexivity of $\nleq$

$$\forall x \; \neg(x \nleq x) \tag{A.19}$$

Definition of $\nless$

$$\forall x \; \forall y \; (x < y) \wedge (x \nless y) \rightarrow \bot \tag{A.20}$$

$$\forall x \; \forall y \; (y \nless x) \rightarrow \text{eqs}(Int, x, y) \vee (x < y) \tag{A.21}$$

$<$ and $\not\leq$

$$\forall x \, \forall y \, (x \not\leq y) \rightarrow (y < x) \tag{A.22}$$

$$\forall x \, \forall y \, (y < x) \rightarrow (x \not\leq y) \tag{A.23}$$

Irreflexivity of $<$

$$\forall x \, \neg(x < x) \tag{A.24}$$

$<$ and neqs

$$\forall x \, \forall y \, (x < y) \rightarrow \text{neqs}(Int, x, y) \tag{A.25}$$

$$\forall x \, \forall y \, \text{neqs}(Int, x, y) \rightarrow (x < y) \vee (y < x) \tag{A.26}$$

Substituitivity for $\leq$

$$\forall x \, \forall y \, \forall s \, (x \leq y) \wedge \text{rew}(Int, x, s) \rightarrow (s \leq y) \tag{A.27}$$

$$\forall x \, \forall y \, \forall s \, (x \leq y) \wedge \text{rew}(Int, y, s) \rightarrow (x \leq s) \tag{A.28}$$

Substituitivity for $<$

$$\forall x \, \forall y \, \forall s \, (x < y) \wedge \text{rew}(Int, x, s) \rightarrow (s < y) \tag{A.29}$$

$$\forall x \, \forall y \, \forall s \, (x < y) \wedge \text{rew}(Int, y, s) \rightarrow (x < s) \tag{A.30}$$

Substituitivity for $\not\leq$

$$\forall x \, \forall y \, \forall s \, (x \not\leq y) \wedge \text{rew}(Int, x, s) \rightarrow (s \not\leq y) \tag{A.31}$$

$$\forall x \, \forall y \, \forall s \, (x \not\leq y) \wedge \text{rew}(Int, y, s) \rightarrow (x \not\leq s) \tag{A.32}$$

Substituitivity for $\not<$

$$\forall x \, \forall y \, \forall s \, (x \not< y) \wedge \text{rew}(Int, x, s) \rightarrow (s \not< y) \tag{A.33}$$

$$\forall x \, \forall y \, \forall s \, (x \not< y) \wedge \text{rew}(Int, y, s) \rightarrow (x \not< s) \tag{A.34}$$

A.3   Arithmetic $(x - 1)$

Nothing between zero and one

$$\forall x \ (0 < x) \wedge (x < 1) \rightarrow \bot \tag{A.35}$$

$0 < 1$

$$0 < 1 \tag{A.36}$$

$$\neg \operatorname{eqs}(Int, 0, 1) \tag{A.37}$$

Monotonicity of minus

$$\forall x \ \forall y \ (x < y) \wedge (\operatorname{minus}(y, 1) < x) \rightarrow \bot \tag{A.38}$$

$$\forall x \ \neg (x \leq \operatorname{minus}(x, 1)) \tag{A.39}$$

$$\forall x \ \neg (\operatorname{minus}(x, 1) \not< x) \tag{A.40}$$

$$\forall x \ \neg \operatorname{eqs}(Int, \operatorname{minus}(x, 1), x) \tag{A.41}$$

if $x - 1 = 0$ then $x = 1$

$$\forall x \ \operatorname{eqs}(Int, 0, \operatorname{minus}(x, 1)) \rightarrow \operatorname{eqs}(Int, x, 1) \tag{A.42}$$

A.4   Arithmetic for plus and times

Commutativity of plus

$$\forall x \ \forall y \ \operatorname{eqs}(Int, \operatorname{plus}(y, x), \operatorname{plus}(x, y)). \tag{A.43}$$

Commutativity of times

$$\forall x \ \forall y \ \operatorname{eqs}(Int, \operatorname{times}(y, x), \operatorname{times}(x, y)). \tag{A.44}$$

Associativity of plus

$$\forall x \ \forall y \ \forall z \ \operatorname{eqs}(Int, \operatorname{plus}(x, \operatorname{plus}(y, z)), \operatorname{plus}(\operatorname{plus}(x, y), z)) \tag{A.45}$$

Associativity of times

$$\forall x \ \forall y \ \forall z \ \operatorname{eqs}(Int, \operatorname{times}(x, \operatorname{times}(y, z)), \operatorname{times}(\operatorname{times}(x, y), z)) \tag{A.46}$$

Distributivity

$$\forall x \, \forall y \, \forall z \ \text{eqs}(Int, \text{times}(x, \text{plus}(y, z)), \text{plus}(\text{times}(x, y), \text{times}(x, z)))$$
(A.47)

Zero is neutral element of plus

$$\forall x \ \text{rew}(Int, \text{plus}(0, x), x) \tag{A.48}$$

One is neutral element of times

$$\forall x \ \text{rew}(Int, \text{times}(1, x), x) \tag{A.49}$$

Integer multiplication

$$\forall x \ \text{rew}(Int, \text{times}(1, x), x) \tag{A.50}$$

$$\forall x \ \text{rew}(Int, \text{times}(2, x), \text{plus}(x, x)) \tag{A.51}$$

$$\forall x \ \text{rew}(Int, \text{times}(3, x), \text{plus}(x, \text{plus}(x, x))) \tag{A.52}$$

$$\forall x \ \text{rew}(Int, \text{times}(4, x), \text{plus}(x, \text{plus}(x, \text{plus}(x, x)))) \tag{A.53}$$

$x + x = 2x$
$$\forall x \ \text{eqs}(Int, \text{plus}(x, x), \text{times}(2, x)) \tag{A.54}$$

Cancellation

$$\forall x \, \forall y \, \forall z \ \text{neqs}(Int, \text{plus}(x, y), \text{plus}(x, z)) \rightarrow \text{neqs}(Int, y, z) \tag{A.55}$$

$$\forall x \, \forall y \, \forall z \ \text{neqs}(Int, \text{times}(x, y), \text{times}(x, z)) \rightarrow \text{neqs}(Int, y, z) \tag{A.56}$$

$$\forall x \, \forall y \ \text{neqs}(Int, \text{neg}(x), \text{neg}(y)) \rightarrow \text{neqs}(Int, x, y) \tag{A.57}$$

Inverse of plus
$$\forall x \ \text{rew}(Int, \text{plus}(\text{neg}(x), x), 0) \tag{A.58}$$

Inverse of times

$$\forall x \ \text{rew}(Int, \text{times}(\text{neg}(1), x), \text{neg}(x)) \tag{A.59}$$

Twice neg
$$\forall x \ \text{rew}(Int, \text{neg}(\text{neg}(x)), x) \tag{A.60}$$

plus **and** neg
$$\forall x \ \forall y \ \text{rew}(Int, \text{neg}(\text{plus}(x, y)), \text{plus}(\text{neg}(x), \text{neg}(y))) \tag{A.61}$$

times **and** neg
$$\forall x \ \forall y \ \text{rew}(Int, \text{neg}(\text{times}(x, y)), \text{times}(\text{neg}(x), y)) \tag{A.62}$$

Substituitivity for plus
$$\forall x \ \forall y \ \forall s \ \text{plus}(x, y) \land \text{rew}(Int, x, s) \rightarrow \text{plus}(s, y) \tag{A.63}$$

$$\forall x \ \forall y \ \forall s \ \text{plus}(x, y) \land \text{rew}(Int, y, s) \rightarrow \text{plus}(x, s) \tag{A.64}$$

Substituitivity for times
$$\forall x \ \forall y \ \forall s \ \text{times}(x, y) \land \text{rew}(Int, x, s) \rightarrow \text{times}(s, y) \tag{A.65}$$

$$\forall x \ \forall y \ \forall s \ \text{times}(x, y) \land \text{rew}(Int, y, s) \rightarrow \text{times}(x, s) \tag{A.66}$$

A.5   Arrays

Array axioms
$$\forall a \ \forall i \ \forall e \ \forall r \ \text{r\_store}(a, i, e, r) \rightarrow \text{r\_select}(r, i, e) \tag{A.67}$$

$$\forall a \ \forall i \ \forall j \forall e \ \forall d \forall r \ \text{r\_store}(a, i, e, r) \land \text{r\_select}(a, j, d) \rightarrow$$
$$\text{r\_select}(r, j, d) \lor \text{eqs}(Int, i, j) \tag{A.68}$$

Functionality of select
$$\forall a \ \forall i \ \forall e_1 \ \forall e_2 \ \text{r\_select}(a, i, e_1) \land \text{r\_select}(a, i, e_2) \rightarrow \text{eqs}(Int, e_1, e_2) \tag{A.69}$$

Totality of select
$$\forall a \ \forall i \ \text{index}(i) \rightarrow \text{r\_select}(a, i, \text{select\_val}(a, i)) \tag{A.70}$$

Functionality of store

$$\forall a \ \forall i \ \forall e \ \forall a_1 \ \forall a_2 \ \text{r\_store}(a, i, e, a_1) \wedge \text{r\_store}(a, i, e, a_2) \rightarrow \text{eqs}(Array, a_1, a_2)$$
(A.71)

Substituitivity for r_select

$$\forall a \ \forall i \ \forall e \ \forall s \ \text{r\_select}(a, i, e) \wedge \text{rew}(Array, a, s) \rightarrow \text{r\_select}(s, i, e) \quad \text{(A.72)}$$

$$\forall a \ \forall i \ \forall e \ \forall s \ \text{r\_select}(a, i, e) \wedge \text{rew}(Int, i, s) \rightarrow \text{r\_select}(a, s, e) \quad \text{(A.73)}$$

$$\forall a \ \forall i \ \forall e \ \forall s \ \text{r\_select}(a, i, e) \wedge \text{rew}(Int, e, s) \rightarrow \text{r\_select}(a, i, s) \quad \text{(A.74)}$$

Substituitivity for r_store

$$\forall a \ \forall i \ \forall e \ \forall r \ \forall s \ \text{store}(a, i, e, r) \wedge \text{rew}(Array, a, s) \rightarrow \text{store}(s, i, e, r) \quad \text{(A.75)}$$

$$\forall a \ \forall i \ \forall e \ \forall r \ \forall s \ \text{store}(a, i, e, r) \wedge \text{rew}(Int, i, s) \rightarrow \text{store}(a, s, e, r) \quad \text{(A.76)}$$

$$\forall a \ \forall i \ \forall e \ \forall r \ \forall s \ \text{store}(a, i, e, r) \wedge \text{rew}(Int, e, s) \rightarrow \text{store}(a, i, s, r) \quad \text{(A.77)}$$

$$\forall a \ \forall i \ \forall e \ \forall r \ \forall s \ \text{store}(a, i, e, r) \wedge \text{rew}(Array, r, s) \rightarrow \text{store}(a, i, e, s) \quad \text{(A.78)}$$

Bibliography

[BBC$^+$05]   Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Tools and Algorithms for the Construction and Analysis of Systems. 11th International Conference, TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 317–333. Springer Berlin / Heidelberg, 2005.

[BFdNT07] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic. To appear, preliminary version, April 2007.

[BFT04]   Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Darwin: A Theorem Prover for the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *Proceedings of the 1st Workshop on Empirically Successful First Order Reasoning (ESFOR'04), Cork, Ireland, 2004*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[BG98]   Leo Bachmair and Harald Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction: A Basis for Applications. Volume I, Foundations: Calculi and Methods*. Kluwer Academic Publishers, Dordrecht, 1998.

[BHS07]   Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[BMS05]   Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's Decidable About Arrays. In *Verification, Model Checking, and Abstract Interpretation. 7th International Conference, VMCAI*

2006, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer Berlin / Heidelberg, 2005.

[BNOT06]   Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on Demand in SAT Modulo Theories. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer Berlin / Heidelberg, 2006.

[Bor05]   Tomasz Borzyszkowski. Generalized Interpolation in First Order Logic. *Fundamenta Informaticae*, 66(3):199–219, 2005.

[BPF07]   Peter Baumgartner, Björn Pelzer, and Ulrich Furbach. The Hyper Tableaux Calculus with Equality. Submission to CADE-21, 2007.

[Bra75]   D. Brand. Proving Theorems with the Modification Method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[BT03]   Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. Technical Report 1, Universität Koblenz-Landau, Fachbereich Informatik, 2003.

[BT05]   Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus with Equality. In *Automated Deduction – CADE-20. 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 392–408. Springer Berlin / Heidelberg, 2005.

[Bur78]   James E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2):42–47, 1978.

[DdM06]   Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification. 18th International Conference, CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer Berlin / Heidelberg, 2006.

[DLL62]   Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DNS05]   David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[DP60]   Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[FP05]   Christian G. Fermüller and Reinhard Pichler. Model Representation via Contexts and Implicit Generalizations. In *Automated Deduction – CADE-20. 20th International Conference on Automated Deduction, Tallinn, Estonia*, volume 3632 of *Lecture Notes in Computer Science*, pages 409–423. Springer Berlin / Heidelberg, July 2005.

[GBT07]   Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving Quantified Verification Conditions using Satisfiability Modulo Theories. Submitted to CADE, 2007.

[GHN+04]   Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification. 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer Berlin / Heidelberg, 2004.

[GP98]   G. Gottlob and R. Pichler. Towards a compact knowledge representation by AR models. In *Proceedings of WLP'98 (13th Workshop on Logic Programming)*, 1998.

[LM05]   Shuvendu Lahiri and Krishna K. Mehra. Interpolant based Decision Procedure for Quantifier-Free Presburger Arithmetik. Technical Report MSR-TR-2005-121, Microsoft Research, 2005.

[MP67]   John McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In J. T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 33–41. American Mathematical Society, Providence, RI, 1967.

[NOT06]   Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[NR01]   Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[Ran06]   Silvio Ranise. Satisfiability Solving for Program Verification: towards the Efficient Combination of Automated Theorem Provers and Satisfiability Modulo Theory Tools. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *Proceedings of the*

IJCAR'06 Workshop DISPROVING: Non-Theorems, Non-Validity, Non-Provability, pages 49–58, 2006.

[RRT05]   Silvio Ranise, Christophe Ringeissen, and Duc-Khanh Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*, pages 372–386. Springer Berlin / Heidelberg, 2005.

[RT06]   Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2, 2006. available at `http://www.smt-lib.org`.

[RT07]   Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2007. Available at `http://www.smtlib.org/`.

[SBDL01]   Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 29–37, 2001.

[Sch04]   Stephan Schulz. System Description: E 0.81. In *Automated Reasoning. Second International Joint Conference, IJCAR 2004*, volume 2097 of *Lecture Notes in Computer Science*, pages 223–228. Springer Berlin / Heidelberg, 2004.

[Smu95]   Raymond M. Smullyan. *First-Order Logic*. Dover Publications, New York, second corrected edition edition, 1995. First published 1968 by Springer-Verlag.

[SS98]   G. Sutcliffe and C. B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Tin02]   Cesare Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer, 2002.