Fall 2009

Evolving model evolution

Alexander Fuchs
University of Iowa

EVOLVING MODEL EVOLUTION

by

Alexander Fuchs

An Abstract

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2009

Thesis Supervisor: Associate Professor Cesare Tinelli

ABSTRACT

Automated theorem proving is a method to establish or disprove logical theorems. While these can be theorems in the classical mathematical sense, we are more concerned with logical encodings of properties of algorithms, hardware and software. Especially in the area of hardware verification, propositional logic is used widely in industry. Satisfiability Module Theories (SMT) is a set of logics which extend propositional logic with theories relevant for specific application domains. In particular, software verification has received much attention, and efficient algorithms have been devised for reasoning over arithmetic and data types. Built-in support for theories by decision procedures is often significantly more efficient than reductions to propositional logic (SAT). Most efficient SAT solvers are based on the DPLL architecture, which is also the basis for most efficient SMT solvers. The main shortcoming of both kinds of logics is the weak support for non-ground reasoning, which noticeably limits the applicability to real world systems.

The Model Evolution Calculus (ME) was devised as a lifting of the DPLL architecture from the propositional setting to full first-order logic. In previous work, we created the solver Darwin as an implementation of ME, and showed how to adapt improvements from the DPLL setting. The first half of this thesis is concerned with ME and Darwin. First, we lift a further crucial ingredient of SAT and SMT solvers, lemma-learning, to Darwin and evaluate its benefits. Then, we show how to use Darwin for finite model finding, and how this application benefits from lemma-learning.

In the second half of the thesis we present Model Evolution with Linear Integer Arithmetic (ME(LIA)), a calculus combining function-free first-order logic with linear integer arithmetic (LIA). ME(LIA) is based on ME and supports similar inference rules and redundancy criteria. We prove the correctness of the calculus, and show how to obtain complete proof procedures and decision procedures for some interesting classes of ME(LIA)'s logic. Finally, we explain in detail how ME(LIA) can be implemented efficiently based on the techniques employed in SMT solvers and Darwin.

Abstract Approved: _____
Thesis Supervisor

_____
Title and Department

_____
Date

EVOLVING MODEL EVOLUTION

by

Alexander Fuchs

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2009

Thesis Supervisor: Associate Professor Cesare Tinelli

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Alexander Fuchs

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2009 graduation.

Thesis Committee: _____
Cesare Tinelli, Thesis Supervisor

_____
Peter Baumgartner

_____
Aaron Stump

_____
Kasturi Varadarajan

_____
Hantao Zhang

ACKNOWLEDGEMENTS

I thank my advisor Professor Cesare Tinelli for his guidance and support. He tended to find the right balance between pushing me to make progress, and giving me enough latitude to pursue my own interests. I appreciate the opportunities he opened up for me, especially the internships with Intel in Oregon and the visit to Chalmers in Sweden. I am grateful for his family's support and hospitality during the final days of finishing the thesis and preparing for the defense.

I thank Peter Baumgartner for his support for almost a decade now, starting with my studies in Koblenz. If he had not initiated my first visit to Iowa, this thesis would not exist.

I am grateful to Jed Hagen and Ruzica Piskac for technical support and proof-reading my thesis. I especially thank the Hagens for welcoming me to their home whenever I needed to escape. I appreciate the conversations with my lab mates, present and past, especially in the beginning they helped me to settle in. I am thankful to Sheryl Semler and Catherine Till for reducing bureaucratic hurdles to the bare minimum.

Finally, I thank my family for their continuous support and patience. I wrote this thesis in the peaceful quietness of my brother's home, powered by home-cooked food and unlimited supplies of coffee and sugar.

This work was partially supported by grant #0237422 from the National Science Foundation.

ABSTRACT

Automated theorem proving is a method to establish or disprove logical theorems. While these can be theorems in the classical mathematical sense, we are more concerned with logical encodings of properties of algorithms, hardware and software. Especially in the area of hardware verification, propositional logic is used widely in industry. Satisfiability Module Theories (SMT) is a set of logics which extend propositional logic with theories relevant for specific application domains. In particular, software verification has received much attention, and efficient algorithms have been devised for reasoning over arithmetic and data types. Built-in support for theories by decision procedures is often significantly more efficient than reductions to propositional logic (SAT). Most efficient SAT solvers are based on the DPLL architecture, which is also the basis for most efficient SMT solvers. The main shortcoming of both kinds of logics is the weak support for non-ground reasoning, which noticeably limits the applicability to real world systems.

The Model Evolution Calculus (ME) was devised as a lifting of the DPLL architecture from the propositional setting to full first-order logic. In previous work, we created the solver Darwin as an implementation of ME, and showed how to adapt improvements from the DPLL setting. The first half of this thesis is concerned with ME and Darwin. First, we lift a further crucial ingredient of SAT and SMT solvers, lemma-learning, to Darwin and evaluate its benefits. Then, we show how to use Darwin for finite model finding, and how this application benefits from lemma-learning.

In the second half of the thesis we present Model Evolution with Linear Integer Arithmetic (ME(LIA)), a calculus combining function-free first-order logic with linear integer arithmetic (LIA). ME(LIA) is based on ME and supports similar inference rules and redundancy criteria. We prove the correctness of the calculus, and show how to obtain complete proof procedures and decision procedures for some interesting classes of ME(LIA)'s logic. Finally, we explain in detail how ME(LIA) can be implemented efficiently based on the techniques employed in SMT solvers and Darwin.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1
INTRODUCTION

1.1 Motivation

Today's hardware and software systems are already complicated, and are steadily becoming ever bigger and more complex. It is common that the effort that goes into developing a software product exceeds hundreds, thousands, or even many more man-years. Obviously, it is beyond a human's capability to know and understand more than only a small part of a system, and it is impossible to reason about a complete system in all but very abstract and simplified ways. This has given rise to many approaches with the goal of supporting the development of robust and functionally correct large scale systems. High level and domain specific languages aim at reducing complexity, by removing unnecessary clutter, by simplifying software artifacts to express only relevant information, and by enforcing (domain specific) invariants. Many aspects of software engineering have a similar purpose, for example modularization, requirements management, design, and software development processes and best practices in general. While these techniques, broadly speaking, aim at reducing the likelihood of introducing errors, other methods focus directly on finding errors.

Testing has become an important and integral part of software development. Most testing techniques rely on checking properties like invariants and assertions by simulation or test cases. In essence, the software is executed in a test environment and its actual and expected behavior and output are compared. Advantages of this approach are that with some discipline it is not too difficult to implement, and that running tests is usually fast. This makes it possible to

maintain a regression test suit, and to continuously check the software for bugs during development. The main disadvantage is that testing is incomplete, as it verifies the system behavior only for specific input. Thus, testing usually means bug finding, not verification. In principle, properties can be checked not only during development but also in released software, for example by keeping assertions in the release version, or in design by contract approaches. But, this can, firstly, have significant run time overhead, and, secondly, still requires the system to deal with bugs at run time.

An alternative to run time testing, i.e., dynamic testing, are static approaches. Static analysis reasons over the source code of software, and tries to find property violations for all possible inputs. The tools have often a narrow focus on properties like access of invalid pointers, out of bounds array accesses, integer overflows, or missing variable initializations. Other tools, especially ones based on logical formalizations of software and its properties, tend to pursue more ambitious goals. In these scenarios, proving a logical theorem usually corresponds to showing that a property holds, while disproving a theorem in the best case gives rise to a concrete counterexample, e.g. the trace of a program run, which exposes a bug. Among these approaches are, for example, tools that execute a program symbolically, i.e., simultaneously for all possible inputs, and are able to perform automatic unit test generation with high (or full) code coverage based on this information [70, 35], or to verify with the help of annotations and interactive user input that contracts or properties like loop termination hold [20]. Other approaches translate a program its properties and run a theorem prover on the resulting logical formula. For example, in extended static checking simple

properties, like e.g. dereferencing of null pointers, can be inferred and checked automatically, while in more complicated cases, especially termination arguments and contracts, user annotations are required [28, 6]. The model checking approach models a system as a state machine, and properties as reachability of states. The symbolic representation of state machines makes it possible to reason over infinite states, and automatic abstraction and abstraction refinement make it possible to scale to large systems, by focusing only on details which are relevant for the currently investigated properties. Applying model checking is most prevalent in hardware verification [36], but is also employed successfully for software verification [5, 43].

If a method can show that a property holds for all inputs, it is not merely testing for bugs but (partially) verifying software correctness. This is a crucial difference, as verification provides a much higher degree of confidence in robustness and correctness than testing. Many kinds of static analysis can be quite efficient and well worth the effort for some classes of bugs, in particular those which can be detected with a local analysis of the source. In contrast, full program analysis for complex properties usually requires user guidance, for example through annotations or interactive proofs, and comes with high computational and financial costs. This is not surprising, considering that problems involving e.g. termination or integer constraints are often undecidable. One goal of current research is to extend the reach of verification techniques, but research is also focused on making algorithms and tools more efficient and easier to use in practice. While there has been significant success in both regards [22], verification is at the moment only feasible and being used in parts of industry where failure is an

extreme cost factor, for example in hardware companies or the air traffic sector.

For the remainder of this thesis, we will mostly take an application-agnostic stance, and evaluate our approaches on standard sets of benchmarks provided by the automated reasoning community. Inspecting the applications of logics in the verification methods sketched above, it becomes clear that reductions to propositional logic, i.e., the SAT problem, are widely used, especially in industrial settings. For many applications, notably in software verification, propositional logic is not sufficiently expressive. Satisfiability Modulo Theories (SMT) provides extensions of propositional logic which make it possible to reason natively and efficiently about aspects of properties involving arithmetic, memory management, and many data structures [23]. A shortcoming of both approaches is the limited support for quantification, i.e., the capacity to reason about all possible values of a variable, not only finitely many. While this is at least in theory adequate for finite data types like fixed sized integers, it usually leads to an explosion of the logical encoding, as it has to resort to quantifier instantiation. Furthermore, it does not always suffice to reason about unbounded and infinite types, such as the memory heap or inductive data structures.

This thesis is concerned with combining SAT and SMT with first-order approaches to support quantification natively, such that the algorithm and improvements that make SAT and SMT solvers efficient and successful carry over to this extended setting. As most efficient SAT and SMT systems are based on the DPLL architecture, our lifting to the first-order setting tries to build on that architecture. The lifting of DPLL techniques to the Model Evolution (ME) calculus is already quite sophisticated, and is accompanied by an efficient implementation

incorporating adaptations of crucial DPLL ingredients. In contrast, the lifting of SMT is still in its early stages. We have devised the Model Evolution with Linear Integer Arithmetic (ME(LIA)) calculus in the spirit of DPLL and ME. Its logic is the extension of propositional logic with quantifiers, linear arithmetic (LIA), and uninterpreted predicate and constant symbols. While we do not have an implementation and an empirical evaluation of the calculus yet, we give detailed guide lines of how to obtain efficient procedures and implementations, based on our experience with implementations of SAT, SMT, and ME solvers.

1.2 Outline

The thesis builds on the ME calculus, which has been developed by Peter Baumgartner and Cesare Tinelli in [16, 18]. Darwin, the implementation of ME, is previous work developed by the author as part of his Master's thesis. The remainder of this thesis consists of new contributions based on publications co-authored by the author.

In Chapter 2 we will be concerned with ME. We will first briefly introduce the DPLL architecture, and then give a quick overview of the ME calculus, as far as needed for the further discussion.

The first contribution in this chapter, in Section 2.3, is based on [10, 9]. We explain several possible ways of adapting one of the core ingredients of efficient DPLL implementations, learning lemmas from conflicts. Lemma-learning is crucial for learning from mistakes made during the search. It enables a solver to prune search space by avoiding making the same, or similar, mistakes in the future. Like DPLL, ME works with clauses. It processes quantifiers by unification-guided clause instance generation. We show how to lift lemma-learning from a

ground, i.e., propositional, to a first-order setting. The algorithm has the flexibility to learn lemmas which are closely based on the clause instances which caused a conflict, but also provides the flexibility to learn more general lemmas, thus incorporating real learning. Empirically, the later comes with significant computational overhead, so that the former is more effective in practice.

The second half of the chapter, Section 2.4, is based on [13]. We use finite model finding as an example application for which SAT-based approaches, and in particular the DPLL-based solver Paradox, are highly effective in practice. We adapt SAT methods to the ME setting, explore which adaptations are useful, and add some ME specific improvements. Finally, we show that Darwin benefits from the lemma-learning capabilities introduced in the previous section when searching for finite models. Furthermore, evaluating Darwin against competitive SAT implementations shows complementary behavior. While Darwin scales better in terms of memory performance, it can be significantly slower in solving a problem. This is not surprising, given its (additional) support of quantifiers.

Chapter 3 introduces the ME(LIA) calculus. While it is based on the presentation of the calculus in [14], it has been modified significantly. Foremost, in ME as well as in the original presentation of ME(LIA), clause instantiation was implicit in the concept of a context unifier. In its new incarnation, the calculus creates and maintains clause instances explicitly. In our experience, this makes the presentation much clearer and the calculus easier to understand. Additionally, we have extended ME(LIA) with notions that were the basis for obtaining an efficient implementation of ME. These include more general redundancy criteria, a concept of universality which makes it possible to strongly constrain the search

space, and additional inference rules which serve to exploit universality and to process constraints more efficiently. We have also generalized the input logic of the calculus, by substituting a notion of compactness for the original restriction that all free constants may range only over finite domains of integers. Compactness still guarantees completeness; it covers the case of bounded constant domains as well as other classes of restrictions to the ME(LIA) logic as special cases. We present a basic proof procedure and prove it complete, as well as a sophisticated proof procedure and its instantiations to the above mentioned classes. Finally, the ingredients which are necessary for an efficient and robust implementation are discussed, including heuristics, learning, non-chronological backtracking, and proof-generation.

1.3 Preliminaries

In this section we introduce general formal preliminaries which are needed for the discussions in the following chapters. We will be mostly concerned with standard notions of first-order logic in automated deduction [60]. Notions that are too specific to be of interest in all chapters, like for example LIA constraints, will be introduced only where needed. We first introduce the syntax of first-order logic, followed by its semantics, and finally provide some notions of calculi. Intuitively, terms denote specific objects, formulas express statements about specific and arbitrary objects via predicates and quantification, and calculi reason about the truth of statements according to the semantics.

1.3.1 Syntax

A *signature* (Σ) consists of a set of function symbols ($f, g, h, \dots$) and a set of predicate symbols ($P, Q, R, \dots$). A function symbol of arity 0 is also called a *constant* ($a, b, c, \dots$), a predicate symbol of arity 0 is also called a *propositional variable*. We assume a denumerable infinite set W of variables ($x, y, z, \dots$).

A (Σ-)*term* is either a variable x or an expression $f(t_1, \dots, t_n)$, where f is an n-ary function symbol, and $t_1, \dots, t_n$ are terms. An *atom* is an expression $P(t_1, \dots, t_n)$, where P is an n-ary predicate symbol, and $t_1, \dots, t_n$ are terms. A *formula* is defined inductively. If $F, F_1, \dots, F_n$ are formulas, then all of the following are formulas:

- $\top$ and $\bot$,
- an atom $P(t_1, \dots, t_n)$,
- a negation $\neg F$,
- a conjunction $F_1 \wedge \dots \wedge F_n$,
- a disjunction $F_1 \vee \dots \vee F_n$,
- a universal quantification $\forall x F$,
- an existential quantification $\exists x F$

The implication $F \to G$ of two formulas is defined as $\neg F \vee G$. In the case of a quantification $\forall x F$ or $\exists x F$, F and all its sub-formulas and sub-terms are in the *scope* of the quantifier, and x is *bound* in F. A variable is *free* in a formula if it is not bound. A term or formula is *ground* or *propositional* if it contains no variables, a formula is *closed* if it contains no free variables. We may introduce a

fresh variable, if we want to use a variable that is not contained in any term or formula considered so far.

A *literal* is an atom A (*positive*) or its negation $\neg A$ (*negative*), or $\top$ or $\bot$. A positive (negative) literal has positive (negative) *polarity*. If A is an atom, then $\neg A$ is the *complement* of A, and vice versa. A *clause* is a disjunction of literals $L_1 \vee \cdots \vee L_n$. A *unit* clause consists of one literal, a *Horn* clause contains at most one positive literal. All variables occurring in the literals of a clause are implicitly considered to be universally quantified, e.g., $P(x) \vee Q(x, y)$ stands for $\forall x \forall y (P(x) \vee Q(x, y))$. A clause is often represented as the set of its literals. A set of clauses (or formulas) stands for the conjunction of the contained formulas, the empty clause stands for $\bot$, and the empty clause set stands for $\top$.

Propositional logic can be seen as the subset of first-order logic where the signature contains only propositional variables.

1.3.2 Semantics

A *(Σ)-structure* $\mathfrak{Z} = (D, \mathfrak{I})$ for a signature Σ consists of a domain D and an interpretation $\mathfrak{I}$. The *domain* or *universe* D of a structure $\mathfrak{Z}$ is a non-empty set. The *interpretation* $\mathfrak{I}$ of a Σ-structure $\mathfrak{Z}$ with domain D maps each n-ary function symbol f to an n-ary function $f^{\mathfrak{I}}$ from D^n to D, and each n-ary predicate symbol p to an n-ary relation $p^{\mathfrak{I}}$ over D^n. The domain of an interpretation $\mathfrak{I}$ may also be denoted by $|\mathfrak{I}|$. A *variable assignment* $\mathfrak{A}$ for a Σ-structure $\mathfrak{Z} = (D, \mathfrak{I})$ maps each variable of W to an element of D.

The value $t^{\mathfrak{A}}$ of a Σ-term t for a variable assignment $\mathfrak{A}$ is defined inductively. If t is a variable x then $t^{\mathfrak{A}}$ is $\mathfrak{A}(x)$, and if t is a term $f(t_1, \ldots, t_n)$ then $t^{\mathfrak{A}}$ is $f^{\mathfrak{I}}(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}})$. We write $\mathfrak{A}[x \mapsto d]$ to denote the variable assignment which is

identical to $\mathfrak{A}$ except that it maps x to d.

We assume for simplicity w.l.o.g. that no variable occurs free and bound in a formula, or bound by more than one quantifier. This can be ensured by standard renaming methods. The truth value of a formula for a structure $\mathfrak{Z} = (D, \mathfrak{I})$ and a variable assignment $\mathfrak{A}$ is defined inductively. A formula is either *true* (*satisfied*, *holds*) or it is *false* (*falsified*) in $(\mathfrak{Z}, \mathfrak{A})$.

- $\top$ is *true*,
- $\bot$ is *false*,
- an atom $P(t_1, \ldots, t_n)$ is true iff $(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}}) \in P^{\mathfrak{I}}$,
- a negation $\neg F$ is true iff F is false,
- a conjunction $F_1 \wedge \cdots \wedge F_n$ is true iff all $F_1, \ldots, F_n$ are true,
- a disjunction $F_1 \vee \cdots \vee F_n$ is true iff one of $F_1, \ldots, F_n$ is true,
- a quantification $\forall x F$ is true iff for all $d \in D$, F is true in $(\mathfrak{Z}, \mathfrak{A}[x \mapsto d])$,
- a quantification $\exists x F$ is true iff for some $d \in D$, F is true in $(\mathfrak{Z}, \mathfrak{A}[x \mapsto d])$,

A closed Σ-formula is *valid* if it is true in $(\mathfrak{Z}, \mathfrak{A})$ for each structure $\mathfrak{Z}$ and variable assignment $\mathfrak{A}$ for Σ, it is *satisfiable* if it is true in some $(\mathfrak{Z}, \mathfrak{A})$, and it is *unsatisfiable* otherwise. $(\mathfrak{Z}, \mathfrak{A})$ is a *model* of a closed formula F if F is true in $(\mathfrak{Z}, \mathfrak{A})$, also written $(\mathfrak{Z}, \mathfrak{A}) \models F$. A closed formula F *entails* a closed formula G, written $F \models G$, if each model of F is a also model of G. Then G is a *consequence* of F. F and G are logically *equivalent* if each model of F is a model of G and vice versa. F and G are *equisatisfiable*, if either both F and G are satisfiable, or neither is.

The *Herbrand universe* of a signature Σ is the set of all ground terms over Σ. The *Herbrand base* of a signature Σ is a subset of all ground atoms over Σ. A *Herbrand interpretation* $\mathfrak{I}$ of a signature Σ interprets all terms as themselves, i.e., $t^{\mathfrak{I}} = t$ for each ground term t. That is, Herbrand interpretations differ only in the interpretation of predicate symbols, as induced by the Herbrand base: a ground atom is true if it is in the Herbrand base, it is false otherwise. A *Herbrand structure* of a signature Σ consists of a Herbrand universe and a Herbrand interpretation.

1.3.3 Calculus

A *calculus* defines a set of derivation rules that operate on some formal representation of a first-order formula F. A *proof procedure* for a calculus is an algorithm that constructs *derivations* by applying the rules of the calculus. A *refutation* is (an artifact of) a derivation that proves the unsatisfiability of a formula, or, equivalently, its validity. A calculus is *(refutationally) complete*, if it has a proof procedure that terminates with a refutation for each unsatisfiable formula. A calculus is *sound*, if when it derives a refutation for a formula, then the formula is unsatisfiable. A calculus is a *decision procedure* if it is sound and terminates for any formula. Proving a formula valid usually gives rise to a *proof* artifact, a calculus specific witness for the unsatisfiability of a formula.

While propositional logic is decidable, first-order logic is undecidable. It is in fact semi-decidable, i.e., there are complete calculi but no decision procedures.

Any set of first-order formulas can be converted into an equisatisfiable clause set, i.e., the conjunction of the formulas in the first set is equisatisfiable to the conjunction of the clauses in the second set. Thus calculi usually operate

on (finite) clause sets, as this leads to significant simplifications in devising a calculus and proving its correctness, i.e., its soundness and completeness. Furthermore, Herbrand interpretations are a purely syntactic concept, and a formula is satisfiable if and only if it has a Herbrand model. Therefore, operations of a first-order calculus are commonly semantically justified by the search for a Herbrand interpretation, conceptually or factually.

Two main categories of calculi are resolution and tableaux. Resolution calculi infer clauses as consequences of the initial clause set until the clause set is saturated under some redundancy criterion. Sound and complete resolution calculi derive the empty clause as the consequence of any unsatisfiable clause set. As mentioned above, this is equivalent to being able to prove the validity of a formula, as a formula is valid if and only if its negation is unsatisfiable. Tableaux calculi construct *derivation trees* by case analysis on input clauses or formulas. If a tree can be closed, the unsatisfiability of the input clause set has been shown. A closed tree serves as a proof of the unsatisfiability of the input. Furthermore, nodes are labeled with *inference rules* (*derivation rules*), which define the relation between a node (*premise*) and its children (*conclusion*s). ME is a sound and complete calculus in the tableaux tradition. It constructs binary trees by case analysis of instances of literals from the input clause set. Each branch can be closed individually, and if a branch is open and exhausted under some redundancy criteria, a Herbrand model can be extracted from its leaf. The same applies in essence to ME(LIA), modulo the detail that it operates on a combination of first-order logic and arithmetic. Thus, ME(LIA) does not work with Herbrand models, as will be explained in Section 3.4.

CHAPTER 2
MODEL EVOLUTION

2.1 Introduction

In propositional satisfiability the DPLL procedure, named after its authors Davis, Putnam, Logemann, and Loveland [33, 32], is the dominant method for building (complete) SAT solvers. Its popularity is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics for reducing the search space. These solvers are so powerful that many developers of automated reasoning-based tools use them as back-ends to solve *first-order* satisfiability problems, albeit often in an incomplete way, by means of ingenious domain specific translations into propositional logic [72, 37, 48]. The treatment of quantifiers is highly inefficient, however, because it is based on enumerating all possible ground instances of an input formula's clause form. The Model Evolution calculus (ME) properly lifts the DPLL procedure to first-order clausal logic. While it relies on clause instantiation as well, this process is guided by unification and produces instances only as necessary. Furthermore, most of the powerful search heuristics that make DPLL effective at the propositional level can be successfully adapted to the first-order case, as shown with Darwin [12, 11], an implementation of the calculus.

In this chapter we are concerned with lifting one further improvement from DPLL to ME, learning, and demonstrating that applications benefit from switching from propositional logic to first order logic as a target language. We will first briefly introduce the DPLL architecture and its most important ingredients, as this is the basis for our efficient implementation of ME, as well as the discussion

of implementations of ME(LIA) (see Section 3.7). As we are interested in proof procedures, i.e., in making design decisions which result in an efficient implementation, the presentation of the calculus based on derivation rules as in [18] is too abstract for our purposes. We will therefore first introduce the ME calculus as an abstract transition system, in the style of [58]. We will then show in Section 2.4 how to integrate learning in ME by adding new rules to the transition system. We will accompany this by a discussion of the design choices and an experimental evaluation of the implementation of learning in Darwin. In Section 2.5 we will show that Darwin is competitive with and complementary to DPLL based approaches for the application of finite model finding. This supports our thesis that a translation to first-order logic can be beneficial for applications that are currently based on a translation to propositional logic.

2.2 The DPLL Procedure

The DPLL procedure can be used to decide the satisfiability of sets of propositional clauses. The essential ingredients of the procedure are recursive reduction to smaller problems, unit propagation, and incremental construction of Herbrand models.

The procedure can be described as a sequent-style calculus [71] with three core rules, Split, Assert, and Close, where a sequent is of the form $\Lambda \vdash \Phi$. Conceptually, it creates a derivation tree where each node corresponds to a literal of an input clause set Φ. We call the set of literals on a branch a *context* Λ. The conjunctions of the literals of a context Λ induce a (partial) Herbrand interpretation. It can be extended to those Herbrand interpretations, where the Herbrand base contains P for each positive literal P in Λ and does not contain

P for each negative literal $\neg P$ in Λ. We call these Herbrand interpretations the models of a context (or branch). The Split rule creates binary branching points, where the two children nodes are labeled with a literal and its complement. The Assert rule extends a branch with one child node, which is labeled with a literal implied by all extensions of the partial Herbrand model of the branch and Φ. If there is no such model for a branch, it is closed with the Close rule. To obtain a decision procedure it suffices in principle to apply Split to all literals in the input, and to apply Close whenever it is applicable. As each branch of the tree induces a (complete) Herbrand model, and each possible Herbrand model for the input signature is induced by one branch, the input clause set is unsatisfiable if all branches of the tree are closed. Furthermore, each open branch induces a Herbrand model.

It is of course not efficient in practice to build a full exponential binary decision tree for all literals in the input. Implementations try to avoid constructing an exponential tree, generally speaking, by closing branches as early as possible, performing unit propagation with Assert, learning from closed branches (lemma learning), pruning parts of the search space (backjumping), and heuristically choosing Split literals to facilitate the above. The impressive capabilities of modern DPLL solvers arise from efficient and effective implementations of all these techniques and more, by employing sophisticated data structures and from intricate interactions between the different components [57, 66].

All implementations, conceptually, construct the derivation tree by a depth-first exploration. An application of Close is justified by a conflicting input clause that is falsified in all Herbrand models of the branch. The conflict is detected

by the same mechanism that makes unit propagation efficient [73], in essence by watching which literals of a clause are not complementary with branch literals. Thus a branch is closed as soon as one clause is falsified by all branch models. When backtracking along the branch, the analysis of the conflict enables backjumping. Instead of continuing the derivation on the right side of the most recent left Split application on the branch, it is continued on the right side of the most recent left Split that is relevant for the conflict. Furthermore, basically by the same mechanism it is possible to learn lemmas, clauses which via unit propagation effectively help to avoid repeating the decisions which led to the current conflict. Finally, conflict analysis plays an important rule in dynamically adapting the heuristics that decides the order of Split and Assert applications.

2.3 The Model Evolution Calculus

2.3.1 Introduction

The Model Evolution calculus is a lifting of the DPLL calculus to the first-order level. This is achieved by suitable first-order versions of the rules Split, Assert, and Close. Similarly to DPLL, the derivation rules of the Model Evolution calculus apply to and produce a derivation tree with sequents of the form $\Lambda \vdash \Phi$. However, this time the literals and clauses may contain variables, with the context context Λ in a sequent $\Lambda \vdash \Phi$ determining a single interpretation I_Λ. The purpose of the main rules of the calculus is to recognize when I_Λ is not a model of Φ, via syntactic unification between elements of Λ and Φ, and evolve it into an actual model. We will in this chapter ignore the optional simplification rules of the calculus, presented in [18]. They are not relevant to the discussion and complicate the presentation of the proof procedure below.

2.3.2 Preliminaries

The ME calculus uses two disjoint, infinite sets of variables, the set X of *universal* variables $(x, y, z, \dots)$, and the set V of *parametric* variables (or just *parameters*) $(u, v, w, \dots)$. We denote by Σ^{sko} the expansion of Σ obtained by adding to Σ an infinite number of (*Skolem*) constants not already in Σ. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t's variables and by $\mathcal{P}ar(t)$ the set of t's parameters. A term is *universal* if it contains no parameters, it is *parametric* otherwise. A *substitution* $(\sigma,\ \rho,\ \dots)$ is a mapping from variables to terms, which is the identity for all but finitely many places. We will denote by $\{w_1 \mapsto t_1,\ \dots,\ w_n \mapsto t_n\}$ the substitution σ such that $w_i\sigma = t_i$ for all $i = 1, \dots, n$ and $w\sigma = w$ for all $w \in X \cup V \setminus \{w_1, \dots, w_n\}$. Also, we will denote by $\mathcal{D}om(\sigma)$ the set $\{w_1, \dots, w_n\}$ and by $\mathcal{R}an(\sigma)$ the set $\{w_1\sigma, \dots, w_n\sigma\}$. The result of the *application* of a substitution σ to a term t, written $t\sigma$, is obtained from t by replacing each occurrence of each variable x in t by $\sigma(x)$. If s and t are two terms, *s is more general than t* ($s \gtrsim t$) iff there is a substitution σ such that $s\sigma = t$. We say that s is a *variant of* t ($s \approx t$) iff there is a renaming ρ such that $s\rho = t$. We write $s \gtrsim\!\!\!\!\!\!\sim\ t$ if $s \gtrsim t$ but $s \not\approx t$. We say that *s is parameter-preserving more general than t* ($s \geq t$), iff there is a parameter-preserving substitution σ such that $s\sigma = t$. When $s \geq t$ we will also say that t is a *p-instance of* s. We say that s is a *parameter-preserving variant, or p-variant, of* t ($s \simeq t$), iff there is a parameter-preserving renaming ρ such that $s\rho = t$. We write $s \geq\!\!\!\!\!\!\sim\ t$ if $s \geq t$ but $s \not\simeq t$. A *unifier* σ of two terms s and t is a substitution such that $s\sigma = t\sigma$. A unifier σ of two terms s and t is *most general*, if there is no unifier ρ of s and t such that $s\sigma$ is an instance of $s\rho$ and ρ is not a variant of σ. All of the above is

extended to formulas in the obvious way.

An ME proof procedure can be described abstractly as a transition system over states. An *annotated literal* is a literal with an annotation which marks it as a *decision* or a *propagated* literal. An *(ordered) context* is a sequence of annotated literals. When convenient, we will see an ordered context simply as a set of literals, ignoring both the annotations and multiple occurrences of its elements. The concatenation of two ordered contexts will be denoted by simple juxtaposition. When we want to stress that a context literal L is annotated as a decision literal we will write it as L^{d}. With an ordered context of the form $\Lambda_0\, L_1\, \Lambda_1 \cdots L_n\, \Lambda_n$, where $L_1, \ldots L_n$ are all the decision literals of the context, we say that the literals in Λ_0 are at *decision level* 0, and those in $L_i\, \Lambda_i$ are at decision level i, for all $i = 1, \ldots, n$. A *state* is of the form $\bot$ or $\Lambda \vdash \Phi$, where $\bot$ is a distinguished fail state, Λ is an *(ordered) context*, and Φ is a clause set. For a given state S, a *transition rule* defines whether there is a transition from S by this rule and, if so, to which state S'. A proof procedure is then a *transition system*, a set of transition rules defined over some given set of states. Given a transition system R, we denote by $\Longrightarrow_{\mathrm{R}}$ the transition relation defined by R. If $\Longrightarrow$ is a transition relation between states we write, as usual, $S \Longrightarrow S'$ instead of $(S, S') \in \Longrightarrow$. We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$. We call any sequence of transitions of the form $S_0 \Longrightarrow_{\mathrm{R}} S_1, \quad S_1 \Longrightarrow_{\mathrm{R}} S_2, \quad \ldots$ a *derivation in R*, and denote it by $S_0 \Longrightarrow_{\mathrm{R}} S_1 \Longrightarrow_{\mathrm{R}} S_2 \Longrightarrow \ldots$

Where L is a literal and Λ a context, we will write $L \in_{\sim} \Lambda$ if L is a p-variant of a literal in Λ. A literal L is *contradictory with* a context Λ iff $L\sigma = \overline{K}\sigma$ for some $K \in_{\sim} \Lambda$ and some p-preserving substitution σ. A context Λ is contradictory

if one of its literals is contradictory with Λ. Every (ordered) context the proof procedure works with will start with a *pseudo-literal* of the form $\neg v$. In examples we will usually not write $\neg v$ explicitly. Each non-contradictory context starting with $\neg v$ defines a Herbrand interpretation I^Λ, see [18] for more details. In a state of the form $\Lambda \vdash \Phi$, the interpretation I^Λ is meant to be a *candidate model* for Φ. The purpose of the proof procedure is to recognize whether the candidate model is in fact a model of Φ, or whether it potentially falsifies a clause of Φ. The latter situation is detectable syntactically through the computation of *context unifiers*.

Definition 2.3.1 (Context Unifier) Let Λ be a context and

$$C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_n$$

a parameter-free clause, where $0 \leq m \leq n$. A substitution σ is *a context unifier of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$* iff there are fresh p-variants $K_1, \ldots, K_n \in_\simeq \Lambda$ such that

1. σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
2. for all $i = 1, \ldots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,
3. for all $i = m+1, \ldots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$.

A context unifier σ of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ is *admissible (for* Split*)* iff for all distinct $i, j = m+1, \ldots, n$, $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$. □

If σ is a context unifier with remainder D of a clause C against a context Λ, we call each literal of D *a remainder literal* of σ. We say that C is *conflicting (in Λ because of σ)* if σ has an empty remainder.

Example 2.3.2 Let Λ be $\{\neg v,\ p(v_1, u_1)\}$ and let C_1 be $r(x) \vee \neg p(x, y)$, where x, y, x_1 are universal and v, v_1, u_1, v_2 are parametric. Then, the substitutions

$$\begin{aligned} \sigma_1 &:= \{v \mapsto r(x),\ v_1 \mapsto x,\ u_1 \mapsto y\} \\ \sigma_2 &:= \{v \mapsto r(v_1),\ x \mapsto v_1,\ u_1 \mapsto y\} \\ \sigma_3 &:= \{v \mapsto r(v_1),\ x \mapsto v_1,\ y \mapsto u_1\} \end{aligned}$$

are all context unifiers of C_1 against Λ with respective remainders $r(x) \vee \neg p(x, y)$, $r(v_1) \vee \neg p(v_1, y)$, and $r(v_1)$. But only σ_2 and σ_3 are admissible. The context unifier σ_1 is not admissible because its remainder literals are not variable-disjoint. □

2.3.3 A Basic Proof Procedure

A basic proof procedure for ME is the transition system B, defined by the rules Decide, Propagate, Backjump and Fail below. The relevant derivations in this system are those that start with the state $\{\neg v\} \vdash \Phi$, where Φ is the input clause set.

Decide: $\quad \Lambda \vdash \Phi,\ C \vee L \implies \Lambda\ (L\sigma)^{\mathsf{d}} \vdash \Phi,\ C \vee L$

where

1. σ is an admissible context unifier of $C \vee L$ against Λ with at least two remainder literals,
2. $L\sigma$ is a remainder literal,
3. neither $L\sigma$ nor $(\overline{L\sigma})^{\text{sko}}$ is contradictory with Λ.

We call the literal $L\sigma$ above a *decision literal* of the context unifier σ and the clause $C \vee L$. This rule corresponds to an application of the left-hand side of

the Split rule in ME, with the additional restriction that the context unifier must have at least two remainder literals. Decide makes the non-deterministic decision of adding the literal $L\sigma$ to the context. It is the only rule that adds a literal as a decision literal.

Propagate: $\quad \Lambda \vdash \Phi,\, C \vee L \implies \Lambda,\, L\sigma \vdash \Phi,\, C \vee L$

where

1. σ is an admissible context unifier of $C \vee L$ against Λ with a single remainder literal $L\sigma$,
2. $L\sigma$ is not contradictory with Λ,
3. there is no $K \in \Lambda$ s. t. $K \geq L\sigma$.

We call the literal $L\sigma$ in the rule above the *propagated literal* of the context unifier σ and the clause $C \vee L$. Propagate models variations of Split and Assert in ME. It corresponds to applying the left-hand side of Split in ME with a context unifier with a unit remainder, and ignoring the right-hand side, an optimization justified as Close is immediately applicable to the right-hand side. Furthermore, it puts less stringent restrictions on Assert, making it possible to assert parametric literals. For simplicity, we ignore here the case in which L is negative. That case somewhat complicates the definition of Propagate and is needed neither for the proof procedure's completeness nor for describing the results of this work.

Backjump: $\quad \Lambda\, L^{\mathsf{d}} \Lambda' \vdash \Phi,\, C \implies \Lambda\, \overline{L}^{\text{sko}} \vdash \Phi,\, C$

where

1. C is conflicting in $\Lambda\, L^{\mathsf{d}} \Lambda'$ but not in Λ

This rule corresponds to the application of the Close rule in ME, followed by a right-hand side Split application higher up in the closed branch. Backjump models both chronological and non-chronological backtracking by allowing but not requiring that the undone decision literal L is the most recent one. Note that L's complement is added as a propagated literal, after all (and only) the universal variables of L have been Skolemized, which is needed for soundness. More general versions of Backjump are conceivable, for instance along the lines of the backjumping rule of Abstract DPLL [58]. Again, we present this one here mostly for simplicity.

Fail: $\quad \Lambda \vdash \Phi, C \implies \bot$

where

1. C is conflicting in Λ,

2. Λ contains no decision literals.

This rule corresponds to the application of the Close rule in ME to the last unexplored branch of the derivation tree, with all other branches being already closed.

Restart: $\quad \Lambda \vdash \Phi \implies \{\neg v\} \vdash \Phi$

Restart is used to generate fair derivations that explore the search space in an iterative-deepening fashion.

One can show that there are (deterministic) rule application strategies for this transition system that are refutationally sound and complete. That is, they

reduce a state of the form $\{\neg v\} \vdash \Phi$ to the state $\bot$ if and only if Φ is unsatisfiable. Furthermore, for all (finite) derivations ending with an irreducible state of the form $\Lambda \vdash \Phi$, i.e., a state to which no transition rule applies, Λ determines a Herbrand model of Φ.

2.4 Lemma-Learning

2.4.1 Introduction

In order to obtain an efficient proof procedure for ME it is crucial to adapt effective techniques that have been developed in the context of DPLL. For some techniques, in particular backjumping, this is relatively straightforward and does lead to performance improvements, as has been shown in [12]. Complementing backjumping with a lemma-learning mechanism turns out to be significantly more complicated. Firstly, because the notion of lemmas and lemma-generation has to be lifted to first-order logic, and the mechanism in particular has to take into account that ME makes use of two kinds of variables with different semantics. Secondly, first-order lemma-generation depends on unification, which adds a significant computational overhead that can offset the potential advantages of learning. We will present and prove correct three lemma-generation procedures for ME with various degrees of power, effectiveness in pruning the search space, and computational overhead. Even if formally correct, each of these procedures presents issues and complications that do not exist at the propositional level, but need to be addressed for learning to be effective for ME in practice. We also present experimental results based on the implementation of all three lemma-generation procedures in Darwin.

2.4.2 Related Work

The paper [42] is about satisfiability checking of function-free first-order formulas. Instead of the widely used reduction to propositional satisfiability followed by running a SAT solver, the authors propose to keep some of the original formulas (clauses) and extend a SAT solver to reason with them natively, on the first-order level, instead of grounding them right away. Their main result is that this idea may well pay off when problem instances become larger.

There is potential to connect their approach to ours by observing that these first-order formulas obviously need not come from the input formula—they could be lemma clauses learned by the techniques we propose. From the perspective of the Model Evolution calculus, the mentioned result allows to speculate that even for ground derivations the learning of non-ground lemma clauses, which is supported by our techniques, may pay off when properly implemented.

Regarding directly related work, i.e., conflict-driven lemma-learning in first-order theorem proving, not much has been done. The only approaches we are aware of have been formulated for the model elimination calculus. One of them is described in [2] and consists of the "caching" and "lemmaizing" techniques. Caching means to store solutions to sub-goals (which are single literals) in the proof search. The idea is to look up a solution (a substitution) that solves the current sub-goal, based on the solution of a previously computed solution of a compatible sub-goal. This idea of replacing search by look-up is thus conceptually related to lemma learning as we consider it here. However, caching corresponds to learning of *unit* clauses, and it works only for Horn clause sets. Closer to our approach is the lemmaizing technique, which allows to generate and use

new clauses as the derivation proceeds. Lemma clauses are generated on branch closure and by taking the so-called A-literals of a branch into consideration. The basic motivation for doing so is, like in our approach, to represent a sub-proof by a single clause. Unsurprisingly, the lemmas are consequences of the input clauses, potentially there are a lot of them, and their use may and should be subject to (arbitrary) heuristics. As far as we can tell from [2], the use of lemmas there seems having been restricted to *unit* lemmas, perhaps for pragmatic reasons, although the mechanism has been defined more generally (already in [51]). A related approach of learning for model elimination, which generalizes this caching technique, is described in [49]. By caching the solutions in a more context-dependent ("local") way, it works for non-Horn clauses, too. In particular the variant of *failure caching*, which can be used to conclude that a sub-goal does not have a solution, turned out to be very useful in practice.

Another source for related work is Explanation-Based Learning (EBL), an established deductive learning approach in Artificial Intelligence. Generally speaking, it allows for learning logical descriptions of a concept from the description of a single concept instance and a preexisting knowledge base. In our discussion, we follow the rather general and powerful framework in [64]. Using standard terminology, its basis consists essentially of the language of definite logic programs and the calculus of SLD-resolution. EBL then means to derive from a given SLD proof a (definite) clause which shall represent parts of the proof or even generalizations thereof. The rationale is to derive clauses that are of high *utility*, that is, help to find shorter proofs of similar theorems without broadening the search space too much. Deriving such clauses can be explained using resolu-

tion terminology, although [64] use their own language. It basically amounts to *partially* repeating parts of the given SLD proof and further modifying the clause derived this way in a sound way. For instance, one of the heuristics prescribed is to exhaustively apply resolution steps with binary clauses, i.e., implications with one body literal. Another heuristics amounts to factoring, while still another one is based on removing redundant sub-goals. The connection to our approach becomes apparent from the *regression processes* defined below. Structurally, these are SLD-derivations and play a comparable role in deriving lemma clauses. It should thus be not too difficult to adapt the heuristics developed in [64] to our setting. The heuristics currently used in our approach are modeled after the ones successfully used for lemma-learning in propositional SAT solvers.

2.4.3 Adding Learning to ME Proof Procedures

To illustrate the potential usefulness of learning techniques for a transition system like the system B defined in the previous subsection, it is useful to look first at an example of a derivation in B.

Example 2.4.1 Let Φ be a clause set containing, among others, the clauses:

$$(1) \quad \neg Q(x) \vee R(x, y)$$

$$(2) \quad \neg P(x) \vee \neg R(y, x) \vee S(y)$$

$$(3) \quad \neg R(x, y) \vee T(x)$$

$$(4) \quad \neg S(x) \vee \neg T(x)$$

Figure 2.1 provides a trace of a possible derivation of Φ. We use the notation $t(x)$ to suggest that t is a term containing the variable x. The first column shows the literal added to the context by the current derivation step, the

second column specifies the rule used in that step, and the third one indicates which instance of a clause in Φ was used by the rule. From the instance alone it should be easy to see which context unifier was used in the last four rule applications. A row with ellipses stands for zero or more intermediate steps. Note that Backjump *replaces* the whole sub-sequence $Q(u)^{\mathsf{d}}\, R(u,y)\, S(u)\, T(u)$ of the current context with $\neg Q(u)$.

Context Literal	**Derivation Rule**	**Clause Instance**
$\cdots$	$\cdots$	$\cdots$
$P(t(x))$	Propagate	instance $P(t(x)) \vee \cdots$ of some clause in Φ where $t(x)$ is a term in the variable x.
$\cdots$	$\cdots$	$\cdots$
$Q(u)^{\mathsf{d}}$	Decide	instance $Q(u) \vee \cdots$ of some clause in Φ
$R(u,y)$	Propagate	instance $\neg Q(u) \vee R(u,y)$ of (1)
$S(u)$	Propagate	instance $\neg P(t(x)) \vee \neg R(u,t(x)) \vee S(u)$ of (2)
$T(u)$	Propagate	instance $\neg R(u,y) \vee T(u)$ of (3)
$\neg Q(u)$	Backjump	instance $\neg S(u) \vee \neg T(u)$ of (4)

Figure 2.1: Trace of a derivation in the system B.

It is clear by inspection of the trace that any intermediate decisions made between the additions of $P(t(x))$ and $Q(u)$ are irrelevant in making clause (4) conflicting at the point of the Backjump application. The fact that (4) is conflicting depends only on the decisions that lad to the propagation of $P(t(x))$—say, some decision literals $L_1, \ldots, L_n$ with $n \geq 0$—and the decision to add $Q(u)$. This means that the decision literals $L_1, \ldots, L_n, Q(u)$ will eventually produce a conflict, i.e., make some clause conflicting, in any context that contains them. The basic goal of this work is to define efficient conflict analysis procedures that can come to this conclusion automatically and store it in the system in such a way that Backjump is

applicable, possibly after some propagation steps, whenever the current context happens to contain again the literals $L_1, \ldots, L_n, Q(u)$. Even better would be the possibility to avoid altogether the addition of $Q(u)$ as a decision literal in any context containing $L_1, \ldots, L_n$, and instead to add the literal $\neg Q(u)$ as a propagated literal. We discuss how to achieve these in the rest of the section. □

Within the abstract framework of Section 2.4.3, and in perfect analogy to the Abstract DPLL framework of [58], learning can be modeled very simply and generally by the addition of the following two rules to the transition system B, in a system we will call L:

Learn: $\Lambda \vdash \Phi \implies \Lambda \vdash \Phi, C$

where

1. $\Phi \models C$.

Forget: $\Lambda \vdash \Phi, C \implies \Lambda \vdash \Phi$

where

1. $\Phi \models C$.

In this very general formulation, learning is simply the addition of an entailed clause to the clause set. While in principle one could learn any entailed clause, Learn is meant to be used to add only clauses that are more likely to cause further propagations and correspondingly reduce the number of needed decisions. The Forget rule's intended use is to control the growth of the clause set, by removing entailed clauses that cause little propagation.

Because of the potentially high overhead involved in generating lemmas and propagating them in practice, we focus in this work on only the kind of *conflict-driven* learning that has proven to be very effective in DPLL-based solvers. This technique can be described proof-theoretically as a linear resolution derivation whose initial central clause is a conflicting clause in the DPLL computation, and whose side clauses are clauses used in unit propagation steps. In terms of the abstract framework above, the linear resolution derivation proceeds as follows. The central clause $C \vee \overline{L}$ is resolved with a clause $L \vee D$ in the clause set only if L was added to the current context by a Propagate step with clause $L \vee D$. Since the net effect of each resolution step is to replace $\overline{L}$ in $C \vee \overline{L}$ by L's "causes" D, we can also see this resolution derivation as a *regression* process.

Both of the first two methods we present below lift this regression to the first-order case, although with different degrees of generality. The first method is, at least in theory, strictly subsumed by the second. We present it here because it does have some advantages in practice, and because it can be used to greatly simplify the presentation of the second method. The third and last method is less general than the other two. In our experiments we used it mostly as a sanity check against the other methods, because of its much lower overhead.

2.4.4 The Grounded Method

Let $\mathbf{D} = (\{\neg v\} \vdash \Phi_0 \Longrightarrow_{\mathrm{L}} \ldots \Longrightarrow_{\mathrm{L}} \Lambda \vdash \Phi)$ be a derivation in the transition system L where Λ contains at least one decision literal and Φ contains a clause C_0 conflicting in Λ. We describe a process for generating from $\mathbf{D}$ a *lemma*, a clause logically entailed by Φ, which can be *learned* in the derivation

by an application of Learn to the state $\Lambda \vdash \Phi$.

We describe the lemma-generation process itself as a transition system, this time applied to *annotated clauses*, pairs of the form $C \mid S$ where C is a clause and S is a finite mapping $\{L \mapsto M, \ldots\}$ from literals in C to context literals of $\mathbf{D}$. A transition invariant for $C \mid S$ will be that C consists of negated ground instances of context literals, while S specifies for each literal L of C the context literal M of which $\overline{L}$ is an instance, provided that M is a propagated literal. The mapping $L \mapsto M$ will be used to *regress* L, that is to resolve it with M in the clause used in $\mathbf{D}$ to add M to the context.

The initial annotated clause A_0 will be built from the conflicting clause of $\mathbf{D}$, and will be regressed by applying to it the GRegress rule, defined below, one or more times. In the definition of A_0 and of GRegress we use the following notational conventions. If σ is a substitution and C a clause or a literal, $C\underline{\sigma}$ denotes the expression obtained by replacing each universal variable or parameter of $C\sigma$ by a fresh Skolem constant (one per universal variable or parameter). If σ is a context unifier of a clause $L_1 \vee \cdots \vee L_n$ against some context, we denote by L_i^σ the context literal paired with L_i by σ.

Assume that C_0 is conflicting in Λ because of some context unifier σ_0. Then A_0 is defined as the annotated lemma

$$A_0 = C_0\underline{\sigma_0} \mid \{L\underline{\sigma_0} \mapsto L^{\sigma_0} \mid L \in C_0 \text{ and } L^{\sigma_0} \text{ is a propagated literal}\}$$

consisting of a fresh grounding of $C_0\sigma_0$ by Skolem constants and a mapping of each literal of $C_0\underline{\sigma_0}$ to its paired literal in Λ if that literal is a propagated literal. The regression rule is

GRegress: $D \vee M \mid S,\ M \mapsto L\sigma \implies_{\mathsf{gr}} D \vee C\sigma\underline{\mu} \mid S, T$

where

1. $L\sigma$ is the propagated literal of some context unifier σ and clause $L \vee C$,
2. μ is a most general unifier of M and $\overline{L\sigma}$,
3. $T = \{N\sigma\underline{\mu} \mapsto N^{\sigma} \mid N \in C\}$ and N^{σ} is a propagated literal.

Note that the mapping is used by GRegress to guide the regression so that no search is needed. The regression process simply repeatedly applies the rule GRegress an arbitrary number of times starting from A_0 and returns the last clause. While this clause is ground by construction, it can be generalized to a non-ground clause C by replacing each of its Skolem constants by a distinct universal variable. As proved in the next two results, this generalized clause is a logical consequence of the current clause set Φ in the derivation, and so can be learned with an application of the Learn rule.

To start, every regression of the initial annotated lemma with GRegress generates a logical consequence of the clause set.

Lemma 2.4.2 *If $A_0 \implies^{*}_{\mathsf{gr}} C \mid S$, then the following holds.*

1. *For every $M \mapsto N \in S$, M is a (ground) instance of $\overline{N}$,*
2. *The (ground) clause C is a consequence of Φ.*

Proof. Suppose a regression derivation $A_0 \implies^{*}_{\mathsf{gr}} C \mid S$ of length $l \geq 0$ as given. We directly prove the claim by induction on l.

$\mathbf{l = 0}$) 1. By construction of A_0, $M = L\underline{\sigma_0}$ for some literal L and N is the propagated literal L^{σ_0}. By definition of context unifier we have that $L\sigma_0 = \overline{N}\sigma_0$. So M is a ground instance of $\overline{N}$.

2. Immediately by construction, as $C_0\underline{\sigma_0}$ is a ground instance of the closing clause.

$\mathbf{l > 0}$) 1. For the mappings of S added by the application of the rule, the proof is analogous to the base case. For the others, the claims holds by induction.

2. Using the notation as introduced in the GRegress rule above, we prove first that GRegress preserves consequenceship.

With M being a (ground) instance of $\overline{L\sigma}$, as obtained by the induction hypothesis and 1, the most general unifier μ is in fact a matcher such that $\overline{L\sigma}\mu = M$. Thus, the clause $D \vee C\sigma\underline{\mu}$ is a (ground) instance of the resolution resolvent $D \vee C\sigma$ of the parent clause $D \vee M$, which is ground, and the parent clause $L\sigma \vee C\sigma$, where the most general unifier used is μ.

Now, the (ground) clause $D \vee M$ is a consequence of Φ by induction assumption and $L\sigma \vee C\sigma$ is an instance of a clause in Φ by construction. With the soundness of resolution it follows that $D \vee C\sigma\underline{\mu}$ is a consequence of Φ. □

Below we write $\underline{C}$ as a suggestive notation to denote a ground clause standing in a certain relation with another clause C.

Theorem 2.4.3 *If $A_0 \Longrightarrow^*_{\mathsf{gr}} \underline{C} \mid S$, the clause C obtained from $\underline{C}$ by replacing each constant of $\underline{C}$ not in Φ by a fresh universal variable is a consequence of Φ_0.*

Proof. By Lemma 2.4.2 and the Free Constants Theorem of first order logic. □

From a practical viewpoint, an important invariant is that one can continue regressing the initial clause until it contains only decision literals. This result, expressed in the next proposition, gives one great latitude in terms of how far to push the regression. In practice, to reduce the regression overhead and following a common practice in DPLL solvers, in our implementation we regress only propagated literals belonging to the last decision level of Λ.

Theorem 2.4.4 *If $A_0 \Longrightarrow^*_{\mathsf{gr}} A$ and A has the form $D \vee M \mid S, M \mapsto N$, then the* GRegress *rule applies to A.*

Proof. It is enough to show that M is an Assert literal in $\mathbf{D}$ and M is an instance of $\overline{N}$. The latter holds by Lemma 2.4.2, the former is easily provable again by induction on the length of regression derivations. □

Example 2.4.5 We are going to show a possible regression of the conflicting clause $\neg S(x) \vee \neg T(x)$ in the derivation of Example 2.4.1. This clause is conflicting because of the context unifier $\sigma_0 = \{x \mapsto u\}$, pairing the clause literals $\neg S(x)$ and $\neg T(x)$ respectively with the context literals $S(u)$ and $T(u)$. So we start with the initial annotated clause:

$$\begin{aligned} A_0 &= (\neg S(x) \vee \neg T(x))\underline{\sigma_0} \mid \{(\neg S(x))\underline{\sigma_0} \mapsto (\neg S(x))^{\sigma_0},\ (\neg T(x))\underline{\sigma_0} \mapsto (\neg T(x))^{\sigma_0}\} \\ &= \neg S(a) \vee \neg T(a) \mid \{\neg S(a) \mapsto S(u),\ \neg T(a) \mapsto T(u)\}\ . \end{aligned}$$

To ease the notation burden, we represent the regression in the more readable form of a linear resolution tree in Figure 2.2.

At each step the central clause is the regressed clause, the literal in bold font is the regressed literal, and the side clause is the clause $(L \vee C)\sigma$ identified

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg S(a) \vee \neg\mathbf{T(a)} \quad \neg R(u,y) \vee T(u)}{\neg\mathbf{S(a)} \vee \neg R(a,b)} \quad \neg P(t(x)) \vee \neg R(u,t(x)) \vee S(u)}{\neg\mathbf{R(a,b)} \vee \neg P(t(c)) \vee \neg R(a,t(c))} \quad \neg Q(u) \vee R(u,y)}{\neg P(t(c)) \vee \neg\mathbf{R(a,t(c))} \vee \neg Q(a)} \quad \neg Q(u) \vee R(u,y)}{\neg P(t(c)) \vee \neg Q(a)}$$

Figure 2.2: Grounded regression of $\neg S(x) \vee \neg T(x)$.

in the precondition of GRegress. The introduced fresh Skolem constants are a, b and c. Stopping the regression with the last resolvent in the derivation gives the lemma $\neg P(t(c)) \vee \neg Q(a)$. After abstracting away the Skolem constants we get the lemma $\neg P(t(x)) \vee \neg Q(y)$. The fact that the literals in this lemma are variable disjoint is not typical of the regression process. It is just a (nice) feature of this particular example. □

To judge the effectiveness of lemmas learned with this process in reducing the explored search space we also need to argue that they let the system later recognize more quickly, or possibly avoid altogether, the set of decisions responsible for the conflict in **D**. This is not immediately obvious within the ME calculus because of the role played by parameters in the definition of a conflicting clause. Recall that a clause is conflicting because of some context unifier σ iff it moves parameters only to parameters in the context literals associated with the clause. To show that lemmas can have the intended consequences, we start by observing that, by construction, every literal L_i in a lemma $C = L_1 \vee \cdots \vee L_m$ generated with the process above is a negated instance of some context literal K_i in Λ. Let us write C^{Λ} to denote the set $\{K_1, \ldots, K_m\}$.

Lemma 2.4.6 *If $A_0 \Longrightarrow^*_{\mathsf{gr}} \underline{E} \mid S$ and the clause E is obtained from $\underline{E}$ by replacing each constant of $\underline{E}$ not in Φ by a fresh universal variable, then E is conflicting*

in any context that contains E^Λ.

See Section 2.4.10 for a proof.

Theorem 2.4.7 *Any lemma* C *produced from* **D** *by the regression method in this section is conflicting in any context that contains* C^Λ.

Proof. Follows immediately from Lemma 2.4.6 □

Proposition 2.4.7 implies, as we wanted, that having had the lemma C in the clause set from the beginning could have led to the discovery of a conflict sooner, that is, with less propagation work and possibly also less decisions than in **D**. Moreover, the more regressed the lemma, the sooner the conflict would have been discovered.

Example 2.4.8 Looking back at the lemmas generated in Example 2.4.5, it is easy to see that the lemma $\neg R(x, y) \vee \neg P(t(z)) \vee \neg R(x, t(z))$ becomes conflicting in the derivation of Figure 2.1 as soon as $R(u, y)$ is added to the context. In contrast, the more regressed lemma $\neg P(t(x)) \vee \neg Q(y)$ becomes conflicting as soon as the decision $Q(u)$ is made. □

Since a lemma generated from **D** is typically conflicting once a *subset* of the decisions in Λ is taken, learning it in the state $\Lambda \vdash \Phi$, C_0 will help recognize more quickly these wrong decisions later, in extensions of **D** that undo parts of Λ by backjumping. In fact, if the lemma is regressed enough, one can often do even better and completely avoid the conflict later on, for example with a derivation strategy that prefers applications of Propagate to applications of Decide.

Example 2.4.9 Consider an extension of the derivation in Figure 2.1, where the context has been undone enough that now its last literal is $P(t(x))$. By applying Propagate to the lemma $\neg P(t(x)) \vee \neg Q(y)$ it is possible to add $\neg Q(y)$ to the context. This prevents the addition of $Q(u)$ as a decision literal, because $Q(u)$ is contradictory with $\neg Q(y)$, and avoids the conflict with clause (4). □

So far, what we have described mirrors what happens with propositional clause sets in DPLL based SAT solvers. What is remarkable about learning at the ME level, in addition to that it does have the same nice effects obtained in DPLL, is that its lemmas are not just caching compactly the reasons for a specific conflict. For being a *first-order* formula, a lemma in ME represents an *infinite* class of conflicts of the same form. For instance, the lemma $\neg P(t(x)) \vee \neg Q(y)$ in our running example will become conflicting once the context contains *any* instance of $P(t(x))$ and $Q(y)$, not just the original $P(t(x))$ and $Q(u)$. In fact, due to (p-preserving) unification, a lemma can be conflicting in a context even due to context literals that are more general than the literals of the lemma. Our lemma-generation process then does perform learning in a more proper sense of the word, as it can generalize over a single instance of a conflict, and so leads to additional pruning of the search space.

A slightly more careful look at the derivation in Figure 2.1 shows that the lemma $\neg P(t(x)) \vee \neg Q(y)$ is actually not as general as it could be. The reason is that a conflict arises also in contexts that contain, in addition to any instance of $Q(y)$, also any *generalization* of $P(t(x))$. So a better possible lemma is $\neg P(x) \vee \neg Q(y)$. We can produce such generalized lemmas by lifting the regression process similarly as in Explanation-Based Learning (see Section 2.1). We describe

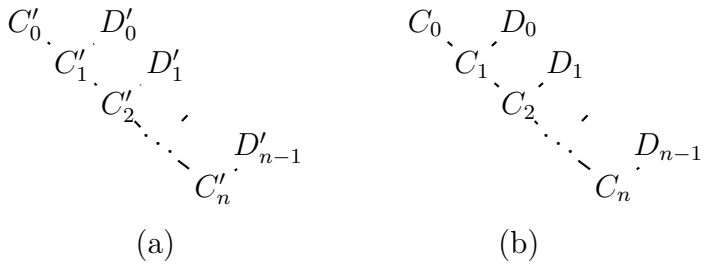


Figure 2.3: Grounded regression derivation and its lifting.

this lifted process next.

2.4.5 The Lifted Method

Consider again the derivation **D** from the previous subsection, whose last state $\Lambda \vdash \Phi$ contains a clause C_0 that is conflicting in Λ because of some context unifier σ_0. Starting with the annotated lemma

$$C'_0 \mid S'_0 \;=\; C_0\underline{\sigma_0} \mid \{L\underline{\sigma_0} \mapsto L^{\sigma_0} \mid L \in C_0 \text{ and } L^{\sigma_0} \text{ is a propagated literal}\},$$

one can build a regression of the form

$$C'_0 \mid S'_0 \Longrightarrow_{\mathrm{gr}} C'_1 \mid T'_1 \Longrightarrow_{\mathrm{gr}} \ldots \Longrightarrow_{\mathrm{gr}} C'_n \mid T'_n \ .$$

We have seen that this regression determines a linear resolution derivation, whose derivation tree is depicted in Figure 2.3(a), where C'_0 and D'_i are instances of clauses in Φ, and C'_{i+1} is a resolvent of C'_i and D'_i for all $i = 0, \ldots, n-1$.

Using basic results about resolution and unification, this derivation can be lifted to one of the form shown in Figure 2.3(b), where C_0 and each D_i are the clauses in Φ that C'_0 and D'_i are instances of, and each C_{i+1} is a resolvent of C'_i and D'_i and a generalization of C'_{i+1}.

Conceptually, the lifted derivation can be built simply by following the steps of the grounded derivation, but this time using the original clauses in Φ for

the initial central clause and the side clauses. In practice the lifted derivation can of course be built directly, without building the grounded derivation first. We do this by starting with the annotated lemma

$$C_0 \mid S_0 \;\; = \;\; C_0 \mid \{L \mapsto L^{\sigma_0} \mid L \in C_0 \text{ and } L^{\sigma_0} \text{ is a propagated literal}\},$$

and regressing that lemma with the following lifted version of GRegress:

Regress: $\quad D \vee M \mid S,\, M \mapsto L\sigma \implies_{\mathsf{r}} (D \vee C_{\mathrm{v}})\mu \mid S, T$

where

1. $L\sigma$ is the propagated literal of some context unifier σ and clause $L \vee C$,
2. $L_{\mathrm{v}} \vee C_{\mathrm{v}}$ is a fresh variant of $L \vee C$,
3. μ is a most general unifier of M and $\overline{L_{\mathrm{v}}}$,
4. $T = \{N_{\mathrm{v}}\mu \mapsto N^{\sigma} \mid N \in C\}$ and N^{σ} is a propagated literal.

Theorem 2.4.10 *For every grounded regression*

$$C'_0 \mid S'_0 \implies_{\mathsf{gr}} C'_1 \mid T'_1 \implies_{\mathsf{gr}} \ldots \implies_{\mathsf{gr}} C'_n \mid T'_n \;,$$

there is a lifted regression

$$C_0 \mid S_0 \implies_{\mathsf{r}} C_1 \mid T_1 \implies_{\mathsf{r}} \ldots \implies_{\mathsf{r}} C_n \mid T_n \;,$$

such that $C_i \gtrsim C'_i$ *and* $\Phi \models C_i$ *for all* $i = 0, \ldots, n$.

Proof. The grounded regression can be written as a linear resolution derivation from ground instances of clauses from Φ. Using standard lifting arguments (see [29]) this derivation can be lifted to a derivation using the clauses from Φ

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg S(x) \vee \mathbf{\neg T(x)} \quad \neg R(x_1, y_1) \vee T(x_1)}{\mathbf{\neg S(x)} \vee \neg R(x, y_1)} \quad \neg P(x_2) \vee \neg R(y_2, x_2) \vee S(y_2)}{\mathbf{\neg R(x, y_1)} \vee \neg P(x_2) \vee \neg R(x, x_2)} \quad \neg Q(x_3) \vee R(x_3, y_3)}{\neg P(x_2) \vee \mathbf{\neg R(x, x_2)} \vee \neg Q(x)} \quad \neg Q(x_4) \vee R(x_4, y_4)}{\neg P(x_2) \vee \neg Q(x)}$$

Figure 2.4: Lifted regression of $\neg S(x) \vee \neg T(x)$.

instead of their instances, which, in turn, can be written as the lifted regression as stated. This proves $C_i \gtrsim C'_i$.

Regarding $\Phi \models C_i$, observe that $C_0 \in \Phi$ and, according to the above, C_i is a resolution resolvent of C_{i-1} and some clause from Φ, for all $i = 1, \ldots, n$. Then $\Phi \models C_i$ follows by the soundness of the resolution inference rule. □

As in the grounded case then, we can use any regressed clause C as a lemma. In contrast, this time there are no constants to abstract, as the regression process resolves only input clauses of C. Again, the resulting clause is a logical consequence of Φ.

Example 2.4.11 Figure 2.4 shows the lifting of the grounded regression in Figure 2.2 for the conflicting clause $\neg S(x) \vee \neg T(x)$ in the derivation of Example 2.4.1. This time, we start with the initial annotated clause:

$$(\neg S(x) \vee \neg T(x)) \mid \{\neg S(x) \mapsto S(u),\ \neg T(x) \mapsto T(u)\}\ .$$

As before, we represent the regression as a linear resolution tree, where this time at each step the central clause is the regressed clause, the literal in bold font is the regressed literal, and the side clause corresponds to the clause $L_{\mathrm{v}} \vee C_{\mathrm{v}}$ in the precondition of Regress. The lemma learned in this case is $\neg P(x) \vee \neg Q(y)$.

□

2.4.6 The Propositional Method

Recall that each propagated literal L in a context is the result of a unification of a clause in the clause set with some previous context literals $K_1, \ldots, K_n$. This sort of dependency of L on $K_1, \ldots, K_n$ defines a dependency graph over context literals (whose roots are the context's decision literals) that can be used for conflict analysis. In fact, if C is a conflicting clause in a context Λ because of some context unifier σ, starting from the context literals used by σ and tracing the dependency graph backwards, one can precisely determine the set $\{L_1, \ldots, L_k\}$ of decision literals that are ultimately responsible for the conflict. Then one could simply remember this set of decisions and make sure that they are not repeated again. The way we do this is to abstract each L_i by a unique (modulo p-renaming) propositional variable P_i, add the propositional clause $\overline{P_1} \vee \cdots \vee \overline{P_k}$ to the clause set, and from then on add P_i to the context each time L_i is added. Then the clause will become conflicting every time $L_1, \ldots, L_k$ occur together again in a context. By applying Propagate to these propositional clauses, one can even avoid the conflict by not adding L_i again as a decision literal if $\overline{P_i}$ is present in the context.

The appeal of this method is that it is relatively cheap to generate and process this sort of lemmas. The downside is that these lemmas are less general than those computed with the previous methods, as they just cache *one* specific set of conflicting decisions.

2.4.7 Implementation

We implemented the three learning methods described in the previous section in Darwin [12]. We discuss this implementation and comment on a few

details.

2.4.7.1 Lemma-Generation

Since we merely take the decision literals relevant for a conflict as the pseudo-lemma in the propositional abstraction, the lemma-generation methods of interest are grounded and lifted regression. For both cases we employ memoization to avoid regressing the same context literal more than once.

In the case of the grounded regression, memoization is achieved implicitly. Recall that here each literal to regress corresponds to a (negated) ground instance of a propagated context literal, which in turn depends itself on previous context literals in the context. It is easy to see that these dependencies between context literals determine a directed graph, called a *conflict graph* in the SAT literature, whose roots are the context literals associated to the current clause to regress (the *regression clause*) and whose leaves are decision literals. The regression process explores a conflict graph by a breadth-first traversal. The literals in the regression clause are regressed in the order of addition to the context, where instances of more recently added context literals are regressed first. This makes sure that all instances of the same context literal are regressed in a row. Now, simply because the regression clause is represented as a set, each literal is automatically regressed only once.

The process is not as simple for the lifted method, as it in general involves unification operations, as opposed to just matching operations as in the grounded case. More precisely, the regression process is implemented by maintaining three data structures. Firstly, a set of all the literals in the current regression clause, the *regression set*, secondly, a set of *regressed* literals, literals that will not be

regressed further due to some stop criterion, and thirdly, a set of unification constraints. If a literal chosen from the regression set is not to be regressed, for example because it is paired with a decision literal, then it is simply moved to the set of regressed literals. Otherwise, it is replaced by the literals in the corresponding side clause, and the unifier of the corresponding resolution step is added to the set of unification constraints. The regression stops when the regression set is empty. At that point, the unification constraints are solved, and the resulting unifier is applied to the set of regressed literals, thus producing the lemma clause.

As the literals of a regression clause are not ground, memoization for variants of a literal is not implicit by using a simple set representation, as in the grounded method. Thus, in the lifted process, memoization is achieved by doing the regression depth-first, again in the order of addition of literals to the context. For each regressed literal its regressed literals and constraints are stored. Whenever the same literal is to be regressed again, this information is reused by creating a copy using fresh variables. As described in [12], this does not require the creation of new terms, but merely replacing integer offsets. As an optimization and similar to the grounded case, a context literal is regressed only once if it is an instance of a ground clause literal.

2.4.7.2 Regression Depth

In analogy to the common procedure in SAT solvers, only literals propagated after the most recent decision literal responsible for a conflict are regressed. The favored backjumping method backtracks up to but excludes the *second* most recent responsible decision literal, L. That is, the derivation is continued right

after the decision of L. In contrast, as allowed by the Backjump rule, Darwin only backtracks up to and including the most recent decision literal K responsible for the conflict. That is, the derivation is continued right before deciding on K, keeping all propagations and decisions made after the decision of L. Thus, propositional backjumping backtracks farther and is in a sense more eager. Experimental results have shown that this more eager form of backjumping is not beneficial in Darwin, as the right split does not in general prevent the jumped over decision literals and the subsequent propagations from being reasserted. That is, most of the times eager backjumping does not change the search space in a beneficial way, but instead introduces additional overhead. This has the effect that in Darwin a negated decision literal does not necessarily depend on the current decision level, and therefore a branch closure might not depend on the most recent decision literal, which makes the algorithm somewhat more complicated.

An important optimization for propositional solvers is to stop the regression at a unique implication point (UIP) [52]. In essence, a UIP is a node of the conflict graph such that each path from the most recent decision literal involved in the conflict to the literals in the regression set goes through the UIP. In particular, the (node of the) decision literal is a UIP. Thus, when backjumping to the split decision and undoing it, all literals of a lemma regressed up to a UIP except for the UIP literal are conflicting. Applying Propagate to it makes the application of the right-hand side of the split decision redundant, as it is implied by the obtained context.

It is unclear how to lift this idea to the first-order level, though, as in general there may be several, distinct instances of a propagated literal used in a

closure. The naive approach of treating all instances of the same context literal as a potential UIP did not turn out to be efficient in practice. Furthermore, while the UIP can be found automatically in the grounded regression, namely when the regression set contains only instances of exactly one context literal, this is not possible using the depth-first approach of the lifted regression. Here either the regression needs to be done breadth-first, but then memoization cannot be used, or the UIP must be computed before the actual regression is performed.

As a side note we point out that, for the same reason as above, unlike in the propositional case lemmas cannot be used in general to make the explicit addition of the negated decision literal unnecessary after backjumping. The following example makes this clear.

Example 2.4.12 A part of a derivation based on the clauses

$$(1) \quad P(a,b) \vee Q(a,x)$$

$$(2) \quad \neg P(a,b) \vee \neg Q(a,c)$$

$$(3) \quad P(x,b) \vee \neg Q(x,x)$$

might be Decide of $Q(a,x)$ based on (1), Propagate of $\neg P(a,b)$ based on (2), and Fail based on (3). Learning yields the grounded and lifted lemma $\neg Q(a,c) \vee \neg Q(a,a)$. While this lemma does become conflicting if Decide of $Q(a,x)$ is applied again, it does not prevent that application of Decide. The reason can be seen by looking at Split, the inference rule on which Decide is based. As x is a universal variable, the left split in essence stands for $\forall x Q(a,x)$, and the right split stands for $\neg\forall x Q(a,x)$ or $\exists x \neg Q(a,x)$. This is modeled in ME by introducing a fresh Skolem constant s and using $\neg Q(a,s)$ as the right split literal. The lemma $\neg Q(a,c) \vee \neg Q(a,a)$ gives us in a sense something stronger, as it limits the

number of instances which can be false to just two, instead of merely saying that "some" instance must be false. Unfortunately, as the lemma is not unit, it is not effective. □

It is unclear how to detect situations of this kind, where it is beneficial to abstract a set of literals by one literal through the introduction of Skolem constants, and we have not pursued this any further. This problem tends to occur more frequently with lifted lemmas, as the more general learning process often introduces several partially instantiated literals instead of one common instance, making Propagate sometimes less efficient with lifted lemmas.

2.4.7.3 Simplification

A lemma computed in the grounded and lifted regression is simplified before usage. Note that a context literal asserted in the root decision level, that is before any decision literal is added to the context, is never regressed according to the description above. But, as it is implied by the clause set, its grounded regression does in essence correspond to a unit resolution step. As a consequence, root assert context literals do not need to be and are not added to the lemma. For the lifted regression the case is more complicated, as it is only directly applicable if the context literal is an instance of a unit clause. As in addition the constraint has to be computed as usual, which in some cases leads to a significant overhead due to a large number of constraints, root asserts are not treated specially here.

Simplifying the lemma is particularly important in the lifted case because it is not unusual for the lemma regression process to produce very long lemmas, with several instances or variants of the same literal. As condensing is too expen-

sive, we employ a simpler method which produces good results in practice with a number of unification tasks linear in the number of literals. If all instances of the same context literal in a lemma have a common instance, they are replaced by their most general common instance, and the corresponding unifier is applied to the remainder of the lemma. Then, if still several variants of a literal occur, they are condensed into one literal, and the renaming is again applied to the remainder of the lemma. Finally, duplicates of literals are removed as clauses are treated as sets of literals.

Unfortunately, this method sometimes simplifies the lemma in an unwanted way, making the lemma in effect useless. For example, if in a context the literals $\neg P(a)$, $\neg Q(a)$, $\neg Q(b)$, and $\neg R(b)$ lead to a conflict, then the learned grounded lemma might be $P(a) \vee Q(a) \vee Q(b) \vee R(b)$, and the more general lifted lemma might be $P(x) \vee Q(x) \vee Q(y) \vee R(y)$. Now, its simplification, $P(x) \vee Q(x) \vee R(x)$, cannot be used to prevent the recreation of the conflicting context, and in fact not even to close on it.

2.4.7.4 Application

In principle, during a derivation of the proof procedure lemmas can be used like any other clause as far as the rules Decide, Propagate, and Fail are concerned. As a lemma's purpose is to prune the search space, applying Decide to lemmas does not seem like a sensible choice, as confirmed by our experimental results. Using lemmas only for Fail applications and for selected applications of Propagate turned out to be the most efficient usage.

Furthermore, to reduce the context unifier computation overhead, potential propagations for a new lemma are computed only based on future extensions

of the context. Therefore, when after an application of Backjump a lemma is learned, it does not propagate the negated decision literal $\overline{L}^{\text{sko}}$. As an optimization this happens only if the lemma is unit, which might in fact propagate a strictly p-preservingly more general literal than $\overline{L}^{\text{sko}}$. In conjunction with the above described shortening of grounded lemmas with root context literals, this case occurs more often in the grounded than the lifted case, making a grounded lemma sometimes more effective than the corresponding lifted one.

In general, applications of Propagate are restricted in Darwin to those with *universal* propagated literals, i.e., parameter-free literals. Adding non-universal propagated literals to the context is not only unnecessary for completeness, but also counterproductive for efficiency, as it substantially increases the number of context unifiers usable by Decide or Propagate. On the other hand, adding literals propagated by a lemma is useful to avoid conflicts, as we discussed earlier. In the current implementation, we strike a balance between these two conflicting needs by adding to the context a non-universal propagated literal only if the propagating clause has been learned as a lemma at least n times in the derivation—a crude but easily computed estimate of the lemma's usefulness in avoiding future conflicts. Experimentally, a value of $n = 3$ seems to give the best results.

2.4.7.5 Forget

At the moment we have implemented only a relatively crude scheme for forgetting lemmas, again inspired by similar schemes in the SAT literature [66]. In this scheme, there exists an upper limit u and a lower limit l on how many lemmas are stored at any time. If a new lemma is learned after u has been reached, the *worst* lemmas are removed until there are only l lemmas left. The

new lemma is then added to this smaller lemma set.

The value of a lemma is determined by a score, which is initially set to the worst score among the existing lemmas. Whenever the lemma is responsible for an application of Fail, i.e., the lemma is involved in the regression of a conflict, its score is incremented by 1. When the worst lemmas are removed, all scores are divided by 2. As an alternative, we also tried to decay the score periodically after a certain number of Backjump applications. This score is not currently used in the heuristics for choosing which lemma to propagate on, mostly because it is not trivial to integrate properly into the system's architecture. Unfortunately, these schemes did not lead to any improvement over not applying Forget at all.

2.4.8 Experimental Evaluation

We evaluated the effectiveness of lemma-learning over two problem sets, firstly directly over the TPTP problem library, and secondly on transformations of TPTP problems obtained in an application of finite model finding.

2.4.8.1 TPTP

We first evaluated the effectiveness of lemma-learning in Darwin over the TPTP problem library version 3.1.1 [68]. Since Darwin can handle only clause logic, and has no dedicated inference rules for equality, we considered only clausal problems without equality. Furthermore, as Darwin never applies the Decide rule in Horn problems [12], and thus also never backtracks, we further restricted the selection to non-Horn problems only. All tests were run on Xeon 2.4Ghz machines with 1GB of RAM. The imposed limit on the prover were 300s CPU time and 512MB of RAM.

Method	Solved Probls	Avg Time	Total Time	Speed up	Failure Steps	Propag. Steps	Decide Steps
no lemmas	896	2.7	2397.0	1.00	24991	597286	45074
propositional	895	2.8	2507.6	0.96	20056	570962	37102
grounded	895	2.4	2135.6	1.12	9476	391189	18935
lifted	898	2.4	2173.4	1.10	9796	399525	19367
no lemmas	244	3.0	713.9	1.00	24481	480046	40766
propositional	243	3.4	821.1	0.87	19546	453577	32794
grounded	243	1.8	445.1	1.60	8966	273849	14627
lifted	246	2.0	493.7	1.45	9286	282600	15059
no lemmas	108	5.2	555.7	1.00	23553	435219	38079
propositional	107	4.5	478.8	1.16	18703	392616	30209
grounded	108	2.2	228.5	2.43	8231	228437	12279
lifted	111	2.6	274.4	2.02	8535	238103	12688
no lemmas	66	5.0	323.9	1.00	21555	371145	34288
propositional	66	4.5	289.7	1.12	17044	333648	27026
grounded	67	1.7	111.4	2.91	6973	183292	9879
lifted	70	2.3	151.4	2.14	7275	193097	10294

Table 2.1: Effect of lemma-learning on the TPTP

The four sections of Table 2.1 correspond to problems that are solved within 300s and Darwin takes respectively at least 0, 3, 20, and 100 applications of Backjump without lemmas. The four rows of each section summarize the results for various configurations of Darwin, namely, not using lemmas and using lemmas with the propositional, grounded, and lifted regression methods. **Solved Problems** gives the number of problems solved by a configuration, while the remaining values are for the subsets of 894, 241, 106, 65 problems solved by *all* configurations. **Avg Time** (**Total Time**) gives the average (total) time needed for the 894 problems solved by all configurations, **Speed up** shows the run time speed up factor of each configuration versus the one with no lemmas. **Failure**, **Propagate**, and **Decide** give the number of rule applications, with **Failure** including both Backjump and Fail applications.

Focusing on the first section, a significant observation is that all configurations solve almost exactly the same number of problems, which is somewhat disappointing. The situation is similar even with an increased timeout of one hour per problem. A sampling of the derivation traces of the unsolved problems reveals, however, that they contain only a handful of Backjump steps, suggesting that the system spends most of the time in propagation steps and supporting operations such as the computation of context unifiers. The second observation is that for the solved problems the search space, measured in the number of Decide applications, is significantly pruned by all learning methods (with 18% to 58% less decisions), although this improvement is only marginally reflected in the run times. This too seems to be due to the fact that most derivations involve only a few applications of Backjump. Indeed, 652 of the 898 solved problems require at most 2 backjumps. This implies that only a few lemmas can be learned, and thus their effect is limited and the run time of most problems remains unchanged. Based on these tests, it is not clear if this an intrinsic property of the calculus, an artifact of the specific proof procedure implemented by Darwin, or a feature of the TPTP library.

For a more meaningful comparison, the other three sections of Table 2.1 show the same statistics, but restricted to the problems solved with more applications of Backjump. There, the effect of pruning the search space is more pronounced and does translate into reduced run times. In particular, the speed up of each lemma configuration with respect to the no lemmas one steadily increases with the difficulty of the problems, reaching a factor of almost 3 for the most difficult problems in the grounded case. Moreover, the lifted lemmas con-

figuration always solves a few more problems than the no lemmas one.

Because of the way Darwin's proof procedure is designed, in addition to pruning search space, lemmas may also cause changes to the order in which the search space is explored. Since experimental results for unsatisfiable problems are usually more stable with respect to different space exploration orders, it is instructive to separate the data in Table 2.1 in unsatisfiable and satisfiable problems.

Method	**Solved Probls**	**Avg Time**	**Total Time**	**Speed up**	**Failure Steps**	**Propag. Steps**	**Decide Steps**
no lemmas	563	3.3	1827.4	1.00	22741	495924	35831
propositional	562	3.5	1975.5	0.92	18478	476066	28959
grounded	561	3.0	1705.2	1.07	8336	294819	11620
lifted	562	3.1	1731.8	1.06	8610	300273	12004
no lemmas	193	1.9	364.4	1.00	22283	419920	35121
propositional	192	2.7	508.9	0.71	18020	399969	28249
grounded	191	1.2	234.7	1.55	7878	218739	10910
lifted	192	1.4	271.4	1.34	8152	224587	11294
no lemmas	89	2.9	255.6	1.00	21589	388200	34109
propositional	89	2.4	216.2	1.18	17390	352350	27328
grounded	90	0.8	68.2	3.74	7352	188032	10216
lifted	90	1.2	103.1	2.48	7615	194755	10581
no lemmas	61	3.7	226.4	1.00	20157	351521	32011
propositional	61	3.1	190.8	1.19	16169	317696	25570
grounded	61	0.9	54.0	4.19	6484	163481	9058
lifted	62	1.4	88.2	2.57	6748	170424	9429

Table 2.2: Effect of lemma-learning on unsatisfiable TPTP problems

The results for unsatisfiable problems in Table 2.2 show the same pattern as the aggregate results. The speed up factors for grounded lemmas in particular are respectively 1.07, 1.55, 3.74, and 4.19, which actually compares more

Method	Solved Probls	Avg Time	Total Time	Speed up	Failure Steps	Propag. Steps	Decide Steps
no lemmas	333	1.7	569.6	1.00	2250	101362	9243
propositional	333	1.6	532.1	1.07	1578	94896	8143
grounded	334	1.3	430.4	1.32	1140	96370	7315
lifted	336	1.3	441.6	1.29	1186	99252	7363
no lemmas	51	7.0	349.5	1.00	2198	60126	5645
propositional	51	6.2	312.2	1.20	1526	53608	4545
grounded	52	4.2	210.4	1.66	1088	55110	3717
lifted	54	4.4	222.3	1.57	1134	58013	3765
no lemmas	18	17.7	300.1	1.00	1964	47019	3970
propositional	18	15.4	262.6	1.14	1313	40266	2881
grounded	19	9.4	160.3	1.87	879	40405	2063
lifted	21	10.1	171.3	1.75	920	43348	2107
no lemmas	5	24.4	97.5	1.00	1398	19624	2277
propositional	5	24.7	98.9	0.99	875	15952	1456
grounded	6	14.4	57.4	1.70	489	19811	821
lifted	8	15.8	63.2	1.54	527	22673	865

Table 2.3: Effect of lemma-learning on satisfiable TPTP problems

favorably overall to the corresponding speed up factors in Table 2.1, respectively 1.12, 1.60, 2.43, and 2.91. The speed up factors for satisfiable problems are a lot smaller, always well below 2. In general, most solved satisfiable problems require only very few applications of Decide and Backjump.

Plotting the individual run times of the no lemmas configuration against the lemma configurations, and the grounded against the lifted lemmas configuration for all solved problems with at least 3 backjumps, as seen on a log-log scale in Figure 2.5, clearly shows the positive effect of learning.

For readability, the cutoff is set at 100s instead of 300s, because in all cases less than a handful of problems are solved in the 100-300s range. For nearly all of the problems, the performance of the grounded lemmas configuration is better, often by a large margin, than the one with no lemmas. A similar situation occurs

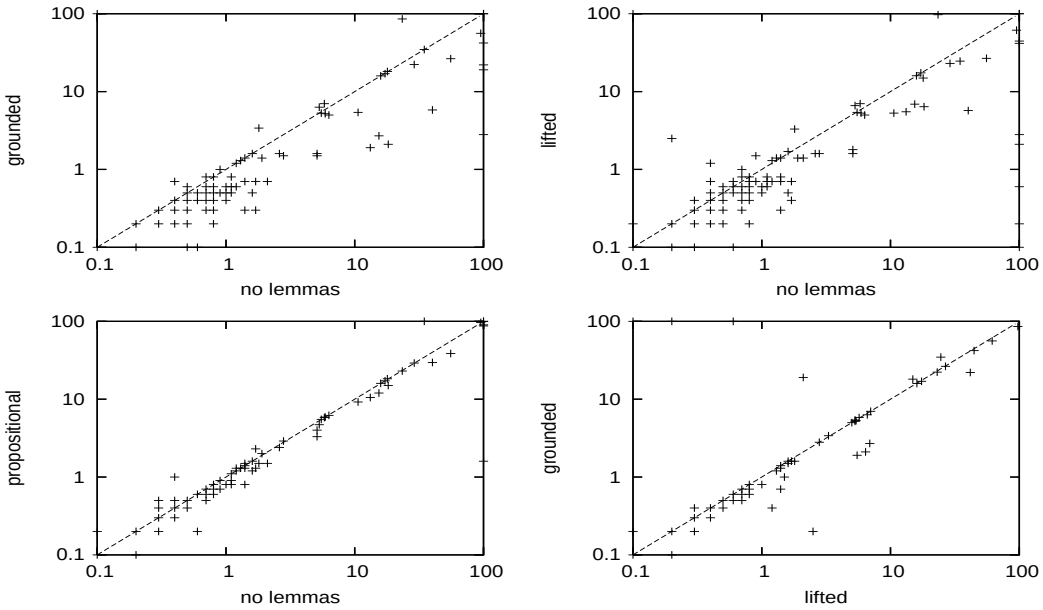


Figure 2.5: Comparison of lemma-learning methods

with lifted lemmas, although there are more problems for which the no lemmas configuration is faster. In contrast, the plot for the propositional configuration looks considerably different, with few outliers for either configuration, and basically all points closely clustered around the diagonal. Finally, the comparison of the grounded and the lifted learning method shows that the gained generality of the latter almost never pays off in terms of run time, except that it allows the system to solve three additional problems.

Overall, the results above indicate that the propositional method is not nearly as effective at pruning the search space or decreasing the run time as the other two learning methods, confirming our hypothesis that generalizing pays off. They also show that lifted lemmas generate more Decide applications and have higher overhead than grounded lemmas. The larger number of decision steps of the lifted method versus the grounded one seems paradoxical at first sight, but can be explained by observing that lifted lemmas also cause the addition

of more general propagated literals to a context, leading to a higher number of (possibly useless) context unifiers. Furthermore, due to the increased generality of lifted lemmas and the way they are simplified when they are too long, sometimes Propagate applies to a grounded lemma but not to the corresponding lifted lemma, making the latter *less* effective at avoiding conflicts (see Section 2.4.7).

The higher overhead of the lifted method can be attributed to two main reasons. The first is the increased number of context unifiers to be considered for rule applications. The second is the intrinsically higher cost of the lifted method versus the grounded one, because of its use of unification—as opposed to matching—operations during regression, and its considerable post-processing work in removing multiple variants of the same literals from a lemma—something that occurs quite often.

2.4.8.2 Finite Model Finding

It is somewhat surprising that only a minority of the TPTP problems used in the first experiment cause a considerable amount of search and backtracking, and that, on the other hand, many decidable fragments of first-order logic are NP-hard. We considered a second problem set, stemming from the application of Darwin to finite model finding described in Section 2.5, which exhibits more search. In this scenario, Darwin is run over transformations of a problem for increasing domain sizes, until a model is found or unsatisfiability can be concluded. In the configurations with learning, Darwin learns and uses lemmas during each iteration, and carries the ones that are independent of the domain size over to the next iterations. Thus it is reasonable to consider together all iterations of a run when measuring the effect of learning.

Method	Solved Probls	Average Time	Total Time	Speed up	Failure Steps	Propagate Steps	Decide Steps
no lemmas	657	5.6	3601.3	1.00	404237	16122392	628731
propositional	658	4.4	2827.1	1.27	198023	7859965	351236
grounded	669	3.3	2106.3	1.71	74559	4014058	99865
lifted	657	4.7	3043.9	1.18	41579	1175468	68235
no lemmas	162	17.8	2708.6	1.00	398865	15911006	614572
propositional	163	13.0	1971.1	1.37	193302	7659591	338074
grounded	174	7.9	1203.1	2.25	70525	3833986	87834
lifted	162	14.0	2126.2	1.27	38157	1023589	57070
no lemmas	52	36.2	1702.9	1.00	357663	14580056	555015
propositional	53	20.5	961.9	1.77	161851	6540084	291492
grounded	64	10.5	495.3	3.44	53486	3100339	64845
lifted	57	11.5	538.7	3.16	26154	678319	39873

Table 2.4: Effect of lemma-learning on finite model finding

Table 2.4 shows the results for the 815 satisfiable clause problems of the TPTP library. The sections of the table are structured as before, except that without lemmas Darwin applies Backjump respectively at least 0, 100, and 1000 times. To give an idea how our results compare to other approaches, we remark that Mace 4 and Paradox 1.3, currently the fastest finite model finders available, respectively solve 553 and 714 of those problems, making Darwin second only to Paradox.

In general, solving a problem in Darwin with the process above requires significantly more applications of Backjump than for the set of experiments presented earlier. As a consequence, the grounded lemmas configuration performs significantly better than the no lemmas configuration, solving the same problems in about half the time, and also solving 12 new problems. The lifted configuration, on the other hand, performs only moderately better. Although the search space is significantly reduced, the overhead of lemma simplification almost outweighs

the positive effects of pruning. Restricting the analysis to harder problems shows that the speed up factor of grounded lemmas increases gradually to about 3.5. This confirms that lemmas do have a significant positive effect if the focus in solving a problem lies on search instead of constraint propagation.

2.4.9 Conclusion

We have introduced three methods for implementing conflict-based learning in proof procedures for the Model Evolution calculus. The methods have various degrees of generality, implementation difficulty, and practical effectiveness. The experimental results indicate that for problems that are not trivially solvable by the Darwin implementation, all methods have a dramatic pruning effect on the search space. The grounded method, however, is the most effective at reducing the run time as well.

The main focus of future work is the development of good heuristics for simplifying and forgetting lemmas, and to tune the decision heuristics of Darwin based on the data obtained from regressing conflict sets.

2.4.10 Proofs

Lemma 2.4.6 *If* $A_0 \Longrightarrow^*_{\mathsf{gr}} \underline{E} \mid S$ *and the clause* E *is obtained from* $\underline{E}$ *by replacing each constant of* $\underline{E}$ *not in* Φ *by a fresh universal variable, then* E *is conflicting in any context that contains* E^Λ.

Proof. Suppose a regression derivation $A_0 \Longrightarrow^*_{\mathsf{gr}} \underline{E} \mid S$ of length $l \geq 0$ as given. We directly prove the claim by induction on l.

$\mathbf{l = 0}$) By construction of A_0, $\underline{E}$ is the clause $C_0\underline{\sigma_0}$, a ground instance of some clause C_0 which is conflicting in Λ because of the context unifier σ_0. If $C_0 =$

$L_1 \vee \cdots \vee L_n$, for some $n \geq 0$, the context unifier σ_0 of C_0 exists against any context containing $\{L_1^{\sigma_0}, \ldots, L_n^{\sigma_0}\}$, which is, by definition, the set E^{Λ}.

The constants in $C_0\underline{\sigma_0}$ and not in Φ are just the fresh constants introduced by $\underline{\sigma_0}$. Thus, $E = C_0\sigma_0$. Because any standard unification algorithm computes idempotent unifiers, it is safe to assume that the context unifier σ_0 is idempotent. It follows $C_0\sigma_0 = C_0\sigma_0\sigma_0$, and thus σ_0 is a context unifier of E against any context containing $\{L_1^{\sigma_0}, \ldots, L_n^{\sigma_0}\}$.

l > 0) We use notation similar as introduced in the GRegress rule above. Thus let

$$\underline{D} \vee \underline{M} \mid S',\ \underline{M} \mapsto L\sigma \Longrightarrow_{\mathsf{gr}} \underline{D} \vee C\sigma\underline{\mu} \mid S', T$$

be the last GRegress application (i.e., $\underline{E} = \underline{D} \vee C\sigma\underline{\mu}$). We assume by induction the result to hold for $\underline{D} \vee \underline{M}$, i.e., that $D \vee M$ is conflicting in any context that contains $(D \vee M)^{\Lambda}$, where $D \vee M$ is obtained from $\underline{D} \vee \underline{M}$ by replacing each constant of $\underline{D} \vee \underline{M}$ not in Φ by a fresh universal variable. We will directly show that under these assumptions $E = D \vee C\sigma\mu$ is conflicting in any context that contains E^{Λ}.

Let $(D \vee M)^{\Lambda} = \{K_1^{\delta}, \ldots, K_m^{\delta}, M^{\delta}\}$, where $D = K_1 \vee \cdots \vee K_m$, for some $m \geq 0$, and δ the context unifier such that $D \vee M$ is conflicting with Λ because of δ. As $D \vee M$ is conflicting with Λ because of δ, δ moves parameters to parameters only. More precisely, by construction $D \vee M$ is parameter-free, and the only parameters moved by δ can thus be assumed to be those in $\{K_1^{\delta}, \ldots, K_m^{\delta}, M^{\delta}\}$, and it holds $(\mathcal{P}ar(\{K_1^{\delta}, \ldots, K_m^{\delta}, M^{\delta}\}))\delta \subseteq V$. We will need this result further below.

By the above notation, the clause $\underline{E}$ is of the form $\underline{D} \vee C\sigma\underline{\mu}$, for some

context unifier σ of a clause $L \vee C$ against some context literals of Λ, where $L\sigma$ is a propagated literal and μ is a most general unifier of $\underline{M}$ and $\overline{L\sigma}$ (in fact, μ is a matcher of $\overline{L\sigma}$ to $\underline{M}$, as $\underline{M}$ is ground). For further use below, we write the clause $L \vee C$ as $L \vee L_1 \vee \cdots \vee L_n$, for some $n \geq 0$. The literals paired with $L \vee C$ by σ then are denoted by $\{L^\sigma, L_1^\sigma, \ldots, L_n^\sigma\}$.

The substitution μ can be written as $\mu = \mu' \circ \gamma$, where μ' is a most general unifier of M and $\overline{L\sigma}$ (in fact, a matcher of $\overline{L\sigma}$ to M) and γ is a substitution that moves all the parameters and variables in M to the fresh constants such that $M\gamma = \underline{M}$. Assume that γ has furthermore been extended to move all the remaining parameters and variables in $C\sigma\mu$ to fresh constants. It follows $C\sigma\underline{\mu} = C\sigma\mu'\gamma$, and, since $\underline{E} = \underline{D} \vee C\sigma\underline{\mu}$ we get $E = D \vee C\sigma\mu'$.

With $(D \vee M)^\Lambda = \{K_1^\delta, \ldots, K_m^\delta, M^\delta\}$ from above and $\{L^\sigma, L_1^\sigma, \ldots, L_n^\sigma\}$ being the literals paired with $L \vee C$ by σ we get $(D \vee C\sigma\mu')^\Lambda = \{K_1^\delta, \ldots, K_m^\delta, L_1^\sigma, \ldots, L_n^\sigma\}$ $(= E^\Lambda)$.

It remains to prove that $D \vee C\sigma\mu'$ is conflicting in any context that contains $(D \vee C\sigma\mu')^\Lambda$. For this, we show that the substitution $\sigma\mu'\delta$ is a context unifier of $D \vee C\sigma\mu'$ against Λ with paired literals $\{K_1^\delta, \ldots, K_m^\delta, L_1^\sigma, \ldots, L_n^\sigma\}$. It suffices to take a literal K_i from D and a literal L_j from C arbitrary and to show that

(1) (a) $K_i\sigma\mu'\delta = \overline{K_i^\delta}\sigma\mu'\delta$ and (b) $(L_j\sigma\mu')\sigma\mu'\delta = \overline{L_j^\sigma}\sigma\mu'\delta$, and

(2) (a) $(\mathcal{P}ar(K_i^\delta))\sigma\mu'\delta \subseteq V$ and (b) $(\mathcal{P}ar(L_j^\sigma))\sigma\mu'\delta \subseteq V$.

First we show that neither σ nor μ' act on K_i, i.e., that $K_i\sigma = K_i$ and $K_i\mu' = K_i$ hold: the literal K_i is a literal from D, which is obtained from the (ground) clause $\underline{D}$ by replacing each constant in $\underline{D}$ not in Φ by a fresh universal variable. Thus,

K_i is parameter-free and all its variables are disjoint from the variables in L and $L\sigma$. Thus, the context unifier σ (of the clause $L \vee C$) need not act on the clause $D \vee M$, and hence in particular not on K_i, which implies $K_i\sigma = K_i$. Similarly, recall that μ' is a matcher of $\overline{L\sigma}$ to M. Clearly we may assume μ' to move the variables and parameters of $\overline{L\sigma}$ only. With the freshness of the variables in $D \vee M$, thus, $K_i\mu' = K_i$.

Next we show, similarly, that neither σ nor μ' acts on $\overline{K_i^\delta}$, i.e., that $\overline{K_i^\delta}\sigma = \overline{K_i^\delta}$ and $\overline{K_i^\delta}\mu' = \overline{K_i^\delta}$ hold. For this, we need the fact that context literals used in context unifiers are fresh. Thus, neither σ nor μ' acts on $\overline{K_i^\delta}$, which is a context literal of the context unifier δ, and the stated equalities follow.

From $\overline{K_i^\delta}\sigma = \overline{K_i^\delta}$ and $\overline{K_i^\delta}\mu' = \overline{K_i^\delta}$ and together with$K_i\sigma = K_i$, $K_i\mu' = K_i$ the above equation (1-a) is equivalent to $K_i\delta = \overline{K_i^\delta}\delta$, which holds trivially by notation.

By $\overline{K_i^\delta}\sigma = \overline{K_i^\delta}$ and $\overline{K_i^\delta}\mu' = \overline{K_i^\delta}$, condition (2-a) is equivalent to $(\mathcal{P}ar(K_i^\delta))\delta \subseteq V$. Recall that $D \vee M$ is conflicting with Λ because of δ. By definition, δ thus does not have a remainder, and $(\mathcal{P}ar(K_i^\delta))\delta \subseteq V$ follows in particular.

It remains to prove conditions (1-b) and (2-b).

Regarding (1-b), it is safe to assume that σ is idempotent. Recall that μ' is a matcher form $L\sigma$ to M, and all variables of M are fresh, as argued further above. It is not difficult to see that $\sigma\mu'$ must be idempotent, too. Thus (1-b) is equivalent to $L_j\sigma\mu'\delta = \overline{L_j^\sigma}\sigma\mu'\delta$. This, however, follows trivially from $L_j\sigma = \overline{L_j^\sigma}\sigma$, which holds by notation.

Regarding (2-b), notice first $(\mathcal{P}ar(L_j^\sigma))\sigma \subseteq V$. This holds because $L\sigma$ is a propagated literal and, by definition of Propagate, none of the literals in $C\sigma$ is a

remainder literal. In particular, thus $(\mathcal{P}ar(L_i^\sigma))\sigma \subseteq V$. Next we will extend this inequality and obtain $(\mathcal{P}ar(L_j^\sigma))\sigma\mu'\delta \subseteq V$, which will complete the proof.

Recall from the regression step we are considering that $\underline{M}$ is paired with $L\sigma$. Recall further that $D \vee M$ is conflicting with Λ because of δ. The literal paired with M in the context unifier δ is thus a fresh p-variant of $L\sigma$, say, $L\sigma\rho$ for some appropriate p-renaming ρ. That is, $M\delta = \overline{L\sigma}\rho\delta$. We also know that $\overline{L\sigma}$ can be instantiated to M by μ'. That is, $\overline{L\sigma}\mu' = M$. Applying δ yields $\overline{L\sigma}\mu'\delta = M\delta = \overline{L\sigma}\rho\delta$. Now, as $D \vee M$ is conflicting because of δ, δ has no remainder literals. In particular, thus, $(\mathcal{P}ar(L\sigma\rho))\delta \subseteq V$. From this it is not too difficult to see that $\rho\delta$ maps all parameters of $L\sigma$ to parameters. Since $\overline{L\sigma}\mu'\delta = M\delta = \overline{L\sigma}\rho\delta$, $\mu'\delta$ maps all parameters of $\overline{L\sigma}$ to parameters. Recall that μ' can be restricted to move parameters and variables of $L\sigma$ only. All other parameters that $\mu'\delta$ moves are moved by δ to parameters (because δ has no remainder literals). Together thus we obtain from $(\mathcal{P}ar(L_j^\sigma))\sigma \subseteq V$ the desired result $(\mathcal{P}ar(L_j^\sigma))\sigma\mu'\delta \subseteq V$. □

2.5 Finite Model Finding

2.5.1 Introduction

Recent years have seen considerable interest in procedures for computing finite models of first-order logic specifications [72]. One of the major paradigms, MACE-style model building, is based on reducing model search to a sequence of propositional satisfiability problems and applying (efficient) SAT solvers to them. A problem with this method is that it does not scale well as the propositional formulas to be considered may become very large.

We propose a new approach in the MACE tradition that exploits new

advances in instantiation-based first-order theorem proving. Instead of using propositional logic as a target logic, we use function-free first-order clause logic, for which ME is a decision procedure. The main appeal of this method is that first-order clause sets grow more slowly than their propositional counterparts, thus allowing for more space efficient reasoning.

Apart from this difference, the general idea in our approach follows the MACE tradition. To find a model with n elements for a given a clause set (possibly with equality), the clause set is first converted into function-free clause logic by means of the following transformations:

1. Each clause is flattened (nested function symbols are removed).

2. Each n-ary function symbol is replaced by an $n + 1$-ary predicate symbol and equality is eliminated.

3. Clauses are added to the clause set that impose totality constraints on the new predicate symbols, but over a domain of cardinality n.

The details of our transformation differ in various aspects from the propositional approach, as explained below. In particular, we add no functionality constraints over the new predicate symbols. The main difference, however, is that due to the different target logic, we use Darwin instead of a SAT solver to look for models. While we do take advantage of some of the distinguishing features of Darwin and the ME calculus, especially in the way models are constructed, our method depends neither on Darwin nor ME. Without much additional effort, we could use any other decision procedure for function-free clause logic, such as Inst-Gen [38, 45] or DCTP [50].

We illustrate our method in some detail, presenting the main translation and its implementation within Darwin, and discuss our experimental results in comparison with Paradox [30] and Mace4 [54], competitive finite model finders. The results indicate that our method is rather promising as it can solve 1074 of the 1251 satisfiable problems in the TPTP library [68]. These problems are neither a subset nor a super-set of the sets of 1083 and 802 problems respectively solved by Paradox and Mace4.

2.5.2 Related Work

Methods for model computation can be classified as those that directly search for a finite model, like the extended PUHR tableau method [26], the methods in [21, 34] and the methods in the SEM-family [65, 74, 54], and those that are based on transformations into certain fragments of logic and which rely on readily available systems for these fragments (see [15] for a recent approach).

One of the most popular solvers following the first approach is perhaps Mace4. It reduces the finite model finding problem to the construction of multiplication tables for all function and predicate symbols occurring in the input ground clauses via constraint solving. Initially, each cell of a table ranges over all elements of the current domain, i.e., 1 to n for a given domain size n. Constraint propagation based on the ground clauses, and in particular the equality constraints in the clauses, is used to strengthen the range of each cell. Together with backtracking search this makes it possible to phrase a finite model finding problem as a constraint satisfaction problem, and to solve it by using constraint solving techniques.

The second approach includes the family of MACE-style model builders [53],

named after Mace2, not Mace4. These systems search for finite models, essentially by constructing a sequence of translations corresponding to interpretations with domain sizes $1, 2, \ldots$, in increasing order, until a model has been found. MACE-style model builders usually use propositional logic as their target logic. The model builder from this class with the best performance today is probably Paradox [30]. As mentioned before, our method follows this approach.

2.5.3 Preliminaries

We assume as given a signature Σ. We denote the function symbols of a signature Σ by Σ_{F}, and the predicate symbols by Σ_{P}. As we are working with equality, we assume Σ_{P} contains a distinguished binary predicate symbol $\approx$, used in infix form, with $\not\approx$ denoting its negation. For a given atom $P(t_1, \ldots, t_n)$ (possibly an equation) the terms $t_1, \ldots, t_n$ are also called the *top-level* terms (of $P(t_1, \ldots, t_n)$). An E-interpretation interprets the equality relation as the *identity relation*, i.e., for every E-interpretation $\mathfrak{I}$, $\approx^{\mathfrak{I}} = \{(d, d) \mid d \in |\mathfrak{I}|\}$.

We are primarily interested in computing *finite* models, that is models with a finite domain. A calculus is *finite model complete* if it is guaranteed to detect the existence of a finite model for any formula in finite time if such a model exists. In the remainder of the section, we assume that M is a given (finite) clause set.

2.5.4 Finite Model Transformation

In this section we describe a set of transformations that we apply to the input problem to reduce it to an equisatisfiable problem in function-free clause logic without equality, for a given domain size.

We write $L \vee C \rightsquigarrow C' \vee C$ to indicate that the clause $C' \vee C$ is obtained from the clause $L \vee C$ by a (single) application of one of the rules.

2.5.4.1 Basic Transformation

We first describe the basic transformation $\mathcal{B}$.

Algorithm 2.5.1 (Basic Transformation $\mathcal{B}$)

(1) Abstraction of positive equations.

$s \approx y \vee C \rightsquigarrow s \not\approx x \vee x \approx y \vee C$

if s is not a variable and x is a fresh variable

$x \approx t \vee C \rightsquigarrow t \not\approx y \vee x \approx y \vee C$

if t is not a variable and y is a fresh variable

$s \approx t \vee C \rightsquigarrow s \not\approx x \vee t \not\approx y \vee x \approx y \vee C$

if s and t are not variables and x and y are fresh variables

These rules make sure that all (positive) equations are between variables.

(2) Flattening of non-equations.

$(\neg)P(\ldots, s, \ldots) \vee C \rightsquigarrow (\neg)P(\ldots, x, \ldots) \vee s \not\approx x \vee C$

if P is not $\approx$, s is not a variable, and x is a fresh variable

(3) Flattening of negative equations.

$f(\ldots, s, \ldots) \not\approx t \vee C \rightsquigarrow f(\ldots, x, \ldots) \not\approx t \vee s \not\approx x \vee C$

if s is not a variable and x is a fresh variable

(4) Separation of negative equations.

$s \not\approx t \vee C \rightsquigarrow s \not\approx x \vee t \not\approx x \vee C$

if s and t are not variables, and x and y are fresh variables

This rule makes sure that at least one side of a (negative) equation is a variable. Notice that this property is also satisfied by the transformations (2) and (3).

(5) Removal of trivial negative equations.

$x \not\approx y \vee C \rightsquigarrow C\sigma$

where $\sigma = \{x \mapsto y\}$

(6) Orientation of negative equations.

$x \not\approx t \vee C \rightsquigarrow t \not\approx x \vee C$

if t is not a variable

□

For a clause C, let the *basic transformation of* C, denoted as $\mathcal{B}(C)$, be the clause obtained from C by applying the transformations (1)-(6), in this order, each as long as it is applicable. It is easy to see that this process is guaranteed to terminate.

We extend this notation to clause sets in the obvious way, i.e., $\mathcal{B}(M)$ is the clause set consisting of the basic transformation of all clauses in M. Strictly speaking, $\mathcal{B}(C)$ and $\mathcal{B}(M)$ are unique only up to variable renaming, and even this does not necessary hold anymore with some of the improvements described below.

Note that this transformation follows the one applied by the MACE-style model finder Paradox closely. The only difference is in (1), where, in contrast to Paradox, equations of the form $s \approx x$ resp. $x \approx s$ are rewritten. This

allows us to omit functionality axioms, as explained below. Although this change may introduce more variables, this has not the negative impact it would have in Paradox's case, as the transformed clause set is not grounded (Section 2.5.6).

The two flattening transformations (2) and (3) alone, when applied exhaustively, turn a clause into a *flat* one, where a clause is flat if each of its literals is flat:

1. each top-level term of each of its negative equations is a variable or has the form $f(x_1, \ldots, x_n)$, where f is a function symbol, $n \geq 0$, and $x_1, \ldots, x_n$ are variables;

2. each top-level term of each of its non-equations is a variable.

Similar flattening transformations have been considered before as a means to deal more efficiently with equality within calculi for first-order logic without equality [25, 3].

The basic transformation above is correct in the following sense:

Lemma 2.5.2 (Correctness of $\mathcal{B}$) *The clause set M is E-satisfiable if and only if $\mathcal{B}(M)$ is E-satisfiable.*

Proof. That flattening preserves E-satisfiability (both ways) is well-know (see [25]). Regarding transformations (1), (4), (5) and (6), the proof is straightforward or trivial. □

2.5.4.2 Conversion to Relational Form

It is not hard to see that, for any clause C, the following holds for the clause set $\mathcal{B}(C)$:

1. each of its positive equations is between two variables,

2. each of its negative equations is flat and of the form $f(x_1, \ldots, x_n) \not\approx y$, and

3. each of its non-equations is flat.

After the basic transformation, we apply the following one, turning each n-ary function symbol f into a (new) $n+1$-ary predicate symbol R_f.

(7) Elimination of function symbols.

$$f(x_1, \ldots, x_n) \not\approx y \vee C \rightsquigarrow \neg R_f(x_1, \ldots, x_n, y) \vee C$$

Let $\mathcal{B}_R(M)$ be the clause set obtained from an exhaustive application of this transformation to $\mathcal{B}(M)$.

For example, application of (1) - (6) transforms the clause $a \approx f(z)$ into $a \not\approx x \vee f(z) \not\approx y \vee x \approx y$, and applying (7) as well yields $\neg R_a(x) \vee \neg R_f(z, y) \vee x \approx y$.

Recall that an $n+1$-ary relation R over a set A is *left-total* if for every $a_1, \ldots, a_n \in A$ there is an $b \in A$ such that $(a_1, \ldots, a_n, b) \in R$. The relation R is *right-unique* if whenever $(a_1, \ldots, a_n, b) \in R$ there is no other tuple of the form $(a_1, \ldots, a_n, b')$ in R.

Because of the above properties (1)–(3) of $\mathcal{B}(M)$, the transformation $\mathcal{B}_R(M)$ is well-defined, and will produce a clause set with no function symbols. This transformation however is not unsatisfiability preserving unless one considers only left-total interpretations for the predicate symbols R_f. More formally:

Lemma 2.5.3 (Correctness of $\mathcal{B}_R$) *The clause set M is E-satisfiable if and only if there is an E-model $\mathfrak{I}$ of $\mathcal{B}_R(M)$ such that $(R_f)^{\mathfrak{I}}$ is left-total, for every function symbol $f \in \Sigma_{\mathrm{F}}$.*

Proof. The direction from left to right is straightforward. For the other direction, let $\mathfrak{I}$ be an E-model of $\mathcal{B}_R(M)$ such that $(R_f)^{\mathfrak{I}}$ is left-total for every function symbol $f \in \Sigma_{\mathrm{F}}$.

Recall that functions are nothing but left-total and right-unique relations. We will show how to obtain from $\mathfrak{I}$ an E-model $\mathfrak{I}'$ of $\mathcal{B}_R(M)$, that preserves left-totality and adds right-uniqueness, i.e., such that $(R_f)^{\mathfrak{I}'}$ is both left-total and right-unique for all $f \in \Sigma_{\mathrm{F}}$. Since such an interpretation is clearly a model of $\mathcal{B}(M)$, it will then follow immediately by Lemma 2.5.2 that M is E-satisfiable.

We obtain $\mathfrak{I}'$ as the interpretation that is like $\mathfrak{I}$, except that $(R_f)^{\mathfrak{I}'}$ contains exactly one tuple $(d_1, \ldots, d_n, d)$, for every $d_1, \ldots, d_n \in |\mathfrak{I}|$, chosen arbitrarily from $(R_f)^{\mathfrak{I}}$ (this choice exists because $(R_f)^{\mathfrak{I}}$ is left-total). It is clear from the construction that $(R_f)^{\mathfrak{I}'}$ is right-unique and left-total. Trivially, $\mathfrak{I}'$ interprets $\approx$ as the identity relation, because $\mathfrak{I}$ does, as $\mathfrak{I}$ is an E-interpretation. Thus, $\mathfrak{I}'$ is an E-interpretation, too.

What is left to prove is that when $\mathfrak{I}$ is a model of $\mathcal{B}_R(M)$ so is $\mathfrak{I}'$. This follows from the fact that every occurrence of a predicate symbol R_f, with $f \in \Sigma_{\mathrm{F}}$, in the clause set $\mathcal{B}_R(M)$ is in a negative literal. But then, since $(R_f)^{\mathfrak{I}'} \subseteq (R_f)^{\mathfrak{I}}$ by construction, it follows easily that any clause of $\mathcal{B}_R(M)$ satisfied by $\mathfrak{I}$ is also satisfied by $\mathfrak{I}'$. □

The significance of this lemma is that it requires us to interpret the predicate symbols R_f as left-total relations, *but not necessarily as right-unique ones.* Consequently, right-uniqueness will not be axiomatized below.

2.5.4.3 Adding Finite Domain Constraints

In order to enforce left-totality, one could add the Skolemized version of axioms of the form

$$\forall x_1, \ldots, x_n \exists y \; R_f(x_1, \ldots, x_n, y)$$

to $\mathcal{B}_r(M)$. The resulting set would be E-satisfiable exactly when M is E-satisfiable. But the axioms would require the introduction of function symbols, which are not part of our target logic.

However, since we are interested in finite satisfiability, we can use finite approximations of these axioms. To this end, let d be a positive integer, the *domain size*. We consider the expansion of the signature of $\mathcal{B}_r(M)$ with d *domain values*, that is, d fresh constant symbols, which we name $1, \ldots, d$. Intuitively, for each E-interpretation of cardinality d, instead of the totality axiom above we can now use the axiom

$$\forall x_1, \ldots, x_n \exists y \in \{1, \ldots, d\} \; R_f(x_1, \ldots, x_n, y) \; .$$

Concretely, if f is an n-ary function symbol let the clause

$$R_f(x_1, \ldots, x_n, 1) \vee \cdots \vee R_f(x_1, \ldots, x_n, d)$$

be the *d-totality axiom for f*, and let $\mathcal{D}(d)$ be the set of all d-totality axioms for all function symbols $f \in \Sigma_{\mathrm{F}}$. The set $\mathcal{D}(d)$ axiomatizes the left-totality of $(R_f)^{\mathfrak{I}}$, for every function symbol $f \in \Sigma_{\mathrm{F}}$ and interpretation $\mathfrak{I}$ with $|\mathfrak{I}| = \{1, \ldots, d\}$.

2.5.4.4 Putting all Together

Since we want to use clause logic *without* equality as the target logic of our overall transformation, the only remaining step is the explicit axiomatization of

the equality symbol $\approx$ over domains of size d—so that we can exploit Lemma 2.5.3 in the (interesting) right-to-left direction. This is easily achieved with the clause set

$$\mathcal{E}(d) = \{i \not\approx j \mid 1 \leq i, j \leq d \text{ and } i \neq j\} \ .$$

Finally then, we define the *finite-domain transformation of* M as the clause set

$$\mathcal{F}'(M, d) := \mathcal{B}_R(M) \cup \mathcal{D}(d) \cup \mathcal{E}(d) \ .$$

Notice that equality is now completely axiomatized, and we can use a first-order calculus without equality to reason about the transformed input. To make this point explicit, we define the transformation $\mathcal{F}(M, d)$ as identical to $\mathcal{F}'(M, d)$, except that each occurrence of $\approx$ is replaced by the fresh predicate symbol E.

Putting all together we arrive at the following first main result:

Theorem 2.5.4 (Correctness of the Finite-Domain Translation) *Let d be a positive integer. Then M is E-satisfiable by some finite interpretation with domain size d if and only if $\mathcal{F}(M, d)$ is satisfiable.*

Proof. Follows from Lemma 2.5.3 and the comments above on $\mathcal{D}(d)$ and $\mathcal{E}(d)$, together with the observation that, for being a set of universal formulas with no function symbols other than the constants $1, \ldots, d$, the set $\mathcal{F}(M, d)$ is satisfiable if and only if it is satisfiable in a Herbrand interpretation with universe $\{1, \ldots, d\}$.

More precisely, for the only-if direction assume as given a Herbrand model $\mathfrak{I}$ of $\mathcal{F}(M, d)$ with universe $\{1, \ldots, d\}$. It is clear from the axioms $\mathcal{E}(d)$ that $\mathfrak{I}$ assigns false to the equation $E(d', d'')$, for any two different elements $d', d'' \in$

$\{1, \ldots, d\}$. Now, the model $\mathfrak{I}$ can be modified to assign true to all equations $E(d', d')$, for all $d' \in \{1, \ldots, d\}$ and the resulting E-interpretation will still be a model for $\mathcal{F}(M, d)$. This is, because the only occurrences of negative equations in $\mathcal{F}(M, d)$ are those contributed by $\mathcal{E}(d)$, which are still satisfied after the change. Notice, in particular, that $\mathcal{B}_R(M)$ contains only positive occurrences of equations, if any. It is this modified model that can be turned into an E-model of M. □

This theorem suggests immediately a practical procedure to search for finite models, by testing $\mathcal{F}(M, d)$ for satisfiability, with $d = 1, 2, \ldots$, and stopping as soon as the first satisfiable set has been found. Moreover, any reasonable such procedure will return in the satisfiable case a Herbrand representation of some finite model.

2.5.5 Implementation

We implemented the transformation described so far within our theorem prover Darwin. In addition to being a full-blown theorem prover for first-order logic without equality, Darwin is a decision procedure for the satisfiability of function-free clause sets, and thus is a suitable back-end for our transformation. We call the combined system FM-Darwin (for Finite Models Darwin).

Conceptually, FM-Darwin builds on Darwin by adding to it as a front-end an implementation of the transformation $\mathcal{F}$, and invoking Darwin on $\mathcal{F}(M, d)$, for $d = 1, 2, ...$, until a model is found. In reality, FM-Darwin is built *within* Darwin and differs from the conceptual procedure described so far as detailed below.

2.5.5.1 Preprocessing Improvements

Initial Transformation FM-Darwin implements some obvious optimizations over the transformation rules described in Section 2.5.4. For instance, the transformations (1)–(4) are done in parallel, depending on the structure of the current literal. Transformation (6) is done implicitly as part of transformation (7), when turning equations into relations. Also, when flattening a clause, the same variable is used to abstract different occurrences of identical sub-term, which leads to significant improvements in a number of cases.

Naming Subterms Clauses with deep terms lead to long flat clauses. To avoid this, deep subterms can be extracted and named by an equation. For instance, the clause set

$$P(h(g(f(x)), y)), \qquad Q(g(f(z)))$$

can be replaced by the clause set

$$P(h_2(x, y)), \qquad Q(h_1(z)), \qquad h_2(x, y) = h(h_1(x), y), \qquad h_1(x) = g(f(x))$$

where h_1 and h_2 are fresh function symbols. When carried out repeatedly, reusing definitions across the whole clause set, this transformation yields shorter flattened clauses.

We tried some heuristics for when to apply the transformation, based on how often a term occurs in the clause set, and how big the flattened definition is, i.e., how much it is possible to save by using the definition. The only consistent improvement on TPTP problems was achieved when introducing definitions only for ground terms. This solves 16 more problems, 14 of which are Horn. Thus, currently only ground terms are flattened by default with this transformation in

FM-Darwin.

Splitting Clauses Systems like Paradox and Mace2 use transformations that, by introducing new predicate symbols, can split a flat clause with many variables into several flat clauses *with fewer variables*. For instance, a clause of the form

$$P(x, y) \vee Q(y, z)$$

whose two subclauses share only the variable y can be transformed into the two clauses

$$P(x, y) \vee S(y) \qquad \neg S(y) \vee Q(y, z)$$

where the predicate symbol in the *connecting* literal $S(y)$ is fresh. This sort of transformation preserves (un-)satisfiability. Thus, in this example, where the number of variables in a clause is reduced from 3 to 2, procedures based on a full ground instantiation of the input clause set may benefit from having to deal with the $O(2n^2)$ ground instances of the new clauses instead of the $O(n^3)$ ground instances of the original clause. A similar observation was made in [44] and exploited beneficially to solve planning problems by reduction to SAT.

As it happens, reducing the number of variables per clause is not necessarily helpful in our case. Since (FM-)Darwin does not perform an exhaustive ground instantiation of its input clause set, splitting clauses can actually be counter-productive because it forces the system to populate its model representation with instances of connecting literals like $S(y)$ above. Our experiments indicate that this is generally expensive unless the connecting literals do not contain any variables. Still, in contrast to plain Darwin, where in general clause splitting is only an improvement for ground connecting literals, for FM-Darwin

splitting in all cases gives a slight improvement. In particular, in our experiments on the TPTP library (see Section 2.5.6) it helped to solve eight additional satisfiable problems.

$$
\begin{array}{l}
R_{c_1}(1) \vee \cdots \vee R_{c_1}(d) \\
R_{c_2}(1) \vee \cdots \vee R_{c_2}(d) \\
\qquad \vdots \\
R_{c_m}(1) \vee \cdots \vee R_{c_m}(d)
\end{array}
\qquad\qquad
\begin{array}{l}
R_{c_1}(1) \\
R_{c_2}(1) \vee R_{c_2}(2) \\
\qquad \vdots \\
R_{c_d}(1) \vee \cdots \vee R_{c_d}(d) \\
R_{c_{d+1}}(1) \vee \cdots \vee R_{c_{d+1}}(d) \\
\qquad \vdots \\
R_{c_m}(1) \vee \cdots \vee R_{c_m}(d)
\end{array}
$$

(a) (b)

Figure 2.6: Totality axioms for constants and their triangular form

Symmetry Breaking Symmetries, in particular *value symmetries*, have been identified as a major source of inefficiencies in constraint solving. A constraint satisfaction problem exhibits value symmetry if permuting the values of a partial solution for the problem, i.e., a satisfying assignment of values to a subset of the problem's variables, gives another partial solution. Breaking such symmetries often produces considerable efficiency gains—with no loss of generality.

In our context, it is easy to break some value symmetries introduced by assigning domain values to constant symbols. Extending this mechanism to unary function symbols, as done by Paradox, did not result in an improvement, so it is not discussed here. Suppose Σ_{F} contains m constants $c_1, \ldots, c_m$. Recall that $\mathcal{D}(d)$ contains, in particular, the axioms shown in Figure 2.6(a). Similarly to what is done by Paradox, these axioms can be replaced by the more "triangular"

form shown in Figure 2.6(b). It is easy to see that the triangular form has less satisfying interpretations over the domain $\{1, \ldots, d\}$ than the first form, and that, nevertheless, any interpretation satisfying the first form is isomorphic to an interpretation satisfying the second. In fact, one could further strengthen the symmetry breaking axioms by adding (unit) clauses like $\neg R_{c_1}(2), \ldots \neg R_{c_1}(d)$. We do not add them, as they do not constrain the search for a model further.

As this optimization allowed to solve about 40 more satisfiable TPTP problems, we consider it to be highly effective, especially in combination with the optimization described next.

Sort Inference Like Paradox, FM-Darwin performs a kind of sort inference in order to improve the effectiveness of symmetry breaking. Each function and predicate symbol of arity n in Σ is assigned a type respectively of the form $S_1 \times \ldots \times S_n \rightarrow S_{n+1}$ and $S_1 \times \ldots \times S_n$, where all sorts S_i are initially distinct. Each term in the input clause set is assigned the result sort of its top symbol. Two sorts S_i and S_j are then identified based on the input clause set by applying a union-find algorithm with the following rules. First, all sorts of different occurrences of the same variable in a clause are identified; second, the result sorts of two terms s and t in an equality $E(s, t)$ are identified; third, for each term or atom of the form $f(\ldots, t, \ldots)$ the argument sort of f at t's position is identified with the sort of t.

All sorts left at the end are taken to be disjoint and of the same size. In essence, this is achieved by using annotated versions $\{1^S, \ldots, d^S\}$ of the domain values for each sort S. This way, when a sorted model is found it can be translated into an unsorted model by an isomorphic translation of each sort into a single

domain of size d. Thus, in order to find a model for the original problem, one can instead search for a sorted model of this kind. This allows for applying the symmetry breaking axioms discussed above independently for each sort. For example, if Σ contains the constant symbols a, b, c and d, then the totality axioms for the unsorted problem could be $R_a(1)$, $R_b(1) \vee R_b(2)$, $R_c(1) \vee R_c(2) \vee R_c(3)$, and $R_d(1) \vee R_d(2) \vee R_d(3) \vee R_d(4)$. In contrast, if a, b, and c were of one sort, and d of a different sort, then the axioms would be $R_a(1)$, $R_b(1) \vee R_b(2)$, $R_c(1) \vee R_c(2) \vee R_c(3)$, and $R_d(1)$, a significant reduction of the search space.

It turns out that distinct sorts can be inferred for a large number of TPTP problems, leading to almost 30 more solved problems and a speed up of a factor of two, compared to symmetry breaking for only one sort.

Meta Modeling Recall from step (7) of the transformation $\mathcal{F}$ that every function symbol is turned into a predicate symbol. In our actual implementation, we go one step further and use a meta modeling approach that can make the final clause set produced by our translation more compact, and generally speed up the search as well, thanks to the way models are built in the Model Evolution calculus. The idea is the following.

For every $n > 0$, instead of generating an $n+1$-ary relation symbol R_f for each n-ary function symbol $f \in \Sigma_{\mathrm{F}}$ we use an $n+2$-ary relation symbol R_n, for all n-ary function symbols. Then, instead of translating a literal of the form $f(x_1, ..., x_n) \not\approx y$ into the literal $\neg R_f(x_1, ..., x_n, y)$, we translate it into the literal $R_n(f, x_1, ..., x_n, y)$, treating f as a zero-arity symbol. The advantage of this translation is that instead of needing one totality axiom per relation symbol R_f with $f \in \Sigma_{\mathrm{F}}$, we only need one per function symbol *arity* (among those found

in Σ_{F}). For example, if Σ_{F} contains the function symbols $f_1, \ldots, f_n$ of arity n, then instead of one totality axiom per function symbol

$$R_{f_1}(x_1, \ldots, x_n, 1) \vee \cdots \vee R_{f_1}(x_1, \ldots, x_n, d)$$

$$\ldots$$

$$R_{f_n}(x_1, \ldots, x_n, 1) \vee \cdots \vee R_{f_n}(x_1, \ldots, x_n, d)$$

it suffices to have the following single totality axiom for all function symbols of arity n

$$R_n(y, x_1, \ldots, x_n, 1) \vee \cdots \vee R_n(y, x_1, \ldots, x_n, d)$$

where the variable y is meant to be quantified over the (original) function symbols in Σ_{F}. Furthermore, in all reasonable proof procedures based on the Model Evolution calculus y will be instantiated in a totality axiom only if there is a complementary literal of the form $\neg R_n(f, x'_1, \ldots, x'_n, d)$, thus ensuring that y will be instantiated only with zero-arity symbols representing function symbols of arity n. While this is not required for correctness, it ensures that the transformation does not increase the search space. Note that the zero-arity symbols representing the original function symbols in the input are in addition to the domain constants, and never interact with them. They are intuitively of a different sort S. Moreover, by the Herbrand theorem, we can consider with no loss of generality only interpretations that populate the sort S precisely with these constants, and no more.

Meta modeling turns out to provide only a very modest improvement, in terms of time as well as memory, for a number of reasons. First, the generalization can only pay off (and is used only) if for a given arity there are at least two symbols of that arity, otherwise it merely introduces unification overhead. Second,

the symmetry breaking axioms prevent its application to constant symbols. And third, because of sort inference in combination with the Per^S predicates introduced in conflict-based learning below, it does not suffice to generalize function symbols by arity alone, instead their sorts have to be taken into account as well. Altogether this makes it questionable if the increase in complexity introduced by this transformation is justified.

Initial Domain Size Following again the example of Paradox, FM-Darwin performs some static analysis of the input clause set to quickly determine a (possibly suboptimal) lower bound k on the cardinality of any model of the clause set. Roughly, this is done by identifying cliques of disequations entailed by the clause set. Then, the search starts with k as the initial domain size instead of 1.

The computation of a lower bound is done by default because of its very small overhead. However, we must add that in our experiments it did not lead to any substantial performance improvement overall.

2.5.5.2 Run-time Improvements

Restarts The search for models of increasing size is built in Darwin's own restarting mechanism. For refutational completeness Darwin explores its search space in an iterative-deepening fashion with respect of certain *depth* measures. The same mechanism is used in FM-Darwin to restart the search with an increased domain size $d + 1$ if the input problem has no models of size d.

By modifying the treatment of equality we could allow for increasing the domain size in steps greater than 1. That is, when going from domain size d to domain size $d + m$, we would add the axioms $\mathcal{E}(d + 1)$ instead of $\mathcal{E}(d + m)$. This

would enforce a lower domain size of $d+1$ instead of $d+m$. Furthermore, as Darwin has no native support for equality we would need to add the standard axioms of equality, that is reflexivity, symmetry, transitivity, and predicate substitution axioms. This ensures that the domain elements are in an equivalence relation, if a model of domain size smaller than $d+m$ is found. As it turned out in our experiments that this is significantly less efficient, we consider only a fixed increment of 1 in the following.

Because the clause sets $\mathcal{F}(M,d)$ and $\mathcal{F}(M,d+1)$, for any d, differ only in the their subsets $\mathcal{D}(d) \cup \mathcal{E}(d)$ and $\mathcal{D}(d+1) \cup \mathcal{E}(d+1)$, respectively, there is no need to re-generate the constant part, and this is not done.

Conflict-based Learning Some of the lemmas learned by FM-Darwin are domain size independent and can therefore be carried over to later iterations with bigger domain sizes. To do that, each clause in $\mathcal{D}(d+1)$ is actually *guarded* by an additional literal D_d standing for the current domain size. Note that symmetry breaking axioms replacing totality axioms do not need to be guarded. In FM-Darwin, the lemmas depending on the current domain size d retain the guard D_d when they are built, making it easy to eliminate them when moving to the next size $d+1$. We remark that the original problem is shown to be unsatisfiable if all lemmas are domain size independent, which might be worth considering in lemma-learning and especially lemma-forgetting heuristics, or when domain sizes are incremented separately for each sort.

Lemmas produced by Darwin can be slightly generalized in FM-Darwin by making the following observation. Assume for now, just for simplicity, that sort inference on the input clause set M produces a single sort S. When the

number m of input constants (of sort S) is smaller than the current domain size d, the symmetry breaking triangular form for the totality axioms forces the first m domain values to be the interpretation of the input constants, but imposes no constraints on the remaining $d - m$ domain values. As a consequence, every Herbrand model $\mathfrak{I}$ of the clause set $\mathcal{F}(M, d)$ is invariant under any permutation p of $(m + 1, \ldots, d)$. In other words, if the model satisfies a ground literal L, it will also satisfy the literal obtained by applying p to L. This means that whenever $\mathcal{F}(M, d)$ entails a formula $\varphi(v_1, \ldots, v_k)$ containing the domain values $v_1, \ldots, v_k$ from $\{m + 1, \ldots, d\}$, it will also entail the formula

$$\forall x_1, \ldots, x_k. \bigwedge_{1 \leq i \leq k} x_i \in \{m + 1, \ldots, d\} \ \wedge \bigwedge_{1 \leq i < j \leq k} \neg E(x_i, x_j) \ \Rightarrow \ \varphi(x_1, \ldots, x_k)$$

where $\varphi(x_1, \ldots, x_k)$ is obtained from $\varphi(v_1, \ldots, v_k)$ by replacing, for each i, every occurrence of the value v_i with the fresh variable x_i.

In the general case of more than one sort, this kind of generalization is applied to lemmas containing unconstrained domain constants as follows. During preprocessing, the system adds to $\mathcal{F}(M, d)$ a unit clause of the form $Per^S(v)$ (for "v is permutable in S") for each inferred sort S and domain value v of sort S that is unconstrained by the symmetry breaking axioms for S. Then, during search, every computed lemma $C(v_1, \ldots, v_k)$ containing unconstrained values $v_1, \ldots, v_k$ of sort S, say, is generalized to the lemma

$$C(x_1, \ldots, x_k) \ \vee \bigvee_{1 \leq i \leq k} \neg Per^S(x_i) \ \vee \bigvee_{1 \leq i < j \leq k} E(x_i, x_j)$$

where $x_1, \ldots, x_k$ are fresh variables (of sort S). This lemma is then further generalized by applying to it the same process but for another sort, until all unconstrained domain values have been eliminated.

With the resulting generalized lemma the system can break more symmetries at run time than with the original lemma. In fact, the search process will avoid not just any (candidate) model $\mathfrak{I}$ that falsifies the original lemma but also any model obtained from $\mathfrak{I}$ by a well-sorted permutation of the unconstrained domain values.

A subtle point is that naming of subterms and splitting of clauses might introduce Skolem constants. Symmetry breaking must be applied to these constants just as to input constants, thus potentially increasing the number of constrained domain elements.

Combined with the improvement described next, which reduces the overhead of using longer clauses, generalized lemmas lead to shorter derivations, a smaller search space *and* smaller run-times overall. While using the original lemmas leads in general to a significant speed up of a factor of 2 to 4 (see Section 2.4.8), the magnitude of the additional improvement of the lemma generalization in our experimental evaluation so far is, however, minimal. For instance, the speed up factor is only 1.11 over the whole TPTP library. More important, the number of solved problems is essentially unchanged.

Constraint-based approach FM-Darwin has a facility for treating equality and permutability predicates as built-in constraints. In this approach, every clause of the form

$$C \vee \bigvee_{i,\iota} \neg Per^{S_\iota}(x_i) \vee \bigvee_{i,j} E(x_i, x_j)$$

where C contains no disequations and no permutability literals, is rewritten as a *constrained clause* of the form $C \leftarrow \Gamma$ where Γ is the constraint set

$$\bigcup_{i,\iota}\{Per^{S_\iota}(x_i)\} \;\cup\; \bigcup_{i,j}\{x_i \not\approx x_j\}\ .$$

Recall that Darwin's inference process is based on generating instances of input clauses and lemmas by computing context unifiers. In the regular approach, if the clause contains an equation E(x, y), with x and y of some sort S, the computation of the context unifiers will attempt to instantiate x and y to all domain values for S. Similarly, if the clause contains a permutability literal $Per^S(x)$, it will attempt to instantiate x to all unconstrained domain values for S.

In the constraint-based approach, context unifiers are computed as usual but using only the clause part C of a constrained clause $C \leftarrow \Gamma$. Then, each context unifier σ for C is further refined into the unifier $\sigma\theta$ for each solution θ of the constraint $\Gamma\sigma$ over the sort domains. These solutions are computed using constraint satisfaction techniques that treat sort assignments to variables as well as permutability constraints as domain constraints, and disequations as disequality constraints.

The main advantages of this approach are that (i) it is not necessary to include in $\mathcal{F}(M, d)$ the quadratically many ground disequations $\neg E(d, d')$ for all distinct domain values nor the linearly many ground permutability predicates $Per^S(d)$, (ii) Darwin's inference rules operate on shorter clauses, especially in case of generalized lemmas, and (iii) computing the context unifiers $\sigma\theta$ using the specialized constraint solving algorithm for the θ part is more efficient than computing $\sigma\theta$ directly with Darwin's context unification procedure.

We finally remark that meta modeling and lemma-learning are the only optimization specific to the targeted function-free clause logic, and of those only lemma-learning is specific to Model Evolution calculus. All other optimizations are applicable in the original propositional MACE-style setting as well.

2.5.6 Experimental Evaluation

2.5.6.1 Space Efficiency

As we have seen, our reduction to a clause set $\mathcal{F}(M, d)$ encoding finite E-satisfiability is heavily influenced by the one done in Paradox, but with the difference that in Paradox the whole counterpart of our clause set $\mathcal{F}(M, d)$ is grounded out, simplified, and fed to a SAT solver.

Feeding the set $\mathcal{F}(M, d)$ directly instead to a theorem prover capable of deciding the satisfiability of function-free clause sets has the advantage of often being more space-efficient. In Paradox, as the domain size d is increased, the number of ground instances of a clause grows exponentially in the number of variables in the clause [30]. In contrast, in our transformation no ground instances of the clause set $\mathcal{F}$ are produced. The subsets $\mathcal{D}$ and $\mathcal{E}$ do grow with the domain size d; however, the number of clauses in $\mathcal{D}(d)$ remains constant in d while their length grows only linearly in d. The number of clauses in $\mathcal{E}(d)$, which are all unit, grows instead quadratically.

As far as preprocessing the input clause set is concerned then, our approach already has a significant space advantage over Paradox's. This is crucial for problems that have models of a relatively large size. The only data structure that grows unbounded in size in Darwin is the context, the data structure representing the current candidate model for a problem. With function-free clause

sets the size of the context depends on the number of possible ground instances of input *literals*, a much smaller number than the number of possible ground instances of input *clauses*. In addition, our experiments show that the context basically never grows to its worst-case size.

The different asymptotic behaviors between FM-Darwin and Paradox can be verified experimentally with the following simple problem.

Example 2.5.5 (Too big to ground) Let P be an n-ary predicate symbol, let c_1,...,c_n be distinct constants, and let $x, x_1, \ldots, x_n$ be distinct variables. Then consider the clause set consisting of the following $n \cdot (n-1)/2 + 1$ unit clauses, for $n \geq 0$:

$$P(c_1, \ldots, c_n)$$

$$\neg P(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{j-1}, x, x_{j+1}, \ldots, x_n) \qquad \text{for all } 1 \leq i < j \leq n$$

The first clause just introduces n constants. Any (domain-minimal) model has to map them to at most n domain elements. The remaining clauses force the constants to be mapped to pairwise distinct domain elements. Thus, the smallest model has exactly n elements. This clause set is perhaps the simplest clause set to specify a domain with n elements in first-order logic without equality. □

We ran the example for $n = 3, \ldots, 10$ on FM-Darwin, Mace4 and Paradox and obtained the results in Table 2.5.

All **Time** results are CPU time in seconds, **Mem** is the required memory size in megabytes with a memory limit of 400 MB. "Fail at size d" means that the memory limit was exhausted while searching for a model with size d, "Inconclusive" that Paradox gave up in that domain size after the time stated.

	FM-Darwin			Mace4	Paradox		
n	**|Cont|**	**Mem**	**Time**	**Time**	**Vars**	**Clauses**	**Time**
3	14	1	< 1	< 1	14	0	< 1
4	24	1	< 1	< 1	301	123	< 1
5	37	1	< 1	< 1	3192	534	< 1
6	53	1	< 1	< 1	46749	7919	< 1
7	72	1	1.1	178	823666	46749	12
8	94	1	5.1	Fail at size 7	Inconclusive, size ≥ 7		36
9	119	1	50	Fail at size 6	Inconclusive, size ≥ 5		9.6
10	147	1	566	Fail at size 4	Inconclusive, size ≥ 4		3.6

Table 2.5: Finite model finding for Too-big-to-ground

|Cont| is the maximum context size needed by FM-Darwin in a derivation, **Vars** and **Clauses** are the number of propositional variables and clauses introduced by Paradox in the translation to propositional logic. These results confirm our expectations on FM-Darwin's greater scalability with respect to space consumption. The growth of the (propositional) variables and clauses within Paradox clearly shows exponential behavior. In contrast, Darwin's context grows much more slowly.

2.5.6.2 Comparative Evaluation on TPTP

We evaluated the effectiveness of our approach on all the satisfiable problems of the TPTP 3.1.1 in comparison to Paradox 1.3 and Mace4. Since Darwin's native input language is clause logic, we used the E prover [63] version 0.91 to convert non-clausal TPTP problems into clause form. All tests were run on Xeon 2.4Ghz machines with 1GB of RAM, with the imposed limits of 300s of CPU time and 512MB of RAM. The results are grouped based on being Horn and/or containing equality. **Sol** gives the number of problems solved by a configuration, and **Time** the average time used to solve these problems. FM-Darwin was run

with the *grounded* learning option and with an upper limit of 500 lemmas (see Section 2.4 for details), Paradox and Mace4 were run in the default configuration.

Problem Type		Problems	FM-Darwin		Mace4		Paradox 1.3	
Horn	≈		Sol	Time	Sol	Time	Sol	Time
no	no	607	575	3.9	394	3.0	578	0.9
no	yes	383	312	4.3	190	7.8	264	0.4
yes	no	65	51	17.5	37	0.2	59	2.1
yes	yes	196	136	7.0	181	3.6	182	5.3
All		1251	1074	5.1	802	4.1	1083	1.6

Table 2.6: Finite model finding over the TPTP

The results given in Figure 2.6 show that in terms of solved problems FM-Darwin significantly outperforms Mace4. Overall, our system is almost as good as Paradox, outperforming it over the non-Horn problems in the set. We speculate that a factor in Paradox's superior performance for Horn problems might be its very efficient unit propagation algorithm based on the watched literals mechanism. As the only non-Horn clauses introduced by the transformation are the totality axioms, and as lemmas learned from Horn clauses are still Horn, solving Horn problems probably requires only a minimal amount of actual search and reduces to a large degree to unit propagation.

More precisely, FM-Darwin solves 328 problems that Mace4 cannot solve, and solves 82 problems that Paradox cannot solve. Mace4 runs out of time for 169 of these problems and out of memory for the remaining ones, while Paradox runs out of memory or gives up. We sampled some of these problems and re-ran Paradox without memory and time limits, but to no avail. For problem `NLP049-1`,

for instance, about 10 million (ground) clauses were generated for a domain size of 8, consuming about 1 GB of memory, and the underlying SAT solver could not complete its run within 15 minutes.

In contrast, on all problems FM-Darwin never uses more than 200 MB of memory, and in most cases less than 50 MB. In conclusion then, both the artificial problem in Example 2.5.5 and the more realistic problems in the TPTP library support our thesis that FM-Darwin scales better on bigger problems, that is, problems with a larger set of ground instances for non-trivial domain sizes. While both approaches have the same complexity for a satisfiable problem, that is exponential for each domain size, this cost is paid eagerly in the propositional approach. On the other hand, Paradox and, to a lesser extent, Mace4 tend to solve problems faster than FM-Darwin.

2.5.6.3 CASC J3

Finally, we report on the results of CASC-J3, the CADE ATP System Competition that was part of the 2006 Federated Logic Conference [69]. CASC is an annually held competition of first-order provers, based on the TPTP library.

Table 2.6 shows the results for FM-Darwin for the SAT and EPR divisions of CASC-J3. The SAT division contains only satisfiable problems, while the EPR division contains only function-free clausal problems. Mace4 did not participate in CASC-J3 and Paradox only in the SAT division, therefore only Paradox is part of the evaluation. FM-Darwin finished third in the SAT division, after two versions of Paradox, and likewise third in the EPR division, after Darwin and DCTP, another decision procedure for function clause logic.

Consistent with the previous evaluation, FM-Darwin performs reasonably

Division	FM-Darwin		Darwin 1.3		Paradox 1.3	
	Sol	**Time**	**Sol**	**Time**	**Sol**	**Time**
SAT	70	13.6	18	31.6	90	5.7
	FM-Darwin		**Darwin 1.3**		**Vampire 8.0**	
	Sol	**Time**	**Sol**	**Time**	**Sol**	**Time**
EPR	92	10.33	100	4.7	78	4.19

Table 2.7: Finite model finding results of CASC-J3

well on the satisfiable problems in the SAT division. An explanation for Paradox apparently being significantly superior to FM-Darwin is that satisfiable function-free clause problems are included only in the EPR division, but not in the SAT division, and that among these are a large number of the problems for which FM-Darwin succeeds, but Paradox fails.

The results show furthermore that FM-Darwin is hugely superior to Darwin in the SAT division, while for function-free problems, which are decided by both systems, Darwin is superior. In more detail, Darwin can solve only 21 problems that FM-Darwin cannot solve. Of those 16 turn out to be function-free clause problems, and only for one of the other problem does Darwin return an infinite model. So, at least on the TPTP problems, Darwin's capability to find infinite Herbrand models does not seem to be an advantage.

In contrast the results show that Darwin is very efficient for EPR problems, thus providing the basis for FM-Darwin's efficiency. This does not only hold compared to FM-Darwin, but especially compared to other systems, like Vampire, a saturation based prover. In detail, Vampire solves 48 unsatisfiable EPR problems, but only 30 satisfiable ones. This highlights that while systems

such as Vampire are highly efficient in a general refutation setting, they are not well-suited for the transformation presented in this section. First, because they are not decision procedures for a function-free clause set and in practice often fail to determine its satisfiability, and second, because they are usually not capable of providing a model. In contrast, recent instantiation based calculi such as ME, Inst-Gen [38], and Disconnection Calculus [50] satisfy both criteria.

2.5.7 Conclusion

Recent years have seen considerable interest in procedures for computing finite models of first-order logic specifications. Established, leading methods based on propositional reasoning, and embodied by systems like Paradox and Mace4, do not scale well with the required domain size of the (smallest) models. We approached this major problem by instead reducing model search to function-free first-order clause sets, and to apply (efficient) theorem provers capable of deciding such problems. We presented our approach in some detail and argued for its correctness. We then provided a comparative evaluation showing that our prover, FM-Darwin, is competitive with the state-of-the-art model builders Paradox and Mace4. The results also demonstrate that the expected space advantages do indeed occur.

While FM-Darwin scales better memory-wise than the other systems considered, it generally struggles like all other finite model-finders with problems (such as the TPTP problem `LAT053-1`) whose smallest model is relatively large (20 or more elements). Increasing the scalability towards larger domain sizes is then certainly a main area of further research.

CHAPTER 3
MODEL EVOLUTION WITH LINEAR INTEGER ARITHMETIC CONSTRAINTS

3.1 Introduction

In Chapter 2 we have introduced an efficient instantiation based approach for reasoning about first-order formulas, the Model Evolution calculus. While first-order logic is a very powerful logic, many applications of automated deduction require reasoning over some form of integer arithmetic. Efficient theory reasoning is crucial in these cases, as approximations of arithmetic by axiomatization in first-order logic are incomplete as well as inefficient.

Theory reasoning techniques developed within first-order theorem proving are often impractical as they require the enumeration of complete sets of theory unifiers (in particular those in the tradition of Stickel's Theory Resolution [67]) or feature only weak or no redundancy criteria (e.g., Bürckert's Constrained Resolution [27]). The family of *Satisfiability Modulo Theories* solvers is quite efficient for ground problems modulo some well supported theories, but lacks support for quantifiers and resorts to incomplete or inefficient heuristics when dealing with quantified formulas [40, 41]. A popular class of SMT solvers is based on the DPLL(T) architecture, which combines a DPLL based SAT solver with decision procedures for the supported theories.

We will be concerned with combining some of the strengths of the two approaches. We have developed a calculus, Model Evolution with Linear Integer Arithmetic Constraints (ME(LIA)), which supports efficient reasoning over the theory of Linear Integer Arithmetic (LIA) as well as first-order clause logic. It re-

lies on a decision procedure for the full fragment of LIA instead of a complete enumerator of LIA-unifiers. The main restriction imposed on the calculus is that it does not support uninterpreted function symbols except for constants. To obtain (semi-)decidability additional restrictions are needed. In a sense, ME(LIA) is to DPLL(LIA) what ME is to DPLL, a lifting from the ground to a restricted quantified case, while keeping and building on proven architecture and design choices. For example, splitting, propagation, backjumping, and learning are techniques applicable and common to all these approaches.

In more detail, the restrictions are as follows. Firstly, every variable is restricted to range over a bounded below interval of $\mathbb{Z}$, which we for simplicity take to be $\mathbb{N}$. This restriction is not essential, it is used here only because it simplifies the treatment of the calculus. Secondly, the input needs to be restricted further in order to make complete proof procedures possible. One example for a sufficient condition is to bound the free constants to range over finite domains. Another one is to restrict the input to ground clauses. This, together with the exclusion of free function symbols other than constants, makes (entailment in) the logic recursively enumerable, and semi-decided by the calculus we present here. Further restricting the variables to finite intervals makes the logic decidable, and our calculus terminating.

In spite of these restrictions, the logic is quite powerful. For instance, functions with a finite range can be easily encoded into it. This makes the logic particularly well-suited for applications that deal with bounded domains, such as, for instance, bounded model checking and planning. As has been argued in Section 2.5, these sorts of applications can benefit from a reduction to a more

powerful logic for which efficient decision procedures are available. On top of the function-free fragment of clause logic used there, we now add integer constraints to the picture. The ability to reason natively about the integers can provide a reduction in search space even for problems that do not originally contain integer constraints.

The ME(LIA) calculus is derived from the ME calculus. Recall that the main data structures of ME are the context Λ, consisting of a finite set of first-order literals inducing the candidate models in a derivation, and Φ, consisting of the input clause set. Similarly, in ME(LIA) the context consists of the pair $\Lambda \cdot \Gamma$ and the clause sets are represented by the pair $\Phi \cdot \Psi$. Here, Λ is a set of first-order *constrained literals*, in essence literals over free constant and predicate symbols and LIA constraints, and Γ is a set of closed LIA constraints. Again, the context induces a candidate model, but this time for an expansion of the integer structure by free predicate and constant symbols. The input clauses are kept in Φ like in ME. But in contrast to ME, where clause instances were used implicitly when computing context unifiers and applying inference rules, clause instances are created explicitly in ME(LIA) and put in Ψ. A clause instance is now obtained by strengthening LIA constraints, instead of by term instantiation as in ME. In fact, one could represent clause instantiation in ME as gathering unification constraints, thus making the two calculi even more similar on the presentation level.

The crucial insight that leads from ME to ME(LIA) lies in the use of the ordering $<$ on integers instead of the ordering induced by term instantiation. This then allows ME(LIA) to work with concepts over integers that are similar

to concepts over free terms used in ME. For instance, productivity can be used as a redundancy notion in a very similar fashion, but now formulated as LIA constraints instead of unification problems.

3.2 Related Work

Most of the related work has been carried out in the framework of the resolution calculus. One of the earliest related calculi is theory resolution [67]. In our terminology, theory resolution requires the enumeration of a complete set of solutions of constraints. The same applies to various "theory reasoning" calculi introduced later [7, 39]. In contrast, in ME(LIA) all background reasoning tasks can be reduced to *satisfiability checks* of (quantified) constraint formulas. This weaker requirement facilitates the integration of a larger class of solvers (such as quantifier elimination procedures) and leads to potentially far less calls to the background reasoner. For an extreme example, the clause $\neg(0 < x) \vee P(x)$ has infinitely many most general solutions with respect to the term instantiation ordering, namely $\{x \mapsto 1\}, \{x \mapsto 2\}, \ldots$ Thus, any calculus based on the computation of complete sets of (most general) solutions of LIA-constraints may need to consider all of them. In contrast, in ME(LIA), or in other calculi based on *satisfiability* alone, like the constrained tableaux calculus in [61], or, notably Bürckert's *constrained resolution* [27], it is enough just to check that a constraint like $(0 < x)$ is LIA-satisfiable.

Constrained resolution admits background theories with (infinitely, essentially denumerable) many models, as opposed to the single fixed model that ME(LIA) works with. In fact, the proof-generation algorithm for ME(LIA) that we will present in Section 3.7.5.4 can be easily phrased in terms of constrained

resolution, as the RQ-resolution rule of constrained resolution is exactly the combination of the resolution and factoring rules used by us. On the other hand, ME(LIA) provides a goal oriented calculus with strong redundancy criteria, both of which are still absent in current research on constrained resolution.

The importance of powerful redundancy criteria has been emphasized in the development of the modern theory of resolution in the 1990s [59]. With slight variations they carry over to *hierarchical superposition* [4], a calculus that is related to constrained resolution. The recent calculus in [1] instantiates hierarchical superposition to Linear Rational Arithmetic. Similarly, the calculus in [46] integrates dedicated inference rules for Linear Rational Arithmetic into superposition. In [18] Baumgartner et al. describe conceptual differences between ME, further *instance based methods* [8] and other (resolution) calculi. For instance, like ME, ME(LIA) explicitly maintains a candidate model, which gives rise to a redundancy criterion different to the ones in superposition calculi. Also, it is known that instance-based methods naturally decide different fragments of first-order logic, and the same holds true for the constrained case.

Over the last years, *Satisfiability Modulo Theories* (SMT) has become a major paradigm for theorem proving modulo background theories. In one of its main approaches, DPLL(T), a DPLL-style SAT-solver is combined with a decision procedure for the quantifier-free fragment of the background theory T [58]. The treatment of quantified formulas is incomplete and usually guided by heuristic instantiation, although progress has been made in obtaining complete procedures for some classes of quantified formulas [40, 24]. In fact, addressing this intrinsic limitation by lifting DPLL(T) to the first-order level is one of the main moti-

vations for the ME(LIA) calculus, much like ME was motivated by the goal of lifting the propositional DPLL procedure to the first-order level while preserving its good properties. The calculus mirrors the DPLL part of the DPLL(T) approach closely, lifting the data structures and the splitting and unit propagation operations to quantified constrained literals. Support for other crucial ingredients like lemma-learning, backjumping, theory learning and theory propagation are described as part of a potential proof procedure. With these rules then ME(LIA) can indeed be seen as a proper lifting of DPLL(T) to the first-order level (within recursion-theoretic limitations).

The original ME calculus has been extended in several ways to support native reasoning over equality. The first extension, Model Evolution with Equality (MEE), extends ME with a resolution based equality treatment [17]. The second extension, MESUP, is a combination of ME and Superposition, which integrates both calculi and their redundancy criteria [19]. All three current further developments of the basic ME calculus, MEE, MESUP, and ME(LIA), natively support equality reasoning, recognizing that this is essential for basically all applications. The first two approaches focus on first-order reasoning, and are in fact complete for full first-order logic with equality. In contrast, ME(LIA) was devised for applications involving integer reasoning. Thus ME(LIA) is not a proper extension of ME, since to be able to obtain complete procedures the support for function symbols is significantly restricted.

3.3 Informal Overview

It is instructive to discuss the main ideas of the ME(LIA) calculus with some simple examples before defining the calculus formally.

The following example from finite model reasoning demonstrates the advantage of native reasoning over integers:

$$a : [1 \,..\, 100] \tag{1}$$

$$P(a) \tag{2}$$

$$\neg P(x) \leftarrow x : [1 \,..\, 100] \tag{3}$$

Here, $a : [1 \,..\, 100]$ restricts the range of the constant a to the given interval. The clause set above is unsatisfiable because the interval declaration for a together with the unit clause $P(a)$ permit only models that satisfy one of $P(1), \ldots, P(100)$. Such models however falsify the third clause. Finite model finders, e.g., need about 100 steps to refute the clause set, one for each possible value of a. The ME(LIA) calculus, on the other hand, reasons directly with integer intervals and allows a refutation in $O(1)$ steps.

For a more complex example, consider the following two unit *constrained clauses*

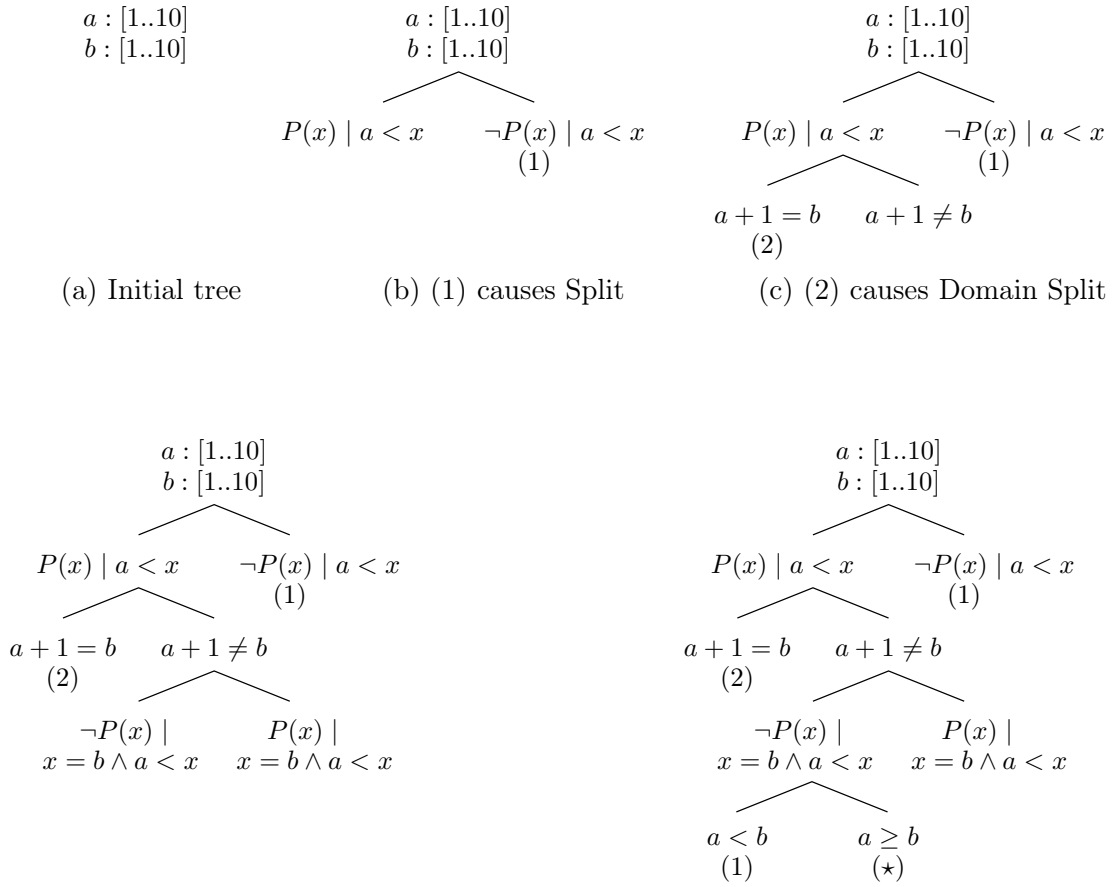
$$P(x) \leftarrow a < x \tag{1}$$

$$\neg P(x) \leftarrow x = b \tag{2}$$

where a, b are free constants, which we call *parameters*, x, y are (implicitly universally quantified) variables, and $a < x$ and $x = b$ are the respective constraints of clause (1) and (2). A constrained clause is merely a suggestive way to write a standard clause, where the constrained part of the clause implies the non-constrained part. For example, $P(x) \leftarrow a < x$ is equivalent to $P(x) \vee \neg a < x$. We also restrict the parameters to finite domains with the global constraints $a : [1 \,..\, 10]$,

$b : [1 \,..\, 10]$. Informally, clause (1) states that for each value of a in $\{1, \ldots, 10\}$, $P(x)$ holds for all integers x greater than a. Similarly for clause (2).

The clause set above is satisfiable in any expansion of the integers structure $\mathcal{Z}$ to $\{a, b, P\}$ that maps a, b into $\{1, \ldots, 10\}$ with $a \geq b$. The calculus discovers this and computes a data structure that denotes exactly these expansions. To see how this works, it is best to describe the calculus' main operations using a semantic tree construction, illustrated in Figure 3.1.



(a) Initial tree (b) (1) causes Split (c) (2) causes Domain Split

(d) (2) causes Split (e) (1) causes Domain Split

Figure 3.1: ME(LIA) derivation example

Each branch in the semantic tree denotes a finite set of first-order in-

terpretations that are expansions of $\mathcal{Z}$. These interpretations are the key to understanding the working of the calculus. The calculus' goal is to construct a branch denoting a set of interpretations that are each a model of the given clause set and the global parameter constraints, or to show that there is no such model, by closing all branches. Closed branches are marked with the number of the clause used to close them.

In the example in Figure 3.1(a), the initial single-node tree denotes all interpretations that interpret a and b over $\{1, \ldots, 10\}$ and falsify *by default* all ground atoms of the form $P(n)$ where n is an integer constant (e.g., $P(-1)$, $P(4)$, ...). Each of these (100) interpretations falsifies clause (1).

The calculus detects this by using the inference rule Inst to generate a falsified clause instance of the input clause (1), which in this case is identical to (1) itself. It then applies the Split inference rule to the *constrained literal* $P(x) \mid a < x$, derived from clause (1), and extends the tree as in Figure 3.1(b). The left branch of the new tree now denotes all interpretations that interpret a and b as before but satisfy $P(n)$ (only) for values n greater than a. The right branch still denotes the same set of interpretations as in the original branch. However, the presence of $\neg P(x) \mid a < x$ now imposes a restriction on later extensions of the branch.

For each interpretation that satisfies the constraint of a constrained literal, the calculus singles out a unique minimal solution, i.e., a tuple of integers, as its least solution. In the case of the constraint $a < x$ whose tuple of variables is the singleton (x), its least solution is $(a + 1)$ for all interpretations. This entails that in the right branch, which contains $\neg P(x) \mid a < x$, the ground literal

$\neg P(a+1)$ is *permanently satisfied* in the sense that (i) $\neg P(\mathcal{Z}(a)+1)$ is true in each interpretation $\mathcal{Z}$ of the branch and (ii) no extension of the branch is allowed to change that. As a consequence, the right branch permanently falsifies clause (1), and so it can be closed using the Close inference rule. Similarly, $P(a+1)$ is permanently satisfied in the left branch of Figure 3.1(b). In DPLL terms, the split with $P(x) \mid a < x$ and $\neg P(x) \mid a < x$ is akin to a split on the complementary ground literals $P(a+1)$ and $\neg P(a+1)$. The calculus' soundness proof relies in essence on this observation.

In interpretations of the branch where $a + 1 = b$, this poses a problem because there clause (2) is falsified. Since the branch does also have interpretations where $a + 1 \neq b$, the calculus makes progress by splitting on $a + 1 = b$. This is done with the Domain Split rule, leading to the tree in Figure 3.1(c). The leftmost branch there denotes only interpretations where $a + 1 = b$, and can be closed because it permanently falsifies clause (2). It is worth pointing out that Domain Splits like the above, identifying "critical" cases of parameter assignments, can be computed deterministically. They do not need not be guessed.

The branch ending in $a + 1 \neq b$ still contains interpretations that falsify the clause set. For instance, those that map a to 2 and b to 4 will satisfy the literal $P(4)$ and so falsify clause (2). The calculus handles this by generating the clause instance $\neg P(x) \leftarrow x = b \wedge a < x$ based on clause (2) and the branch literal $P(x) \mid a < x$. This makes the Split rule applicable with the literal $\neg P(x) \mid x = b \wedge a < x$, which yields the tree in Figure 3.1(d).

Moving to the branch ending in $\neg P(x) \mid x = b \wedge a < x$, let us consider its interpretations where $a < b$. As defined later, those interpretations satisfy

$P(a+1), \ldots, P(b-1)$ and falsify, among others, $P(b), P(b+1)$ and so on. This is a consequence of the fact that when $a < b$ then the minimal solution of the constraint in $P(x) \mid a < x$, namely $a+1$, is smaller than the minimal solution of the constraint in $\neg P(x) \mid x = b \wedge a < x$, namely b. If the branch had only such interpretations, it would permanently falsify clause (1) and could then be closed. This situation is achieved by applying Domain Split with the literal $a < b$, resulting in the tree of Figure 3.1(e). As for the branch $\star$, all its interpretations satisfy $P(n)$ for all $n > a$, because the constraint in $\neg P(x) \mid x = b \wedge a < x$ is now unsatisfiable, and falsify $P(b)$ by default, because $a \geq b$. It follows that they all satisfy the clause set. The calculus recognizes this and stops. With a similar argument the branch ending in $P(x) \mid x = b \wedge a < x$ can be extended with a Domain Split on $a < b$, whose left child would be closed due to clause (2) and whose right child would induce the same interpretations as $\star$. Which of these two branches is found first depends on the proof procedure that builds the tree. Had the clause set been unsatisfiable, the calculus would have generated a tree with closed branches only.

Note how the calculus found a model, in fact a set of models, for the input clause set without having to enumerate all possible values for the parameters a and b, resorting instead to much more course-grained domain splits. In its full generality the calculus still works as sketched above. Its formal description is, however, more complex because the calculus handles constraints with more than one (free) variable, it does not require the computation of explicit (symbolic) representations of minimal solutions, and it supports universal constraints which permanently satisfy not only their least but all their solutions.

3.4 Preliminaries

Detailed proofs of the results in this and the following sections are provided in Section 3.9.

3.4.1 Constraints

The language of our logic is made of sets of *admissible constrained* Σ-*clauses*, defined below. A (parametric linear integer) *constraint* is a first-order formula over the signature $\Sigma_{\mathcal{Z}}^{\Pi} = \{\doteq, \dot{<}, +, -, 0, \pm 1, \pm 2, \ldots\} \cup \Pi$, where Π is an infinite (denumerable) set of constant symbols not in $\Sigma_{\mathcal{Z}} = \Sigma_{\mathcal{Z}}^{\Pi} \setminus \Pi$. The symbols of $\Sigma_{\mathcal{Z}}$ have the expected arity and usage. Following a common math terminology, we will call the elements of Π *parameters*.

We will use, possibly with subscripts, the letters m, n to denote the *integer constants* (the constants in $\Sigma_{\mathcal{Z}}$); a, b to denote *parameters* (the constants in Π); c, d, s, t to denote constraints; and l, k to denote literals; We will use $\dot{\leq}$ to denote less than or equal, definable using $\doteq$ and $\dot{<}$. We write $t : [m \mathinner{..} n]$ as an abbreviation of $m \dot{\leq} t \wedge t \dot{\leq} n$. We write $c(\mathbf{x})$ to denote that c is a constraint whose argument tuple is *exactly* $\mathbf{x}$. We denote by $\bar{\exists}\, c$ (resp. $\bar{\forall}\, c$) the existential (resp. universal) closure of the constraint c, and by $\pi\, \mathbf{x}\, c$ the *projection of* c *on* $\mathbf{x}$, i.e., $\exists \mathbf{y}\, c$ where $\mathbf{y}$ is a tuple consisting of all the free variables of c that are not in the variable tuple $\mathbf{x}$. The notations $\forall \mathbf{x}\, c$ and $\exists \mathbf{x}\, c$ stand just for c when $\mathbf{x}$ is empty.

A constraint is *ground* if it contains no variables, *closed* if it contains no free variables. We define a satisfaction relation $\models_{\mathcal{Z}}$ for closed parameter-free constraints as follows: $\models_{\mathcal{Z}} c$ if c is satisfied in the standard sense in the structure $\mathcal{Z}$ of the integers—the one interpreting the symbols of $\Sigma_{\mathcal{Z}}$ in the usual way over

the universe $\mathcal{Z}$. A *parameter assignment* α, a mapping from Π to $\mathcal{Z}$, determines an expansion $\mathcal{Z}_\alpha$ of $\mathcal{Z}$ to the signature $\Sigma_{\mathcal{Z}}^{\Pi}$ that interprets each $a \in \Pi$ as $\alpha(a)$. For each parameter assignment α and closed constraint c we write $\alpha \models_{\mathcal{Z}} c$ to denote that c is satisfied in $\mathcal{Z}_\alpha$. A (possibly non-closed) constraint c is *α-satisfiable* if $\alpha \models_{\mathcal{Z}} \bar{\exists}\, c$. For finite sets Γ of closed constraints we denote by $\mathit{Mods}(\Gamma)$ the set of all assignments α such that $\alpha \models_{\mathcal{Z}} \Gamma$. We write $\Gamma \models_{\mathcal{Z}} c$ to denote that $\alpha \models_{\mathcal{Z}} c$ for all $\alpha \in \mathit{Mods}(\Gamma)$. We might abuse α to denote the constraint which equates each parameter a with the value $\alpha(a)$.

If e is a term or a constraint, $\mathbf{y} = (y_1, \ldots, y_k)$ is a tuple of distinct variables containing the free variables of e, and $\mathbf{t} = (t_1, \ldots, t_k)$, we denote by $e[\mathbf{t}/\mathbf{y}]$ the result of simultaneously replacing each free occurrence of y_i in e by t_i, possibly after renaming e's bound variables as needed to avoid variable capturing. We will write just $e[\mathbf{t}]$ when $\mathbf{y}$ is clear from context. With a slight abuse of notation, when $\mathbf{x}$ is a tuple of distinct variables, we will write $e[\mathbf{x}]$ to denote that the free variables of e are included in $\mathbf{x}$. For any assignment α, a tuple $\mathbf{m}$ of integer constants is *an α-solution of* a constraint $c[\mathbf{x}]$ if $\alpha \models_{\mathcal{Z}} c[\mathbf{m}]$. For instance, $\{a \mapsto 3\} \models_{\mathcal{Z}} c[4, 1]$ with $c[x, y] = a \doteq x - y$, and $a : [1 \,..\, 10] \models_{\mathcal{Z}} \exists x\ x \dot{<} a$.

As indicated in Section 3.3, the calculus needs to analyze constraints and their minimal solutions. In short, to make progress the calculus requires that there exist minimal solutions for each (satisfiable) constraint. In order to achieve completeness, the calculus also requires that the number of minimal solutions is denumerable. As we show below, it is possible to build orderings and restrictions on constraints such that both of these requirements are satisfied, and such that the number of minimal solutions is in fact finite. One solution would be to

base minimal solutions on an ordering which ensures that both requirements are satisfied, for example $0, -1, 1, -2, 2, \ldots$ (see Lemma 3.9.1 in Section 3.9.1 for further details). Instead we choose to go with the natural order $\dot{<}$ on integers, as this makes some notions simpler and makes operational reasoning about the calculus more intuitive. On the other hand, the predicate $\dot{<}$ does not guarantee that minimal solutions always exist—consider e.g. the constraint $x \dot{<} 0$. We will address this problem below, but first we need to introduce the extension of $\dot{<}$ from integers to constraint solutions.

With tuples $\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathbf{t} = (t_1, \ldots, t_n)$ of terms, we will use the following abbreviations, where $\mathbf{s} \dot{\leq} \mathbf{t}$ denotes the component-wise extension of the integer ordering $\dot{\leq}$ to integer tuples, $\mathbf{s} \dot{\leq}_\ell \mathbf{t}$ denotes the lexicographic extension of $\dot{\leq}$, and $\mathbf{s} \dot{<} \mathbf{t}$ and $\mathbf{s} \dot{<}_\ell \mathbf{t}$ the strict versions of those:

$$
\begin{aligned}
\mathbf{s} \doteq \mathbf{t} &\stackrel{\text{def}}{=} s_1 \doteq t_1 \wedge \cdots \wedge s_n \doteq t_n \\
\mathbf{s} \dot{\leq} \mathbf{t} &\stackrel{\text{def}}{=} s_1 \dot{\leq} t_1 \wedge \cdots \wedge s_n \dot{\leq} t_n \\
\mathbf{s} \dot{<} \mathbf{t} &\stackrel{\text{def}}{=} \mathbf{s} \dot{\leq} \mathbf{t} \wedge \neg(\mathbf{s} \doteq \mathbf{t}) \\
\mathbf{s} \dot{<}_\ell \mathbf{t} &\stackrel{\text{def}}{=} s_1 \dot{<} t_1 \vee (s_1 \doteq t_1 \wedge (s_2, \ldots, s_n) \dot{<}_\ell (t_2, \ldots, t_n)) \\
\mathbf{s} \dot{\leq}_\ell \mathbf{t} &\stackrel{\text{def}}{=} \mathbf{s} \doteq \mathbf{t} \vee \mathbf{s} \dot{<}_\ell \mathbf{t}
\end{aligned}
$$

We note that if $n = 0$ then $\mathbf{s} \doteq \mathbf{t}$ and $\mathbf{s} \dot{\leq} \mathbf{t}$ hold by convention, and thus $\mathbf{s} \dot{\leq}_\ell \mathbf{t}$ holds as well, while $\mathbf{s} \dot{<} \mathbf{t}$ and $\mathbf{s} \dot{<}_\ell \mathbf{t}$ do not hold.

We say that a constraint c is *admissible* iff for all parameter assignments α, if c is α-satisfiable then the set of α-solutions of c contains a finite but non-zero number of minimal elements with respect to $\dot{\leq}$, each of which we call a *minimal*

α-solution of c. This can be easily enforced by conjoining a given constraint $c[\mathbf{x}]$ with the constraint $\mathbf{n} \dot{\leq} \mathbf{x}$ for some tuple $\mathbf{n}$ of integer constants.

Lemma 3.4.1 *Let* $c[\mathbf{x}]$ *be a constraint and* $\mathbf{n}$ *a tuple of integer constants of the same length as* $\mathbf{x}$ *such that* $\models_{\mathcal{Z}} \exists \mathbf{x}\, \mathbf{n} \dot{\leq} \mathbf{x}$. *Then c is admissible.*

From now on we always assume that all constraints are admissible.

We stress that for the calculus to be effective, it need not actually *compute* minimal solutions. Instead, it is enough for it to work with constraints that *denote* each of the n minimal α-solutions $m_1, ..., m_n$ (in strict lexicographic order) of an α-satisfiable constraint $c[\mathbf{x}]$. This is achieved with the formulas defined below, where $\mathbf{y}$ is a tuple of fresh variables with the same length as $\mathbf{x}$ and $k \geq 1$.

$$\mu\, c \stackrel{\text{def}}{=} c \wedge \forall \mathbf{y}\, (c[\mathbf{y}] \rightarrow \neg(\mathbf{y} \dot{<} \mathbf{x}))$$

$$\mu_\ell\, c \stackrel{\text{def}}{=} c \wedge \forall \mathbf{y}\, (c[\mathbf{y}] \rightarrow \mathbf{x} \dot{\leq}_\ell \mathbf{y})$$

$$\mu_k\, c \stackrel{\text{def}}{=} \mu_\ell\, (\neg(\mu_1\, c) \wedge \cdots \wedge \neg(\mu_{k-1}\, c) \wedge (\mu\, c))$$

Recalling that c is admissible, it is easy to see that for any assignment α, $\mu\, c$ has at most n α-solutions (for some n): the n minimal α-solutions of c, if any. If c is α-satisfiable, let $m_1, ..., m_n$ be the enumeration of these solutions in the lexicographic order $\dot{<}_\ell$. Observing that $\dot{<}_\ell$ is a linearization of $\dot{<}$, it is also easy to see that $\mu_\ell\, c$ has exactly one α-solution: m_1. Similarly, for $k = 1, \ldots, n$, $\mu_k\, c$ has exactly one α-solution: m_k. This is thanks to the additional constraint $\neg(\mu_1\, c) \wedge \cdots \wedge \neg(\mu_{k-1}\, c)$, which excludes the previous minimal α-solutions, denoted by $\mu_1\, c, \ldots, \mu_{k-1}$. For $k > n$, $\mu_k\, c$ is never α-satisfiable.

Lemma 3.4.2 *Let α be an assignment and c an admissible constraint. Then, there is an $n \geq 0$ such that $\mu_1 c, \ldots \mu_n c$ have unique, pairwise different α-solutions, which are all minimal α-solutions of c. Furthermore, for all $k > n$, $\mu_k c$ is not α-satisfiable.*

For example, if $c[x, y] = a \dot{\leq} x \wedge a \dot{\leq} y \wedge x \neq y$ then

- $\{(x \doteq a \wedge y \doteq a+1), (x \doteq a+1 \wedge y \doteq a)\}$ is the set of all minimal α-solutions of c for any α,
- $\mu\, c$ is semantically equivalent ($\equiv$) to $(x \doteq a \wedge y \doteq a+1) \vee (x \doteq a+1 \wedge y \doteq a)$,
- $\mu_\ell c \equiv (x \doteq a \wedge y \doteq a+1)$,
- $\mu_1\, c \equiv (x \doteq a \wedge y \doteq a+1)$,
- $\mu_2\, c \equiv (x \doteq a+1 \wedge y \doteq a)$, and
- $\mu_3\, c$ is not α-satisfiable, for any α.

For convenience, we will implicitly consider the ordering $\dot{<}$ when we talk about minimal solutions of a constraint, while we will consider the ordering $\dot{\leq}_\ell$ when we talk about the least solution of a constraint.

3.4.2 Constrained Clauses

We now expand the signature $\Sigma^{\Pi}_{\mathcal{Z}}$ with a finite set of free predicate symbols, and denote the resulting signature by Σ. The language of our logic is made of sets of *admissible constrained* Σ*-clauses*, defined below. The semantics of the logic consists of all the expansions of the integer structure to the signature Σ, the Σ*-expansions* of $\mathcal{Z}$.

A *normalized literal* is an expression of the form $(\neg)P(\mathbf{x})$ where P is an n-ary free predicate symbol of Σ and $\mathbf{x}$ is an n-tuple of *distinct* variables. We write $L(\mathbf{x})$ to denote that L is a normalized literal whose argument tuple is *exactly* $\mathbf{x}$.

A *normalized clause* is an expression $C = L_1(\mathbf{x}_1) \vee \cdots \vee L_n(\mathbf{x}_n)$ where $n \geq 0$ and each $L_i(\mathbf{x}_i)$ is a normalized literal, called a *literal in* C. We write $C(\mathbf{x})$ to indicate that C is a normalized clause whose variables are exactly $\mathbf{x}$. We denote the empty clause by $\square$.

A *(constrained Σ-)literal* K is a pair $L(\mathbf{x}) \mid c$ where $L(\mathbf{x})$ is a normalized literal and c is a constraint with free variables included in $\mathbf{x}$. We denote by $\overline{K}$ the constrained literal $\overline{L}(\mathbf{x}) \mid c$, where $\overline{L}$ is the complement of L. For convenience we may use the literal $L(\mathbf{t})$ instead of its normalized version $L(\mathbf{x}) \mid \pi\, \mathbf{x}\, (\mathbf{x} \doteq \mathbf{t}[\mathbf{z}/\mathbf{x}])$ where $\mathbf{z}$ is a tuple of fresh variables, e.g. $P(a)$ instead of $P(x) \mid x = a$.

A *(constrained Σ-)clause* $D[\mathbf{x}]$ is an expression of the form

$$C(\mathbf{x}) \leftarrow c$$

where $C(\mathbf{x})$ is a normalized clauses and c is a constraint with free variables included in $\mathbf{x}$. When C is $\square$ we call D a *constrained empty clause*. We call $C(\mathbf{x}) \leftarrow d$ an *instance of* $C(\mathbf{x}) \leftarrow c$, if for any parameter assignment α it holds that $\alpha \models_{\mathcal{Z}} \bar{\forall}\, (d \rightarrow c)$. For a constrained clause $D = L_1(\mathbf{x}_1) \vee \cdots \vee L_n(\mathbf{x}_n) \leftarrow c$ we call each $L_i(\mathbf{x}_i) \mid c_i$, with $c_i = \pi\, \mathbf{x}_i\, c$, a *(constrained) literal of* D.

A clause $C(\mathbf{x}) \leftarrow c$ is *LIA-(un)satisfiable* if there is a (no) Σ-expansion of the integer structure $\mathcal{Z}$ that satisfies the formula $\forall \mathbf{x}\, (c \rightarrow C(\mathbf{x}))$. A set S of clauses and constraints is *LIA-(un)satisfiable* if there is an (no) Σ-expansion of $\mathcal{Z}$ that satisfies every element of S. As we are interested in LIA-satisfiability only, we will from now on for convenience shorten LIA-(un)satisfiable to just

(un)-satisfiable, unless we want to stress that we are working with the standard interpretation of LIA.

We will consider only *admissible clauses*, i.e., constrained clauses $C(\mathbf{x}) \leftarrow c$ where (i) $C \neq \Box$ and (ii) c is an admissible constraint. Condition (i) above is motivated by purely technical reasons. It is, however, no real restriction, as any clause $\Box \leftarrow c$ in a clause set S can be replaced by $\mathit{false} \leftarrow c$, where false is a 0-ary predicate symbol not in S, once S has been extended with the clause $\neg \mathit{false} \leftarrow \top$.

Requiring admissible constraints is the real restriction, needed to guarantee the existence of minimal solutions, as explained earlier. To simplify the presentation, we will further restrict ourselves to clauses with (trivially admissible) constraints of the form $c[\mathbf{x}] \wedge \mathbf{0} \dot{\leq} \mathbf{x}$, where $\mathbf{0}$ is the tuple of all zeros. For brevity, in our examples we will usually leave the constraint $\mathbf{0} \dot{\leq} \mathbf{x}$ implicit.

We may mark a constrained literal as *universal*, otherwise we call it *non-universal*. The *permanent constraint* $\mathsf{perm}(L(\mathbf{x}) \mid c)$ of a universal constrained literal is c, the permanent constraint of a non-universal literal is $\mu_\ell\, c$. That is, the permanent constraint of a non-universal literal is just its least α-solution for any assignment α, while for a universal literal it is the constraint itself. This will enable the calculus to make stronger assumptions during a derivation. For convenience, we will write $\mathsf{perm}(c)$ instead of $\mathsf{perm}(L(\mathbf{x}) \mid c)$ when the constrained literal in question is clear from context.

As informally introduced in Section 3.3, the calculus relies on least α-solutions in order to make progress. In fact, the calculus compares least α-solutions of constraint literals with respect to $\dot{\leq}_\ell$ with the following comparison

operators, where $\mathbf{x}$ and $\mathbf{y}$ are disjoint vectors of variables of the same length:

$$c \dot{<}_{\mu_\ell} d \stackrel{\text{def}}{=} \exists \mathbf{x} \exists \mathbf{y} \, (\mu_\ell \, c[\mathbf{x}] \wedge \mu_\ell \, d[\mathbf{y}] \wedge \mathbf{x} \dot{<}_\ell \mathbf{y}) \ .$$

In words, for every α, the formula $c \dot{<}_{\mu_\ell} d$ is α-satisfiable iff c and d are α-satisfiable, and the least α-solution of c is $\dot{<}_\ell$-smaller than the least α-solution of d.

Similarly, we write

$$c \dot{=}_{\mu_\ell} d \stackrel{\text{def}}{=} \exists \mathbf{x} \, (\mathsf{perm}(c[\mathbf{x}]) \wedge \mathsf{perm}(d[\mathbf{x}]))$$

to denote the formula expressing that the permanent constraints of $c[\mathbf{x}]$ and $d[\mathbf{x}]$ of two constrained literals have common solutions.

From the above, it is not difficult to show the following.

Lemma 3.4.3 (Total ordering) *Let α be a parameter assignment, and $c[\mathbf{x}]$ and $d[\mathbf{x}]$ two α-satisfiable admissible constraints. Then, one of the following cases applies:*

(i) $\alpha \models_{\mathcal{Z}} c \dot{<}_{\mu_\ell} d$

(ii) $\alpha \models_{\mathcal{Z}} c \dot{=}_{\mu_\ell} d$

(iii) $\alpha \models_{\mathcal{Z}} d \dot{<}_{\mu_\ell} c$

We stress that the restriction to α-satisfiable constraints is essential here. If c or d is not α-satisfiable, then none of the listed cases applies. If c and d are not constraints of universal literals, then the three cases are mutually exclusive, otherwise Case (ii) as well as one of the other cases can apply.

3.4.3 Constrained Contexts

A *(constrained) context* is a pair $\Lambda \cdot \Gamma$ where Λ is a finite set of constrained literals and Γ is a finite set of closed constraints. We will call a constrained literal in Λ a *context literal*. We will implicitly identify the set Λ with its closure under renamings of a context literal's free variables.

In the discussion of Figure 3.1 we explained informally the meaning of parameter constraints and constrained literals. In terms of the semantic tree presentation, each branch there corresponds roughly to a context $\Lambda \cdot \Gamma$, where Γ are the parameter constraints along the branch and Λ are the constrained literals. The purpose of this section is to provide a formal account for that.

Definition 3.4.4 (α-Covers) Let α be a parameter assignment. A constrained literal $L(\mathbf{x}) \mid c_1$ *α-covers* a constrained literal $L(\mathbf{x}) \mid c_2$ if $\alpha \models_{\mathcal{Z}} \bar{\exists}\, c_2$ and $\alpha \models_{\mathcal{Z}} \bar{\forall}\, (c_2 \rightarrow c_1)$.

If Γ is a set of closed constraints, $L(\mathbf{x}) \mid c_1$ *Γ-covers* $L(\mathbf{x}) \mid c_2$ if it α-covers it for all $\alpha \in \mathit{Mods}(\Gamma)$. □

The intention of the previous definition is to compare context literals with respect to their set of solutions for a fixed assignment α. This is expressed basically by the second condition in the definition of α-covers. For example, $P(x) \mid a \dot{<} x$ α-covers $P(x) \mid a{+}1 \dot{<} x$, for any α. The first condition ($\alpha \models_{\mathcal{Z}} \bar{\exists}\, c_2$) is needed to exclude α-coverage for the trivial reason that c_2 is not α-satisfiable. Without it, for example, $P(x) \mid x \doteq 2$ would α-cover $P(x) \mid x \doteq a \wedge a \doteq 5$ when, say, $\alpha(a) = 3$, which is not intended. But note that $\alpha \not\models_{\mathcal{Z}} \exists x\, (x \doteq a \wedge a \doteq 5)$ in this case. Also note that the two conditions $\alpha \models_{\mathcal{Z}} \bar{\exists}\, c_2$ and $\alpha \models_{\mathcal{Z}} \bar{\forall}\, (c_2 \rightarrow c_1)$ in combination enforce that c_1 is α-satisfiable as well.

Definition 3.4.5 (α-Extends) Let α be a parameter assignment. A constrained literal $L(\mathbf{x}) \mid c_1$ *α-extends* a constrained literal $L(\mathbf{x}) \mid c_2$ if $L(\mathbf{x}) \mid c_1$ α-covers $L(\mathbf{x}) \mid c_2$ and $L(\mathbf{x}) \mid \mathsf{perm}(c_1)$ α-covers $L(\mathbf{x}) \mid \mathsf{perm}(c_2)$.

If Γ is a set of closed constraints, $L(\mathbf{x}) \mid c_1$ *Γ-extends* $L(\mathbf{x}) \mid c_2$ if it α-extends it for all $\alpha \in \mathit{Mods}(\Gamma)$. □

The notion of α-extension is similar to that of α-coverage, but requires in addition that the permanent constraints themselves are α-covering.

For instance, $P(x) \mid 0 \dot{\leq} x \wedge x \dot{<} 7$ α-extends and α-covers $P(x) \mid 0 \dot{\leq} x \wedge x \dot{<} 3$ for any α, the least solution being (0) for both literals. On the other hand, $P(x) \mid 0 \dot{\leq} x \wedge x \dot{<} 7$ α-covers $P(x) \mid 1 \dot{\leq} x \wedge x \dot{<} 3$, but only when $P(x) \mid 0 \dot{\leq} x \wedge x \dot{<} 7$ is universal does it α-extend $P(x) \mid 1 \dot{\leq} x \wedge x \dot{<} 3$ as well.

The concepts introduced in the next three definitions allow us to associate a set of structures to each context satisfying certain well-formedness conditions.

Definition 3.4.6 (α-Contradictory) Let $\Lambda \cdot \Gamma$ be a context and $\alpha \in \mathit{Mods}(\Gamma)$. A constrained literal $L(\mathbf{x}) \mid c$ is *α-contradictory with* Λ if there is a context literal $\overline{L}(\mathbf{x}) \mid d$ in Λ such that $\alpha \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$. It is *$\Gamma$-contradictory with* Λ if there is a $\overline{L}(\mathbf{x}) \mid d$ in Λ such that $\Gamma \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$. The literal $L(\mathbf{x}) \mid c$ is *contradictory with* the context $\Lambda \cdot \Gamma$ if it is α-contradictory with Λ for some $\alpha \in \mathit{Mods}(\Gamma)$. The context $\Lambda \cdot \Gamma$ itself is *contradictory* if some context literal in Λ is contradictory with it. □

Note that the calculus always uses Γ-contradictory in such a way that the following more general definition would be justified as well: $L(\mathbf{x}) \mid c$ is α-contradictory with the context for each $\alpha \in \mathit{Mods}(\Gamma)$. That is, we do not use the

same context literal for all α but can choose a different one for each. We decided to go with the less general definition as it is probably more efficient to implement and puts less stringent requirements on a proof procedure.

The notion of Γ-contradictory is based on conflicts between context literals and clause literals due to overlaps in the permanent constraints of α-solutions for all $\alpha \in \mathit{Mods}(\Gamma)$. It underlies the abandoning of candidate models due to permanently falsified clauses in Section 3.3, which is captured precisely in the Close rule of the calculus.

The following definition is analogous to the notion of productivity in ME. To achieve that either a ground literal or its complement is produced (and not both), a universal literal has to trump a non-universal literal. Just comparing minimal solutions does not suffice, all solutions of a universal constraint must be taken into account.

Definition 3.4.7 (α-Produces) Let $\Lambda \cdot \Gamma$ be a context and $\alpha \in \mathit{Mods}(\Gamma)$. A constrained literal $L(\mathbf{x}) \mid c_1$ *α-produces* a constrained literal $L(\mathbf{x}) \mid c_2$ *wrt.* Λ iff

1. $L(\mathbf{x}) \mid c_1$ α-covers $L(\mathbf{x}) \mid c_2$, and
2. (a) $L(\mathbf{x}) \mid c_1$ is universal, or
   (b) there is no $\overline{L}(\mathbf{x}) \mid d \in \Lambda$ such that
      i. $\overline{L}(\mathbf{x}) \mid d$ α-covers $\overline{L}(\mathbf{x}) \mid c_2$, and
         A. $\overline{L}(\mathbf{x}) \mid d$ is universal, or
         B. $\alpha \models_{\mathcal{Z}} c_1 \dot{<}_{\mu_\ell} d$.

The set Λ *α-produces* a constrained literal K if some literal in Λ α-produces K wrt. Λ. A context $\Lambda \cdot \Gamma$ *produces* K if there is an $\alpha \in \mathit{Mods}(\Gamma)$ such that Λ α-

produces K. If a literal $L(\mathbf{x}) \mid c_1$ in Λ α-covers $L(\mathbf{x}) \mid c_2$ for some $\alpha \in \mathit{Mods}(\Gamma)$, but does not α-produce it wrt. Λ due to some literal $\overline{L}(\mathbf{x}) \mid d$ in Λ, we say that $\overline{L}(\mathbf{x}) \mid d$ *α-blocks* $L(\mathbf{x}) \mid c_1$ *from α-producing* $L(\mathbf{x}) \mid c_2$ *in* $\Lambda \cdot \Gamma$ □

We require our contexts not only to be non-contradictory but also to guarantee that the associated Σ-expansions of $\mathcal{Z}$ are total over tuples of natural numbers. All this is achieved with *admissible* contexts.

Definition 3.4.8 (Γ, Admissible Context) A context $\Lambda \cdot \Gamma$ is *admissible* if

1. Γ is satisfiable.

2. For each free predicate symbol P of non-zero arity in Σ, the set Λ contains $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$.

3. $\Lambda \cdot \Gamma$ is not contradictory.

□

Thanks to Condition 2 in the above definition, an admissible context α-produces a literal $\neg P(\mathbf{n})$ with $\mathbf{n}$ consisting of non-negative integer constants, if no other literal in the context α-produces $P(\mathbf{n})$. While this does not cover propositional literals, i.e., predicate symbols of arity 0, a simple preprocessing step can take care of that. For each predicate symbol P of arity 0 in the signature we introduce a fresh predicate symbol P' of arity 1. Then, we transform the input clause set by exhaustively replacing each clause of the form $(\neg)P \vee C \leftarrow c$, where P is a propositional symbol, by the clause $(\neg)P'(x) \vee C \leftarrow c \wedge x \doteq 0$, where x is a fresh variable.

Observe that admissible contexts $\Lambda \cdot \Gamma$ may contain context literals whose constraint is not α-satisfiable for some (or even all) $\alpha \in \mathit{Mods}(\Gamma)$. For those α's, such literals simply do not matter as their effect is null.

Admissible contexts are always consistent in the following sense.

Lemma 3.4.9 (Consistent α-Productivity) *Let $\Lambda \cdot \Gamma$ be an admissible context and $\alpha \in \mathit{Mods}(\Gamma)$. For any constrained literal $L(\mathbf{x}) \mid c$, Λ cannot α-produce both $L(\mathbf{x}) \mid c$ and its complement $\overline{L}(\mathbf{x}) \mid c$.*

The following definition provides the formal account of the meaning of contexts.

Definition 3.4.10 (Induced Structure) Let $\Lambda \cdot \Gamma$ be an admissible context and let $\alpha \in \mathit{Mods}(\Gamma)$. The Σ-*structure $\mathcal{Z}_{\Lambda,\alpha}$ induced by Λ and α* is the expansion of $\mathcal{Z}$ to all the symbols in Σ that agrees with α on the parameters and satisfies a positive ground literal $L(\mathbf{s})$ iff Λ α-produces $L(\mathbf{s})$. □

Lemma 3.4.11 *Let $\Lambda \cdot \Gamma$ be an admissible context and $\alpha \in \mathit{Mods}(\Gamma)$. For any ground literal $L(\mathbf{s})$ such that $\alpha \models_{\mathcal{Z}} \mathbf{0} \dot{\leq} \mathbf{s}$, $\mathcal{Z}_{\Lambda,\alpha}$ satisfies $L(\mathbf{s})$ if and only if Λ α-produces $L(\mathbf{s})$.*

Thus, Definition 3.4.10 connects syntax (α-productivity) to semantics (truth) in a one-to-one way. For convenience, we might say that a context satisfies (falsifies) a clause if one of the structures induced by it satisfies (falsifies) the clause.

As an important consequence all ground instances of a literal corresponding to its permanent constraint are satisfied in an induced structure.

Lemma 3.4.12 *Let $\Lambda \cdot \Gamma$ be an admissible context, let $L(\mathbf{x}) \mid c$ be a context literal in Λ, and let $\alpha \in Mods(\Gamma)$. If c is α-satisfiable then each literal in $\{L(\mathbf{m}) \mid \alpha \models_{\mathcal{Z}} \mathsf{perm}(c)[\mathbf{m}/\mathbf{x}]\}$ is satisfied by $\mathcal{Z}_{\Lambda,\alpha}$.*

That is, the permanent constraints of context literals are responsible for refining the current and, as will become clear later, all future structures induced by a context. We make the effect of a permanent constraint more explicit, by using it to define the notion of permanently satisfied ground literals on top of it.

Definition 3.4.13 (Permanently α-Satisfied) Let $\Lambda \cdot \Gamma$ be an admissible context, and let $L(\mathbf{x}) \mid c \in \Lambda$ be a context literal in Λ. A ground literal $L(\mathbf{s})$ is *permanently α-satisfied* by a $L(\mathbf{x}) \mid c \in \Lambda$ in $\Lambda \cdot \Gamma$ for some $\alpha \in Mods(\Gamma)$, if $\mathbf{s}$ is an α-solution of $\mathsf{perm}(c)$.

A ground literal $L(\mathbf{s})$ is *permanently α-falsified* by a context literal K in a context $\Lambda \cdot \Gamma$, if its complement $\overline{L}(\mathbf{s})$ is permanently α-satisfied by K in $\Lambda \cdot \Gamma$.

□

These notions extend to ground clauses in the natural way.

The definition implies that if a ground literal $L(\mathbf{s})$ is permanently α-satisfied in an admissible context $\Lambda \cdot \Gamma$, then $\mathcal{Z}_{\Lambda,\alpha}$ satisfies $L(\mathbf{s})$ in $\Lambda \cdot \Gamma$. The calculus will further justify the understanding that, if a $L(\mathbf{s})$ is permanently α-satisfied in $\Lambda \cdot \Gamma$, then in any context $\Lambda' \cdot \Gamma'$ obtainable from $\Lambda \cdot \Gamma$ with $\alpha \in Mods(\Gamma')$ it is the case that $\mathcal{Z}_{\Lambda',\alpha}$ satisfies $L(\mathbf{s})$ as well.

That is, intuitively, for a given $\alpha \in Mods(\Gamma')$ such that c is α-satisfiable, a universal context literal $L(\mathbf{x}) \mid c$ causes all its ground instances $L(\mathbf{m}_\alpha)$, with m_α an α-solution of c, to be permanently α-satisfied in the current and all future

contexts. In contrast, a non-universal context literal $L(\mathbf{x}) \mid c$ causes only the ground instance $L(\mathbf{m}_\alpha)$ with m_α being the lexicographic least α-solution of c to be permanently α-satisfied.

The inference rules of the calculus will use the notion of a Γ-universal literal to detect if a context literal can be made universal, i.e., if it permanently satisfies only its least or all of its solutions.

Definition 3.4.14 (α-Universal) Let $C[\mathbf{x}] \leftarrow c(\mathbf{x})$ be a clause with literals $L_i(\mathbf{x}_i) \mid c_i(\mathbf{x}_i)$ for $1 \leq i \leq k$, and $\Lambda \cdot \Gamma$ an admissible context. Let for a given assignment $\alpha \in \mathit{Mods}(\Gamma)$ the set S be defined as the set consisting of all α-solutions of c, and let S_i be defined as the set consisting of the projections of each α-solution in S over $\mathbf{x}_i$. Then *the literal* $L_i(\mathbf{x}_i) \mid c_i$ *of the clause* $C \leftarrow c$ *is* α*-universal in the context* $\Lambda \cdot \Gamma$ iff for each $\mathbf{m}_i \in S_i$ there is an $\mathbf{m} \in S$ such that i) $\mathbf{m}_i$ is the projection of $\mathbf{m}$ over $\mathbf{x}_i$, and ii) for each $L_j(\mathbf{x}_j) \mid c_j$ with $i \neq j$, $L_j(\mathbf{m}_j)$ is permanently α-falsified in $\Lambda \cdot \Gamma$.

If a literal of a clause $C \leftarrow c$ is α-universal in $\Lambda \cdot \Gamma$ for all $\alpha \in \mathit{Mods}(\Gamma)$, then it is Γ-*universal* in Λ. □

That is, in a given context the Γ-universal literals of a clause instance are literals that, solely due to permanently falsified literals, must be Γ-satisfied in the induced structure in order to satisfy $C \leftarrow c$. Note that the notion of an α-universal (Γ-universal) literal depends on a clause and the current context, while a universal constrained literal is independent of both. The calculus connects these two notions by marking a context literal as universal if it is Γ-universal at the point when it is added to the context. Universal constrained literals are an optimization of the calculus, they are needed neither for completeness nor for

soundness. It is safe to (consistently) consider a universal context literal as non-universal. This makes it possible to use cheap approximations of the potentially expensive check for the Γ-universality criterion given above (see Section 3.7.5.1).

3.4.4 Context Unifier

In Section 3.3 we explained the derivation in Figure 3.1 as being driven by semantic considerations, to construct a model by successive branch extensions. The calculus' inference rules achieve that in their core by generating falsified clause instances based on *context unifiers*.

Definition 3.4.15 (Context Unifier) Let $\Lambda \cdot \Gamma$ be an admissible context and $C[\mathbf{x}] = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c[\mathbf{x}]$ a constrained clause with free variables $\mathbf{x}$. A *context unifier of C against $\Lambda \cdot \Gamma$* is a constraint

$$d[\mathbf{x}] = d'[\mathbf{x}] \wedge \exists \mathbf{y}\, (\mathbf{y} \dot{\leq} \mathbf{x} \wedge \mu_j\, d'[\mathbf{y}]), \quad \text{where } d'[\mathbf{x}] = c[\mathbf{x}] \wedge e_1[\mathbf{x}_1] \wedge \cdots \wedge e_k[\mathbf{x}_k]$$

with each e_i coming from a context literal $\overline{L_i}(\mathbf{x}_i) \mid e_i$ in Λ, and $j \geq 1$. We say that each literal $L_i(\mathbf{x}_i) \mid c_i$ of C with $c_i = \pi\, \mathbf{x}_i\, c$ has been *paired with* $\overline{L_i}(\mathbf{x}_i) \mid e_i$. □

The following lemma follows directly by construction of the constraint of a literal of a clause.

Lemma 3.4.16 *Let $\Lambda \cdot \Gamma$, C, d, and $\overline{L_i}(\mathbf{x}_i) \mid e_i$ be as in Definition 3.4.15. Then for all $\alpha \in Mods(\Gamma)$, the least α-solution of each literal $L_i(\mathbf{x}_i) \mid d_i$ of $C \leftarrow d$ cannot be smaller than the least solution of the corresponding context literal $\overline{L_i}(\mathbf{x}_i) \mid e_i$.*

The constraint d can perhaps be best understood as follows. Its component $d' = c[\mathbf{x}] \wedge e_1[\mathbf{x}_1] \wedge \cdots \wedge e_k[\mathbf{x}_k]$ denotes any simultaneous solution of C's constraint and the constraints coming from pairing each of C's literals with a context literal with the same predicate symbol but opposite sign. The component $\mu_j d'[\mathbf{y}]$ denotes the j^{th} minimal solution of d', which bounds from below the solutions of d. A simple, but important consequence (for completeness) is that for any α and concrete solution $\mathbf{m}$ of d', j can be always chosen so that $d[\mathbf{m}]$ is α-satisfied. As a special case, when $\mathbf{m}$ is the j-th minimal solution of d', it is also the least (and single minimal) solution of d.

Example 3.4.17 As a first example, without parameters, for simplicity, let d' $= c[x_1, x_2] \wedge e_1[x_1] \wedge e_2[x_2]$ where

$$c = \neg(x_1 \doteq x_2), \quad e_1 = 1 \dot{\leq} x_1, \quad e_2 = 1 \dot{\leq} x_2 \ .$$

Then, the (unique) solution of $\mu_j\, d'$ for $j = 1$ is $(1, 2)$; for $j = 2$ it is $(2, 1)$. By fixing $j = 1$ now let us commit to $(1, 2)$. Then the solutions of d_1 are $(1), (2), \ldots$ and the solutions of d_2 are $(2), (3), \ldots$ The least solution of d_1, (1), coincides with the projection over x_1 of the committed minimal solution $(1, 2)$. Similarly for d_2. This is no accident and is crucial in proving the soundness of the calculus. It relies on the property that the least (individual) solutions of all the d_i's are, in combination, the least solution of d—which is in turn the j.th minimal solution of d'. In the example, the least solutions of d_1 and d_2 are 1 and 2, respectively, and combine into $(1, 2)$, the least solution of d. □

We stress that all the notions in the above definition are effective thanks to the decidability of LIA. A subtle point here is the choice of j in (3.4.4), as

j is not bounded *a priori*. However, all these notions hold only if d and thus d' is α-satisfiable for some $\alpha \in \mathit{Mods}(\Gamma)$. As d' as a conjunction of admissible constraints is itself admissible, it follows as argued before that only finitely many members of any family of context unifiers are admissible. By this argument, the possible values for j are effectively bounded.

The calculus performs clause instantiation solely based on context unifiers. The next example takes a closer look at the role played by context unifiers in the third step in the example from the introduction.

Example 3.4.18 Consider the context

$$\{P(x) \mid a \dot{<} x\} \cdot \{a : [1 \,..\, 10], b : [1 \,..\, 10]\}$$

and the input clause

$$\neg P(x) \leftarrow x \doteq b.$$

The context corresponds to the left branch in Figure 3.1(b). There is a context unifier, for any $j \geq 1$:

$$d = x \doteq b \wedge a \dot{<} x \wedge \exists y \, (y \dot{\leq} x \wedge \mu_j \, (y \doteq b \wedge a \dot{<} y)).$$

The sole literal of the clause instance $\neg P(x) \leftarrow d$ is $K' = \neg P(x) \mid d_1$, where $d_1 = \pi \, \mathbf{x} \, d = d$. The constraint $y \doteq b \wedge a \dot{<} y$ has a unique minimal α-solution, which is then also its least α-solution. Thus, d is equivalent to $x \doteq b \wedge a \dot{<} x$, obtained with $j = 1$. Let us analyze if K' is Γ-contradictory, that is, if $\Gamma \models_{\mathcal{Z}} (a \dot{<} x) \doteq_{\mu_\ell} d$. That is true iff

$$\Gamma \models_{\mathcal{Z}} \exists x \, \mu_\ell \, (a \dot{<} x) \wedge \mu_\ell \, (x \doteq b \wedge a \dot{<} x) \ . \tag{3.1}$$

By quantifier elimination we can show that checking (3.1) reduces to checking if $\Gamma \models_{\mathcal{Z}} a + 1 \doteq b$. Since that entailment does not hold, K' is not Γ-contradictory.

But since some $\alpha \in \mathit{Mods}(\Gamma)$ do satisfy $a + 1 \doteq b$, a literal split is not applicable to K', as the resulting context would be contradictory. Splitting the domain with $a + 1 \doteq b$ makes K' Γ-contradictory in the left conclusion, and the literal split applicable in the right conclusion. This case analysis gives rise to the domain split in Figure 3.1(c). □

The last example gives a preview of the guiding role that context unifiers can play in determining the universality of context literals.

Example 3.4.19 Let $\Lambda \cdot \Gamma$ be a context containing the (non-universal) literals $\overline{L_1}(x) \mid x \dot{\leq} b$, $\overline{L_1}(x) \mid a \dot{\leq} x$, and $\overline{L_2}(x, y) \mid -1 \dot{\leq} x \wedge -1 \dot{\leq} y$. Let d be a context unifier of the clause $L_1(x) \vee L_2(x, y) \leftarrow (x \doteq a \vee x \doteq b) \wedge x \dot{<} y$ paired with the literals $\overline{L_1}(x) \mid a \dot{\leq} x$ and $\overline{L_2}(x, y) \mid -1 \dot{\leq} x \wedge -1 \dot{\leq} y$. If $\Gamma \models_{\mathcal{Z}} d$ then the literal $L_2(x, y) \mid d_2$ is Γ-universal while the literal $L_1(x) \mid d_1$ is not. □

3.5 The Calculus

The input language of the calculus consists of a (finite) set Φ of constrained clauses and a satisfiable (finite) set Γ of closed constraints. We call the logic described by this language the ME(LIA)-logic.

The inference rules of the calculus are defined over quadruples, *sequents*, of the form $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$, where $\Lambda \cdot \Gamma$ is an admissible context, and Φ and Ψ are sets of constrained clauses. We will write Λ, L to denote the set of constrained literals $\Lambda \cup \{L\}$, and similar with closed constraints, resp. constrained clauses, for Γ, resp. Φ and Ψ. We will implicitly define the components of a sequent S as Λ_S,

Γ_S, Φ_s, and Ψ_s. Φ is initialized with a set Φ_0 of admissible input clauses, while Ψ is initially empty and is extended by the calculus with instances of clauses from Φ.

The completeness of the calculus guarantees that each of its branches terminates with failure, i.e., contains the constrained empty clause $\square \leftarrow \top$, when Φ_0 is LIA-unsatisfiable. Contrapositively, from any unfailing branch it is possible to extract (possibly in the limit) a set of models for Φ_0. When the branch is finite and ends with a sequent $\Lambda_n \cdot \Gamma_n \vdash \Phi_n \cdot \Psi_n$, these models are precisely those denoted by $\Lambda_n \cdot \Gamma_n$.

Context unifiers play a crucial role in the evolution of $\Lambda \cdot \Gamma$. To illustrate their use, consider a sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$. If for some $\alpha \in \mathit{Mods}(\Gamma)$ the structure $\mathcal{Z}_{\Lambda,\alpha}$ induced by Λ and α falsifies Φ, it must falsify a "ground" instance $C[\mathbf{m}]$ of some clause $C(\mathbf{x}) \leftarrow c$ in Φ. This implies the existence of a context unifier d of $C(\mathbf{x}) \leftarrow c$ against $\Lambda \cdot \Gamma$, where $\mathbf{m}$ is an α-solution of d. The Inst rule, described later, will add the clause instance $C(\mathbf{x}) \leftarrow d$ to Ψ. This will trigger other rules of the calculus, which try to extend the context to make it satisfy $C(\mathbf{x}) \leftarrow d$.

If $C(\mathbf{x}) \leftarrow d$ in Ψ has a literal $L(\mathbf{x}_i) \mid d_i$ which is not contradictory with the context, the problem with $C[\mathbf{m}]$ can be fixed by adding $L(\mathbf{x}_i) \mid d_i$ to Λ using the rules Split or Extend introduced below. In essence, if $\mathbf{m}_i$ is the projection of $\mathbf{m}$ over $\mathbf{x}_i$, then for each $\alpha \in \mathit{Mods}(\Gamma)$ that satisfies d, $L(\mathbf{x}_i) \mid d_i$ will α-produce $L_i[\mathbf{m}_i]$ in the new context (assuming it is not blocked by other context literals, see Definition 3.4.7), since its least solution is no greater than $\mathbf{m}_i$. That will make the new $\mathcal{Z}_{\Lambda,\alpha}$ satisfy $L_i[\mathbf{m}_i]$ and so $C[\mathbf{m}]$ as well. This is the analogous of "lifting" in Herbrand-based theorem proving.

If a literal of $C \leftarrow d$ is contradictory with the context, it must be β-contradictory with Λ for one or more $\beta \in \mathit{Mods}(\Gamma)$. Then, it is necessary to strengthen Γ to eliminate offending βs. This is achieved with the Domain Split rule. Strengthening Γ using Domain Split eventually makes either Split or Extend applicable to a literal of $C(\mathbf{x}) \leftarrow d$, or makes all literals of $C(\mathbf{x}) \leftarrow d$ Γ-contradictory. In the latter case, the calculus will close the corresponding branch with the Close rule.

3.5.1 Derivation Rules

The ME(LIA) calculus consists at its core of five mandatory rules, which are sufficient to obtain a sound and complete proof procedure. The original calculus in [14] is extended here with a number of redundancy criteria, simplification rules, and optional derivation rules, which are geared towards improving the efficiency of practical proof procedures. To simplify the presentation, the core and optional derivation rules only extend components of a sequent, while the simplification rules only shrink components of a sequent. Furthermore, redundancy criteria apply only to clause instances in Ψ, while the clause simplification rules apply only to clauses in Φ. An implementation would probably integrate obvious simplifications, for example due to redundancy criteria, into a rule application.

It is also noteworthy that for some rules, for example Close and Extend, the application of a rule instance does not make it non-applicable in the conclusions. That is, the rule instance could in principle be applied infinitely often, even in the case where the premise and conclusions of a rule instance are identical. This is of course pointless in practice. When a rule instance has been applied to a sequent, then it does not need to be applied to its conclusions or any sequent obtained from its conclusions by further rule applications. This is justified by

the redundancy criteria and side conditions of the derivation rules defined below. Thus an implementation does not have to apply any rule instance more than once.

We call the clause in Ψ singled out in the premises of Close, Split, Extend, Domain Split, and Assert the *selected clause* of the rule instance. We call the clause in Ψ singled out in the conclusions of Inst and Inst Assert the *derived clause* of the rule instance.

3.5.1.1 Redundancy

Redundancy criteria serve to determine clauses in Ψ which are redundant and can be safely ignored when considering which inference rules need to be applied. Redundancy is a persistent notion. If a clause is redundant in a sequent S, then it is redundant in all conclusions obtained from S. Thus an implementation can opt to remove a clause D from Ψ as soon as it recognizes that D is redundant.

Let S be a sequent, and let $D(\mathbf{x}) = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow d$ be a clause in Ψ_S. Then there must be a sequent S_{Inst} to which Inst was applied with derived clause D, and from which S is derived by a sequence of inference rule applications. Furthermore, d must be a context unifier of a clause $C(\mathbf{x}) = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c$ in Φ_{Inst} paired with context literals $\overline{L_i}(\mathbf{x}_i) \mid e_i$ in Λ_{Inst}.

Definition 3.5.1 (Redundancy) Let C, D and $\overline{L_i}(\mathbf{x}_i) \mid e_i$ be as above. The clause D is *α-redundant in a sequent* $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ for an $\alpha \in \mathit{Mods}(\Gamma)$ if any of the following holds:

(i) R-Unsatisfiable: $\alpha \models_{\mathcal{Z}} \bar{\forall} \neg d$

That is, D is trivially satisfied because its constraint is α-unsatisfiable.

(ii) R-Unproductive: There is an i, $1 \leq i \leq k$ such that $\overline{L_i}(\mathbf{x}_i) \mid e_i$ does not α-produce $\overline{L_i}(\mathbf{x}_i) \mid c_i$ wrt. Λ.

That is, one of the context literals of the context unifier does not α-produce the complement of the instance of the clause literal it was paired with.

(iii) R-Subsume: There is (modulo variable renaming) a clause $E = F \leftarrow f$ in Ψ such that $F \subset \{L_1, \cdots, L_k\}$ and each literal of E α-extends some literal of D.

This is analogous to strict subsumption in first-order logic.

A clause in Ψ is Γ-*redundant in* S if it is α-redundant in S for all $\alpha \in \mathit{Mods}(\Gamma)$. An inference rule application is α-*redundant in* S if its selected or derived clause is α-redundant. It is Γ-*redundant in* S if it is α-redundant in S for all $\alpha \in \mathit{Mods}(\Gamma)$. □

The definition formalizes that an inference rule application is Γ-redundant in a sequent $S = \Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ if it is not applicable to the sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi'$, where Ψ' is obtained from Ψ by removing all clauses which are redundant in S.

As a side note, the following is a natural candidate for a redundancy criterion: If a literal of a clause instance D is α-produced by a context $\Lambda \cdot \Gamma$, then D is α-redundant in $\Lambda \cdot \Gamma$. Intuitively, in this case the context literals which were used to compute the context unifier d are not sufficient for falsifying any ground instance of C. As it turns out, this redundancy criterion is a special case of R-Unproductive, and thus it is not necessary to specify it explicitly.

3.5.1.2 Core Rules

The core rules of the calculus are crucial for obtaining sound and complete proof procedures.

$$\textsf{Inst}\ \frac{\Lambda \cdot \Gamma \ \vdash (\Phi, C \leftarrow c) \cdot \Psi}{\Lambda \cdot \Gamma \ \vdash (\Phi, C \leftarrow c) \cdot (\Psi, C \leftarrow d)}$$

where

1. d is a context unifier of $C \leftarrow c$ against $\Lambda \cdot \Gamma$.

This rule recognizes through the existence of a context unifier d that the context (possibly) falsifies the instance $C \leftarrow d$ of the input clause $C \leftarrow c$. It adds the instance to Ψ, which enables other rules.

$$\textsf{Close}\ \frac{\Lambda \cdot \Gamma \ \vdash \Phi \cdot (\Psi, C \leftarrow c)}{\Lambda \cdot \Gamma \ \vdash \Phi \cdot (\Psi, C \leftarrow c, \square \leftarrow \top)}$$

where

1. each literal of $C \leftarrow c$ is Γ-contradictory with Λ.

This rule recognizes that the context permanently Γ-falsifies some (ground instance of a) clause and adds the empty clause as a marker for that.

A potential special case worth noting is when the selected clause is a constrained empty clause, i.e., $\square \leftarrow c$. Close then applies, its side condition is trivially satisfied as $\square \leftarrow c$ has no literal. But Close is only sound if c is entailed by Γ, which is usually ensured precisely by having Γ-contradictory literals. This is the reason why we insist that all input clauses are admissible, as this ensures that Φ cannot contain a constrained empty clause. Thus, it is impossible for this special case to occur.

We note that the correctness proof justifies to weaken Condition 1 in Close so that each literal of $C \leftarrow c$ is α-contradictory with Λ for each $\alpha \in \mathit{Mods}(\Gamma)$. Thus, an implementation can make use of this fact to apply Close more eagerly, but it is not required to do so to be complete.

$$\textsf{Split}\ \frac{\Lambda \cdot \Gamma \ \vdash\ \Phi \cdot (\Psi, C \leftarrow c)}{(\Lambda, L_i \mid c_i) \cdot \Gamma \ \vdash\ \Phi \cdot (\Psi, C \leftarrow c) \qquad (\Lambda, \overline{L_i} \mid c_i) \cdot \Gamma \ \vdash\ \Phi \cdot (\Psi, C \leftarrow c)}$$

where

1. $L_i \mid c_i$ is a literal of $C \leftarrow c$,
2. neither $L_i \mid c_i$ nor $\overline{L_i} \mid c_i$ is contradictory with $\Lambda \cdot \Gamma$.

This rule, analogous to the main rule of the DPLL procedure, derives one of two possible sequents non-deterministically. The left conclusion fixes the context by adding the *split literal* $L_i \mid c_i$, which then Γ-produces a literal of the potentially falsified clause $C \leftarrow c$. The right conclusion is needed for soundness and forces the calculus to consider other alternatives.

Split is intended to be applied only if the clause, and in particular the split literal, is not currently produced by the context, i.e., it is not R-Unproductive Γ-redundant. Its application makes $C \leftarrow c$ R-Unproductive Γ-redundant in the left conclusion, and the split literal contradictory in the right conclusion.

$$\textsf{Extend}\ \frac{\Lambda \cdot \Gamma \qquad \vdash \Phi \cdot (\Psi, C \leftarrow c)}{(\Lambda, L_i \mid c_i) \cdot \Gamma \ \vdash \Phi \cdot (\Psi, C \leftarrow c)}$$

where

1. $L_i \mid c_i$ is a literal of $C \leftarrow c$,
2. $\overline{L_i} \mid c_i$ is Γ-contradictory with Λ.

This rule can be seen as a one-branched Split, where the split literal must be produced by all $\alpha \in \mathit{Mods}(\Gamma)$. Its application makes $C \leftarrow c$ R-Unproductive Γ-redundant in the left conclusion.

To illustrate the need for Extend, suppose $\Lambda = \{\neg P(x) \mid -1 \dot{\leq} x,\ P(x) \mid x : [1\,..\,5]\}$, $\Gamma = \emptyset$ and $C = P(x) \leftarrow x : [1\,..\,7]$. The clause C is falsified in the (single) induced interpretation, because, for instance, $\neg P(6)$ is satisfied. Adding $P(x) \mid x : [1\,..\,7]$ to Λ will fix the problem. However, Split cannot be used for that since $\neg P(x) \mid x : [1\,..\,7]$ is Γ-contradictory with Λ—for having the same least solution, 1, as the constraint of $P(x) \mid x : [1\,..\,5]$. Extend will do instead.

$$\textsf{Domain Split}\ \frac{\Lambda \cdot \Gamma \ \vdash\ \Phi \cdot (\Psi, C \leftarrow c)}{\Lambda \cdot (\Gamma, d) \ \vdash\ \Phi \cdot (\Psi, C \leftarrow c) \qquad \Lambda \cdot (\Gamma, \neg d) \ \vdash\ \Phi \cdot (\Psi, C \leftarrow c)}$$

where

1. $L_i \mid c_i$ is a literal of $C \leftarrow c$,
2. there is an $\alpha \in \mathit{Mods}(\Gamma)$, and an $L_i \mid e$ or $\overline{L_i} \mid e$ in Λ such that $L_i \mid c_i$ is α-contradictory but not Γ-contradictory with $\overline{L_i} \mid e$,
3. $d = \bar{\exists}(c_i \doteq_{\mu_\ell} e)$.

The purpose of this rule is to enable applications of Close, Split, and Extend, which are not applicable to the current context due to too strong or insufficient contradictions between clause literals and the context. It achieves this by partitioning the current $\mathit{Mods}(\Gamma)$ into two non-empty parts thanks to the *split constraint* d.

3.5.1.3 Optional Rules

The following rules are optional, but are often preferable to the core rules.

$$\textsf{Inst Assert}\ \frac{\Lambda \cdot \Gamma \ \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi}{\Lambda \cdot \Gamma \ \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot (\Psi, C \vee L_0 \leftarrow d)}$$

where

1. $K = \overline{L_0}(\mathbf{x}_0) \mid -\mathbf{1} \dot{\leq} \mathbf{x}_0$,
2. d is a context unifier of $C \vee L_0 \leftarrow c$ against $(\Lambda, K) \cdot \Gamma$, where $L_0(\mathbf{x}_0)$ is paired with K and no other literal of $C \vee L_0 \leftarrow c$ is paired with K,
3. $L_0 \mid d_0$ is Γ-universal in $\Lambda \cdot \Gamma$.

The intent of this rule is to detect if universal literals can be added to the context, and to add a clause instance suitable for Assert. In the conclusion either Close applies, or Assert applies with assert literal $L_0 \leftarrow d_0$, possibly after an application of Domain Split.

While K is contained in any admissible context if L is positive, the K used here is not actually in Λ but merely used for technical reasons. This achieves in essence that $L_0(\mathbf{x}_0)$ is not paired with any literal, thus generating the most general clause instance possible. As a design choice, K could be restricted to positive literals only, in order to keep the applicability of Inst and Inst Assert mutually exclusive.

$$\textsf{Assert}\ \frac{\Lambda \cdot \Gamma \qquad\qquad \vdash \Phi \cdot (\Psi, C \leftarrow c)}{(\Lambda, L_i \mid c_i) \cdot \Gamma \ \vdash \Phi \cdot (\Psi, C \leftarrow c)}$$

where

1. $L_i \mid c_i$ is a literal of $C \leftarrow c$,
2. $L_i \mid c_i$ is not contradictory with $\Lambda \cdot \Gamma$,

3. $L_i \mid c_i$ is Γ-universal in $\Lambda \cdot \Gamma$,

4. $L_i \mid c_i$ is marked as universal in $(\Lambda, L_i \mid c_i) \cdot \Gamma$.

This rule is intended to strengthen the context and make applications of Split and Extend Γ-redundant by adding universal literals to the context.

It might be surprising that the side conditions of Assert do not explicitly require that each $L_j \mid c_j$ of $C \leftarrow c$ with $j \neq i$ must be Γ-contradictory with Λ. This is justified by the observation that the *assert literal* $L_i \mid c_i$ is Γ-universal in $\Lambda \cdot \Gamma$. This implies that $L_j \mid c_j$ is α-contradictory with Λ for each $\alpha \in \mathit{Mods}(\Gamma)$. While this condition is weaker it is still sufficient for soundness.

The decision to introduce universal literals only here, but not in the Split rule, keeps the calculus simpler. Otherwise, the reduction to ground DPLL style splitting is not easily justified anymore, and, similarly to ME, we would need to introduce a mechanism like Skolem constants in order to deal with right conclusions.

$$\textsf{Ground Split}\ \frac{\Lambda \cdot \Gamma \ \vdash\ \Phi \cdot \Psi}{\Lambda \cdot (\Gamma, l) \ \vdash\ \Phi \cdot \Psi \qquad \Lambda \cdot (\Gamma, \overline{l}) \ \vdash\ \Phi \cdot \Psi}$$

where

1. l is a ground constraint literal over the parameters Π,

2. $\alpha \models_{\mathcal{Z}} l$, for some $\alpha \in \mathit{Mods}(\Gamma)$,

3. $\Gamma \not\models_{\mathcal{Z}} l$.

This rule partitions Γ by extending it with ground literals only, with the goal of keeping Γ as ground as possible, thus keeping tests involving it (relatively) cheap.

In particular, an application of Domain Split to d can be replaced by a succession of applications of Ground Split to ground literals l, such that l occurs in some constraint e with $\alpha \models_{\mathcal{Z}} e \leftrightarrow d$, and $\alpha \models_{\mathcal{Z}} l$ and $\Gamma \not\models_{\mathcal{Z}} l$. The constraint e could, for example, be computed from d by quantifier elimination.

3.5.1.4 Simplification Rules

The simplification rules below are the direct counterparts to the simplification rules of ME [18], and simplify the input clause set Φ as well as the set of constrained literals Λ. Removing clauses from Φ with Subsume leads to less clause instances generated by Inst, while shortening clauses with Resolve leads to shorter clause instances generated by Inst. As most of the calculus's rule operate on the (literals of the) generated clause instances in Ψ, these simplifications can significantly reduce the number of possible rule applications.

$$\textsf{Subsume}\ \frac{(\Lambda, L_0 \mid e) \cdot \Gamma\ \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi}{(\Lambda, L_0 \mid e) \cdot \Gamma\ \vdash \Phi \cdot \Psi}$$

where

1. $\Gamma \models_{\mathcal{Z}} \bar{\forall}(c_0 \rightarrow \mathsf{perm}(L_0 \mid e))$.

This rule captures standard first-order unit subsumption. It removes an input clause which contains a literal that is permanently satisfied by the context. Recall that c_0 is the constraint of the literal $L_0 \mid c_0$ of $C \vee L_0 \leftarrow c$.

This implies either that $L_0 \mid e$ is universal, or that $L_0 \mid c_0$ has at most one α-solution for each $\alpha \in \mathit{Mods}(\Gamma)$, i.e., it is in essence ground.

$$\textsf{Resolve}\ \frac{(\Lambda, \overline{L_0} \mid e) \cdot \Gamma\ \vdash (\Phi, C(\mathbf{x}) \vee L_0 \leftarrow c) \cdot \Psi}{(\Lambda, \overline{L_0} \mid e) \cdot \Gamma\ \vdash (\Phi, C(\mathbf{x}) \leftarrow d) \cdot \Psi}$$

where

1. C is not empty,

2. $\Gamma \models_{\mathcal{Z}} \bar{\forall}(c_0 \to \mathsf{perm}(\overline{L_0} \mid e))$,

3. $d = \pi\, \mathbf{x}\, c$.

This rule captures standard first-order unit resolution. It removes a literal from an input clause whose complement is permanently satisfied. Recall that c_0 is the constraint of the literal $L_0 \mid c_0$ of $C \vee L_0 \leftarrow c$. The constraint d of the simplified clause is obtained from c by existentially quantifying the variables which are local to L_0 in $C \vee L_0$.

Condition 1 makes the calculus behave somewhat more nicely, by ensuring that Close is the only rule that adds a constrained empty clause to Ψ. But as simplifying a clause with Resolve to an empty constrained clause $C = \square \leftarrow d$ makes Close applicable to C (in a sound way), it is not really necessary.

$$\textsf{Compact}\ \frac{(\Lambda, L \mid e) \cdot \Gamma\ \vdash \Phi \cdot \Psi}{\Lambda \cdot \Gamma\qquad\qquad \vdash \Phi \cdot \Psi}$$

where

1. For each $\alpha \in \mathit{Mods}(\Gamma)$ there is a literal in Λ that α-extends $L \mid e$.

This rule simplifies Λ by removing literals that have no effect. The intended use is that a literal is only added to the context if Compact is not applicable to it, and that a literal is removed from the context when a Γ-extending literal is added to the context.

3.5.2 Derivations

Derivations in the ME(LIA) calculus are defined in terms of *derivation trees*, where each node corresponds to a particular application of a derivation rule, and each of the node's children corresponds to one of the conclusions of the rule. More precisely, a derivation tree is a labeled tree inductively defined as follows.

Let Φ be an admissible clause set and Γ a satisfiable set of closed constraints. A one-node tree is a derivation tree (of Φ and Γ) iff its root is labeled by an *initial sequent for* Φ *and* Γ, that is, a sequent of the form $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$, where Λ contains (only) the constrained literal $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$ for each free predicate symbol P in Σ, and Ψ is the empty set. It is easy to see that the context $\Lambda \cdot \Gamma$ is admissible.

A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to a leaf node N in $\mathbf{T}$ new children nodes $N_1, \ldots, N_m$ so that the sequents labeling $N_1, \ldots, N_m$ can be derived by applying a rule of the calculus to the sequent labeling N. In this case, we say that $\mathbf{T}'$ *is derived from* $\mathbf{T}$. When it is convenient and it does not cause confusion, we will identify the nodes of a derivation tree with their labels.

We say that a branch in a derivation tree is *closed* if its leaf is labeled by a sequent of the form $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi, \square \leftarrow \top$; otherwise, the branch is *open*. A derivation tree is *closed* if each of its branches is closed, and it is *open* otherwise. We say that a derivation tree (of Φ and Γ) is a *refutation tree* (of Φ and Γ) iff it is closed.

If in a derivation tree the branch B is an extension of the branch B', where

the leaf node of B is labeled with the segment S and the leaf node of B' is labeled with the segment S', we call S' an *extension* of S.

Definition 3.5.2 (Derivation) Let Φ be an admissible clause set and Γ a satisfiable set of closed constraints. A *derivation (in ME(LIA)) of* Φ *and* Γ is a possibly infinite sequence of derivation trees $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$, such that $\mathbf{T}_0$ is a one-node tree whose root is labeled with an initial sequent for Φ and Γ, and for all i with $0 < i < \kappa$, $\mathbf{T}_i$ is derived from $\mathbf{T}_{i-1}$. □

We say that $\mathcal{D}$ is a *refutation of* Φ *and* Γ iff $\mathcal{D}$ is finite and ends with a refutation tree of Φ and Γ.

We show below that the ME(LIA) calculus is sound and (strongly) complete in the following sense: for all admissible clause sets Φ and satisfiable sets of closed constraints Γ, $\Phi \cup \Gamma$ is unsatisfiable iff every fair derivation of Φ and Γ is a refutation of Φ and Γ.

3.6 Correctness

3.6.1 Soundness

Theorem 3.6.1 (Soundness) *For all admissible clause sets* Φ *and satisfiable sets of closed constraints* Γ*, if there is a refutation tree of* Φ *and* Γ*, then* $\Phi \cup \Gamma$ *is LIA-unsatisfiable.*

In essence, and leaving Γ aside, the proof is by first deriving a binary tree over parameter-free literals that reflects the applications of the derivation rules in the construction of the given refutation tree. For instance, a Split application with split literal $L(\mathbf{x}) \mid c$ gives rise to the literal $L(\mathbf{m})$, where $\mathbf{m}$ is the least α-solution of c for a given α. In the resulting tree neighboring nodes will be

labeled by complementary literals, like $L(\mathbf{m})$ and $\neg L(\mathbf{m})$. In the second step it is shown that this binary tree is closed by ground instances from the input set. It is straightforward then to argue that $\Phi \cup \Gamma$ is LIA-unsatisfiable.

3.6.2 Fairness

To prove the calculus' completeness we will introduce the notion of an *exhausted branch*, in essence, a (limit) derivation tree branch that need not be extended any further by the calculus and that is obtained by a fair derivation.

The specific notion of fairness that we adopt is defined formally in the following. For that, it will be convenient to describe a tree $\mathbf{T}$ as the pair $(\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is the set of the nodes of $\mathbf{T}$ and $\mathbf{E}$ is the set of the edges of $\mathbf{T}$. In the rest of the section, we will use κ to denote a countable (possibly infinite) ordinal, and i, j to denote finite ordinals.

Each derivation $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa} = (\mathbf{N}_i, \mathbf{E}_i)_{i<\kappa}$ in the calculus determines a *limit tree* $\mathbf{T} := (\bigcup_{i<\kappa} \mathbf{N}_i, \bigcup_{i<\kappa} \mathbf{E}_i)$. It is easy to show that a limit tree of a derivation $\mathcal{D}$ is indeed a tree. But it will not be a derivation tree unless $\mathcal{D}$ is finite.

We will assume the following setup for the remainder of this section, including Section 3.6.3. Let Φ be an admissible clause set, Γ a finite satisfiable set of closed constraints, and assume that $\mathcal{D}$ is a derivation of Φ and Γ. Let $\mathbf{T}$ be the limit tree of $\mathcal{D}$, and let $\mathbf{B} = (N_i)_{i<\kappa}$ be a branch in $\mathbf{T}$ with κ nodes. For all $i < \kappa$, let $S_i = \Lambda_i \cdot \Gamma_i \vdash \Phi_i \cdot \Psi_i$ be the sequent labeling node N_i, where $\Lambda_0 \cdot \Gamma_0 \vdash \Phi_0 \cdot \Psi_0$ is the initial sequent for Φ and Γ.

Definition 3.6.2 (Limit Context and Clause Set) We define the *limit con-*

text of $\mathbf{B}$ as $\Lambda_{\mathbf{B}} \cdot \Gamma_{\mathbf{B}} := (\bigcup_{i<\kappa} \Lambda_i) \cdot (\bigcup_{i<\kappa} \Gamma_i)$ and the *limit clause set* of $\mathbf{B}$ as $\Phi_{\mathbf{B}} := (\bigcup_{i<\kappa} \Phi_i)$. □

Although, strictly speaking, $\Lambda_{\mathbf{B}} \cdot \Gamma_{\mathbf{B}}$ is not a context because $\Lambda_{\mathbf{B}}$ may be infinite, for the purpose of the completeness proof we treat it as one. This is possible because all relevant definitions (in particular Definition 3.4.15) can be applied without change to $\Lambda_{\mathbf{B}} \cdot \Gamma_{\mathbf{B}}$ as well.

One of the main technical notions needed to prove the calculus' completeness is that of an *exhausted (limit) branch*, in essence, a (limit) derivation tree branch that need not be extended any further.

We need to introduce a further notion in order to define an exhausted branch.

Definition 3.6.3 (Compactness) A set of closed constraints is *compact*, if, when each of its finite subsets is satisfiable, then the set itself is satisfiable. The branch $\mathbf{B}$ is *compact* if $\Gamma_{\mathbf{B}}$ is compact. □

An exhausted branch is one that is compact, cannot be closed, and to which only Γ-redundant applications of the core rules are possible.

Definition 3.6.4 (Exhausted branch) The branch $\mathbf{B}$ is *exhausted* if for all $i < \kappa$ all of the following hold:

(i) Close is not applicable to S_i.

(ii) For all $\alpha \in \mathit{Mods}(\Gamma_{\mathbf{B}})$, if Inst is applicable to S_i with derived clause D, then there is a $j \geq i$ with $j < \kappa$ such that D is in Ψ_j or D is α-redundant in S_j.

(iii) For all $\alpha \in \mathit{Mods}(\Gamma_{\mathbf{B}})$, if Split, Extend, or Domain Split is applicable to S_i with selected clause D, then there is $j \geq i$ with $j < \kappa$ such that D is α-redundant in S_j.

(iv) $\mathbf{B}$ is compact.

□

Due to this definition, a proof procedure has, for a given sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$, to consider only clauses from Ψ that are not Γ-redundant in $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$.

Note that Condition (i) implies that an exhausted branch cannot contain the empty constrained clause $\square \leftarrow \top$. Compactness ensures that $\Gamma_{\mathbf{B}}$ is satisfiable.

Lemma 3.6.5 *$\Gamma_{\mathbf{B}}$ is satisfiable, and for all $i < \kappa$, $\mathit{Mods}(\Gamma_{\mathbf{B}}) \subseteq \mathit{Mods}(\Gamma_i)$.*

Definition 3.6.6 (Fairness) A limit tree of a derivation is *fair* if it is a refutation tree or it has an exhausted branch. A derivation is *fair* if its limit tree is fair. A proof procedure is *fair* if it produces only fair derivations. □

We point out that fair proof procedures cannot exist in general, as the ME(LIA)-logic is not semi-decidable and a fair proof procedure gives rise to a complete procedure. But for important subclasses such procedures exist and are effective, as will be shown in Section 3.7.

3.6.3 Completeness

For the rest of the section assume that $\mathcal{D}$ is a fair derivation that is not a refutation. Observe that $\mathcal{D}$'s limit tree must have at least one exhausted branch. Let $\mathbf{B}$ be this branch.

The following proposition is the main result for proving the calculus complete.

Theorem 3.6.7 (Model Construction) *For every* $\alpha \in \mathit{Mods}(\Gamma_{\mathbf{B}})$, $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ *is a model of* $\Phi_{\mathbf{B}}$.

It is worth noting that, due to Lemma 3.6.5, Proposition 3.6.7 never holds for the trivial reason that $\Gamma_{\mathbf{B}}$ is unsatisfiable. The completeness of the calculus is then a consequence of Proposition 3.6.7 and Lemma 3.6.5. We state it here in its contrapositive form to underline the model computation ability of ME(LIA).

Theorem 3.6.8 (Completeness) *Since* $\mathbf{T}$ *is not a refutation tree,* $\Phi \cup \Gamma$ *is satisfiable. More specifically, for every exhausted branch* $\mathbf{B}$ *of* $\mathbf{T}$ *and for every* $\alpha \in \mathit{Mods}(\Gamma_{\mathbf{B}})$, $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ *is a model of* $\Phi \cup \Gamma$.

Note that the theorem includes a proof convergence result, that *every* fair derivation of an unsatisfiable clause set is a refutation. In practical terms, it implies that as long as a derivation strategy guarantees fairness, the order of application of the rules of the calculus is irrelevant for proving an input clause set unsatisfiable, giving to the ME(LIA) calculus the same flexibility enjoyed by the DPLL calculus at the propositional level.

Proof. Since $\mathbf{T}$ is not a refutation tree and $\mathbf{B}$ is an exhausted branch, $\Gamma_{\mathbf{B}}$ is satisfiable by Lemma 3.6.5. Let α be arbitrary in $\mathit{Mods}(\Gamma_{\mathbf{B}})$. By Proposition 3.6.7, $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ is a model of $\Phi_{\mathbf{B}}$. Since $\Phi_{\mathbf{B}}$ is defined as $\bigcup_{i<\kappa} \Phi_i$, $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ is a model of each Φ_i, and in particular of Φ_0 ($= \Phi$). By Lemma 3.6.5, $\alpha \in \mathit{Mods}(\Gamma_0)$ ($= \mathit{Mods}(\Gamma)$). By definition, $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ agrees with α on the parameters.

It follows that $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ is a model of $\Phi \cup \Gamma$. □

When the branch $\mathbf{B}$ in Theorem 3.6.8 is finite, $\Lambda_{\mathbf{B}} \cdot \Gamma_{\mathbf{B}}$ coincides with the context $\Lambda_n \cdot \Gamma_n$, say, in $\mathbf{B}$'s leaf. From a model computation perspective, this is a very important fact because it means that a model of the original clause set—or rather, a finite representation of it, $\Lambda_n \cdot \Gamma_n$—is readily available at the end of the derivation; it does not have to be computed from the branch, as in other model generation calculi.

3.7 Proof Procedures

In this chapter we present some fair proof procedures for restricted versions of the input language of the calculus, and provide guidelines for their efficient implementation. We will first introduce a basic proof procedure that is fair for the restriction of the ME(LIA) input language to parameters with finite ranges. We will then present a more sophisticated and in some details underspecified proof procedure. We will instantiate it in several ways to get fair procedures for, the same language of bounded parameters as above, function-free first-order logic with equality, and ground formulas. Finally, we will explain how to refine the procedure presented before to obtain efficient procedures and implementations. Some of the refinements are specific to ME(LIA), such as heuristics and efficient treatment of universal literals, others are adaptations of ingredients of Darwin and DPLL solvers, such as backjumping and learning.

3.7.1 General Approaches To Achieving Fairness

The core ingredient of any fair strategy is that each rule application must be considered eventually and cannot be postponed indefinitely. We outline two basic approaches for achieving this.

In the first approach, a proof procedure computes all possible rule applications exhaustively at each step in a derivation. That is, when a rule is applied, all potential rule applications to its conclusions are computed eagerly. If all computed applications are stored in a first-in first-out queue in the order they are computed and are considered for application in FIFO order, then the above property holds naturally. Furthermore, it is possible to interleave this chronological rule application scheme with a rule application heuristic more geared towards obtaining short derivations. As long as there are only finitely many steps between two rule applications taken from the FIFO queue, it is still the case that no rule application is postponed indefinitely.

A practical drawback of this approach in the context of derivation tree based calculi is that the tree is in essence constructed breadth-first. A better alternative is to use iterative deepening over some bound imposed on rule applications, which significantly reduces memory requirements. Empirically, in first-order calculi refutations are usually possibly with small bounds. By first exhausting all rule applications within a bound, and then increasing the bound and restarting the proof procedure, no rule application is postponed indefinitely. Common bounds are often based on the depth or weight of the terms involved in a rule application, or the depth of derivation branches, or the number of applications of specific inference rules. While these could be used for ME(LIA) as well, due to the role played by constraints and the high complexity of constraint simplification, the depth or weight of terms might not be a good measure. An alternative would be to impose an upper bound on the minimal solutions of constraints of context literals. A problem with this approach is that minimal

bounds exist only symbolically, and checking them requires constraint solving, which might not be efficient in practice.

Thus, a good starting point for an implementation might be to choose an interactive deepening strategy that imposes limits on the number of rule applications per branch, in particular on Split and Domain Split. This should be compared and combined with limiting the upper bounds of minimal solutions of context literal constraints. Which approach is preferable in practice might depend on the problem structure. In the end, only an implementation can provide solid clues to which kinds of bounds work well for ME(LIA).

3.7.2 A Basic Proof Procedure

In this section we present a basic proof procedure. It serves the purpose of, firstly, showing that fair proof procedures exist for the ME(LIA) calculus, and secondly, to lay the foundation for the more sophisticated proof procedure introduced in the next section. For simplicity, we will consider only the core rules of the calculus and ignore the optional and simplification rules. Furthermore, we have to restrict the input to be able to guarantee that the procedure computes only fair derivations. For this, we will assume that the initial Γ bounds every parameter in the input signature from below and above, i.e., each parameter ranges over a finite domain. The functional style pseudo-code in Figure 3.2 gives a high-level overview of the procedure.

The two main data structures maintained throughout the procedure are the set of open branches of the current derivation tree (OpenBranches), and the set of rule application candidates (Candidates). A rule application candidate is a pair of a potential rule application and the branch to whose leaf node sequent it

Basic Procedure

```
function basic (Φ, Γ)
  let Λ = {¬P(x) | −1 ≤ x} for all predicates in Φ
  let Ψ = ∅ // empty set of clause instances
  let B = Λ · Γ ⊢ Φ · Ψ // initial sequent and single node branch
  let Candidates = candidates ({B}) // initial candidate rule applications
  solve (OpenBranches, Candidates)

function candidates (Branches)
    // returns the set of candidates applicable to the leaf sequents of Branches,
    // where a candidate is a pair (RuleApp, Branch),
    // with RuleApp a potential rule application to a branch Branch ∈ Branches

function select (OpenBranches, Candidates)
    // returns (RuleApp, OpenBranch), where
    // (RuleApp, RuleBranch) is in Candidates,
    // OpenBranch ∈ OpenBranches is an extension of RuleBranch,
    // RuleApp is applicable to OpenBranch, and
    // no applicable candidate in Candidates is smaller than RuleBranch.
    // returns None if there is no such candidate

function solve (OpenBranches, Candidates)
  if OpenBranches is empty then
    ⊥ // unsatisfiable, all branches closed
  else case select (OpenBranches, Candidates) of
    None →
      OpenBranches // satisfiable, each leaf sequent induces a model
    Some (RuleApp,OpenBranch) →
      if RuleApp is an instance of Close then
        solve (OpenBranches \ {OpenBranch}, Candidates)
      else
        let Conclusions = branches obtained by appl. RuleApp to OpenBranch
        let Candidates′ = Candidates ∪ candidates (Conclusions)
        let OpenBranches′ = (OpenBranches \ {OpenBranch}) ∪ Conclusions
        solve (OpenBranches′, Candidates′)
```

Figure 3.2: Basic ME(LIA) proof procedure

applies. Rule applications are considered in the order of the depth of the branch in which they were computed. This gives rise to the breadth-first construction of a derivation tree based on the chronological order of rule applicability, which in essence ensures fairness.

In more detail, the function *basic* takes as input a clause set Φ and a satisfiable set of closed constraints Γ. Additionally, Γ has to impose lower and upper bounds on all parameters occurring in Φ. *basic* then creates the initial sequent of the derivation based on Φ and Γ, by adding the literal $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$ for each predicate symbol P in the signature of Φ, and setting Ψ to the empty set. It also computes all possible applications of the core rules of the calculus to the initial sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$, and adds them to the candidate set Candidates. With the data structures set up control is given to the function *solve*.

solve first checks if the derivation contains any open branches, and terminates with $\perp$ if this is not the case. This signals that a refutation has been found. Otherwise, it checks if Candidates contains any applicable candidates.

A candidate (RuleApp, RuleBranch) is applicable if there is an open branch OpenBranch in the set of OpenBranches which is (identical to or) an extension of RuleBranch, and RuleApp is applicable to the leaf sequent of OpenBranch. If there are no applicable candidates in Candidates then *solve* terminates with OpenBranches, and each branch in OpenBranches induces a model of the input sets Φ and Γ. If there are applicable candidates in Candidates then the smallest one, (RuleApp, RuleBranch), is selected with the function *select*, where a strict order is imposed based on the depth of the RuleBranch of a candidate.

Now, RuleApp is applied to the leaf sequent of its associated open branch, OpenBranch. If RuleApp is an instance of Close, then OpenBranch is simply removed from OpenBranches. Otherwise OpenBranch is replaced by the open branches obtained by extending it with the conclusions of RuleApp. Finally, the candidate set is extended by adding the candidates which are applicable to the

new branches, computed with the function *candidates*. Then *solve* calls itself recursively with the updated sets of open branches and candidates.

The soundness of the procedure is easy to see, as it builds a derivation tree by application of the core rules of the calculus, and only removes branches when Close is applied. Fairness, and thus completeness, requires a more involved argument.

Theorem 3.7.1 (Fairness) *The above proof procedure is fair.*

Let us first give a high level argument why the procedure is fair, to get an intuition about the argument underlying the proof below. It is easy to see that whenever the procedure terminates, it has either found a refutation or a finite exhausted branch, so this case is immediate. It remains to argue that each infinite branch of the limit tree is exhausted, i.e., that conditions (i)-(iv) of Definition 3.6.4 hold. Conditions (i) and (iv) are immediate, see the proof below. We will use a termination argument to show that conditions (ii) and (iii) are satisfied as well.

Let S_i and D be as in condition (ii) resp. condition (iii), and denote the rule application by RuleApp. We must show that D is Γ-redundant at some point. Since all candidate rule applications are computed exhaustively, RuleApp is in *Candidates* at some point. Now, let *Ord* be the lexicographic order over $(place(\mathsf{RuleApp}, \mathsf{Candidates}), |\mathsf{OpenBranches}|)$, where $place(\mathsf{RuleApp}, \mathsf{Candidates})$ is the number of candidates that are applicable and smaller than or equal to RuleApp (according to the strict order imposed by *select*), and $|\mathsf{OpenBranches}|$ is the size of OpenBranches. We will use *Ord* as the order for the termination argument.

Since we have an infinite branch, *solve* must call itself infinitely often. If the first recursive call, in line 29, is executed, |OpenBranches| is decremented. Since |OpenBranches| is always strictly positive, the first recursive call can not be called infinitely often without incrementing |OpenBranches| in between. It follows that the second recursive call in line 34, which increases |OpenBranches| by a finite amount, must be called infinitely often. Now, if the second recursive call is executed, then *place*(RuleApp, Candidates) is decreased, as each call to *solve* decreases *place*(RuleApp, Candidates) by at least one until it is 0. But when *place*(RuleApp, Candidates) is 0, then RuleApp and thus D must be Γ-redundant. It follows that conditions (ii) and (iii) hold.

The actual proof is more detailed and follows the pseudo-code much closer.

Proof. Let us first observe that the procedure constructs a derivation tree which is represented by the set OpenBranches. This is easy to see, as OpenBranches is only modified when a rule is applied to a branch in OpenBranches (lines 29, 31), and then replaced by the new open branches obtained by its conclusions (lines 29, 33). That is, OpenBranches is a representation of a derivation tree simplified by removing its closed branches.

Secondly, observe that if a rule instance becomes applicable to the leaf of some branch B, then in each extension of B the rule instance either becomes non-applicable after a finite number of steps, or the branch is closed. For, upon construction of a branch B all possible core rule applications are computed in line 32 and added to the previously computed set of rule applications. Furthermore, Candidates only grows but never shrinks. If *solve* terminates although Candidates still contains applicable rule instances, it must be in line 23 because

all branches have been closed. Otherwise, if it terminates it must be in line 25, because *select* fails to find an applicable rule instance in Candidates. Now, the only way for the procedure to not terminate is by creating an infinite branch in the limit tree through infinitely many applications of *solve*, with the recursive calls in lines 29 and 34. Recall that the set of open branches is modified only in lines 29 and 33. As any given branch has only finitely many nodes, and the call in line 29 removes exactly one open branch, any infinite sequence of calls must include executing line 33, and thus line 34. Furthermore, as line 34 extends a branch only by the finitely many conclusions of an inference rule application created in line 31, it follows that line 34 must be executed infinitely many times as well. Now, the rule applications in Candidates are applied in the order given by *select* in line 27, which prefers rule applications computed in shorter branches. As each component of a sequence is finite, and by construction of the derivation rules, there can be only finitely many instances of (core) rules that are applicable to a given sequence. It follows that only finitely many rule applications in Candidates are associated with branches of the same length. As B is of finite length, it follows that *select* will consider any rule instance added to Candidates that was added when B was created after finitely many executions of line 24. Thus, there is a $j \geq i$ such that the rule instance has been applied or is not applicable anymore. This of course holds for the finite open branches obtained in case of termination in line 25 as well.

Now, let us consider an arbitrary open branch $\mathbf{B} = (N_i)_{i<\kappa}$ of a limit tree constructed by the procedure. As observed above, as $\mathbf{B}$ is an open branch it must be in OpenBranches. Let us denote each initial segment of $\mathbf{B}$ of length i by $\mathbf{B}_i$

with the leaf sequent $S_i = \Lambda_i \cdot \Gamma_i \vdash \Phi_i \cdot \Psi_i$. Note that all components of a sequent can only grow along a branch, they never shrink, as the procedure makes only use of the core rules. Now let us check by case analysis that each of the conditions (i)-(iv) of Definition 3.6.4 holds, i.e., that $\mathbf{B}$ is an exhausted branch.

(i) If Close were applicable to some S_i, then it would be applicable to all S_j with $j \geq i$. Thus it would be applied to some $\mathbf{B_j}$ in line 28. But then $\mathbf{B_j}$ would be removed eventually from OpenBranches in line 29, and its extension $\mathbf{B}$ could not be in OpenBranches. Thus Close cannot apply to any S_i, and Condition (i) holds.

(ii) Whenever Inst is applicable to some S_i, then it is also applicable to any extension of S_i. Thus, Inst is applied eventually and the derived clause is part of some S_j, $j \geq i$, and Condition (ii) holds.

(iii) Assume that Split, Extend, or Domain Split applies to some S_i with selected clause D, and let $\alpha \in \mathit{Mods}(\Gamma_\mathbf{B})$. By definition of $\Gamma_\mathbf{B}$, $\alpha \in \mathit{Mods}(\Gamma_i)$ for all i.

Firstly, note that by (i) Close does not apply to any S_i with selected clause D. It follows that some literal L of D is not Γ_i-contradictory with Λ_i, and, since $\Gamma_\mathbf{B}$ is satisfiable due to compactness, also not $\Gamma_\mathbf{B}$-contradictory with $\Lambda_\mathbf{B}$. Furthermore, by construction of the core rules, each literal of D that is not $\Gamma_\mathbf{B}$-contradictory gives rise to an application of Split, Extend, or Domain Split. That is, if Split, Extend, or Domain Split is applicable to some literal L of D, and D is $\Gamma_\mathbf{B}$-contradictory in $\Lambda_\mathbf{B} \cdot \Gamma_\mathbf{B}$, then there must be other literals of D which are not $\Gamma_\mathbf{B}$-contradictory in $\Lambda_\mathbf{B} \cdot \Gamma_\mathbf{B}$.

Then L is not relevant for making D $\Gamma_{\mathbf{B}}$-redundant, the other literals are. Also, if Extend is applicable to S_i, then it is also applicable to all S_j with $j \geq i$. Furthermore, the application of Extend makes its selected clause R-Unproductive Γ-redundant in the conclusion. It follows that an reduction of the applicability of Split, Extend, or Domain Split to these two cases, that the literal is Γ_j-contradictory or Extend applies, suffices to guarantee that D is Γ_j-redundant in some S_j and that Condition (iii) holds.

Now consider the case when Domain Split applies to D with some literal L and constraint d. In the two cases when the Domain Split application becomes non-applicable and in the left conclusion of a Domain Split application, L is either Γ_j-contradictory or Extend is applicable to it. It remains the case of the right conclusion of the Domain Split application to some S_j with $j \geq i$. If L is not contradictory with the context of S_j, then Split applies. Otherwise Domain Split applies again, with a different constraint than d. Now notice that due to the finite range imposed on the parameters, Domain Split can be applied only finitely many times. It follows that for some $k \geq j$ it holds that $\Gamma_{k'} = \Gamma_{\mathbf{B}}$ for all $k' \geq k$. As Domain Split does eventually not apply anymore, applications of Domain Split reduce to L being Γ_j-contradictory, or Extend or Split being applicable.

Now consider the case when Split applies with selected literal L of D. If it is applied to some S_j, then D is R-Unproductive α-redundant in the branch obtained from its left conclusion, and Split is not applicable in its right conclusion. If it is not applied, then this can only be because Split is not applicable in some S_j. But if Split is not applicable, then either L or

its complement is contradictory with the context. That is, either L is Γ_j-contradictory, or Extend becomes applicable, or Domain Split becomes applicable.

In summary, as Close does not apply we can focus on literals which are not $\Gamma_{\mathbf{B}}$-contradictory in the context of $\mathbf{B}$ (and thus of any $\mathbf{B_i}$), Split in its left conclusion and Extend make D α-redundant, and as there can be no infinite sequence of Domain Splits and right Splits it follows that they reduce eventually to one of the above cases. It follows that Condition (iii) holds.

(iv) The procedure can modify a Γ_i only by application of Domain Split, which strengthens Γ_i in both conclusions, while keeping both strengthened versions satisfiable. Because there are only finitely many parameters and each parameter is bounded over a finite range, these strengthenings can occur only finitely many times. It follows that $\Gamma_{\mathbf{B}}$ is finite. Thus compactness follows trivially, and Condition (iv) holds.

In summary, it follows that on termination and in the limit each branch in OpenBranches is exhausted, and that the procedure is fair. □

As mentioned before, the ME(LIA) logic is not even semi-decidable, and thus a fair proof procedure cannot exist for all inputs. The following example provides an intuition of how the above procedure fails if parameter bounds are not imposed on the input language.

Example 3.7.2 Let Γ be empty and let Φ consist of the following clauses:

$$\neg P(x) \leftarrow x \geq 0 \wedge x = a \tag{1}$$

$$P(x) \leftarrow x = 0 \tag{2}$$

$$\neg P(x) \vee P(y) \leftarrow y = x + 1 \tag{3}$$

Clearly, Φ is unsatisfiable. Due to clauses (2) and (3), $P(m)$ must be true for all $m \geq 0$, but due to clause (1), $P(m)$ must be false for some $m \geq 0$.

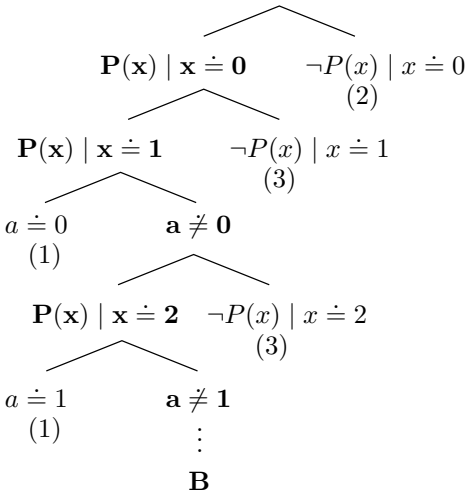


Figure 3.3: Unfairness of basic procedure for unrestricted ME(LIA) input

A possible derivation with the basic procedure is shown in Figure 3.3. It fails to produce a refutation tree, but instead builds an infinite open branch $\mathbf{B}$ in its limit tree. While $\Gamma_{\mathbf{B}}$ contains no models, Γ_S is satisfiable for each sequent S along $\mathbf{B}$. That is, $\mathbf{B}$ is not compact and thus not exhausted, and thus the procedure is not fair. □

The basic procedure has the advantage of being relatively straightforward

to prove fair. Its obvious downside is its inefficiency. It makes no use of redundancy criteria to avoid applying redundant rule instances, and it does not use any of the optional or simplification rules. While the procedure terminates only if each exhausted branch happens to be finite, it can easily be modified so that it stops as soon as one of the open branches is exhausted. It is still performing breadth-first search and has to manage the whole derivation tree at once, while an iterative deepening approach needs to consider only a single branch and naturally terminates on the first exhausted branch. Furthermore, the strictly chronological order of rule applications needs to be enhanced by a candidate selection heuristics. The proof procedure presented next will improve on these shortcomings.

3.7.3 A DPLL(LIA) Based Procedure

As mentioned before, ME is closely modeled on DPLL, lifting most notions from the ground to the first-order level. Similarly, DPLL(LIA) extends DPLL(T) to reason over LIA constraints, by tightly integrating a LIA decision procedure in the DPLL framework. One motivation for developing ME(LIA) was to explore how to integrate a LIA decision procedure in the same spirit in the ME setting. The proof procedure presented in this section tries to achieve exactly that.

We will focus on this aspect of the procedure, while intentionally leaving most details unspecified. We assume that the proof procedure performs a depth-first search. Apart from that, we keep the procedure quite abstract, and refer to Section 3.7.5 for a discussion of the remaining design decisions. Firstly, this greatly simplifies the presentation. Secondly, it makes specializations specific for a given input logic and problem structure possible, in order to obtain a more efficient and, if possible, fair procedure.

Thus, we will first give a short overview of the DPLL(LIA) architecture, followed by a discussion on how to mirror this in ME(LIA). We will then present the proof procedure as a system of cooperating agents.

3.7.3.1 DPLL(T)

The paper [47] provides a detailed presentation of the DPLL(T) architecture in the style of a transition system. A DPLL(T) solver creates a derivation in a depth-first fashion, where each branch corresponds to an incrementally built conjunction of propositional literals and literals of the theory T. The DPLL solver works on propositional literals, which includes theory literals abstracted to propositional literals, whereas the theory solver works on theory literals only. The DPLL solver drives the search, by performing splitting, unit propagation, learning, and backjumping as usual. The theory solver performs (theory) unit propagation and (theory) lemma-learning based on the theory literals on the branch. The purpose of a lemma can be, firstly, to encode that the current branch is T-unsatisfiable. Or, secondly, to serve as justification for a theory implication. Or, thirdly, to encode as a disjunction the set of alternatives which need to be explored before the theory solver can determine if the current branch is unsatisfiable, thus delegating search decisions to the DPLL solver.

The two solvers cooperate closely with each other. Both can extend the current derivation branch by unit propagation, and both can close it, thus reducing the search space aggressively. They need to cooperate on conflict lemma-learning, as a lemma can depend on propagations from the DPLL as well as the theory solver. Finally, the theory solver needs to know which literals are known to the DPLL solver, as this determines which literals should be propagated and

can be part of theory lemmas.

At the moment, the major drawback of the DPLL(T) approach, and its instance DPLL(LIA), is the limited support for quantifiers, which are basically treated by heuristic instantiation. ME(LIA) goes beyond that.

3.7.3.2 Reduction to DPLL(LIA)

ME and ME(LIA) are conceptually very similar, the exceptions being that ME supports functions and that its notions are fundamentally based on syntactic unification, while ME(LIA) supports LIA constraints and relies on constraint solving. As mentioned before, if substitutions are not applied eagerly in ME, but instead unification is represented as a constraint problem, the presentations of ME and ME(LIA) resemble each other even more. The crucial difference in practice is that for syntactic unification efficient algorithms with linear complexity as well as indexing techniques exist, while solving LIA constraints has doubly exponential complexity. In principle, the calculus is formulated abstractly enough to make it possible to make use of an existing efficient LIA solver, for example one based on the DPLL(LIA) architecture or one based on quantifier elimination (see Section 3.7.5.7). But as each operation of the calculus, such as finding applicable rule instances or checking for redundancy, relies on LIA solving, this fact makes it doubtful that treating each LIA problem in isolation is going to be effective in practice. Thus the proof procedure presented here tries to spread the cost of LIA solving among problems with a similar structure. This will also have the effect of replacing any application of Domain Split by a number of applications of Ground Split.

This is achieved by reducing each LIA problem to a ground LIA problem,

and by having the same kind of cooperation between the main solver and the LIA solver as in DPLL(LIA). Recall that all queries involve Γ and variants of constraints taken from constrained clauses and constrained literals. While constraint problems may contain quantifiers, they, and in particular all constraints in Γ, are closed.

We let Γ be managed by a DPLL(LIA)-style LIA solver, i.e., the solver accepts ground literals incrementally and can perform theory propagation and theory learning. Thus we get very efficient handling of Γ and of queries consisting of conjunctions of ground literals only. This implies that Γ can contain only ground LIA literals, and in fact we are going to reduce all queries to a satisfiability test of a conjunction of ground LIA literals. We reduce LIA constraints to ground formulas by quantifier elimination. Due to the high cost of quantifier elimination, we try to do this partially and incrementally, and might thus defer full quantifier elimination and solving of a particular constraint to a later time. Quantifier elimination eventually exposes some ground top level structure, i.e., conjunctions and disjunctions of ground literals. These literals give rise to Domain Split candidates, which constrain Γ such that constraints can be efficiently simplified further, possibly even to trivially true or false constraints. If Domain Splits which benefit the simplification of many LIA problems are preferred, this has the potential to, firstly, amortize the cost of quantifier elimination over many constraints, and, secondly, to make constraint solving relatively cheap, as it is reduced to checking the satisfiability of conjunctions of ground literals.

Let us take a closer look at how constraints are constructed. Firstly, we have context unifiers, which are relevant for redundancy checks and form the

basis for the constraints of all constrained literals. Secondly, we have queries based on context literal constraints. These require renaming and quantification of the free variables of the constraints, followed by putting them in a conjunction, implication, or imposing an order. Finally, the resulting closed constraint is checked for entailment or satisfiability against Γ. For example, determining if $c(\mathbf{x})$ and $d(\mathbf{y})$ are Γ-contradictory translates into checking the unsatisfiability of the constraint $\Gamma \wedge \forall \mathbf{z} \neg(\mathsf{perm}(c[\mathbf{z}]) \wedge \mathsf{perm}(d[\mathbf{z}]))$. Determining if $c(\mathbf{x})$ α-covers $d(\mathbf{y})$, which is part of the test for α-productivity and thus R-Unproductive, translates into checking the satisfiability of $\Gamma \wedge \bar{\exists}\ d$ and $\Gamma \wedge \forall \mathbf{z}\ (d[\mathbf{z}] \rightarrow c[\mathbf{z}])$.

While the queries involving universal quantifiers are especially expensive to check, and in fact a DPLL(LIA) solver might give up on such a query, their exists extensive structural sharing. Context unifiers are at the core constructed by conjunction of a clause constraint with context literal constraints. Furthermore, the constraint of a clause instance, i.e., a context unifier, and the constraints of the literals of the clause differ only in which variables are free, but are identical otherwise. If we can assume that performing quantifier elimination on constraints with shared structures results in constraints which are also similar in structure and ground LIA literals occurrences, the above described interaction between quantifier elimination and strengthening of Γ is an effective way of mitigating the cost of solving the LIA problems coming up in the calculus. This thesis has to be evaluated in practice, though.

A remaining problem is if the LIA solver should perform internal search or not. That is, the satisfiability of a conjunction of LIA literals can in general not be determined without search. This search can be performed internally by

the LIA solver, without impacting the rest of the system. Alternatively, the LIA solver can delegate the search decisions to the DPLL solver. Instead of performing a case split, the LIA solver creates a theory lemma, which encodes the case split as a disjunction. The DPLL solver is then responsible for satisfying the lemma, which amounts to trying out all cases.

We could use the same approach for ME(LIA), for example by generating Domain Splits as needed. Alternatively, we could use a full DPLL(LIA) solver, which can take care of the search internally. In the end, it is an empirical question if information exchange on this micro level is useful and worth the additional complexity. We will for simplicity assume that the LIA solver can take care of conjunctions of literals internally.

3.7.3.3 Architecture

We will present the proof procedure as a system of asynchronously cooperating modules. By organizing the conceptually different parts of the procedure in components, we make their responsibilities and dependencies explicit. Furthermore, by only imposing weak restrictions on the interactions between the components, instantiations of the procedure can use different communication protocols. Apart from making it possible to try out different strategies, this is a core ingredient of customizing the procedure for specific inputs, and in particular, for obtaining fair procedures, as explored in Section 3.7.4.

We organize the system in seven modules. As our architecture makes use of exactly one instance of each module, we might talk about, e.g., "the Eliminator", when strictly speaking we should be talking about "the instance of the module Eliminator". We first give a short overview of the modules:

- Builder: builds the derivation tree by rule application and backtracking, performs lemma-learning.
- Selector: gathers possible rule applications from other modules, selects one heuristically.
- Instantiator: manages Φ and lemmas, computes Inst, Inst Assert, Subsume, Resolve applications.
- Searcher: manages Ψ, computes Close, Split, Extend, Assert, Domain Split applications.
- Interpreter: manages Λ, performs context checks, computes Compact applications.
- Solver: manages Γ, performs constraint solving.
- Eliminator: performs quantifier elimination.

We make a few assumptions about the procedure:

- The system contains only one instance of each module. This suffices for a depth-first exploration, as only one sequent has to be represented at any time. We discuss extensions of the system to support more module instances in Section 3.7.3.4.
- In the initialization phase the Instantiator and the Interpreter are set up with the initial Φ and Γ. Then the Builder is run to build the complete derivation. That is, the proof procedure is not interactive and cannot be guided or queried during the derivation process. Extensions for supporting interactive usage are discussed in Section 3.7.3.4.

- Each module except for the Builder supports a push / pop mechanism, which is globally controlled by the Builder. This suffices to implement backtracking for a depth-first procedure, including advanced versions (see Section 3.7.5.5).

- Each module responsible for computing inference rule applications is exhaustive in the sense that it will consider each possible inference rule application eventually. Furthermore, the responsibility for obtaining a fair procedure lies with the Selector. This is achieved by a mixture of push and pull mechanisms. Interpreter and Solver provide a callback mechanism which is used by Instantiator and Searcher to be notified about changes to Λ resp. Γ, which trigger the computation of new inference rule applications. The Selector has to pull application candidates from Instantiator and Searcher, so that it can select among these. These mechanisms will become clearer when we introduce the module interfaces below.

We now give a more detailed description of each module. We describe each module interface in a functional style, and discuss the module's responsibilities. We assume that the types $Constraint, Clause, Literal, Candidate, Context,$, and $Proof$ are predefined. $Unit$, $Option$, $List$ stand for the usual types and type constructors, | stands for a variant type, and a callback function is represented as a lambda function. For each interface function we first give its **name** and signature, possibly including notes, preconditions, and information about the return value. We omit the push / pop functions, as these are common to all modules except for Builder.

Figure 3.4 provides a high level overview of the dependencies between the

modules.

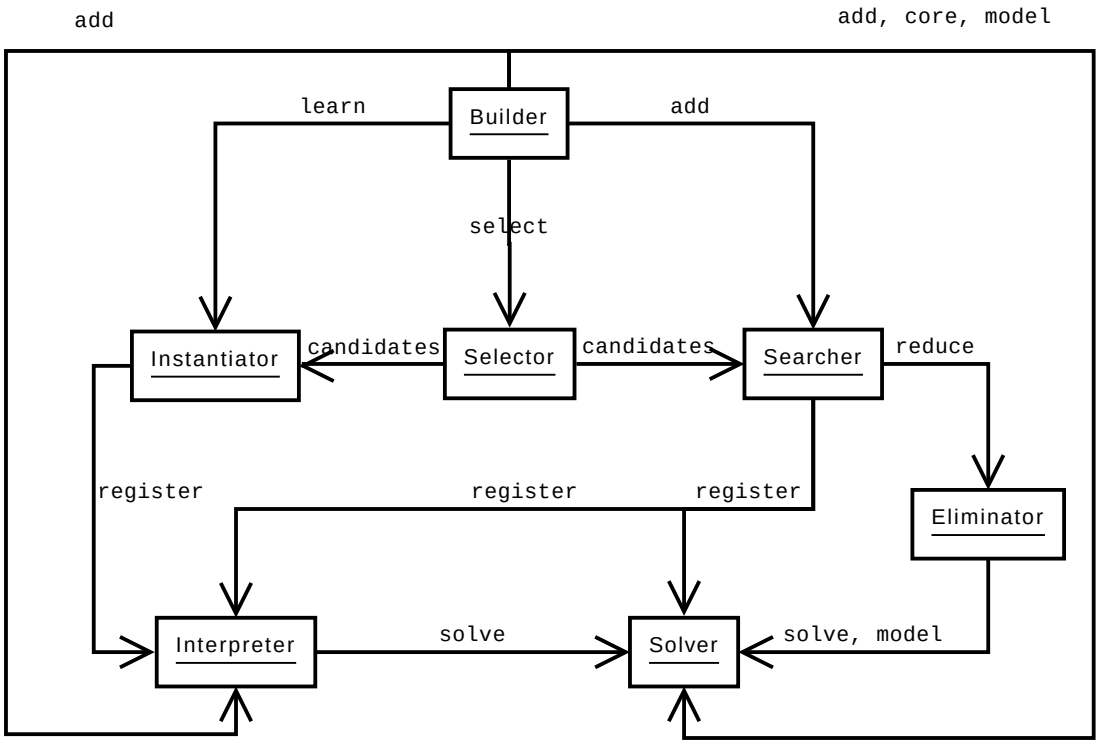


Figure 3.4: DPLL(LIA)-style ME(LIA) proof procedure

Builder	builds the derivation tree, performs lemma-learning.
derive	$Unit \rightarrow Proof \mid Context$
returns	a proof of unsatisfiability, or a context if a model has been found.

Table 3.1: Interface: Builder

The Builder (Table 3.1) is run with **derive**, and returns either a proof or the limit context of a (finite) exhausted branch. In the later case it is possible to refine Γ to a single model by using Solver and Interpreter, and to check for any ground literal if it is satisfied (produced) in it. The Builder asks the Selector for

the next inference rule to apply. It terminates when Selector **select** returns None, or if a refutation has been built.

The Builder manages and builds the derivation tree by applying inference rules, learning lemmas, and performing backtracking. Depending on the inference rule applied, either push and one of Solver **add**, Interpreter **add**, Searcher **add** is called, or a pop is initiated in the case of backtracking.

Lemmas are added with Instantiator **learn**. While lemma-learning is an optional feature and the procedure could be simplified by removing it, we have decided to include it in the architecture, as we expect it to be crucial for efficiency.

The Builder is parametric in the backtracking (Section 3.7.5.5) and learning (Section 3.7.5.6) mechanism. In order to be able to perform advanced versions of backtracking it might need additional information in the form of unsatisfiable cores from the Solver.

Selector	gathers possible rule applications, selects best one heuristically.
select	*Unit* → *Candidate Option*
returns	A candidate returned by Instantiator, or Searcher which was not returned before, or None if no such candidate exists.

Table 3.2: Interface: Selector

The Selector (Table 3.2) is responsible for gathering all possible rule applications with Instantiator **candidates** and Searcher **candidates**. It picks one heuristically when **select** is called. It may only return None if both **candidates** functions return None, and if it has already returned all applicable non-redundant candidates gathered before. In order to obtain fairness, each candidate should be

selected eventually.

Selector is parametric in the selection heuristics (Section 3.7.5.3), as well as in the redundancy criteria applied to returned candidates.

Instantiator	manages Φ, Φ consist of input clauses Φ_i and lemmas Φ_l, computes Inst, Inst Assert, Subsume, Resolve, applications.
learn	$c : Clause \rightarrow Unit$
requires	c is entailed by Φ_i
note	adds c to Φ_l
candidates	$Unit \rightarrow Candidate\ List$
returns	at least one not previously returned rule application candidate, an empty list if none exists.

Table 3.3: Interface: Instantiator

The Instantiator (Table 3.3) is responsible for managing Φ. It also manages lemmas added with **learn**, which may be forgotten heuristically to avoid being flooded with lemmas. It registers callbacks with the Interpreter, so that it can efficiently compute all possible applications of Inst and Inst Assert to input clauses and lemmas. Rule application candidates are returned with **candidates**. Subsume and Resolve are applied internally to simplify Φ. Exporting them with **candidates** would require to extend the interface such that the Builder could upon application feed the simplification information back to the Instantiator. This could potentially be beneficial, if the clause instance in Ψ managed by the Searcher would then be simplified as well.

The Instantiator is parametric in how and when candidates are computed, as well as in the lemma forgetting heuristics.

The Searcher (Table 3.4) is responsible for managing Ψ. It registers call-

Searcher	manages Ψ, computes Close, Split, Extend, Assert, Domain Split applications.
add	$c : Clause \rightarrow ls : Literal\ List \rightarrow Unit$
requires	Inst is applicable to c and ls with derived clause c'.
desc	adds c' to Ψ.
candidates	$Unit \rightarrow Candidate\ List$
returns	at least one not previously returned rule application candidate, an empty list if none exists.

Table 3.4: Interface: Searcher

backs with the Interpreter and the Solver, so that it can efficiently compute all possible applications of inference rules to clause instances. It computes all rule application candidates of (non-redundant) clauses added with **add** to Ψ, and returns them with **candidates**.

The Searcher searches for the best way to make a clause instance redundant. It has to use the Interpreter for checks of clause literals against the context. Due to quantifier elimination, a request to Interpreter might make a Domain Split necessary. Similarly, checking for redundancy, e.g. if a clause is R-Unsatisfiable redundant, might require to use the Eliminator and perform Domain Splits on the returned simplified constraint. In essence, the Searcher is in in both cases blocked on that action and has to wait till the Builder applies the Domain Split, before that particular action can be continued.

The Searcher is parametric in how and when candidates are computed, how quantifier elimination and Domain Splits are interleaved with the candidate computation, and which redundancy criteria are applied.

The Interpreter (Table 3.5) is responsible for Λ, and all checks on the context. Λ is initialized with $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$ for all predicate symbols P. Interpreter is responsible to notify all registered modules of additions to Λ via

Interpreter	manages Λ, performs context checks, computes Compact applications.
add	$l : Literal \to Unit$
requires	the constraint of l is quantifier-free.
note	adds a literal to Λ.
check_contr	$l : Literal \to Literal \mid$ DomainSplit $\mid None$
returns	a Λ literal contradictory with l, if there is any, None, if there is none, Domain Split if Γ-strengthening needed to determine this.
check_extended	$l : Literal \to Literal \mid$ DomainSplit $\mid None$
returns	like **check_contr**, but for a literal extending l.
check_produces	$l : Literal \to k : Literal \to Literal \mid$ DomainSplit $\mid None$
requires	l covers k
returns	like **check_contr**, but looks for a literal blocking l from producing k.
is_produced	$l : Literal \to Bool$
returns	true iff l is Γ-produced in Λ.
register	$(Literal \to Unit) \to Unit$
note	registers a callback function, activated whenever **add** is called

Table 3.5: Interface: Interpreter

callback. **check_contr**, **check_extended**, and **check_produces** exist for the α and Γ versions. If a check depends on a constraint that does not simplify to true or false after eliminating a single quantifier, then a Domain Split that helps to simplify the constraint is returned. Compact is applied internally. **is_produced** allows to check if a ground literal is true in a model.

The Interpreter is parametric in how it checks a query against all context literals.

The Solver (Table 3.6) is responsible for Γ. It accepts only ground literals for addition with **add**, and notifies all registered modules via callback of additions to Γ. An unsatisfiable core is a (small) unsatisfiable subset of Γ, a satisfiable core is a (small) subset of Γ that determines a model for all constants.

The Eliminator (Table 3.7) is responsible for quantifier elimination. It

Solver	contains and performs constraint solving with Γ.
add	$c : Constraint \rightarrow Unit$
requires	c is a ground literal.
solve	$Unit \rightarrow Status$
returns	status of Γ: unsatisfiable or satisfiable.
core	$Unit \rightarrow Literal\ List$
returns	unsatisfiable or satisfiable core depending on status of Γ.
model	$Unit \rightarrow Constraint$
requires	**solve** returns satisfiable.
returns	a model of Γ as equalities between constants and integers.
register	$(Constraint \rightarrow Unit) \rightarrow Unit$
note	registers a callback function, activated whenever **add** is called.

Table 3.6: Interface: Solver

Eliminator	performs quantifier elimination
reduce	$c : Constraint \rightarrow Constraint$
requires	c is a closed constraint.
returns	c itself if it is ground, a constraint equivalent to c with (at least) one less quantifier otherwise.

Table 3.7: Interface: Eliminator

reduces non-ground closed constraints by removing at least one quantifier. Γ should be used to simplify reduced forms.

The Eliminator might cache constraints and their simplified forms, so that it can avoid repeated applications of the expensive quantifier elimination algorithm if possible.

In case that the Monniaux based quantifier elimination algorithm is used (see Section 3.7.5.7), the Eliminator needs a LIA solver which can solve arbitrary ground constraints. Then the Eliminator might maintain an internal DPLL(LIA) solver which is kept in sync with Γ, i.e., the observable state of the Solver. Alternatively, the Solver itself might be a DPLL(LIA) solver, in which case the

Eliminator can use it directly. But as the Monniaux algorithm requires a series of queries and changing the state of the LIA solver, this also requires that the Eliminator has exclusive access to the Solver during the whole process of eliminating a quantifier.

3.7.3.4 Extensions

We are now going to explore a few extensions of the above architecture.

Parallelism Having set up the architecture as a system of asynchronously communicating modules, it is natural to wonder if this can be exploited to obtain speed ups in a parallel setting. This is indeed possible, and a straightforward way to parallelize the search is by having instances of Instantiator and Searcher for each clause, not only for the whole clause sets Φ and Ψ. When the context is extended, each instance would then be responsible for the rule applications for one clause only. The Builder and the Selector have to be modified only slightly, to be able to manage and gather candidates from more than only one instance of each of the other modules. As each Searcher instance potentially needs to use an Interpreter, an Eliminator, and a Solver, it makes sense to have several instances of these modules as well, in order to avoid that accessing them becomes the bottleneck of the system. Of course, keeping them synchronized adds new overhead.

Interactive Usage The interface is geared towards a one run use. That is, setting up the Solver and the Instantiator according to the initial Γ and Φ, running the Builder, and checking the result. For a more interactive use of the system the

modules would need to export more functionality. In particular, it would need to be possible to control the Builder and the Selector. This includes running the Builder inference step by inference step, controlling backjumping and global push / pop synchronization, and injecting Split and Domain Split decisions into the Selector queue, and overriding decisions of the Selector. Furthermore, the Instantiator would need to allow for a client to add and remove clauses at any time. This would complicate lemma management, as after removing a clause from Φ_i a lemma might not be entailed by Φ any longer.

In short, the system must be usable as a (decision) procedure for ME(LIA) problems as a component of a bigger system, just like it itself uses a LIA solver as a component.

Heuristics As mentioned in Section 2.2, it is common in DPLL systems to base the selection heuristic in part on the analysis of a Close application. Similarly, the lemma forgetting process is guided by the relevance of a lemma for recent branch closures. As the Builder obtains this information during backtracking and lemma-learning, it would make sense to share it with the Instantiator and the Selector.

3.7.4 Some Fair Proof Procedures And Decision Procedures

In this section we are going to instantiate the proof procedure presented in the previous chapter for a number of special cases of the ME(LIA) language.

3.7.4.1 ME(LIA) With Bounded Parameters

For applications like bounded model checking and planning a logic which makes it possible to efficiently reason over finite domains is appropriate. The specialization of the ME(LIA) logic where all parameters range only over finite domains gives that. Encoding functions with finite ranges is immediate. In principle, parameters with finite domains can be eliminated by an exhaustive case analysis over all the possible values. In practice, however, doing that can be prohibitively expensive, depending on the size of the parameters' domains. In contrast, ME(LIA) refines the possible ranges of each parameter by splitting on Γ, which is much more coarse grained and driven by the current interpretation. See Section 3.3 for examples for this input language.

We have already given a fair procedure for this case with the basic proof procedure in Section 3.7.2. Furthermore, the DPLL(LIA) based procedure introduced above instantiates straightforwardly to a fair procedure. We only need to require that each applicable and non-redundant core inference rule is eventually made available to the Selector by the **candidates** function of the Instantiator and the Searcher, and that the Selector eventually returns it via **select** to the Builder. With the core of the argument given for the fairness of the basic procedure, i.e., that Γ can be strengthened only finitely many times, it is easy to see that the procedure is fair.

3.7.4.2 A Decision Procedure For Function-free First-order Clause Logic

As mentioned in Chapter 2, ME is a decision procedure for first-order clause logic without (non-constant) function symbols. Considering that this class is in the language of both ME and ME(LIA), it is natural to wonder if ME(LIA)

is a decision procedure as well. As it turns out, a decision procedure is easily obtained, and the approach used for finite model finding in Section 2.5 provides valuable insights. In fact, ME(LIA) is a decision procedure even for the extension of this language with equality.

For a given problem, we first convert it to constrained clauses and then extend Γ such that each of the n constants in the problem is restricted to range from 1 to n. We also impose the same range bound on each variable occurring in a clause constraint, i.e., the free variables shared with the clause literals as well as the local quantified variables. Then we can apply the fair procedure described above in Section 3.7.4.1, and immediately get a complete procedure, and, in fact, a decision procedure.

Intuitively, this is clear. While ME(LIA) works with infinite interpretations, each finite model of a problem is isomorphic to the substructure obtained from some ME(LIA) model by restricting its domain to the range from 1 to n. Thus the procedure is sound. Since all variables are bounded, there is only a finite number of different least solutions of all constraints. As an admissible context is not contradictory and Γ cannot be strengthened infinitely often, at some point none of Split, Ground Split, Assert, Domain Split can be applied anymore (or is redundant). Similarly, Extend can be applied only finitely many times, as due to the finite range imposed on each variable there can be only finitely many ways that any literal can be Γ-extended. Note that while Compact removes literals from a context, any application of Split or Extend which might potentially add the literal again must be R-Unproductive redundant. Thus, only a finite number of literals is added to a context, which implies that the number of applications of

Inst, Inst Assert, Subsume, Resolve, and Compact is finite as well. It follows that the procedure terminates for any input.

3.7.4.3 A Decision Procedure For Ground Clauses

As the last example, we consider the input for which DPLL(LIA) procedures are most effective, ground first-order formulas with LIA constraints. An example clause of this language, before normalization, is $P(a+2) \vee Q(b-a) \vee a \dot{\leq} b$. If we allow no function symbols except for constants, it is straightforward to obtain a decision procedure for ME(LIA).

We will prove that any derivation can add only finitely many literals to the context. It follows immediately that Split, Extend, Assert, and Domain Split, and thus also Inst and Inst Assert, can be applied only finitely many times. Any reasonable procedure using Domain Split, including any implementation of 3.7.3, will not resort to infinitely many application of Domain Split where finitely many applications of Ground Split suffice. Thus we can conclude that such a procedure terminates, and we have a decision procedure for ground formulas.

We now prove that any derivation can consider only finitely many constrained literals for addition to the context, modulo constraint equivalence. The normalization of ground clauses to constrained clauses introduces variables, but only in such a way that in each constrained clause each variable is equated to a ground LIA constraint. That is, each input clause $C \leftarrow c$ is of the form $L_1(\mathbf{x_1}) \vee \cdots \vee L_k(\mathbf{x_k}) \leftarrow f \wedge \mathbf{x_1} = \mathbf{f_1} \wedge \cdots \wedge \mathbf{x_n} = \mathbf{f_n}$, where all $\mathbf{x_i}$ are pairwise disjoint, f is ground, and each $\mathbf{f_i}$ is a tuple of ground constraints. It follows that for each parameter assignment α each clause constraint is either α-unsatisfiable or has a unique α-solution.

We proceed by induction over the sequents of a derivation, with the hypothesis that for any sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ of a derivation, for each context unifier d of a clause in Φ against $\Lambda \cdot \Gamma$, each d_i is equivalent to a constraint of the form $\bigwedge_{C \leftarrow c \in \Phi'} \pi\ \mathbf{x_i}\ c$, where Φ' is a non-empty subset of Φ. That is, $d_i = \pi\ \mathbf{x_i}\ d$ is equivalent to the conjunction of the top level conjuncts $f_C \wedge \mathbf{x_i} = \mathbf{f_{C_i}}$ of some input clauses $C \leftarrow f_C \wedge \mathbf{x_1} = \mathbf{f_{C_1}} \wedge \cdots \wedge \mathbf{x_n} = \mathbf{f_{C_n}}$. Thus, for each parameter assignment α, either d_i is α-unsatisfiable, or d_i and all $f_C \wedge \mathbf{x_1} = \mathbf{f_{C_1}}$ have the same unique α-solution. We assume that Φ remains fixed during the derivation, as simplifications achieved by applications of Subsume and Resolve require only minor changes to our argument.

For the base case we have to consider the initial sequent of a derivation, $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$. Here, the only literals in Λ are the default literals of the form $\overline{P}(\mathbf{x_i}) \mid -\mathbf{1} \dot{\leq} \mathbf{x_i}$. It is immediate that each context unifier d of a clause $C \leftarrow c$ is equivalent to c. Thus each d_i is identical to $c_i = \pi\ \mathbf{x_i}\ c$, and the claim follows.

For the inductive case consider any sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ of a derivation. Let d be an arbitrary context unifier of a clause $L_1(\mathbf{x_1}) \vee \cdots \vee L_k(\mathbf{x_k}) \leftarrow c$ in Φ against context literals $\overline{L_i} \mid e_i$ from Λ. Each context literal is either one of the default literals, or it has been added with Split, Extend, or Assert. In the latter case, by hypothesis, its constraint is equivalent to a constraint of the form $\bigwedge_{C \leftarrow c \in \Phi'} \pi\ \mathbf{x_i}\ c$, where Φ' is a non-empty subset of Φ. From the above it follows that d has at most one solution for each $\alpha \in \mathit{Mods}(\Gamma)$, and by Definition 3.4.15 it follows then that d is equivalent to $c \wedge e_1 \wedge \cdots \wedge e_n$. As c is of the shape $f \wedge \mathbf{x_1} = \mathbf{f_1} \wedge \cdots \wedge \mathbf{x_n} = \mathbf{f_n}$, and by the shape of each e_i, the claim follows.

As the input clause set is finite, there can then be only finitely many

constrained literals with constraints that are not equivalent. Furthermore, no two literals $L \mid e$ and $\overline{L} \mid f$ with equivalent constraints e and f can be added to an admissible context, and only one of two literals $L \mid e$ and $L \mid f$ with equivalent constraints e and f need to be added to a context, as justified by redundancy. It is now immediate that Split, Extend, and Assert, and thus also Inst, Inst Assert, and Ground Split, are applicable only finitely many times (without being redundant), as was claimed above.

3.7.5 Improvements

This section describes a number of ingredients and improvements which are likely to be important for any efficient implementation of ME(LIA). We will first discuss simple adaptations of techniques which proved effective in Darwin [12]. In the remaining sections we will describe more complex methods in more detail.

An obvious technique from Darwin on the data structure level that apply with none or small modifications, is to use offsets to make the creation of variants of literals and constraints cheap. Another one is use a database to memoize constraints. This is especially efficient if constraints are aggressively normalized and simplified.

3.7.5.1 Universal Literals

Universal context literals are preferable to non-universal ones, as they in general constrain further modifications of the context significantly more, which leads to shorter derivations. Furthermore, universal literals often require significantly less complicated constraints in order to check basic concepts like α-

contradictory or α-productivity, which are crucial for checking the applicability and redundancy of most inference rule applications. Thus it would be desirable to recognize and mark literals as universal whenever possible. That is, each application of Inst Assert should be detected, which in turn always triggers an application of Assert (or Close). As this is expensive to detect in general, a practical implementation has to rely on heuristics and approximations.

First, we note that if all but one literal of a clause can be paired with universal context literals, it is immediate that Inst Assert applies. Secondly, we note that for a literal whose constraint has at most one α-solution for each $\alpha \in \mathit{Mods}(\Gamma)$ it makes no difference if it is universal or not, as its permanent constraint is the same in both cases. This is easy to detect for some classes of literals. For example, literals whose non-constrained part is propositional have the empty tuple as the only solution for all α. Also, literals whose constraints equate all free variables with constants, parameters, or ground terms, like $L(x_1, x_2) \mid x_1 \doteq 2 \wedge x_2 \doteq a$, have at most one solution for each α. Thus, these types of context literals can be treated as universal, even if they were not added by Assert.

Now, a simple way of cheaply detecting universal literals is to perform basic unit propagation on universal literals. For the base case, Inst Assert trivially applies to a unit clause. For the more general case, if all but one literal of a clause can be paired with universal context literals, then it is immediate from the above that Inst Assert applies. Thus, checks for Γ-universality in Inst Assert and Assert are reduced to checking if the paired context literals are marked as universal.

A further optimization which can enable more applications of Inst Assert resp. Assert is clause splitting, a common optimization in theorem provers. If a

clause consists of several variable disjoint literal sets, then it is split into several clauses which are connected by fresh propositional literals. For example, the clause (1) $P(x) \vee Q(y) \leftarrow x \dot{>} 0 \wedge y \dot{>} 0$ can be split into the two clauses (1a) $P(x) \vee \neg A(z) \leftarrow x \dot{>} 0 \wedge z \doteq 0$ and (1b) $A(z) \vee Q(y) \leftarrow y \dot{>} 0 \wedge z \doteq 0$, which are connected by the fresh predicate symbol A. Originally, we would have to split on a non-universal literal in clause (1), e.g. $P(x) \mid x \dot{>} 0$. But now it is possible to first split on $A(z) \mid z \doteq 0$ from clause (1b). As explained above, $A(z) \mid z \doteq 0$ can be considered to be universal, which makes the assertion of the universal literal $P(x) \mid x \dot{>} 0$ from clause (1a) possible in the left-hand conclusion, and the assertion of the universal literal $Q(y) \mid y \dot{>} 0$ from clause (1b) in the right-hand conclusion. For the rest of the derivation the contexts are now far more constrained than they would have been without splitting.

3.7.5.2 Context Unifier Computation

Note that the definition of context unifier gives rise to a family of context unifiers parametric in j, where j denotes the j-th least solution of the conjunction c of the clause and context literal constraints. While there can be only finitely many least solutions due to the admissibility of the involved constraints, j is not bounded in general. But as the $(i+1)$-th context unifier constraint is stronger than the i-th constraint, it follows that it suffices to enumerate the context unifiers for increasing j. When a context unifier is unsatisfiable in Γ, for some i, then the context unifiers for $1 \ldots i-1$ are all that exist for this family.

There might be a significant number of context unifiers, all of which give rise to an application of Inst, which in turn gives rise to rule applications on the derived clause instance. Thus computing all context unifiers eagerly might

be expensive, especially if only a few are actually needed to close a derivation branch. As with a stronger Γ the number of least solutions of c can shrink, it might thus pay off to delay the enumeration of context unifiers based on heuristics and fairness criteria.

3.7.5.3 Heuristics

The ME(LIA) calculus is highly non-deterministic. With only minor restrictions to achieve fairness, any order of rule applications ensures correctness. This property makes it possible for a proof procedure to make well thought-out design choices, about when and how to introduce determinism. It implies that the decision heuristics, which selects among all known applicable rule instances the one which is used to extend the current derivation, is a crucial part of any efficient proof procedure. While the effectiveness of a heuristics is highly dependent on the calculus, the application domain, other design choices, and even implementation details, some guidelines can be formulated for ME(LIA) based on experience with implementations of other calculi, especially Darwin.

Initial Interpretation A first observation is that ME(LIA) provides some flexibility in the initial interpretation induced by a context. Although an admissible context is defined to contain $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$ for each free predicate symbol, the complement $P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$ works just as well. This makes it possible to decide independently for each predicate symbol if all its instances are by default interpreted as true or as false, for example based on the polarity of its occurrences in the input clause set.

Note that the unit propagation scheme described above in Section 3.7.5.1

does not depend on the default interpretation, as Inst Assert makes use of universal literals only. Thus it is possible to defer committing to an initial interpretation until after then end of an initial unit propagation and simplification phase. In fact, like in ME, a Horn problem can be refuted with the rules Inst Assert, Assert, and Close alone, without any need for an initial interpretation of the free predicate symbols.

Derivation Rules Some derivation rules are clearly preferable to others. Close is obviously the rule with the highest priority. If a derivation branch can be closed, then applying any other rule is just wasted work.

The simplification rules are in principle also high on the priority list. But they are only worth applying in practice if the savings offset the work needed to compute and apply the rules. The same argument applies for checking if a rule application is redundant, as in principle no redundant rule should be applied. For example, in Darwin it turned out that redundancy and Resolve are often very beneficial, while Subsume and Compact have basically no effect.

Assert and Inst Assert are next in the priority ordering, followed by Extend. The splitting rules need to make a case distinction, introducing all the complexity of search, learning and backtracking. A literal split actively modifies a context in order to satisfy a clause, while a domain split enables applications of other rules. Thus Split might be preferable to Domain Split and Ground Split, but some interleaving seems to be necessary. Finally, Ground Split should be preferred over Domain Split, as it introduces only ground constraints, which should be cheaper to process.

Clause Literals Each of the non-Γ-contradictory literals of a clause instance gives rise to a rule application, with the goal of making the clause redundant. If Extend is applicable to a literal of a clause, then it can always be applied eventually. Thus, all other clause literals can be completely ignored by the selection heuristics. If Assert is applicable to a literal, then it can either be applied eventually, or the literal becomes Γ-contradictory. The problematic cases are again the splits.

Thus an important measure of a clause instance are the number of (different) literals which are not Γ-contradictory and to which neither Extend nor Assert are applicable. In the worst case, each of these *remainder* literals gives rise to one split on the same branch. Thus, when a heuristic has to select from a number of split applications, it should prefer splits on literals from clauses with fewer remainder literals. We remark that in contrast to ME it is not possible in ME(LIA) to focus on one remainder literal only and to ignore the others [12].

A case worth pointing out is when all literals but one of a clause is closing, and Split or Domain Split applies to this literal. If Split applies, then in the left conclusion the clause is redundant, and in the right conclusion Close applies to the clause. If Domain Split applies, then in the left conclusion the clause is closing, and in the right conclusion Split is applicable. That is, in this case no search is needed, as three of the four branches created by the two splits can be closed immediately.

Preference should also be given to candidate Split literals that occur in many clauses.

Generation The *generation* of an input clause is 0. The *generation* of a derived clause of an application of Inst and Inst Assert is the maximum of the generations of the context literals paired with the context unifier. The *generation* of a clause literal is the same as the generation of its clause. Choosing literals with a small generation from time to time helps to avoid getting lost deep in the search space and focusing on a small subset of the input clauses only.

Constraints It is difficult to compare constraints in a meaningful way, even if they have been simplified extensively. Ground constraints are highly preferable, as they are cheap to process and context literals with ground constraints can be treated as universal (see Section 3.7.5.1). Quantifier alternations are the worst that can happen, they are the core reason for the exponential complexity of the LIA theory. It is not clear if strong or weak constraints are preferable, as context literals with strong constraints are useful for making a specific clause redundant, while context literals with more general constraints impact more clauses.

Conflicts A crucial part of the selection heuristics of DPLL solvers is to give preference to clauses and literals which were responsible for closing branches [66]. For this, each clause and literal is assigned a score. When a branch is closed and a lemma is learned, then all clauses and literals regressed are considered to be relevant and their score is increased. Periodically, all scores are decreased, in order to ensure that recent conflicts carry more weight than less recent conflicts. This provides some locality to the heuristic, as in a depth-first traversal more recent conflicts are more likely to be effective in the current context of the derivation tree.

Combining Heuristics While some of the above guide lines, like delaying the application of splitting rules, are probably effective for any input, most will be effective in one scenario but not in others, giving rise to numerous competing decision heuristics.

One approach to tackle this problem is to use machine learning to learn heuristics for specific classes of input problems [62]. An alternative is to alternate between different heuristics, a common scheme in theorem provers [55]. In this context randomization can be a valuable heuristics, although it has the serious drawback that the proof procedure is not deterministic. A solver might not be capable of reproducing a proof, unless it logs the randomized decisions. Finally, using the generation of clause instances as a heuristics can be the essential ingredient for ensuring fairness.

3.7.5.4 Proof-Generation

An important feature of a theorem prover is proof-generation. Apart from the direct benefit of having a proof, e.g., of a theorem, this makes it possible to independently verify the correctness of the result computed by the prover. Considering the complexity of an efficient implementation of a proof procedure, this greatly increases the confidence in a theorem prover. Combined with the model generation capabilities of the ME(LIA) calculus, this gives an implementation the opportunity to gain high confidence of correctness through the use of external model and proof checking tools, which can be very simple pieces of software. We explain below how a resolution proof can be extracted in a straightforward way from a ME(LIA) refutation tree. It follows the same principles as used in other DPLL based procedures, and can be easily simplified and adapted for the

ME calculus.

Overview We will first present a version of the proof-generation algorithm Regress that for simplicity does not compute a resolution proof (from a refutation) of Φ and Γ, but only for Φ and one $\alpha \in \mathit{Mods}(\Gamma)$. Regress performs a recursive bottom-up labeling of a refutation tree with lemmas. It labels nodes to which Close was applied with (an instance of) the selected clause of the Close application, and computes lemmas for inner clauses by resolution based on the lemma labels of their children nodes. The procedure terminates with a constrained empty clause $\square \leftarrow c$ in the root node such that c is α-satisfiable. That is, we have derived a constrained empty clause for α.

We will then show how to extend Regress to the procedure Regress$*$, which computes a proof for Φ and Γ. In essence, the refutation proof gives rise to a finite partition of the models of Γ. Regress$*$ computes a constrained empty clause $\square \leftarrow c_i$ for one α_i of each partition Γ_i. By $\Gamma_i \models_{\mathcal{Z}} c_i$, Γ entails the conjunction of the constraints of all constrained empty clauses. Thus, we can derive the empty clause by resolution modulo the theory LIA.

Before introducing the regression algorithm, we will first define the structure of a label, along with an invariant. We will then describe the algorithm, and prove that the labels computed by it satisfy the invariant. From this we will conclude that the algorithm constructs a resolution proof from a refutation tree.

Labels A label is simply a lemma along with a mapping from each literal of the lemma to a literal of the context. The mapping will guide the resolution steps.

Definition 3.7.3 (Label) A *(regression) label* is a tuple (Lemma, Link), where

Lemma is a clause and Link maps each literal of Lemma to a literal. The *merge of two such literal to literal maps* Link_1 *and* Link_2 is the map Link, where for a literal L, $\mathsf{Link}(L)$ is defined as

- $\mathsf{Link}_1(L)$ or $\mathsf{Link}_2(L)$, if L is in the domain of both Link_1 and Link_2,
- $\mathsf{Link}_1(L)$, if L is in the domain of Link_1 only,
- $\mathsf{Link}_2(L)$, if L is in the domain of Link_2 only,
- undefined otherwise.

□

That is, when we merge the maps of two labels and their domains overlap, we resolve this conflict arbitrarily.

The following invariant on labels will be crucial in showing that the algorithm computes a proof. It states that Lemma is a logical consequence of Φ, the constraint of Lemma is satisfiable, and each literal L of Lemma is α-contradictory with a literal K from the Λ of the labeled node, such that Link maps L to K.

Definition 3.7.4 (Label Invariant) Let T be a refutation tree with root sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$. Fix an $\alpha \in \Gamma$. Let $(C \leftarrow c, Link)$ be the label of node N with sequent $S = \Lambda_S \cdot \Gamma_S \vdash \Phi_S \cdot \Psi_S$. The *$\alpha$-invariant* of the label is that either c is not α-satisfiable or all of the following hold:

(i) $\Phi \models_{\mathcal{Z}} C \leftarrow c$,

(ii) $\alpha \models_{\mathcal{Z}} \bar{\forall}(c(\mathbf{x}) \rightarrow \mathbf{0} \dot{\leq} \mathbf{x})$, and

(iii) For each literal $L_i \mid c_i$ of $C \leftarrow c$, the literal $\mathsf{Link}(L_i \mid c_i)$ is in Λ_S and α-contradictory with $L_i \mid c_i$.

A label is α-invariant if it satisfies the α-invariant. A label is Γ-invariant for some Γ, if it is α-invariant for all α in Γ. □

Regress We now present the algorithm Regress, and show that it computes a resolution proof from a refutation tree.

Algorithm 3.7.5 (Regress) The algorithm Regress takes as input a refutation tree T, and performs a case analysis based on the rule instance applied to its root note N with sequent S. Let in the following in each case the (one or two) children nodes of N be denoted by $N_1, \ldots, N_n$ with sequents $S_1, \ldots, S_n$. Assume w.l.o.g. that all clauses computed by Regress or taken from a component of a sequent, i.e., Φ or Ψ, are variable disjoint. Fix an α in $Mods(\Gamma_S)$. We assume that each label computed for a child node is α-invariant, as will be justified by Lemma 3.7.6. If $\alpha \notin Mods(\Gamma_S)$, then in each case Regress returns $(\Box \leftarrow \bot, Link)$, where Link is an empty map. Thus we will consider below only the cases where $\alpha \in Mods(\Gamma_S)$. If the rule application is an instance of

(i) one of the rules Inst, Inst Assert, Subsume, Resolve, or Compact, then it returns $\mathsf{Regress}(N_1)$.

(ii) the rule Extend, with literal $L_i \mid c_i$ of selected clause $C \leftarrow c$, then there must be a literal $L_i \mid c_i'$ in Λ_S which Γ_S-extends $L_i \mid \mathsf{perm}(L_i \mid c_i)$. This follows as $L_i \mid c_i$ is non-universal and $\overline{L_i} \mid c_i$ is Γ_S-contradictory in Λ_S, by definition of Extend.

Let $(\mathsf{Lemma}_{N_1}, \mathsf{Link}_{N_1})$ be the result of $\mathsf{Regress}(N_1)$. Let Link be obtained from Link_{N_1} by replacing all mappings to $L_i \mid c_i$ by mappings to $L_i \mid c_i'$. Then Regress returns $(\mathsf{Lemma}_{N_1}, \mathsf{Link})$.

(iii) the rule Close, then $\alpha \in \mathit{Mods}(\Gamma_S)$, the selected clause $C(\mathbf{x}) \leftarrow c$ is in Ψ_S, and each of its literals is α-contradictory with Λ_S. By Lemma 3.9.5 there is a clause $C(\mathbf{x}) \vee D \leftarrow d$ in Φ such that each α-solution of c extends to an α-solution of $d \wedge c$. By this and $C(\mathbf{x}) \leftarrow c$ being closed on, for each literal L_i of C, $L_i \mid (d \wedge c)_i$ is α-contradictory with Λ_S. Furthermore, by Lemma 3.9.5 also for each literal L_i of D the literal $L_i \mid (d \wedge c)_i$ is Γ_s-contradictory with Λ_S.

Let Lemma be $C \vee D \leftarrow d \wedge c$. Let Link map each literal of Lemma to a literal in Λ_S with which it is α-contradictory. Then Regress returns (Lemma, Link).

(iv) the rule Split, then the split literal $L_i \mid c_i$ is in Λ_{S_1} and its complement $\overline{L_i} \mid c_i$ is in Λ_{S_2}. Let $(\mathsf{Lemma}_{N_1}, \mathsf{Link}_{N_1})$ be the result of $\mathsf{Regress}(N_1)$, and let $(\mathsf{Lemma}_{N_2}, \mathsf{Link}_{N_2})$ be the result of $\mathsf{Regress}(N_2)$. Note that Γ_S, Γ_{S_1} and Γ_{S_2} are identical.

Link_{N_1} may map some literals of Lemma_{N_1} to $L_i \mid c_i$ and Link_{N_2} may map some literals of Lemma_{N_2} to $\overline{L_i} \mid c_i$.

If more than one is mapped in the case of N_1, then, assuming that the children are α-invariant, Lemma_{N_1} must be of the shape $C \vee \overline{L_i}(\mathbf{x}) \vee \overline{L_i}(\mathbf{y}) \leftarrow d$ where $\overline{L_i}(\mathbf{x})$ and $\overline{L_i}(\mathbf{y})$ are both mapped to $L_i \mid c_i$. We simplify Lemma_{N_1} to $C \vee \overline{L_i}(\mathbf{x}) \leftarrow \pi\, \mathbf{z}\, (d \wedge \mathbf{x} \doteq \mathbf{y})$, where $\mathbf{z}$ are the variables occurring in C and $\overline{L_i}(\mathbf{x})$. By applying this method of factoring repeatedly to Lemma_{N_1} and Lemma_{N_2}, the procedure ensures that eventually at most one literal is mapped in each case.

Now, if none is mapped in Link_{N_1} then the procedure returns $\mathsf{Regress}(N_1)$,

and if none is mapped in Link_{N_2} then the procedure returns $\mathsf{Regress}(N_2)$.

Otherwise, again assuming that the children are α-invariant, Lemma_{N_1} must be of the form $C \vee \overline{L_i} \leftarrow d$ and Lemma_{N_2} must be of the form $C' \vee L_i \leftarrow d'$, We rename the clauses such that the variables of $\overline{L_i}$ and L_i are identical. Let Lemma be the resolvent on the split literal, i.e., $(C \vee C')(\mathbf{x}) \leftarrow \pi\,\mathbf{x}\,(d \wedge d')$. Let Link be the merge of Link_{N_1} and Link_{N_2}. Then $\mathsf{Regress}$ returns (Lemma, Link).

(v) one of the rules $\mathsf{Domain\ Split}$ or $\mathsf{Ground\ Split}$, then by assumption of $\alpha \in \mathit{Mods}(\Gamma_S)$ and definition of the rules, either $\alpha \in \mathit{Mods}(\Gamma_{N_1})$ or otherwise $\alpha \in \mathit{Mods}(\Gamma_{N_2})$. In the first case $\mathsf{Regress}$ returns $\mathsf{Regress}(N_1)$, in the second case it returns $\mathsf{Regress}(N_2)$.

(vi) the rule Assert, then $L_i \mid c_i$ is the assert literal of the selected clause $C \vee L_i \leftarrow c$, and $L_i \mid c_i$ is in Λ_{S_1}. Let (Lemma_{N_1}, Link_{N_1}) be the result of $\mathsf{Regress}(N_1)$. If Link_{N_1} maps no literal to $L_i \mid c_i$, then $\mathsf{Regress}$ returns (Lemma_{N_1}, Link_{N_1}).

Otherwise, assuming that N_1 is α-invariant, Lemma_{N_1} must be of the form $C' \vee \overline{L_i} \leftarrow c'$. We rename $C \vee L_i \leftarrow c$ and $C' \vee \overline{L_i} \leftarrow c'$ such that they agree on the variables of L_i and $\overline{L_i}$. Let Lemma be the resolvent $C \vee C' \leftarrow \pi\,\mathbf{x} c \wedge c'$ of the two clauses, where $\mathbf{x}$ are the variables occurring in $C \vee C'$.

By definition of Assert, $L_i \mid c_i$ of $C \vee L_i \leftarrow c$ is α-universal in Λ_S. By $\pi\,\mathbf{x}(c \wedge c')$ being stronger than c, $L_i \mid \pi\,\mathbf{x}(c \wedge c')_i$ of $(C \vee L_i)(\mathbf{x}) \leftarrow \pi\,\mathbf{x}(c \wedge c')$ is also α-universal in Λ_S. It follows that each literal $L_j \mid \pi\,\mathbf{x}\,(c \wedge c')_j$ of $C \vee L_i \leftarrow \pi\,\mathbf{x}\,(c \wedge c')$ other than $L_i \mid \pi\,\mathbf{x}\,(c \wedge c')_i$ is α-contradictory with Λ_S. Let Link_{N_0} map each $L_j \mid \pi\,\mathbf{x}\,(c \wedge c')_j$ to a literal in Λ_S with which it

is α-contradictory. Let Link be the merge of Link_{N_0} and Link_{N_1}.

In the case that the selected clause of Assert is not an input clause, it must be a simplification of an input clause obtained by applications of Resolve. Then we can adapt Link as explained in Case (iii) for Close.

If Link maps a literal to $L_i \mid c_i$, i.e., Link_{N_1} mapped more than only the one literal considered above to $L_i \mid c_i$, then Regress repeats the steps above using (Lemma, Link) instead of (Lemma_{N_1}, Link_{N_1}). When this recursive process finally terminates, Link maps no literal to $L_i \mid c_i$. Then Regress returns the final (Lemma, Link).

□

For simplicity and w.l.o.g. we will assume in the following that the leaf nodes of a refutation tree are the conclusions of applications of Close, and that Close is applied exactly once on each path. It is easy to see that any refutation tree can be converted to a refutation tree of this shape, by simply cutting off all branches below the first Close application.

The next lemma states that Regress computes only α-invariant labels. The theorem makes then use of the fact that this holds in particular for the root node of a refutation tree, which gives us as a corollary that Regress computes a resolution proof of the unsatisfiability of Φ, α modulo LIA.

Lemma 3.7.6 *Let T be a refutation tree with root sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$. Fix an $\alpha \in \Gamma$. When Regress is applied to T, it computes a label for each inner node such that each label is α-invariant.*

Theorem 3.7.7 *Let T be a refutation tree with root sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$. Fix an $\alpha \in \Gamma$. Then* Regress *labels the root node of T with a clause $\square \leftarrow c$ such that $\Phi \cup \{\alpha\} \cup \{\square \leftarrow c\}$ is unsatisfiable.*

Proof. Let $(C \leftarrow c, \mathsf{Link})$ be the label computed by Regress for T.

As Γ is refined only by partitioning with Domain Split and Ground Split, each model of a sequent of a derivation node is also a model of a sequent of one of the children nodes. It follows that there must be at least one leaf node with sequent S' such that $\alpha \in \Gamma_{S'}$.

By construction of Regress (and as is explained further in the proof of Lemma 3.7.6), whenever the constraint of a lemma in the label of at least one child node is α-satisfiable, then so is the constraint of the lemma in the label of the node itself. Thus c is α-satisfiable.

By Lemma 3.7.6 it follows that $(C \leftarrow c, \mathsf{Link})$ is α-invariant. As $\Lambda \cdot \Gamma$ is admissible, by 3.7.3-(ii) there can be no literals in Λ that are α-contradictory with any literal of $C \leftarrow c$. By 3.7.3-(iii), C is $\square$. As c is α-satisfiable, $\square \leftarrow c$ is unsatisfiable in $\Phi \cup \{\alpha\}$. The claim follows. $\square$

As Regress constructs all lemmas by applying resolution and factoring, it is easy to see that it in fact builds a resolution proof of the empty clause from Φ and Γ, piece by piece for each model α of Γ. Equivalently, it constructs the lemma $\Phi \models_{\mathcal{Z}} \neg\alpha$ for each α, and thus, if there are finitely many α, as a consequence $\Phi \models_{\mathcal{Z}} \neg\Gamma$.

Regress$*$ As a refutation tree is finite, it is intuitively clear that even if there are infinitely many models of Γ, Regress needs to be called only finitely many

times in order to obtain sufficiently many constrained empty clauses to conclude the empty clause. Let us for simplicity first assume that Assert has not been used in the refutation. Then, in essence, the models of Γ can be partitioned by taking all Γ_i of the Close nodes, creating all intersections until a fix point is reached, and then taking the satisfiable partitions which are minimal wrt. to their set of models. Each partition constitutes an equivalence class in the sense that Regress computes the same constrained empty clause for each of its models.

Equivalently, but closer to the actual derivation, the Domain Split and Ground Split applications occurring in T can be combined in all possible ways. Let $d_1, \cdots, d_n$ be an enumeration of all n domain split constraints occurring in T. Then each of the 2^n conjunctions over all d_i, where for each i either d_i or $\neg d_i$ is used as a conjunct, gives rise to a smallest (or empty) partition. It is easy to see that each $\alpha \in \mathit{Mods}(\Gamma)$ is a model of exactly one partition, as Domain Split and Ground Split perform a binary partition and thus α is a model of only one conjunction. Looking closely at Regress it becomes clear that, apart from Assert, when applied to a node with a sequent S it does not distinguish among the models of Γ_S. That is, each label is in fact Γ_i-invariant for each partition Γ_i. Therefore it suffices to run Regress only once for each partition, for any of its models.

Extending this to take Assert into consideration complicates things quite a bit. The mappings created for the literals of the lemma which stem from the selected clause are chosen by α-contradiction, not by Γ-contradiction. In fact, Γ-contradiction might not hold, and thus it is not possible to create the same map Link for each $\alpha \in \mathit{Mods}(\Gamma)$ when regressing an application of Assert. But,

as Λ is finite, each Assert regression can require only finitely many different Link maps. Thus, we can partition Γ further, such that for all α in each partition it holds that for each Assert application of the refutation tree to some sequent S, all literals of the selected clause are α-contradictory with exactly the same set of literals from Λ_S.

The postprocessing approach above might be too inefficient to be useful in practice. An implementation should modify Regress to compute the partitions on the fly, as we will do with the algorithm Regress$*$. It will be the basis for justifying non-chronological backtracking and learning, as described in Sections 3.7.5.5 and 3.7.5.6.

Algorithm 3.7.8 (Regress$*$) We formulate the regression algorithm Regress$*$ as a modification of Regress. It takes the same input, but does not fix an α. Instead of assigning one label to a node N with sequent S, it assigns a map from constraint sets $\Gamma_1, \ldots, \Gamma_n$ to labels. The union of the constraint sets must be equivalent to Γ_S, and when a constraint set Γ_i is mapped to *Label* then *Label* must be α-invariant for all α in $Mods(\Gamma_i)$.

The modifications to the regression of the inference rule applications are as follows.

(i) In the case of an application of Close to sequent S of node N, Regress$*$ maps Γ_S to Regress(N).

(ii) In the case of Split, Regress tries to resolve the two clauses in the left and right children labels. Regress$*$ has to try to resolve over the cross product of the labels in the domains of the left and right children. That is, first the

domains of Regress $*$ (N_1) and Regress $*$ (N_2) are harmonized by creating the intersections of all partitions up to a fix point as described above, and duplicating the map entries accordingly. For example, if Regress $*$ (N_1) maps Γ to *Label*, and the created partitions contain Γ_1 and Γ_2, then after harmonization Regress $*$ (N_1) maps Γ_1 as well as Γ_2 to *Label*. Thus we can now assume that the domains of Regress $*$ (N_1) and Regress $*$ (N_2) are identical. Then Regress $*$ (N) maps each constraint set Γ' in the domain of Regress $*$ (N_1) to the result of the application of Regress to the labels mapped to by Γ' in Regress $*$ (N_1) resp. Regress $*$ (N_2)

(iii) In the case of Domain Split and Ground Split, Regress $*$ (N) is the union of Regress $*$ (N_1) and Regress $*$ (N_2).

(iv) The case of Assert is similar to the case of Split. But, as explained above, harmonization of the partitions is now done based on which subsets of each partition are contradictory with the same context literals for each of the literals of the selected clause. Considering that the Assert application is part of the derivation and its side conditions have thus been checked by the proof procedure, it is reasonable to assume that sufficient information to perform this partition is available.

The remaining cases are straightforward.

□

3.7.5.5 Backtracking

As mentioned before, it is reasonable to assume that a proof procedure will explore a derivation tree by a depth-first left to right traversal. When back-

tracking from a closed branch, chronological backtracking explores the branches of all right splits (Split, Domain Split, Ground Split) on the branch. In contrast, backjumping, as an instance of non-chronological backtracking, jumps over right split branches which are useless for finding a refutation or a model.

The proof-generation procedures Regress and Regress$*$ described above in Section 3.7.5.4 provide a strong intuition of when and how backjumping is justified in ME(LIA). If Regress$*$ does not depend on any label of some branch B of a refutation tree T, then that branch is useless. For example, if for a Split node N with children N_1 and N_2 Regress$*(N)$ returns Regress$*(N_1)$ or Regress$*(N_2)$, then the Split was not needed. For a depth-first exploration this translates into performing the left split at N, closing the sub-tree and computing the lemma for N_1 during backtracking, and finally backjumping over the node N without exploring the sub-tree N_2

Now consider the other splitting rules, Domain Split and Ground Split, applied to a node N with children N_1 and N_2, where S is the sequent of N. By definition, Regress$*(N)$ is the union of the maps Regress$*(N_1)$ and Regress$*(N_2)$. Consider a mapping from Γ to *Label* in Regress$*(N)$. Regress$*$ enforces that *Label* is α-invariant for all $\alpha \in \mathit{Mods}(\Gamma)$. By construction of the mapping, $\mathit{Mods}(\Gamma) \subseteq \mathit{Mods}(\Gamma_S)$. Now *Label* could very well be α-invariant for some $\alpha \in \mathit{Mods}(\Gamma_S)$ although $\alpha \notin \mathit{Mods}(\Gamma)$. Let Γ' be such that $\mathit{Mods}(\Gamma) \subseteq \mathit{Mods}(\Gamma')$, $\mathit{Mods}(\Gamma') \subseteq \mathit{Mods}(\Gamma_S)$, and *Label* is α-invariant for all $\alpha \in \mathit{Mods}(\Gamma')$. Then the domain of *Label* can be generalized to Γ' in Regress$*(N)$, and all mappings from any Γ'' with $\mathit{Mods}(\Gamma'') \subseteq \mathit{Mods}(\Gamma')$ can be removed from Regress$*(N)$. This leads to less fine grained partitions of the constraint sets, and less lemmas to maintain during

the procedure. In particular, if a Domain Split or Ground Split was useless, then Regress $*$ (N_1) can be generalized such that the union of its domains is equal to Γ_S, and Regress $*$ (N_2) is not needed at all. In that case the right split can be omitted.

We stress that proof-generation serves as a justification for backjumping, it is neither a requirement nor a strict guide line for its implementation. For example, to jump over a domain split N with children N_1 and N_2 and with split constraint d, we would need to check if d was relevant for any α-contradictory check used in closing and regressing the derivation branch below N_1. This could be done by extracting unsatisfiable cores from the constraint solver that is used for Γ checks (see Section 3.7.3.3). If the split constraint d is not part of any unsatisfiable core, then the split was not needed and N can be backjumped over.

Similarly, approximations for backjumping over literal splits are possible. We still use the Link structure, but the constraint and the partitions of Γ are completely ignored. Instead of computing lemmas for partitions of Γ by resolution, the literals of all lemmas are considered to be a conflict set. A split is clearly not relevant if the Link structures of the conflict sets of the two children nodes do not contain the split literals. While it is not clear how effective this approximation is in practice, it is simpler and cheaper to compute, which might make it worthwhile.

3.7.5.6 Learning

Like in the case of backtracking, the proof generating procedure Regress$*$ gives clear guide lines of how to compute lemmas analogous to their computation in ME (see Section 2.4). The first crucial difference is that a lemma-learning

algorithm would stop (latest) at split nodes instead of resolving them. In a sense, proof-generation can be seen as an extension of lemma-learning with a focus on reliability instead of efficiency. Thus, the second important difference is that lemma computation does not need to be complete, rather it needs to be cheap and the computed lemmas need to be effective. As mentioned in Section 3.7.5.5, generalizing the partitions for which lemmas are effective in the context of domain splits can reduce the number of computed lemmas, but makes their computation more expensive. Similarly, when regressing a split node it is not necessary to partition the Γs and then consider the cross product of the lemmas. Instead, an approximation worth considering might be to use each lemma only once for resolution. This way significantly less lemmas are learned. The ones that are learned should be relevant in the current context, whereas the missed ones tend to require more regression steps before they can be learned.

A learning procedure based on Regress$*$ corresponds closely to the grounded method in ME (see Section 2.4.4), as it starts with the instances of input clauses used in the applications of Close. The generalization to the lifted method (Section 2.4.5) is immediate, by taking in the Close nodes the input clauses themselves as lemmas, instead of their instances. Then each generated lemma is at least as general as in the original algorithm, and all resolve and factoring steps still apply. This gives some justification to expect that lemma-learning in ME(LIA) can be similarly beneficial as in the ME case.

Not surprisingly, the main disadvantage of lemma-learning in ME compared to DPLL holds for ME(LIA) as well, lemma-learning does not suffice to avoid right splitting on context literals. In fact, Example 2.4.12 demonstrates

this not only for ME, but for ME(LIA) as well.

3.7.5.7 Constraint Simplification

As mentioned before, the most critical part of any implementation is probably how constraints are handled. The core operations of the calculus, inference rule application and redundancy detection, rely fundamentally on constraint solving.

One important step is to enforce strong normalization conventions on constraints. This makes algorithms operating on constraints simpler and faster, and enables far more sub-constraint sharing between constraints, especially when constraints are managed in a hash-consing data structure. Finally, simplifying constraints by quantifier elimination as explained below assumes a specific normalization that is easy to obtain from the one introduced next.

A reasonable normalization would for example rewrite all occurrences of $\forall$, $\dot{\leq}$, $\dot{>}$, $\dot{\geq}$ to equivalent constraints not containing any of these symbols. It might even make sense to rewrite all Boolean connectives to $\neg$, $\wedge$, and $\vee$. Furthermore, multiplication by a constant factor can be introduced as an abbreviation for a fixed number of additions. Then each atomic constraint can be written as either $P[\mathbf{x}] \dot{<} 0$ or $P[\mathbf{x}] \doteq 0$, where $P[\mathbf{x}]$ is a polynomial with variables $\mathbf{x}$. Assuming there is a global order on variables, polynomials should be kept ordered first by variables and secondly by coefficient. Apart from stronger normalization this makes the addition and subtraction of polynomials efficient. A strict total ordering over Boolean terms also allows for the normalization to be extended to conjunctions and disjunctions.

There are a number of obvious and trivial yet often effective simplification

techniques. On the constraint level it is often possible to simplify constraints by evaluation when they contain integer constants. On the Boolean level this basically amounts to the propagation of true and false, the detection of literals subsuming or subsumed by other literals or contradictory with other literals in a conjunction or disjunction, possibly by using cheap LIA properties for example of equations, and the detection of conjunctions and disjunctions that are trivially true or false.

It can also pay off to simplify constraints aggressively by eliminating all quantifiers. The high initial cost might be worth it in the long run, when the same constraint is used repeatedly in constraint solving. Furthermore, it is the basis for the proof procedure described in Section 3.7.3.3. A quantifier elimination algorithm transforms a constraint into an equivalent constraint which does not contain any quantifiers. For example, $\exists x(y \dot{\leq} x \wedge x \dot{\leq} 5)$ (normalized to $\exists x(-x + y - 1 \dot{<} 0 \wedge x - 6 \dot{<} 0)$) is equivalent to $y \dot{\leq} 5$ (normalized to $y - 6 \dot{<} 0$), which contains the free variable y but no quantifiers. LIA allows one to eliminate all quantifiers, which implies that a closed constraint is simplified to either true or false. All quantifier elimination algorithms work from the inside out. That is, innermost quantifiers have to be eliminated first, and a quantifier cannot be eliminated in general if another quantifier is in its scope.

We propose a combination of two quantifier elimination algorithms. The main algorithm is based on a quantifier algorithm for Real Linear Arithmetic by Monniaux [56]. We adapt it to LIA by using Cooper's quantifier elimination algorithm [31] as a component for solving conjunctions of LIA constraints, instead of conjunctions of Linear Rational Arithmetic constraints as in the original

approach. At its core, the Monniaux algorithm eliminates a quantifier of a constraint $\exists xc$, with c quantifier-free, by enumerating all the solutions of c with an SMT solver. A solution s, or rather a class of solutions, is defined by a satisfiable core, i.e., a conjunction of the atoms in c. Cooper's algorithm is used to simplify $\exists xs$ to the quantifier-free formula s'. c is replaced by $c \wedge \neg s'$ to exclude the solutions defined by s, and the procedure is restarted. It terminates when c is simplified to false, i.e., when all solutions have been found. The disjunction of the solutions is a quantifier-free formula that is equivalent to the original constraint $\exists xc$. The integration of quantifier elimination for each solution in the enumeration loop makes this algorithm potentially more performant than either Cooper's algorithm or an SMT based solution enumeration alone [56]. Furthermore, it is easy to implement, as a standard SMT solver can be used, and Cooper has to operate only on literal conjunctions, not arbitrary formulas.

If the SMT solver used for the enumeration takes Γ into account as an additional constraint, it should terminate faster and find fewer solutions.

It must be noted that Cooper introduces divisibility constraints of the form $i \mid p$, where i is an integer constant and p is an arbitrary polynomial. As $i \mid p$ can be expressed as $\exists x(i \mathbin{\dot{*}} x \doteq p)$, this does not change the expressiveness of the language of the calculus. If the SMT solver supports the division predicate natively, it might make sense to extend the input language of the calculus with it, as this makes it possible to formalize a wider range of problems naturally.

3.7.5.8 Candidate Recomputation

Although each candidate rule application can be represented in constant size, it can still happen that too many candidates are computed to be stored

in memory. A solution is to store only the best candidates, forget the others, and recompute them on demand. For example, Darwin achieves this by storing candidates in a min-max heap, which supports efficient removal of the best as well as the worst element. The procedure must keep track of which candidates have been forgotten in a compact way, i.e., by remembering only the best forgotten candidate for some partition of the candidate space. Possible partitions are, for example, by application rule, by depth of the derivation tree, or by the selected clause from Φ or Ψ.

If the number of candidates kept is big enough, recomputation should occur almost never, as the best candidates should suffice to keep the derivation going. If not, then the derivation is probably taking that long that the time spent for recomputation is not going to matter overall anyways.

3.8 Conclusion

We have presented an extended version of the previously published version of the ME(LIA) calculus. It incorporates universal literals, additional redundancy criteria, and additional inference rules.

Clearly, implementing the calculus based on the presented detailed guide lines is the next step. The implementation needs to be evaluated in practice, to see how it compares to alternative approaches, including ME. Although many theorem proving techniques carry over from the ME setting, it is not clear how to achieve something like term indexing for ME(LIA). That is, how to perform context checks such that the context literals are not considered linearly one by one, but that some kind of indexing allows to check many literals simultaneously.

Support for universal literals needs further consideration. It needs to be

explored if Split can be generalized to work with universal literals, and especially without complicating the calculus too much. It is also worth investigating if the notion of universality can be broken down from the level of literals to the level of variables, akin to ME.

While support for finite domain functions and for approximating functions by using predicates can give some leverage, support for proper function symbols would be important. Similarly, it needs to be investigated which other constraint theories are suitable for combination with the ME(LIA) framework. The crucial properties of LIA which were exploited in ME(LIA) are a non-dense ordering, and the decidability of the satisfiability of arbitrary LIA constraints. Finally, it is an interesting question if several theories can be combined instead of using just one, as is done in DPLL(T) [47].

3.9 Proofs

3.9.1 Preliminaries

Lemma 3.9.1 *Let* $\overset{wf}{\leq}$ *be the ordering* $0, -1, 1, -2, 2, \ldots$ *over the integers. Let* $\overset{wf.}{\leq}$ *be defined the same way as* $\dot{\leq}$ *is defined in Section 3.4.1, except that integers are compared by* $\overset{wf}{\leq}$ *instead of* $\dot{\leq}$*. Then any constraint* $c[\mathbf{x}]$ *is admissible.*

Proof. If c is not satisfiable for any assignment α, then c is trivially admissible. Thus, assume in the following that α is an arbitrary assignment such that c is α-satisfiable.

Due to $\overset{wf}{\leq}$ being well-founded, $\overset{wf.}{<}$ is a well-founded partial ordering over the α-solutions of c, and thus c has minimal solutions. Any two minimal α-solutions $\mathbf{l}$ and $\mathbf{m}$ must be incomparable, that is there must be two indices i, j,

with $1 \leq i \leq n$, $1 \leq j \leq n$, and $i \neq j$, such that $m_i \overset{wf}{<} l_i$ and $l_j \overset{wf}{<} m_j$. As $\mathbf{m}$ has finitely many elements, there can be only finitely many such (ordered) combinations of i and j. Thus it suffices to show that for any given pair i, j there are only finitely many minimal α-solutions of c which differ at i, j. Let the set of all such solutions be denoted by S_{ij}. If S_{ij} is empty, it trivially contains only finitely many solutions. So assume in the following that it is not empty.

Now, S_{ij} can be ordered such that the i.th component of the k.th solution is less than the i.th component of the $k+1$.th solution wrt. $\overset{wf}{<}$, and the j.th component of the k.th solution is greater than the j.th component of the $k+1$.th solution wrt. $\overset{wf}{>}$. That is, when traversing S_{ij} in order solution by solution, the i.th component is strictly increasing, and the j.th component is strictly decreasing.

Now pick an arbitrary solution m in S_{ij}. The i.th resp. j.th component of all solutions is bounded below by 0 wrt. $\overset{wf}{<}$. It follows that there can be only finitely many solutions in S_{ij} whose i.th component is smaller than the i.th component of m. Furthermore, if a solution is bigger than m at the j.th position, then it must be smaller than m at the i.th position. Again, it follows that there can be only finitely many solutions in S_{ij} whose j.th component is bigger than the j.th component of m wrt. $\overset{wf}{>}$.

Thus, S_{ij} consists of m, finitely many solutions which are smaller than m at the i.th position, and finitely many solutions which are bigger than m at the j.th position. Thus, S_{ij} is finite, for each pair of positions i, j there are only finitely many minimal solutions which are incomparable due to i, j, and there are only finitely many pairs of positions. The claim follows. □

Lemma 3.4.1 *Let $c[\mathbf{x}]$ be a constraint and $\mathbf{n}$ a tuple of integer constants of the same length as $\mathbf{x}$ such that $\models_{\mathcal{Z}} \exists \mathbf{x}\, \mathbf{n} \dot{\leq} \mathbf{x}$. Then c is admissible.*

Proof. Analogous to the proof of Lemma 3.9.1. The crucial observation is that it follows from $\models_{\mathcal{Z}} \exists \mathbf{x}\, \mathbf{n} \dot{\leq} \mathbf{x}$ that $\dot{<}$ is a well-founded partial ordering over the α-solutions of c with $\mathbf{n}$ as the lower bound. □

Lemma 3.4.2 *Let α be an assignment and c an admissible constraint. Then, there is an $n \geq 0$ such that $\mu_1\, c, \ldots \mu_n\, c$ have unique, pairwise different α-solutions, which are all minimal α-solutions of c. Furthermore, for all $k > n$, $\mu_k\, c$ is not α-satisfiable.*

Proof. Let $\mathbf{m}_1, \ldots, \mathbf{m}_n$ be all minimal α-solutions of c, for some $n \geq 0$. They exist because c is admissible. Without loss of generality assume they are ordered lexicographically, i.e. $\mathbf{m}_i \dot{\leq}_\ell \mathbf{m}_j$ whenever $1 \leq i \leq j \leq n$.

To prove the first part of the lemma it suffices to show that $\mathbf{m}_k$ is the unique α-solution of $\mu_k\, c$, for all $k = 1, \ldots, n$. It then follows that these solutions are pairwise different, because $\mathbf{m}_1, \ldots, \mathbf{m}_n$ are all different.

The case $n = 0$ being trivial, we assume $n > 0$ and prove the statement by induction on k, for all $k = 1, \ldots, n$.

If $k = 1$ then recall that by assumption $\mathbf{m}_1$ is the least of all minimal α-solutions of c. This fact is expressed in our constraint language as the constraint $\mu_\ell(\mu\, c)$ $(= \mu_1\, c)$, which has exactly one α-solution, $\mathbf{m}_1$.

If $k > 1$ then assume by induction that $\mathbf{m}_1, \ldots, \mathbf{m}_{k-1}$ are the unique α-solution of $\mu_1\, c, \ldots \mu_{k-1}\, c$, respectively. We have to prove this for $\mu_k\, c$.

By assumption, $\mathbf{m}_k$ is a minimal α-solution of c (the k-th one). Thus, $\mathbf{m}_k$ is an α-solution of $\mu\, c$. Because $\mathbf{m}_1, \ldots, \mathbf{m}_n$ are all pairwise different, it follows with

the induction hypothesis (and $k \leq n$) that $\mathbf{m}_k$ is not an α-solution of any of the constraints $\mu_1\, c, , ..., \mu_{k-1}\, c$. Thus $\mathbf{m}_k$ is an α-solution of $\neg(\mu_1\, c) \wedge \cdots \wedge \neg(\mu_{k-1}\, c)$. It follows that $\mathbf{m}_k$ is an α-solution of the conjunction $c' = \neg(\mu_1\, c) \wedge \cdots \wedge \neg(\mu_{k-1}\, c) \wedge (\mu c)$.

For the induction step it suffices to show that $\mathbf{m}_k$ is the least α-solution of c'. Because then, with $\mu_k\, c = \mu_\ell\, c'$ it follows with the definition of the μ_ℓ-operator that $\mathbf{m}_k$ is the unique α-solution of $\mu_k\, c$.

By way of contradiction, assume there is a least α-solution $\mathbf{m}$ of c' with $\mathbf{m} \neq \mathbf{m}_k$. We consider two (exhaustive) cases.

In the first case $\mathbf{m}_k \dot{\leq}_\ell \mathbf{m}$. This is a direct contradiction to the assumption that $\mathbf{m}$ is the least α-solution of c.

In the second case $\mathbf{m} \dot{\leq}_\ell \mathbf{m}_k$. Because $\mathbf{m}$ is a (least) α-solution of c', $\mathbf{m}$ is in particular an α-solution of its conjunct $(\mu\, c)$. In other words, $\mathbf{m}$ is a minimal α-solution of c'. Further, recall that $\mathbf{m}_1, \ldots, \mathbf{m}_k, \ldots, \mathbf{m}_n$ are all minimal α-solutions and that they are lexicographically ordered. Altogether, with $\mathbf{m} \dot{<} \mathbf{m}_k$ it follows that $\mathbf{m} = \mathbf{m}_j$, for some $1 \leq j < k$. By the induction hypothesis, $\mathbf{m}_j$ is an α-solution of $\mu_j\, c$. On the other hand, with $\mathbf{m}$ and thus also $\mathbf{m}_j$ being a solution of $\mu_k\, c$, $\mathbf{m}_j$ is also an α-solution of $\mu_k\, c$'s conjunct $\neg(\mu_j\, c)$. A plain contradiction.

From the contradictions in both cases conclude $\mathbf{m} = \mathbf{m}_k$, which remained to be shown for the first part.

It remains to show that $\mu_k\, c$ is not α-satisfiable, for all $k > n$. This, however, is clear with the first part, because any such α-solution $\mathbf{m}$ would provide a minimal α-solution for c that is different to each of $\mathbf{m}_1, \ldots, \mathbf{m}_n$. However $\mathbf{m}_1, \ldots, \mathbf{m}_n$ was assumed to consist of *all* minimal α-solutions. □

Lemma 3.4.3 *Let α be a parameter assignment, and $c[\mathbf{x}]$ and $d[\mathbf{x}]$ two α-satisfiable admissible constraints. Then one of the following cases applies:*

(i) $\alpha \models_{\mathcal{Z}} c \dot{<}_{\mu_\ell} d$

(ii) $\alpha \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$

(iii) $\alpha \models_{\mathcal{Z}} d \dot{<}_{\mu_\ell} c$

Proof. $c[\mathbf{x}]$ and $d[\mathbf{x}]$ contain the same free variables and are thus comparable wrt. $\doteq$ and $\dot{\leq}$. Thus the formulas in cases (i)-(iii) are well-formed.

Since both $c[\mathbf{x}]$ and $d[\mathbf{x}]$ are α-satisfiable, the solutions $\mu_\ell\, c[\mathbf{x}]$ and $\mu_\ell\, d[\mathbf{x}]$ are α-satisfiable as well, with least solutions $\mathbf{m_c}$ and $\mathbf{m_d}$. Clearly, either $\mathbf{m_c} \dot{<} \mathbf{m_d}$ is true, in which case (i) holds, or $\mathbf{m_d} \dot{<} \mathbf{m_c}$ is true in which case (iii) holds, or $\mathbf{m_d} \doteq \mathbf{m_c}$ is true, in which case (ii) holds, as the least α-solution of a constraint is also an α-solution of the permanent constraint of a constraint. □

Lemma 3.4.9 (Consistent α-Productivity) *Let $\Lambda \cdot \Gamma$ be an admissible context and $\alpha \in Mods(\Gamma)$. For any constrained literal $L(\mathbf{x}) \mid c$, Λ cannot α-produce both $L(\mathbf{x}) \mid c$ and its complement $\overline{L}(\mathbf{x}) \mid c$.*

Proof. Suppose, by contradiction, Λ α-produces both $L(\mathbf{x}) \mid c$ and its complement $\overline{L}(\mathbf{x}) \mid c$, for some context literal $\overline{L}(\mathbf{x}) \mid c$. Then there need to be two context literals $K = L(\mathbf{x}) \mid c_1$ and $K' = \overline{L}(\mathbf{x}) \mid c_2$ in Λ that α-produce $L(\mathbf{x}) \mid c$ and $\overline{L}(\mathbf{x}) \mid c$ wrt. Λ, respectively. By definition of α-productivity, K and K' α-cover these literals and so c_1 and c_2 must be α-satisfiable. Now, with Lemma 3.4.3 three cases apply: if $\alpha \models_{\mathcal{Z}} c_1 \doteq_{\mu_\ell} c_2$ then $\Lambda \cdot \Gamma$ would be contradictory, which is impossible by admissibility; if $\alpha \models_{\mathcal{Z}} c_1 \dot{<}_{\mu_\ell} c_2$ then K cannot α-produce $L(\mathbf{x}) \mid c$,

and if otherwise $\alpha \models_{\mathcal{Z}} c_2 \dot{<}_{\mu_\ell} c_1$ then K' cannot α-produce $\overline{L}(\mathbf{x}) \mid c$. Both cases contradict the assumption made. □

Lemma 3.4.11 *Let $\Lambda \cdot \Gamma$ be an admissible context and $\alpha \in Mods(\Gamma)$. For any ground literal $L(\mathbf{s})$ such that $\alpha \models_{\mathcal{Z}} \mathbf{0} \dot{\leq} \mathbf{s}$, $\mathcal{Z}_{\Lambda,\alpha}$ satisfies $L(\mathbf{s})$ if and only if Λ α-produces $L(\mathbf{s})$.*

Proof. Let $L(\mathbf{s})$ be an arbitrary ground literal over Σ. As $\Lambda \cdot \Gamma$ is admissible Λ contains the literal $\neg L(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$. Either this literal α-produces $\neg L(\mathbf{s})$, or it is α-blocked by some context literal $L(\mathbf{x}) \mid c$. Now either $L(\mathbf{x}) \mid c$ α-produces $L(\mathbf{s})$, or it is in turn α-blocked by some context literal. It is clear from Definition 3.4.7 that then there must be a literal that α-covers $L(\mathbf{s})$ or $\neg L(\mathbf{s})$ that is not α-blocked by another context literal, since universal literals cannot be blocked and non-universal literals can only be α-blocked by literals with a bigger least α-solution. By Lemma 3.4.9 only one of $L(\mathbf{s})$ or $\neg L(\mathbf{s})$ can be α-produced. The claim follows by definition of $\mathcal{Z}_{\Lambda,\alpha}$ □

Lemma 3.4.12 *Let $\Lambda \cdot \Gamma$ be an admissible context, let $L(\mathbf{x}) \mid c$ be a context literal in Λ, and let $\alpha \in Mods(\Gamma)$. If c is α-satisfiable then each literal in $\{L(\mathbf{m}) \mid \alpha \models_{\mathcal{Z}} \mathsf{perm}(c)[\mathbf{m}/\mathbf{x}]\}$ is satisfied by $\mathcal{Z}_{\Lambda,\alpha}$.*

Proof. Let $\mathbf{m}$ be an arbitrary α-solution of $\mathsf{perm}(c)$. Observe that because of this, and because by definition also $\alpha \models_{\mathcal{Z}} \bar{\forall}(\mathsf{perm}(c) \rightarrow c)$, it follows that $L(\mathbf{x}) \mid c$ α-covers $L(\mathbf{m})$. If $L(\mathbf{x}) \mid c$ α-produces $L(\mathbf{m})$, then the claim follows by Lemma 3.4.11, since $\mathbf{m}$ was an arbitrary α-solution of $\mathsf{perm}(c)$.

If $L(\mathbf{x}) \mid c$ is universal in $\Lambda \cdot \Gamma$, it α-produces $L(\mathbf{m})$ by definition. So assume that $L(\mathbf{m})$ is not universal. Then, $L(\mathbf{m})$ must be the least α-solution of

$L(\mathbf{x}) \mid c$.

Now assume that a context literal $\overline{L}(\mathbf{x}) \mid d$ from Λ α-blocks $L(\mathbf{x}) \mid c$ from α-producing $L(\mathbf{m})$. $\alpha \models_{\mathcal{Z}} c \dot{<}_{\mu_\ell} d$ cannot hold, as then $L(\mathbf{x}) \mid c$ would not α-cover $L(\mathbf{m})$. $\alpha \models_{\mathcal{Z}} c \doteq_{\mu_\ell} d$ cannot hold, as then $\Lambda \cdot \Gamma$ would be contradictory. $\alpha \models_{\mathcal{Z}} d \dot{<}_{\mu_\ell} c$ cannot hold, as then $L(\mathbf{x}) \mid c$ would not be α-blocked by $\overline{L}(\mathbf{x}) \mid d$. But one of these cases must hold due to Lemma 3.4.3. Thus, there is no such blocking literal in Λ, $L(\mathbf{x}) \mid c$ α-produces $L(\mathbf{m})$, and the claim follows. □

3.9.2 Soundness

To prove soundness we have to show that whenever there is a refutation of Φ and Γ then $\Phi \cup \Gamma$ is LIA-unsatisfiable.

To prove the result, we need to take the context literals Λ into account, as they evolve in the refutation and constrain the candidate models. To this end, for a given parameter assignment α we define a set $\Lambda^{\mu_\ell(\alpha)}$ of constrained unit clauses as follows.

Definition 3.9.2 (μ_ℓ-satisfiability) Let $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ be a sequent and α a parameter assignment. Then $\Lambda^{\mu_\ell(\alpha)}$ consists of the set of all unit clauses $L(\mathbf{x}) \leftarrow \mathsf{perm}(c)$ such that the literal $L(\mathbf{x}) \mid c$ is in Λ and c is α-satisfiable.

We say that $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ is *μ_ℓ-satisfiable* iff for some parameter assignment $\alpha \in \mathit{Mods}(\Gamma)$, $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup \Psi$ is LIA-satisfiable in some Σ-interpretation $\mathcal{Z}$ that agrees with α on the parameters. We call $\mathcal{Z}$ a *μ_ℓ-model of* $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$.

□

In words, for an assignment $\alpha \in \mathit{Mods}(\Gamma)$ the clause set $\Lambda^{\mu_\ell(\alpha)}$ (conceptually) consists of the set of (possibly infinitely many) ground literals which are

permanently α-satisfied in Λ, and μ_ℓ-satisfiability requires that any μ_ℓ-model has to satisfy $\Lambda^{\mu_\ell(\alpha)}$.

We will use the contrapositive of the following theorem to show that whenever there is a refutation of Φ and Γ, then the initial sequent of the refutation is μ_ℓ-unsatisfiable.

Lemma 3.9.3 *For each rule of the ME(LIA) calculus, if the premise of the rule is μ_ℓ-satisfiable, then one of its conclusions is μ_ℓ-satisfiable as well.*

Proof. We carry out a case analysis wrt. the derivation rule applied. We consider the rules in the order of core, optional, simplification.

3.9.2.1 Core Rules

Inst) The premise of Inst has the form $\Lambda \cdot \Gamma \vdash (\Phi, C \leftarrow c) \cdot \Psi$, while its conclusion has the form $\Lambda \cdot \Gamma \vdash (\Phi, C \leftarrow c) \cdot (\Psi, C \leftarrow d)$, where d is a context unifier of $C \leftarrow c$ against $\Lambda \cdot \Gamma$.

As d is a context unifier of $C \leftarrow c$, it follows by construction of d that d is a stronger constraint than c. Thus any model of $C \leftarrow c$ is also a model of $C \leftarrow d$. It follows immediately that any model of $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup (\Phi, C \leftarrow c) \cup \Psi$ is also a model of $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup (\Phi, C \leftarrow c) \cup (\Psi, C \leftarrow d)$.

Thus if the premise of Inst is μ_ℓ-satisfiable then so is the conclusion.

Close) The premise of Close has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$, while its conclusion has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c, \square \leftarrow \top)$, where each literal of $C \leftarrow c$ is Γ-contradictory with Λ.

As $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c, \Box \leftarrow \top)$ is μ_ℓ-unsatisfiable, we must show that $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ is μ_ℓ-unsatisfiable as well. Equivalently, we must show that for all parameter assignments α, $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$ is LIA-unsatisfiable in all Σ-interpretations that agree with α on the parameters.

Let α be any parameter assignment. If $\alpha \notin \mathit{Mods}(\Gamma)$ then there is no Σ-interpretation that agrees with α on the parameters and that satisfies Γ. In this case the claim follows trivially. Hence assume from now on $\alpha \in \mathit{Mods}(\Gamma)$.

Let $C \leftarrow c$ be of the form $L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c[\mathbf{x}]$. As each literal $L_i(\mathbf{x}_i) \mid c_i[\mathbf{x}_i]$ of $C \leftarrow c$ is Γ-contradictory with Λ, there must be literals $\overline{L_i} \mid d_i$ in Λ such that $\alpha \models_{\mathcal{Z}} c_i \doteq_{\mu_\ell} d_i$. By the definition of $\doteq_{\mu_\ell}$ it follows that d_i, c_i, and c are α-satisfiable. By definition of $\doteq_{\mu_\ell}$, by Definition 3.9.2, and since clause literals are not universal, any Σ-model $\mathcal{Z}$ of $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$ that agrees with α on the parameters must assign true to each literal $\overline{L_i}(\mathbf{m}_i)$, where $\mathbf{m}_i$ is the least α-solution of c_i.

Let $\mathbf{n}$ be the least α-solution of c. $\mathcal{Z}$ must then satisfy $L_1(\mathbf{n}_1) \vee \cdots \vee L_k(\mathbf{n}_k)$, as it is a ground instance of the clause $C \leftarrow c$. Since c_i is defined as $\pi\, \mathbf{x}_i\, c[\mathbf{x}]$, $\mathbf{n}_i$ is the least α-solution of c_i, and thus $\alpha \models_{\mathcal{Z}} \mathbf{m}_i \doteq \mathbf{n}_i$. As $\mathcal{Z}$ assigns true to $\overline{L_i}(\mathbf{m}_i)$, for all $i = 1, \ldots, k$, such a model $\mathcal{Z}$ cannot exist, which remained to be shown.

Split) The premise of Split has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$, while its conclusions have respectively the form $(\Lambda, L_i \mid c_i) \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ and $(\Lambda, \overline{L_i} \mid c_i) \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$. Suppose that $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi, C \leftarrow c$ is μ_ℓ-satisfiable. Then there must be a parameter assignment $\alpha \in \mathit{Mods}(\Gamma)$ such that $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$ is LIA-satisfiable in some Σ-interpretation $\mathcal{Z}$ that agrees with α on the parameters.

If c_i is not α-satisfiable then by definition $\Lambda^{\mu_\ell(\alpha)} = (\Lambda, L_i \mid c_i)^{\mu_\ell(\alpha)} = (\Lambda, \overline{L_i} \mid c_i)^{\mu_\ell(\alpha)}$ and it is easy to see that μ_ℓ-satisfiability is preserved even for both conclusions.

If on the other hand c_i is α-satisfiable, then $\{L_i \mid c_i\}^{\mu_\ell(\alpha)} = \{L_i(\mathbf{m}_i)\}$ and $\{\overline{L_i} \mid c_i\}^{\mu_\ell(\alpha)} = \{\overline{L_i}(\mathbf{m}_i)\}$, for some integer vector $\mathbf{m}_i$, since $L_i \mid c_i$ is not universal. Thus $\mathcal{Z}$ must satisfy either $(\Lambda, L_i(\mathbf{m}_i))^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$ or $(\Lambda, \overline{L_i}(\mathbf{m}_i)^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$, which is clearly the case.

In conclusion in any case one of the consequences is μ_ℓ-satisfiable.

Extend) The premise of Extend has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ while its conclusion has the form $(\Lambda, L_i \mid c_i) \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$, where $\overline{L_i} \mid c_i$ is Γ-contradictory with Λ.

Suppose that $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ is μ_ℓ-satisfiable. Then there must be a parameter assignment $\alpha \in \mathit{Mods}(\Gamma)$ such that $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$ is LIA-satisfiable in some Σ-interpretation $\mathcal{Z}$ that agrees with α on the parameters.

As $\overline{L_i} \mid c_i$ is Γ-contradictory with Λ, with the same argumentation as for Close above it follows that (in particular) $\mathcal{Z}$ cannot LIA-satisfy $(\Lambda, \overline{L_i}(\mathbf{m}_i))^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$. Hence $\mathcal{Z}$ must LIA-satisfy $(\Lambda, L_i(\mathbf{m}_i))^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$. Equivalently, with $L_i \mid c_i$ being non-universal, $(\Lambda, L_i \mid c_i) \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ is μ_ℓ-satisfiable, which was to be shown.

Domain Split) The premise of Domain Split has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ while its conclusions have respectively the form $\Lambda \cdot (\Gamma, d) \vdash \Phi \cdot (\Psi, C \leftarrow c)$ and $\Lambda \cdot (\Gamma, \neg d) \vdash \Phi \cdot (\Psi, C \leftarrow c)$.

Suppose that $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi, C \leftarrow c$ is μ_ℓ-satisfiable. Then there must be a parameter assignment $\alpha \in \mathit{Mods}(\Gamma)$ such that $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup \Psi \cup (\Psi, C \leftarrow c)$ is

LIA-satisfiable in some Σ-interpretation $\mathcal{Z}$ that agrees with α on the parameters.

As $\mathcal{Z}$ agrees with α on the parameters, if $\alpha \models_{\mathcal{Z}} d$ then $\mathcal{Z}$ satisfies d. And if $\alpha \not\models_{\mathcal{Z}} d$ then $\alpha \models_{\mathcal{Z}} \neg d$ and so $\mathcal{Z}$ satisfies $\neg d$. Corresponding to the case that applies, it follows that one of the consequences is μ_ℓ-satisfiable.

3.9.2.2 Optional Rules

Inst Assert) The premise of Inst Assert has the form $\Lambda \cdot \Gamma \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi$ while its conclusion has the form $\Lambda \cdot \Gamma \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot (\Psi, C \vee L_0 \leftarrow d)$.

The reasoning is essentially the same as in the case for Inst, as d is a context unifier and thus a stronger constraint than c.

Assert) The premise of Assert has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ while its conclusion has the form $(\Lambda, L_i \mid c_i) \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$. Furthermore, $L_i \mid c_i$ is Γ-universal in $\Lambda \cdot \Gamma$, and $L_i \mid c_i$ is marked as universal in $(\Lambda, L_i \mid c_i) \cdot \Gamma$.

Suppose that $\Lambda \cdot \Gamma \vdash \Phi \cdot (\Psi, C \leftarrow c)$ is μ_ℓ-satisfiable. Then there must be a parameter assignment $\alpha \in \mathit{Mods}(\Gamma)$ such that $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$ is LIA-satisfiable in some Σ-interpretation $\mathcal{Z}$ that agrees with α on the parameters.

Let $C \leftarrow c$ be of the form $L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c[\mathbf{x}]$. Let $\mathbf{m}$ be an arbitrary α-solution of c. Since $L_1(\mathbf{m}_1) \vee \cdots \vee L_k(\mathbf{m}_k) \leftarrow c[\mathbf{m}]$ is an instance of a clause in Φ, it is satisfied by $\mathcal{Z}$.

For all $L_j(\mathbf{x}_j)$ of $C \leftarrow c$ with $i \neq j$ the following holds. Since $L_i(\mathbf{x}_i) \mid c_i$ is Γ-universal in $\Lambda \cdot \Gamma$, by Definition 3.4.14, $L_j(\mathbf{m}_j)$ is permanently α-falsified in $\Lambda \cdot \Gamma$. By Definition 3.4.13, there must be a literal $\overline{L_j}(\mathbf{x}_j) \mid e_j$ in Λ such that $\mathbf{m}_j$ is an α-solution of $\mathsf{perm}(e_j)$. By Definition 3.9.2, $L_j(\mathbf{m}_j)$ is falsified in $\mathcal{Z}$.

Since $\mathbf{m}$ was arbitrary, $\mathcal{Z}$ must then satisfy $L_i(\mathbf{m}_i)$ for all α-solutions of c_i.

In summary, as $\mathcal{Z}$ is a model of $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup (\Psi, C \leftarrow c)$ as well as of $L_i(\mathbf{x}_i) \leftarrow c_i$, and $\Lambda^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c) \cup \{L_i(\mathbf{x}_i) \leftarrow c_i\}$ is the same as $(\Lambda, L_i \mid c_i)^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup (\Psi, C \leftarrow c)$, $\mathcal{Z}$ is also a model of the latter.

Thus, from the premise $\Lambda \cdot \Gamma \ \vdash \ \Phi \cdot (\Psi, C \leftarrow c)$ being μ_ℓ-satisfiable it follows that the conclusion $(\Lambda, L_i \mid c_i) \cdot \Gamma \ \vdash \ \Phi \cdot (\Psi, C \leftarrow c)$ is μ_ℓ-satisfiable as well.

Ground Split) The proof is analogous to the proof of Domain Split.

3.9.2.3 Simplification Rules

Subsume) The premise of Subsume has the form $(\Lambda, L_0 \mid e) \cdot \Gamma \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi$

while its conclusion has the form $(\Lambda, L_0 \mid e) \cdot \Gamma \ \vdash \ \Phi \cdot \Psi$.

Thus any μ_ℓ-model of $(\Lambda, L_0 \mid e) \cdot \Gamma \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi$ is also a μ_ℓ-model of $(\Lambda, L_0 \mid e) \cdot \Gamma \ \vdash \ \Phi \cdot \Psi$, as the clause set $(\Lambda, L_0 \mid e)^{\mu_\ell(\alpha)} \cup \Gamma \cup \Phi \cup \Psi$ is a subset of $(\Lambda, L_0 \mid e)^{\mu_\ell(\alpha)} \cup \Gamma \cup (\Phi, C \vee L_0 \leftarrow c) \cup \Psi$, and thus any model of the former is also a model of the latter.

Resolve) The premise of Resolve has the form $(\Lambda, \overline{L_0} \mid e) \cdot \Gamma \vdash (\Phi, C(\mathbf{x}) \vee L_0 \leftarrow c) \cdot \Psi$ while its conclusion has the form $(\Lambda, \overline{L_0} \mid e) \cdot \Gamma \ \vdash \ (\Phi, C(\mathbf{x}) \leftarrow d) \cdot \Psi$, where $\Gamma \models_{\mathcal{Z}} c_0 \rightarrow \mathsf{perm}(\overline{L_0} \mid e)$ and $d = \pi \, \mathbf{x} \, c$.

Suppose that $(\Lambda, \overline{L_0} \mid e) \cdot \Gamma \ \vdash \ (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi$ is μ_ℓ-satisfiable. Then there must be a parameter assignment $\alpha \in \mathit{Mods}(\Gamma)$ such that $(\Lambda, \overline{L_0} \mid$

$e)^{\mu_\ell(\alpha)} \cup \Gamma \cup (\Phi, C \vee L_0 \leftarrow c) \cup \Psi$ is LIA-satisfiable in some Σ-interpretation $\mathcal{Z}$ that agrees with α on the parameters.

Let C consist of the literals $L_1(\mathbf{x}_1), \ldots, L_k(\mathbf{x}_k)$. From $\Gamma \models_{\mathcal{Z}} \bar{\forall}(c_0 \rightarrow \mathsf{perm}(\overline{L_0} \mid e))$ follows with $\alpha \in \mathit{Mods}(\Gamma)$ that $\bar{\forall}(c_0 \rightarrow \mathsf{perm}(\overline{L_0} \mid e))$ holds in $\mathcal{Z}$. By $\mathcal{Z}$ satisfying $\overline{L_0}(\mathbf{m})$ for each α-solution $\mathbf{m}$ of $\mathsf{perm}(e)$, $L_0(\mathbf{n}_0)$ is falsified in $\mathcal{Z}$ for each α-solution $\mathbf{n}_0$ of c_0. Furthermore, for each α-solution $\mathbf{n}$ of c the ground instance $L_1(\mathbf{n}_1) \vee \cdots \vee L_k(\mathbf{n}_k) \vee L_0(\mathbf{n}_0) \leftarrow c[\mathbf{n}]$ is satisfied in $\mathcal{Z}$. As for each α-solution $\mathbf{n}$ of c, $\mathbf{n}_0$ is an α-solution of c_0, it follows that $L_0(\mathbf{n}_0)$ is falsified in $\mathcal{Z}$ and one of $L_1(\mathbf{n}_1) \vee \cdots \vee L_k(\mathbf{n}_k)$ must be satisfied instead. As d is the projection of c to the variables of C not in L_0, it follows that $C \leftarrow d$ is true in $\mathcal{Z}$.

Thus any μ_ℓ-model of $(\Lambda, \overline{L_0} \mid e) \cdot \Gamma \vdash (\Phi, C \vee L_0 \leftarrow c) \cdot \Psi$ is also a μ_ℓ-model of $(\Lambda, \overline{L_0} \mid e) \cdot \Gamma \vdash (\Phi, C \leftarrow d) \cdot \Psi$.

Compact) The premise of Compact has the form $(\Lambda, L \mid e) \cdot \Gamma \vdash \Phi \cdot \Psi$ while its conclusion has the form $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$.

Analogous to the case of Subsume, it is immediate from Definition 3.9.2 that any μ_ℓ-model of the premise is also a μ_ℓ-model of the conclusion. □

Theorem 3.6.1 (Soundness) *For all admissible clause sets Φ and satisfiable sets of closed constraints Γ, if there is a refutation tree of Φ and Γ, then $\Phi \cup \Gamma$ is LIA-unsatisfiable.*

Proof. Let $\mathbf{T}$ be a refutation tree of Φ and Γ.

By the contrapositive of Lemma 3.9.3 it follows that if $\mathbf{T}$ is a refutation tree of Φ and Γ, then its root sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ must be μ_ℓ-unsatisfiable, as all its leaf node sequents S_i contain the constrained empty clause $\square \leftarrow \top$ in Ψ_i

and are thus trivially μ_ℓ-unsatisfiable.

By admissibility, the only literals in Λ are of the form $\neg P(\mathbf{x}) \mid -\mathbf{1} \dot{\leq} \mathbf{x}$. Thus for all $\alpha \in \mathit{Mods}(\Gamma)$ the α-solution of each ground literal that is permanently α-satisfied in $\Lambda \cdot \Gamma$ is of the form $-\mathbf{1}$. But by admissibility the solutions of each constrained clause in Φ are bounded below by $\mathbf{0}$. Thus the root sequent is μ_ℓ-satisfiable iff $\Phi \cup \Gamma$ is LIA-satisfiable.

The claim follows. □

3.9.3 Fairness

We will assume the same setup as in Section 3.6.2.

Lemma 3.6.5 *$\Gamma_{\mathbf{B}}$ is satisfiable, and for all $i < \kappa$, $\mathit{Mods}(\Gamma_{\mathbf{B}}) \subseteq \mathit{Mods}(\Gamma_i)$.*

Proof. First we show that Γ_i is satisfiable, for all $i < \kappa$. This however follows easily from the facts that, by definition, Γ_0 ($= \Gamma$) is satisfiable, and that the only derivation rules that can manipulate the Γ_i's, Domain Split and Ground Split, preserve satisfiability (and admissibility) in both conclusions. In fact, each Γ_i is finite and each Γ_{i+1} is obtained from Γ_i by adding at most one constraint. By compactness of $\Gamma_{\mathbf{B}}$, $\Gamma_{\mathbf{B}}$ as the union of all Γ_i is satisfiable as well.

Furthermore, as $\Gamma_{\mathbf{B}} = \bigcup_{i<\kappa} \Gamma_i$, $\mathit{Mods}(\Gamma_{\mathbf{B}}) \subseteq \mathit{Mods}(\Gamma_i)$. □

3.9.4 Completeness

We assume the same setup as in Section 3.6.3.

Lemma 3.9.4 *Let α be in $\mathit{Mods}(\Gamma_{\mathbf{B}})$. For any literal $L \mid c$ in Λ_j such that c is α-satisfiable, each $\Lambda_{j'}$ with $j' \geq j$ contains a literal which α-extends L.*

Proof. Let $L \mid c$ be an arbitrary literal in Λ_j such that c is α-satisfiable. First observe that because c is α-satisfiable, $L \mid c$ α-extends itself. Now assume that $S_{j'}$, with $j' > j$, is the first sequent such that $L \mid c$ is not in $\Lambda_{j'}$. This can only be because Compact was applied to the previous sequent. But then, by definition of Compact, there must be a literal $L \mid c'$ in $\Lambda_{j'}$ which α-extends $L \mid c$. □

As said, the following theorem is the main result for proving the calculus complete.

Theorem 3.6.7 (Model Construction) *For every $\alpha \in Mods(\Gamma_{\mathbf{B}})$, $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ is a model of $\Phi_{\mathbf{B}}$.*

Proof. Let α be arbitrarily in $Mods(\Gamma_{\mathbf{B}})$. Clearly, $\Lambda_{\mathbf{B}} \cdot \Gamma_{\mathbf{B}}$ is admissible and hence $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ is indeed defined. Suppose ad absurdum that $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$ is not a model of $\Phi_{\mathbf{B}}$. This implies the existence of a constrained clause $C[\mathbf{x}] = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow c[\mathbf{x}]$ from $\Phi_{\mathbf{B}}$ that is falsified by $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$. It holds $k \geq 1$ because the clause set Φ is admissible. That is, there is a vector $\mathbf{m}$ of constants from $\mathbb{Z}$ such that $C[\mathbf{m}]$ is falsified by $\mathcal{Z}_{\Lambda_{\mathbf{B}},\alpha}$. By Lemma 3.4.11 there must be context literals $\overline{L_1}(\mathbf{x}_1) \mid e_1$, $\ldots$, $\overline{L_k}(\mathbf{x}_k) \mid e_k$ in $\Lambda_{\mathbf{B}}$ which α-produce $\overline{L_1}(\mathbf{m}_1)$, $\ldots$, $\overline{L_k}(\mathbf{m}_k)$. By Lemma 3.9.4 it follows that there must be a $j < \kappa$ such that each $\Lambda_{j'}$, with $j' \geq j$, contains for each $\overline{L_i}(\mathbf{x}_i) \mid e_i$ a literal that α-extends $\overline{L_i}(\mathbf{x}_i) \mid e_i$.

We first show that the clause $D[\mathbf{x}] = L_1(\mathbf{x}_1) \vee \cdots \vee L_k(\mathbf{x}_k) \leftarrow d[\mathbf{x}]$, with d the context unifier of C and the above context literals, is not α-redundant in any $S_{j'}$:

(i) R-Unsatisfiable: D is α-satisfiable in $\Gamma_{\mathbf{B}}$, thus it cannot be R-Unsatisfiable α-redundant in any $S_{j'}$, with $j' \geq j$, as α is a model of each $\Gamma_{j'}$ by

Lemma 3.6.5.

(ii) R-Unproductive: If D is R-Unproductive α-redundant in some $S_{j'}$, with $j' \geq j$, then one of the context literals $\overline{L_i}(\mathbf{x}_i) \mid e_i$ does not α-produce $\overline{L_i}(\mathbf{x}_i) \mid d_i$ in $S_{j'}$. But this cannot be, as then by Definition 3.4.7 some $L_i(\mathbf{x}_i) \mid f_i$ in $\Lambda_{j'}$ blocks $\overline{L_i}(\mathbf{x}_i) \mid e_i$ from α-producing $\overline{L_i}(\mathbf{x}_i) \mid d_i$ in $\Lambda_{j'}$, and thus $\overline{L_i}(\mathbf{x}_i) \mid e_i$ does in particular not α-produce $\overline{L_i}(\mathbf{m}_i)$. By Lemma 3.9.4 some literal in $\Lambda_{\mathbf{B}}$ α-extends $L_i(\mathbf{x}_i) \mid f_i$, which implies that it in turn blocks $\overline{L_i}(\mathbf{x}_i) \mid e_i$ from α-producing $\overline{L_i}(\mathbf{m}_i)$ in $S_{\mathbf{B}}$. But this is a contradiction to the assumptions, and thus there is no such $L_i(\mathbf{x}_i) \mid f_i$ in $\Lambda_{j'}$ and D is not R-Unproductive α-redundant in $S_{j'}$.

(iii) R-Subsume: We can assume that D is not R-Subsume α-redundant in any $S_{j'}$, with $j' \geq j$. Because if it were due to some clause D' in $\Psi_{j'}$, then we would have chosen D' instead of D in our considerations. This is possible, as each literal of D' is α-extended by a literal of D, and thus (a subset of) the context literals $\overline{L_i}(\mathbf{x}_i) \mid e_i$ from above also falsifies the clause instance $D'[\mathbf{m}']$ of D', where $\mathbf{m}'$ is the projection of $\mathbf{m}$ to D''s variables. Furthermore, if D' is α-redundant in $S_{j'}$, then so is D. This is immediate for R-Unsatisfiable and R-Unproductive, and follows for R-Subsume by transitivity of the R-Subsume property. Furthermore, as the chain of replacements due to R-Subsume can be at most as long as the size of the longest clause, this replacement process is well-founded and terminates.

As Inst applies to S_j with selected clause C and derived clause D, by Definition 3.6.4-(ii) D is in $\Psi_{j'}$, for some $j' \geq j$. Let that j be that j' for the

remainder of the proof.

Together, we have now shown that,

$$\text{for all } j' \geq j,\ D \text{ is in } \Psi_{j'} \text{ and } D \text{ is not } \alpha\text{-redundant in } S_{j'}. \tag{3.2}$$

This property will lead to various contradictions below.

We perform a case analysis on the status of the literals of D in $\Lambda_j \cdot \Gamma_j$. Note that as $\Lambda_j \cdot \Gamma_j$ is admissible, it cannot be that a literal and its complement are α-contradictory with Λ_j for any $\alpha \in \mathit{Mods}(\Gamma_j)$.

1. All literals of D are Γ_j-contradictory with Λ_j. Then Close is applicable to S_j. But this is a contradiction to Definition 3.6.4-(i).

Thus, as $k \geq 1$, there must be some literal $L_i(\mathbf{x}_i) \mid d_i$ of D that is not Γ_j-contradictory with Λ_j. We will make use of $L_i(\mathbf{x}_i) \mid d_i$ in the remaining cases.

2. $\overline{L_i}(\mathbf{x}_i) \mid d_i$ is Γ_j-contradictory with Λ_j. Then Extend is applicable to S_j with selected clause D. By Definition 3.6.4-(iii) D is α-redundant in $S_{j'}$, for some $j' \geq j$. This is a direct contradiction to (3.2).

3. Neither $L_i(\mathbf{x}_i) \mid d_i$ nor $\overline{L_i}(\mathbf{x}_i) \mid d_i$ is α-contradictory with Λ_j. Here, Split is applicable to S_j with split literal $L_i(\mathbf{x}_i) \mid d_i$. Again, by Definition 3.6.4-(iii) D is α-redundant in $S_{j'}$, for some $j' \geq j$. This is a direct contradiction to (3.2).

4. $L_i(\mathbf{x}_i) \mid d_i$ or $\overline{L_i}(\mathbf{x}_i) \mid d_i$ is α-contradictory with Λ_j. Then there must be a literal $L_i(\mathbf{x}_i) \mid f_i$ or $\overline{L_i}(\mathbf{x}_i) \mid f_i$ in Λ_j such that $\alpha \models_{\mathcal{Z}} d_i \doteq_{\mu_\ell} f_i$, but not $\Gamma \models_{\mathcal{Z}} d_i \doteq_{\mu_\ell} f_i$. Thus Domain Split is applicable to S_j. Again, by Definition 3.6.4-(iii) D is α-redundant in $S_{j'}$, for some $j' \geq j$. This is a direct contradiction to (3.2).

In sum, each case has led to a contradiction now. Consequently, the assumption that $\mathcal{Z}_{\Lambda_\mathbf{B},\alpha}$ is not a model of $\Phi_\mathbf{B}$ is false, and the proof is complete. □

3.9.5 Proof-Generation

Lemma 3.9.5 *Let B be a derivation branch with root sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$ and N a node of B with sequent $S = \Lambda_S \cdot \Gamma_S \vdash \Phi_S \cdot \Psi_S$.*

Then for each clause $C(\mathbf{x}) \leftarrow d$ in Ψ_S there is a clause $C(\mathbf{x}) \vee D \leftarrow c$ in Φ, such that (i) d is stronger than the projection of c to the variables local to C, and (ii) for each literal L_i in D, $L_i \mid c_i$ is Γ_s-contradictory with Λ_S.

Proof. As $C(\mathbf{x}) \leftarrow d$ is the derived clause of an Inst application it must be an instance of some clause $C(\mathbf{x}') \leftarrow c'$ in Φ_S. Assume that the variables of the clauses are renamed such that $\mathbf{x}$ and $\mathbf{x}'$ are identical. If $C \leftarrow c'$ is in Φ then (i) and (ii) follow trivially, as D is empty and c is stronger than c'. Otherwise, $C \leftarrow c'$ must have been obtained from some $C \vee D \leftarrow c$ in Φ through applications of Resolve.

Observe that by definition of Resolve c' is obtained from c by existentially quantifying the variables local to D. With $C \leftarrow d$ being an instance of $C \leftarrow c'$, (i) follows.

Furthermore, if Resolve is applied to some sequent S_j with literal L_i in D, it follows by definition and with c' being a projection of c, that $L_i \mid c_i$ is Γ_{s_j}-contradictory with Λ_{S_j}. By Γ_S being an extension of all Γ_{S_j} used in resolving $C \vee D \leftarrow c$ to $C \leftarrow c'$, and by Lemma 3.9.4, (ii) follows. □

Lemma 3.7.6 *Let T be a refutation tree with root sequent $\Lambda \cdot \Gamma \vdash \Phi \cdot \Psi$. Fix*

an $\alpha \in \Gamma$. When Regress *is applied to T, it computes a label for each inner node such that each label is α-invariant.*

Proof. The proof will be by induction on the structure of the derivation tree, with the hypothesis that each label computed for a child node is α-invariant. The base cases are the sub-trees where Close was applied to the root node N with sequent S, all other rule applications make use of the induction hypothesis. The proof is by case analysis of the possible rule applications.

If the constraint c of a constrained clause in a label is not α-satisfiable the invariant holds trivially. We will consider only the cases where c is α-satisfiable.

Close Let Regress(N) be ($C \leftarrow c$, Link).

As $C \leftarrow c$ is an instance of a clause in Φ it is immediate that 3.7.6-(i) and (ii) hold. By construction of Link, 3.7.6-(iii) holds as well.

Inst, Extend, Domain Split, Inst Assert, Ground Split, Subsume, Resolve, Compact Immediate by construction, as the label of the child node is α-invariant by the induction hypothesis.

Split Assume the induction hypothesis holds for Regress(N_1) and Regress(N_2). If one of the trivial cases of the algorithm applies, i.e., it returns Regress(N_1) or Regress(N_2), the claim follows immediately. Otherwise, let Regress(N) be $(C \vee C')(\mathbf{x}) \leftarrow \pi\,\mathbf{x}\,(d \wedge d'), Link_N)$, let Regress($N_1$) be $(C \vee \overline{L_i} \leftarrow d$, $\mathsf{Link}_{N_1})$, and let Regress(N_2) be $(C' \vee L_i \leftarrow d'$, $\mathsf{Link}_{N_2})$.

Invariant 3.7.6-(i) and (ii) are immediate by hypothesis, as $(C \vee C')(\mathbf{x})$ $\leftarrow \pi\,\mathbf{x}\,(d \wedge d')$ is obtained by resolution on $C \vee \overline{L_i} \leftarrow d$ and $C' \vee L_i \leftarrow d'$. As $\overline{L_i} \mid d_i$

is α-contradictory with the (non-universal) left split literal $L_i \mid c_i$, and $L_i \mid d'_i$ is α-contradictory with the (non-universal) right split literal $\overline{L_i} \mid c_i$, it follows that d_i and d'_i have the same least α-solution. Thus $d \wedge d'$ is α-satisfiable. As the α-solutions of $\pi\,\mathbf{x}\,(d \wedge d')$ are exactly the α-solutions of d and d' joined over the variables local to d_i resp. d'_i, it follows that each literal of $(C \vee C')(\mathbf{x}) \leftarrow \pi\mathbf{x}(d \wedge d')$ has the same least α-solution as the corresponding literal of $C \vee \overline{L_i} \leftarrow d$ resp. $C' \vee L_i \leftarrow d'$. By construction of Link, 3.7.6-(iii) holds as well.

For the more complicated case in which one of the clauses returned by $\mathsf{Regress}(N_1)$ or $\mathsf{Regress}(N_2)$ are factored, observe that the literals factored must have the same least α-solution, as they are all α-contradictory with the same non-universal literal. It follows immediately by the hypothesis that by replacing in a label the original clause with its factored version, we obtain a label which is α-invariant as well.

Assert Let $L_i \mid c_i$ be the literal of the selected clause $C \vee L_i \leftarrow c$. Let $\mathsf{Regress}(N)$ be $(C \vee C' \leftarrow \pi\,\mathbf{x}\,(c \wedge c'), \mathsf{Link})$, and let $\mathsf{Regress}(N_1)$ be $(C' \vee \overline{L_i} \leftarrow c', \mathsf{Link}_{N_1})$.

As $L_i \mid c_i$ and $\overline{L_i} \mid c'_i$ are α-contradictory in N, $c \wedge c'$ is α-satisfiable. Invariant 3.7.6-(i) and (ii) are immediate by hypothesis, as $C \vee C' \leftarrow \pi\,\mathbf{x}\,(c \wedge c')$ is obtained by resolution on $C' \vee \overline{L_i} \leftarrow c'$ and a clause from Φ. By construction of Link, (iii) holds as well. $\square$

REFERENCES

[1] E. Althaus, E. Kruglov, and C. Weidenbach. Superposition Modulo Linear Arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *7th International Symposium on Frontiers of Combining Systems (FroCos 2009)*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 84–99, Trento, Italy, September 2009. Springer.

[2] O. Astrachan and M. Stickel. Caching and Lemmaizing in Model Elimination Theorem Provers. In *11th International Conference on Automated Deduction (CADE)*, pages 224–238, 1992.

[3] L. Bachmair, H. Ganzinger, and A. Voronkov. Elimination of Equality via Transformation with Ordering Constraints. In C. Kirchner and H. Kirchner, editors, *Automated Deduction — CADE 15*, LNAI 1421, Lindau, Germany, July 1998. Springer-Verlag.

[4] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational Theorem Proving for Hierachic First-Order Theories. *Appl. Algebra Eng. Commun. Comput*, 5:193–212, 1994.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. Mcgarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *In EuroSys 06: European Systems Conference*, pages 73–85, 2006.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. Draft, 2009.

[7] P. Baumgartner. Theory Reasoning in Connection Calculi. In *Lecture Notes in Artificial Intelligence*, volume 1527. Springer, 1998.

[8] P. Baumgartner. Logical Engineering with Instance-Based Methods. In F. Pfenning, editor, *CADE-21 – The 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 404–409. Springer, 2007.

[9] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic. In *Third Workshop on Disproving - Non-Theorems, Non-Validity, Non-Provability (DISPROVING'06)*, July 2006.

[10] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic. *Journal of Applied Logic*, 7(1):58–74, March 2009.

[11] P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin: A Theorem Prover for the Model Evolution Calculus. In S. Schulz, G. Sutcliffe, and T. 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.

[12] P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the Model Evolution Calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.

[13] P. Baumgartner, A. Fuchs, and C. Tinelli. Lemma Learning in the Model Evolution Calculus. In M. Hermann and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 4246 of *LNAI*, pages 572–586. Springer, 2006.

[14] P. Baumgartner, A. Fuchs, and C. Tinelli. ME(LIA) – Model Evolution With Linear Integer Arithmetic Constraints. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*, volume 5330 of *LNAI*, pages 258–273. Springer, November 2008.

[15] P. Baumgartner and R. Schmidt. Blocking and Other Enhancements for Bottom-Up Model Generation Methods. In U. Furbach and N. Shankar, editors, *Automated Reasoning – Third International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNAI*. Springer, 2006.

[16] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.

[17] P. Baumgartner and C. Tinelli. The Model Evolution Calculus with Equality. In R. Nieuwenhuis, editor, *CADE-20 – The 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 392–408. Springer, 2005.

[18] P. Baumgartner and C. Tinelli. The Model Evolution Calculus as a First-Order DPLL Method. *Artificial Intelligence*, 172(4-5):591–632, 2008.

[19] P. Baumgartner and U. Waldmann. Superposition and Model Evolution Combined. In R. Schmidt, editor, *CADE-22 – The 22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 17–34, Montreal, Canada, July 2009. Springer.

[20] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[21] M. Bezem. Disproving Distributivity in Lattices using Geometry Logic. In *Proc. CADE-20 Workshop on Disproving*, 2005.

[22] N. Bjoerner and L. M. de Moura. Z3[10]: Applications, Enablers, Challenges and Directions. In *Sixth International Workshop on Constraints in Formal Verification*.

[23] N. Bjoerner and L. M. de Moura. Satisfiability Modulo Theories: An Appetizer. In *Brazilian Symposium on Formal Methods (SBMF)*, 2009.

[24] M. P. Bonacina, C. Lynch, and L. M. de Moura. On Deciding Satisfiability by DPLL(Γ+T) and Unsound Theorem Proving. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2009.

[25] D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4:412–430, 1975.

[26] F. Bry and S. Torge. A Deduction Method Complete for Refutation and Finite Satisfiability. In *Proc. 6th European Workshop on Logics in AI (JELIA)*, LNAI. Springer, 1998.

[27] H. J. Bürckert. A Resolution Principle for Clauses with Constraints. In *10th International Conference on Automated Deduction (CADE)*, pages 178–192, 1990.

[28] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, , and E. Poll. An overview of JML tools and applications. *Software Tools For Technology Transfer*, 7(3), 2005.

[29] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[30] K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Building. In P. Baumgartner and C. G. Fermüller, editors, *CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications*, 2003.

[31] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence 7*, pages 91–99, 1972.

[32] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[33] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[34] H. de Nivelle and J. Meng. Geometric Resolution: A Proof Procedure based on Finite Model Search. In U. Furbach and N. Shankar, editors, *Proc. International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNAI*. Springer, 2006.

[35] C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer. Integrating Verification and Testing of Object-Oriented Software. In B. Beckert and R. Hähnle, editors, *Tests and Proofs. Second International Conference, TAP 2008, Prato, Italy*, LNCS 4966. Springer, 2008.

[36] L. Fix. Fifteen Years of Formal Property Verification in Intel. In *25 Years of Model Checking*, pages 139–144, 2008.

[37] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *International Conference in Computer Aided Verification (CAV)*, 2007.

[38] H. Ganzinger and K. Korovin. New Directions in Instantiation-Based Theorem Proving. In *LICS*, pages 55–64. IEEE Computer Society, 2003.

[39] H. Ganzinger and K. Korovin. Theory Instantiation. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06)*, volume 4246 of *LNAI*, pages 497–511. Springer, 2008.

[40] Y. Ge, C. Barrett, and C. Tinelli. Solving Quantified Verification Conditions using Satisfiability Modulo Theories. *Annals of Mathematics and Artificial Intelligence*, 2009. DOI 10.1007/s10472-009-9153-6.

[41] Y. Ge and L. M. de Moura. Complete instantiation for quantified SMT formulas. In *International Conference in Computer Aided Verification (CAV)*, 2009.

[42] M. L. Ginsberg and A. J. Parkes. Satisfiability Algorithms and Finite Quantification. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR'2000)*, pages 690–701. Morgan Kauffman, 2000.

[43] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *8th International Conference on Formal Methods in Computer-Aided Design*, 2008.

[44] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, Portland, OR, USA, 1996.

[45] K. Korovin. An Invitation to Instantiation-Based Reasoning: From Theory to Practice. In A. Podelski, A. Voronkov, and R. Wilhelm, editors, *Volume in memoriam of Harald Ganzinger*, LNCS. Springer, to appear. Invited paper.

[46] K. Korovin and A. Voronkov. Integrating Linear Arithmetic into Superposition Calculus. In *Computer Science Logic (CSL'07)*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

[47] S. Krstic and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In B. Konev and F. Wolter, editors, *6th international Symposium on Frontiers of Combining Systems (FroCos 2007)*, volume 4720 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.

[48] S. K. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *International Conference in Computer Aided Verification (CAV)*, 2004.

[49] R. Letz and G. Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 28, pages 2017–2114. Elsevier, 2001.

[50] R. Letz and G. Stenz. The Disconnection Tableau Calculus. *Journal of Automated Reasoning*, 38(1-3):79–126, 2007.

[51] D. Loveland. *Automated Theorem Proving - A Logical Basis*. North Holland, 1978.

[52] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[53] W. McCune. Mace 2.0 Reference Manual and Guide. Technical Report ANL/MCS-TM-249, Argonne National Laboratory, 2001.

[54] W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, 2003.

[55] W. McCune and L. Wos. Otter - The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[56] D. Monniaux. A Quantifier Elimination Algorithm for Linear Real Arithmetic. In *LPAR (Logic for Programming Artificial Intelligence and Reasoning)*, volume 5330 of *Lecture Notes in Computer Science*, pages 243–257. Springer Verlag, 2008.

[57] Moskewicz, Madigan, Zhao, Zhang, and Malik. Chaff: Engineering an Efficient SAT Solver. In *38th Design Automation Conference (DAC)*.

[58] R. Nieuwenhuis, A. Oliveras, and C. 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, Nov. 2006.

[59] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, pages 371–443. 2001.

[60] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier, 2001.

[61] P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

[62] S. Schulz. E - A Brainiac Theorem Prover. *AI Communications*, 15:111–126, 2002.

[63] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[64] A. Segre and C. Elkan. A High-Performance Explanation-Based Learning Algorithm. *Artificial Intelligence*, 69:1–50, 1994.

[65] J. Slaney. FINDER (Finite Domain Enumerator): Notes and Guide. Technical Report TR-ARP-1/92, Australian National University, Automated Reasoning Project, Canberra, 1992.

[66] N. Sörensson and N. Een. An extensible SAT-solver. In *SAT*, 2003.

[67] M. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.

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

[69] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[70] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs. Second International Conference, TAP 2008, Prato, Italy*, LNCS 4966. Springer, 2008.

[71] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In G. Ianni and S. Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.

[72] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '07)*, 2007.

[73] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. 1996.

[74] J. Zhang and H. Zhang. SEM: a System for Enumerating Models. In *IJCAI-95 — Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal*, pages 298–303, 1995.