Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond

Simon Cruanes

École Doctorale de l'École polytechnique

INRIA

THÈSE DE DOCTORAT

Présentée par

Simon Cruanes

Pour obtenir le grade de

DOCTEUR de l'ÉCOLE POLYTECHNIQUE

Spécialité : **Informatique**

Soutenue le **10 Septembre 2015**

Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond

Directeurs de thèse:

    M. Gilles DOWEK   Directeur de Recherche, Inria

    M. Guillaume BUREL   Maître de Conférence, ENSIIE

Rapporteurs:

    M. Stephan SCHULZ   Professeur, DHBW Stuttgart

    M. Uwe WALDMANN   Directeur de Recherche, MPII, Saarbrücken

Examinateurs:

    M. Nicolas PELTIER   Chargé de Recherche, CNRS

    M. Stéphane GRAHAM-LENGRAND   Chargé de Recherche, CNRS

    M. Sylvain CONCHON   Professeur, Université Paris-Sud

Contents

Acknowledgements

First, I wish to thank my directors, the unordered set {Gilles, Guillaume}, for their kind support for the last three years — even longer for Gilles as he used to be my teacher years ago. I also thank the reviewers of the thesis, for their careful reading of the manuscript and their helpful comments and feedback — I feel privileged that they devoted some of their time and expertise to reviewing my work —, and of course the jury for accepting to attend my defense. Before I started my thesis, I had the chance to learn from people from Proval (in particular Jean-Christophe and Andrei), and SRI (Shankar, Sam, Bruno...).

<center>(now switching to french...)</center>

Ensuite, je remercie évidemment mes parents et ma sœur, grâce auxquels j'ai acquis le goût de la lecture et ai eu une scolarité heureuse qui a finalement mené à cette thèse; leur soutien inconditionnel et leur aide ont été précieux pour la soutenance (merci aussi à mes oncles et tantes). Pauline a illuminé le temps que je n'ai pas passé à (cyber-)scribouiller des arbres de symboles. Merci également à mes amis du lycée (aux carrières artistiques ou médicales), de la prépa (dédicace trinôme de colle, bien évidemment : le chevelu et le mercenaire), et de l'X (μ, Shuba, et le reste du groupe fameux pour son intelligence collective; fufu, Anne, Nathaniel, #bll, et bien d'autres geeks; leurs +1, eeepc, etc.)

Deducteam dans son ensemble a été un endroit très agréable pour travailler et discuter science (ou libre, ou autres), ce grâce aux autres doctorants, stagiaires, et chercheurs. En particulier, Raphaël a eu la patience de m'écouter lors de nombreuses pauses café post-prandiales; Pierre Halmagrand pour son dealership efficace de capsules du breuvage sus-mentionné; Ali pour les trolls et Baklavas; Guillaume Bury pour les discussions techniques, en particulier celle où il m'a sauvé d'une calvitie fulgurante liée à une preuve arithmétique épineuse; David pour l'évocation de la super^{2}position; Arnaud m'a encouragé à procrastiner; et tous les autres !

<center>(and back to english...)</center>

Last, I want to thank the authors of the various free software I have been using. In particular, in no specific order: LaTeX(which allowed me to write this document and other papers), vim, linux, git (for my peace of mind), OCaml, merlin, etc. TPTP and its nice maintainer, Geoff Sutcliffe, were also both very helpful.

Chapter 1

Introduction

Logic and Proofs in Computer Science

Logic, the language and art of formal reasoning, is very useful in both Computer Science and Mathematics. Both require correct, unambiguous argumentation to support claims; the central concept of *theorem* precisely designates a claim backed by such an irrefutable argument, called a *proof*. This focus on formal proofs is quite characteristic of Mathematics — a notable exception is the project of Leibniz to design a "calculus ratiocinator" that would make formal, unambiguous reasoning the norm. However, proofs have a major drawback: it is in general very difficult to find a proof to support a given claim. Human experts (usually called "mathematicians" or "logicians") are undoubtedly the best at finding proofs; even more so when the problem is about finding *elegant* proofs. On the other hand, many theorems are truly boring. For instance, theorems generated to ensure that software abides by some specification are neither elegant nor fun to prove. Programs able to discharge automatically those *proof obligations* are therefore quite useful in practice, even though they probably will not be able to prove hard theorems (e.g., the Goldbach's conjecture) in the foreseeable future. The study of programs that (try to) prove theorems is *automated theorem proving*.

From Resolution to Superposition

Automated theorem proving has been an active field of research ever since the 1960s. Within this discipline, first-order logic plays an important rôle, as it occupies a sweet spot in the trade-off between having nice computational properties — as in the case of propositional logic —, and featuring a high level of expressiveness — the climax being arguably reached by the higher-order, dependently typed logics usually found in proof assistants such as Coq [CDT]. Focusing on first-order logic, we can admire a quite diversified ecosystem of calculi; among them, Resolution [Rob65] and its offspring, Superposition [BG90, NR99] — which adds good reasoning about equality over uninterpreted functions —, have benefited from decades of theoretical improvements and implementation efforts in various languages. Nowadays, Superposition-based theorem provers [RV01b, Sch02, WSH$^+$07] are very competitive in the first-order case.

Superposition is not enough

Even then, mere Superposition falls short for many applications: some may require some arithmetic reasoning, some may be heavy with specific algebraic theories, some may need inductive reasoning to reason on inductive structures — in practice, those abound in programming, Mathematics, etc. Extending Superposition has been an active research domain, going back to handling the theory of Associative Commutative symbols — we might say it culminated

with the proof of the Robbins conjecture by the automated theorem prover EQP [McC97]. Recently, an extension called AVATAR [Vor14] proposes to interface a SAT-solver within Superposition, so as to delegate propositional reasoning to it. Another extension, Hierarchic Superposition [BGW94, BW13], adds a background theory solver — for instance a linear arithmetic solver — to Superposition, in order to reason modulo that particular theory.

In this thesis, we aimed at developing new extensions to Superposition. Our claim is that Superposition lends itself very well to being grafted additional inference rules and reasoning mechanisms, mostly *remaining in a clausal saturation framework*. Saturation, that Superposition shares with its ancestor, Resolution, possess many interesting properties for reasoning at the first-order level (as opposed to boolean-level reasoning, found in Hierarchic Superposition): when a clause is deduced, it can be used several times, making proofs DAGs by sharing sub-proofs; in addition, using free variables (implicitly quantified at the clause level) leverages unification to efficiently find relevant instances of terms. Developing extensions as deductive inferences on first-order clauses allow us to deduce new quantified truths even in the presence of theories, or in a more powerful logic (inductive logic).

The importance of Implementation

Automated theorem proving is theoretically solved: the space of proofs is recursively enumerable, so a program that enumerates the possible proofs and checks whether they are a proof of F is a valid procedure to try to prove F. This method is critically inefficient, for several reasons: (i) it enumerates all the uninteresting theorems, for instance every instance of $\underbrace{(A \wedge A \wedge \ldots \wedge A)}_{m \text{ times}} \Rightarrow \underbrace{(A \vee A \vee \ldots \vee A)}_{n \text{ times}}$ for $(m, n) \in \mathbb{N}^+ \times \mathbb{N}$; (ii) it does not use the goal to guide its search. In practice, decades of research have been dedicated to studying algorithms that behave less stupidly on actual theorems. This makes automated theorem proving both an experimental and a theoretical domain. Our work is oriented towards prototyping and experimentation; each extension we built has its own implementation in Zipperposition, a Superposition-based theorem prover developed for this very purpose. A chapter of the thesis is dedicated to presenting Zipperposition, its implementation, as well as a foundational logic library called Logtk.

Organization of this Thesis

Our main contributions, in addition to the pure implementation work, are threefold; consequently, they are detailed in three separate chapters (Chapters 4, 5, 6). The organization of this thesis is:
- In Chapter 2, preliminary mathematical and logic notions are defined, and their notations, introduced. The Superposition and AVATAR calculi are also presented. After this chapter, the reader should have a clear idea of the notions required to understand the next chapters.
- Chapter 3 focuses on the implementation part of the three years we worked on this thesis. It starts by presenting Logtk, a general-purpose OCaml library for representing types, terms, formulas, etc. — notions mathematically defined in Chapter 2 — in addition to a collection of classic algorithms such as unification, CNF transformation, or term indexing. Then, Zipperposition, a theorem prover we built upon Logtk, is introduced. This chapter is not a lecture in the implementation of automated theorem provers; it only underlines some issues pertaining to writing programs that search for proofs.
- In Chapter 4, a Superposition-based calculus for integer linear arithmetic (also called Presburger arithmetic), inspired by the work of Waldmann on combining Superposition with rational arithmetic [Wal01]. Linear Integer Arithmetic is a widely studied and used

theory in other areas of automated deduction, in particular SMT (Satisfiability Modulo Theory).

Linear arithmetic problems are often encountered in automated proving, whether it be directly for program verification, or asserting coherence of compiler optimizations. Indeed, most programs use built-in arithmetic, and often can be formalized in linear arithmetic. A compiler might, for instance, meet the following snippet of C code:

```
for (i=1; i≤10; i++) a[j+i]=a[j];
```

The compiler may be interested in loading the value of `a[j]` once before entering the loop, rather than loading it repeatedly inside the loop, since memory access is usually slow. However, to ensure this optimization is safe, the compiler must assert that the value of `a[j]` does not change within the loop. One way to do so is to prove that there is no index collision in the loop, which can be formalized by proving the arithmetic formula $\forall i \in \mathbb{Z}.\ 1 \le i \le 10 \Rightarrow j \ne j + i$. In addition to pure arithmetic reasoning or computations (including program verification), other problems that have a discrete, totally ordered structure, such as temporal logic, might be encoded into first-order logic with arithmetic efficiently.

- In Chapter 5, we define an extension of Superposition+AVATAR that is able to reason by structural induction on natural numbers, lists, binary trees, etc. Induction is attractive because a *local* reasoning (prove that $P(0)$ holds, and that if P is true on n, then P is true on $n + 1$) allows to prove universal properties ($\forall n : \mathbb{N}.\ P(n)$: the property P holds on all natural numbers). Again, inductive reasoning is extremely prevalent in Computer Science, logic, and programming; yet the two realms of first-order theorem provers and inductive provers are still mostly separate. Whereas specialized inductive provers such as Spike [BKR92, Str12] are very successful in the latter, they do not shine in the former. We try here to bridge the gap from the opposite side.

- Chapter 6 is dedicated to a theory detection system that, given a signature-agnostic description of algebraic theories, detects their presence in sets of formulas. Its integration in Zipperposition can also detect specific inductive theories (such as the Peano axioms for natural numbers, when presented as an inductive type). An early version of this work was published in [BC13].

Each chapter is relatively self-contained — common definitions and techniques from the state of the art are first listed in Chapter 2. Readers interested only in one chapter might then read it directly after Chapter 2.

Chapter 2

Technical Preliminaries

We start with a gentle introduction to the mathematical concepts and basics of Logic that everything else in this thesis is built on top of. Everything takes place in a *classical* setting, meaning the principle of *excluded middle* (for any proposition p, $p \vee \neg p$ holds) is available for the theorem prover to use.

Remark 2.1 (Definitional Equality). *In this thesis, $a \stackrel{def}{=} b$ means that a is equal to b by definition of a. We often define new variables this way.*

2.1 Mathematical Concepts

We use some very classic mathematical notions; in particular, we assume the reader knows about sets. $a \in b$ means that a is a member of the set b. Set comprehension is noted $\{x \in a \mid p(x)\}$ — the set of all x that are members of a and satisfy property p —; the cardinal $\#s$ of a finite set s is the number of elements it contains.

Definition 2.1 (Natural Numbers). *The natural numbers are the positive integers $\{0, 1, 2, \ldots\}$. The set of all natural numbers is denoted $\mathbb{N}$, and the set of strictly positive natural numbers is $\mathbb{N}^+ \stackrel{def}{=} \mathbb{N} \setminus \{0\}$.*

Definition 2.2 (Integers). *The set of integers $\{\ldots, -1, 0, 1, 2, \ldots\}$ is denoted $\mathbb{Z}$.*

Definition 2.3 (Multiset). *A multiset is a collection of objects, like a set, but in which each item can occur several times. More formally, a multiset is a function M from a set S (called the* carrier *of M) to $\mathbb{N}$; an element $x \in S$ has multiplicity i iff $M(x) = i$. We say x belongs to M, or $x \in M$, iff $M(x) \geq 1$. The union operator $\cup$, defined by $(M_1 \cup M_2)(x) \stackrel{def}{=} M_1(x) + M_2(x)$, is often useful. The* support *of a multiset M is the subset of S that have a strictly positive multiplicity. We will only consider finite multisets, that is, multisets whose support is a finite set. In the rest of this thesis, we will generally use set-like notations for multisets.*

Definition 2.4 (Order). *An order is a binary relation $\leq$ such that the following axioms hold:*
Reflexivity *: $\forall x.\ x \leq x$;*
Transitivity *: $\forall x\ y\ z.\ x \leq y \wedge y \leq z \Rightarrow x \leq z$;*
Antisymmetry *: $\forall x\ y.\ x \leq y \wedge y \leq x \Rightarrow x = y$.*

Definition 2.5 (Strict Order). *A strict order is a binary relation $<$ satisfying:*
Irreflexivity *: $\forall x.\ x \not< x$;*
Transitivity *: $\forall x\ y\ z.\ x < y \wedge y < z \Rightarrow x < z$.*

Definition 2.6 (Well-founded Order). *A well-founded order is a strict order $<$ such that there is no infinite sequence $x_1, x_2, \ldots$ such that $\forall i.\ x_{i+1} < x_i$. We might speak of well-founded relations when the relation admits no infinite sequence.*

Definition 2.7 (Partial Order, Total Order)**.** *A strict order $<$ is* total *(up to some equality relation $=$) if, for any two objects x, y, either $x = y$, or $x < y$, or $y < x$; otherwise, the order is* partial. *A non-strict order $\leq$ is total if, for any two objects x, y, either $x \leq y$ or $y \leq x$; otherwise it is partial.*

Remark 2.2 (Termination)**.** *Sometimes, if a transitive relation $\rightarrow$ (intuitively, a rewrite relation) is well-founded, we might say it is* terminating.

Definition 2.8 (Lexicographic Combination)**.** *The lexicographic combination of two strict order $<_1, <_2$, also noted $(<_1, <_2)_{lex}$, is a strict order defined on pairs by*

$$(x_1, x_2)(<_1, <_2)_{lex}(y_1, y_2) \ if \begin{cases} x_1 <_1 y_1, \ or \\ x_1 = y_1 \ and \ x_2 <_2 y_2 \end{cases}$$

Definition 2.9 (Multiset Extension of an Order)**.** *The* multiset extension *of an order $\succ$, noted $\gg$, is defined [DM79] by $M \ll N$ iff there exist multisets X, Y such that:*
- *$X \subseteq N$;*
- *$X \neq \emptyset$;*
- *$M = (N - X) \cup Y$;*
- *X dominates Y, that is, $\forall y \in Y. \exists x \in X. \ x \succ y$*

where $M - N$ is the multiset defined by $(M - N)(x) = M(x) - N(x)$.

If $\succ$ is well-founded, so is $\gg$; if $\succ$ is total, so is $\gg$.

Definition 2.10 (Transitive Closure)**.** *The* transitive closure *of a binary relation $\rightarrow$, noted $\rightarrow^+$, is defined inductively by*
- *if $u \rightarrow v$, then $u \rightarrow^+ v$;*
- *if $u \rightarrow v$ and $v \rightarrow^+ w$, then $u \rightarrow^+ w$.*

Definition 2.11 (Transitive Reflexive Closure)**.** *The* transitive reflexive closure *of a binary relation $\rightarrow$, noted $\rightarrow^*$, is defined by*
- *for any u, $u \rightarrow^* u$;*
- *if $u \rightarrow^+ v$, then $u \rightarrow^* v$.*

Definition 2.12 (Confluence)**.** *A relation $\rightarrow$ is* confluent *if, for any x, y, z such that $x \leftarrow^* y \rightarrow^* z$, there exists some w such that $x \rightarrow^* w \leftarrow^* z$.*

Definition 2.13 (Normal Form)**.** *An object x is a* normal form *for a relation $\rightarrow$ if there is no y such that $x \rightarrow y$. We say y is a normal form of x if $x \rightarrow^* y$ and y is a normal form for $\rightarrow$.*

If $\rightarrow$ is confluent, there is at most one normal form for each object; if $\rightarrow$ is terminating, then every object has a normal form.

Definition 2.14 (Directed Acyclic Graph (DAG))**.** *A directed acyclic graph, or DAG, is a collection of vertices V and edges $E \subseteq V \times V$ (each edge connects two vertices), such that there is no cycle; that is, there is no sequence of edges $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$ such that $v_n = v_1$.*

Definition 2.15 (Smaller Part of a Set)**.** *We denote $N^{<x}$ the set of elements of N that are smaller than x w.r.t. the well-founded order $<$.*

We denote $N^{\leq x}$ the set of elements of N that are smaller than x w.r.t. the order $\leq$ (including x if $x \in N$).

2.2   Boolean Logic

Boolean logic (or *propositional logic*) is the simplest form of logic widely used. It is simple enough that even computers can easily deal with it[1]. It deals with statements, or formulas, built on top of atomic propositions (*atoms*) that only carry a "true" or "false" value, without any further structure.

---

[1] Decision in propositional logic is NP-complete, but this is still much better than semi-decidability!

Definition 2.16 (Boolean Atoms)**.** *A* boolean atom *(or* boolean variable*) a is a variable that can take only two values (true or false). It represents a single atomic proposition that can be either valid or invalid.*

Definition 2.17 (Boolean Formula)**.** *A boolean formula is inductively defined as follows:*

- 1 *and* 0 *are boolean formulas (for truth and falseness respectively);*
- *Any boolean variable is a boolean formula*
- *if F is a boolean formula, then $\neg F$ (its negation) is too;*
- *if F and G are boolean formulas, then $F \sqcap G$ (conjunction), $F \rightarrow G$ (implication), $F \sqcup G$ (disjunction), $F \oplus G$ (exclusive disjunction) are boolean formulas.*

Typically, $\sqcap$ and $\sqcup$ are associative, with $\sqcap$ binding more strongly: $a \sqcap b \sqcup c$ is $(a \sqcap b) \sqcup c$. The other operators bind less strongly.

Definition 2.18 (Boolean Clause)**.** *A boolean clause is a finite disjunction of* literals, *where a literal is either an atom b or its negation $\neg b$. All clauses are formulas, but not all formulas are clauses.*

Definition 2.19 (Boolean Valuation)**.** *A* boolean valuation *is a function v that maps every boolean variable b in the current signature to $v(b) \in \{1, 0\}$. It extends to boolean formulas and clauses using the following* truth table*; given the valuation of F and G, in the two first columns, the other columns define the valuation of $\neg F$ and of $F \circ G$ for the every operator $\circ \in \{\sqcup, \sqcap, \rightarrow, \oplus\}$:*

$v(F)$	$v(G)$	$v(\neg F)$	$v(F \sqcup G)$	$v(F \sqcap G)$	$v(F \rightarrow G)$	$v(F \oplus G)$
0	0	1	0	0	1	0
0	1	1	1	0	1	1
1	0	0	1	0	0	1
1	1	0	1	1	1	0

Definition 2.20 (Comprehension-Style Formulas)**.** *We also use comprehension-style versions of some of those operators that can be defined using $\sqcup$, $\sqcap$ and $\neg$. Those operate on a finite set S of boolean formulas, of cardinality n, and are themselves formulas.*

- $\bigsqcup_{i=1}^{n} F_i$ *is an n-ary disjunction, valued to 1 iff at least one F_i is. $\bigsqcup_{i=1}^{n} F_i$ expands to $F_1 \sqcup F_2 \sqcup \ldots \sqcup F_n$;*
- $\bigsqcap_{i=1}^{n} F_i$ *is an n-ary conjunction (valued to 1 iff all F_i are). $\bigsqcap_{i=1}^{n} F_i$ expands to $F_1 \sqcap F_2 \sqcap \ldots \sqcap F_n$;*
- $\bigoplus_{i=1}^{n} F_i$ *is an n-ary exclusive disjunction, valued to 1 iff exactly one F_i is. $\bigoplus_i F_i$ expands to $\bigsqcup_i \left(F_i \sqcap \bigsqcap_{j \neq i} \neg F_j \right)$ — note that $\bigoplus_{i=1}^{n} F_i$ is not the same as $F_1 \oplus F_2 \oplus \ldots \oplus F_n$.*

Formulas defined by comprehension over a finite set, such as $\bigsqcup_{F \in S} F$ (disjunction of all formulas in S), will also be used.

Definition 2.21 (SAT, SAT solver)**.** *The boolean satisfiability problem, usually called* SAT*, consists in finding whether a given boolean formula F is satisfied by at least one valuation, and returning such a valuation if it exists. SAT is well known to be NP-complete [Coo71]. A* SAT*-* solver *is a program that inputs F (usually in the form of a set of boolean clauses) and outputs one of $\{unsat, sat(v)\}$ where $F(v) = 1$. It returns* unsat *only in case no valuation satisfies F.*

Definition 2.22 (Quantified Boolean Formula (QBF))**.** *A* quantified boolean formula *(or* QBF*) is defined as $Q_1 x_1. Q_2 x_2. \ldots Q_n x_n. F$ where F is a boolean formula, $\{x_1, \ldots, x_n\}$ is the set of all boolean variables that occur in F, and every quantifier Q_i is either $\exists$ or $\forall$.*

Definition 2.23 (QBF Evaluation). *A QBF can be evaluated into a boolean[2] by the rewrite system $\hookrightarrow_{QBF}$ defined as follows (terminating).*

$$\exists x.\, F \quad \hookrightarrow_{QBF} \quad 1\,[x \leftarrow F] \sqcup 0\,[x \leftarrow F]$$
$$\forall x.\, F \quad \hookrightarrow_{QBF} \quad 1\,[x \leftarrow F] \sqcap 0\,[x \leftarrow F]$$

Evaluating QBFs is known to be PSPACE-complete. A QBF solver *is a program that inputs a QBF F and outputs* 1 *or* 0.

It is clear that a QBF can be transformed into a regular boolean formula using the rewrite system $\hookrightarrow_{QBF}$. Why bother with QBF then? The point of QBF is that it allows some boolean formulas to be expressed in an exponentially smaller form, in the hope that QBF solvers will work with such "compressed" formula more efficiently than by just unfolding them and calling a SAT solver.

Example 2.1 (QBF). $\forall a.\, \exists b.\, \forall c.\, ((a \sqcup b) \sqcap (c \sqcup \neg b))$ *is a false QBF.*

Boolean logic has been a popular research topic for decades, and many breakthroughs have made modern SAT solvers very efficient on some kinds of "realistic" problems (excluding random formulas and prime factorization — good news for cryptography). This motivates the use of SAT solvers in other areas of logic; in particular, AVATAR (Section 2.5) successfully combines a SAT solver with a first-order Superposition prover.

2.3 First-Order logic

The first-order logic is the target language for theorem provers we are interested in. It is already expressive enough that provability is semi-decidable [Chu36], yet it remains somehow tractable for automated tools. Compared to higher-order logic, a critical difference is the decidability of unification. This alone makes resolution [Rob65] and Superposition possible and quite efficient, whereas resolution for higher-order logic [BTPF08], for instance, needs inference steps just to perform unification, since this operation is undecidable.

Every construct in first-order logic is built upon *symbols* and *variables*. They will be used in types, terms, and formulas. We assume the existence of infinite countable sets of variables.

2.3.1 Types

As we chose, for several reasons, to use *typed* logic, it is only natural that we first define what types are. The main purpose of types is, fundamentally, to restrict the set of terms a variable can be replaced with: a boolean variable can only be instantiated with true, false, or other boolean variables; a variable of type int will be instantiated only with integers or arithmetic expressions of type int.

Definition 2.24 (Type Constructor). *A* type constructor *is a symbol associated with a natural number called its* arity. *The arity of a type constructor is the number of arguments it admits —* nullary *means arity 0,* unary *means arity 1,* binary *means arity two,* ternary *means arity 3, and n-*ary *means arity n.*

Definition 2.25 (Type Signature). *A* type signature Σ_τ *is a set of type constructors such that no symbol occurs in it twice.*

---

[2] In any other case, the formula is a regular non-quantified formula, that cannot contain any variable, since they must be all quantified in a QBF. Consequently, the formula contains only connectives and $\{1, 0\}$, which makes it easy to evaluate.

Definition 2.26 (Type and Atomic Type)**.** *Given a type signature Σ_τ and an infinite countable set of variables $\mathscr{A}$, we define the set of* atomic types $ATypes(\Sigma_\tau)$ *and the set of* types $Types(\Sigma_\tau)$ *inductively by:*

- $\alpha \in \mathscr{A}$ *implies* $\alpha \in ATypes(\Sigma_\tau)$. α *is called a* type variable*;*
- $c \in \Sigma_\tau$ *with arity* $n \in \mathbb{N}$ *and* $\tau_1, \ldots, \tau_n \in ATypes(\Sigma_\tau)$ *imply* $c(\tau_1, \ldots, \tau_n) \in ATypes(\Sigma_\tau)$. *We say we* apply *the type constructor c to the types $\tau_1, \ldots, \tau_n$;*
- $(\tau_1, \ldots, \tau_n, \tau) \in ATypes(\Sigma_\tau)$ *implies* $(\tau_1 \times \ldots \times \tau_n) \to \tau \in Types(\Sigma_\tau)$. *We call $(\tau_1 \times \ldots \times \tau_n) \to \tau$ a* function type, *or* arrow type*;*
- $ATypes(\Sigma_\tau) \subseteq Types(\Sigma_\tau)$ *(all atomic types are types);*
- $\alpha \in \mathscr{A}$ *and* $\tau \in Types(\Sigma_\tau)$ *imply* $\Pi\alpha.\ \tau \in Types(\Sigma_\tau)$. *This is a* polymorphic type.

Remark 2.3 (Currying)**.** *Some functional languages, and higher-order type theory, tend to use only binary arrow types — all arrows have the form $\tau_1 \to \tau_2$ — and binary applications. However, in the first-order realm, there is no notion of partial application; that makes currying useless. In chapter 6 we will introduce some curried higher-order terms.*

Remark 2.4 (Notations, ι, o)**.** *In the rest of the thesis, we will typically denote type variables as α, β, γ and types as τ, τ_1, etc. We assume given the two nullary constructors ι (the default type of terms) and o (the type of propositions). In first-order logic, o cannot appear as a type argument of a function type; that is, $(\tau_1 \times \ldots \times \tau_n) \to \tau$ where some $\tau_i = o$ is not a valid first-order type.*

Definition 2.27 (Variables of a Type)**.** *The set of variables of a type τ, noted $vars(\tau)$, is defined inductively as*

$$
\begin{aligned}
vars(\alpha) &= \{\alpha\}\ \textit{if}\ \alpha \in \mathscr{A} \\
vars(c(\tau_1, \ldots, \tau_n)) &= \bigcup_{i=1}^{n} vars(\tau_i) \\
vars((\tau_1 \times \ldots \times \tau_n) \to \tau) &= vars(\tau) \cup \bigcup_{i=1}^{n} vars(\tau_i) \\
vars(\Pi\alpha.\ \tau) &= vars(\tau) \cup \{\alpha\}
\end{aligned}
$$

Definition 2.28 (Free Variables of a Type)**.** *The set of* free variables *of a type τ, noted $freevars(\tau)$, is defined by induction on the structure of τ:*

$$
\begin{aligned}
freevars(\alpha) &= \{\alpha\}\ \textit{if}\ \alpha \in \mathscr{A} \\
freevars(c(\tau_1, \ldots, \tau_n)) &= \bigcup_{i=1}^{n} freevars(\tau_i) \\
freevars((\tau_1 \times \ldots \times \tau_n) \to \tau) &= freevars(\tau) \cup \bigcup_{i=1}^{n} freevars(\tau_i) \\
freevars(\Pi\alpha.\ \tau) &= freevars(\tau) \setminus \{\alpha\}
\end{aligned}
$$

Definition 2.29 (Closed Type)**.** *A type τ is* closed *if $freevars(\tau) = \emptyset$; it can still contain bound variables.*

Definition 2.30 (Ground Type)**.** *A type τ is* ground *iff $vars(\tau) = \emptyset$; it contains no free variables nor quantified type variables. Note that a ground type is always closed.*

Example 2.2 (Atomic, Closed and Ground Types)**.** $\Pi\alpha.\ \alpha \times list(\alpha) \to list(\alpha)$ *is closed, but not ground nor atomic.* $list(list(nat))$ *is closed, ground and atomic.* $\iota \to o$ *is closed and ground, but not atomic.*

Definition 2.31 (Type Substitution)**.** *A* type substitution *is a finite injective mapping $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ from type variables to types. The set of variables mapped by σ is the* domain *of σ, denoted $dom(\sigma) \overset{def}{=} \{\alpha_1, \ldots, \alpha_n\}$. A substitution σ can be applied to a type τ (denoted $\tau\sigma$) as follows:*

$$
\begin{aligned}
\alpha_i \sigma &= \tau_i & &\textit{if}\ \sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} \\
\beta\sigma &= \beta & &\textit{otherwise, if}\ \beta \notin dom(\sigma) \\
c(\tau_1, \ldots, \tau_n)\sigma &= c(\tau_1\sigma, \ldots, \tau_n\sigma) \\
((\tau_1 \times \ldots \times \tau_n) \to \tau)\sigma &= (\tau_1\sigma \times \ldots \times \tau_n\sigma) \to \tau\sigma \\
(\Pi\alpha.\ \tau)\sigma &= \Pi\beta.\ \left((\tau\{\alpha \mapsto \beta\})\sigma \right) & &\textit{where}\ \beta\ \textit{is a fresh variable}
\end{aligned}
$$

Remark 2.5 (α-equivalence). *We define syntactic equality on terms, $\tau_1 = \tau_2$, modulo alpha-equivalence. In other words, $\tau_1 = \tau_2$ iff there is a renaming of bound variables in τ_1 that yields a term syntactically identical to τ_2.*

2.3.2 Terms

Now that we have defined types, we can define *terms*. Terms represent objects of discourse; first-order logic is all about expressing and proving properties of terms. We will use the built-in types from Remark 2.4.

Definition 2.32 (Signature). *Given a type signature Σ_τ, a signature Σ is a finite set of symbols along with closed types, with the following restrictions:*
- *types in Σ must have the form $\Pi\alpha_1 \ldots \alpha_m. \, (\tau_1 \times \ldots \times \tau_n) \to \tau$ where $\tau_1, \ldots, \tau_n$ are atomic types distinct from o, and τ is an atomic type (possibly o, see Remark 2.4) — otherwise, we would enter higher order logics. If $m = 0$ the symbol is* monomorphic *or* simply-typed, *otherwise it is* polymorphic; *if $n = 0$ the symbol is a* constant, *if $\tau = o$ it is a* predicate, *and otherwise it is a* function;
- *if $(f, \tau) \in \Sigma$, meaning that the symbol f has type τ, no other pair (f, τ') can appear in Σ; in other words, as a relation, Σ is functional.*

The rôle of a signature is to define which symbols can be used to build terms, and what their type is. This will be useful to define well-typed terms.

Definition 2.33 (Term). *Given a signature Σ and a countably infinite set of variables $\mathcal{X}_\tau$ for each type $\tau \in \mathrm{Types}(\Sigma_\tau)$, the set of terms $\mathrm{Terms}(\Sigma)$ is defined inductively by:*
- *$x \in \mathcal{X}_\tau$ implies $x \in \mathrm{Terms}(\Sigma)$. We will sometimes write x^τ to indicate that x has type τ, sometimes $x : \tau$, and sometimes we will omit type annotations altogether;*
- *$(f, \tau) \in \Sigma$, $(\tau_1, \ldots, \tau_m) \in \mathrm{Types}(\Sigma_\tau)$ and $(t_1, \ldots, t_n) \in \mathrm{Terms}(\Sigma)$ imply $f_{\langle \tau_1, \ldots, \tau_m \rangle}(t_1, \ldots, t_n) \in \mathrm{Terms}(\Sigma)$. We add restrictions on types just below.*

Definition 2.34 (Well-Typed Term). *A term is* well typed *if it has some type according to the following set of type rules, where $\Sigma \vdash t : \tau$ is the judgement that t has type τ in the signature Σ:*

$$\frac{x \in \mathcal{X}_\tau}{\Sigma \vdash x : \tau}$$

$$\frac{(f, \Pi\alpha_1 \ldots \alpha_m. \, (\tau_1 \times \ldots \times \tau_n) \to \tau) \in \Sigma \quad \tau_i' \in \mathrm{Types}(\Sigma_\tau) \quad \Sigma \vdash t_i : \tau_i \sigma}{\Sigma \vdash f_{\langle \tau_1', \ldots, \tau_m' \rangle}(t_1, \ldots, t_n) : \tau \sigma}$$

$$where \; \sigma \overset{def}{=} \{ \alpha_1 \mapsto \tau_1', \ldots, \alpha_m \mapsto \tau_m' \}$$

Remark 2.6 (Notations). *From now on, when we speak of terms, we assume they are well-typed. We will use s, t, u, v to denote (well-typed) terms, x, y, z to denote term variables, and f, g, h, p, q, r to denote symbols of Σ (where f, g, h are function symbols and p, q, r predicates, with return type o). Predefined terms include $\top : o$ and $\bot : o$ (true and false propositions) — other connectives will be defined in Section 2.3.3.*

Example 2.3 (Polymorphic Lists). *The following signature defines the type of polymorphic lists; $list(\tau)$ is the type of lists whose elements have type τ. (::) and (@) are infix operators that denote respectively list construction and list concatenation.*

$$\left\{ \begin{array}{rcl} [] & : & \Pi\alpha. \, list(\alpha) \\ (::) & : & \Pi\alpha. \, (\alpha \times list(\alpha)) \to list(\alpha) \\ (@) & : & \Pi\alpha. \, (list(\alpha) \times list(\alpha)) \to list(\alpha) \\ rev & : & \Pi\alpha. \, list(\alpha) \to list(\alpha) \\ mem & : & \Pi\alpha. \, (\alpha \times list(\alpha)) \to o \end{array} \right\}$$

Assuming Σ also contains $a : \iota$, $b : \iota$, $f : \iota \to \iota$, and the Leibniz equality $\simeq : \Pi\alpha.\ \alpha \times \alpha \to o$ (defined below), Terms(Σ) contains, among others, the following well-typed terms:

$$
\begin{aligned}
x^\iota ::_{\langle \iota \rangle} (f(a) ::_{\langle \iota \rangle} (b ::_{\langle \iota \rangle} [\,]_{\langle \iota \rangle})) &\ :\ \ list(\iota) \\
f(f(a)) &\ :\ \ \iota \\
f(b) \simeq_{\langle \iota \rangle} f(b) &\ :\ \ o \\
mem_{\langle \beta \rangle}(y^\beta, rev_{\langle \beta \rangle}(y^\beta ::_{\langle \beta \rangle} [\,]_{\langle \beta \rangle})) &\ :\ \ o
\end{aligned}
$$

The first term denotes the list $[x, f(a), b]$. The last one is a predicates that asserts y is a member of the list $rev([y])$ (the reverse of $[y]$).

In some cases, the signature contains only simply-typed symbols (with ground types); in this case all terms are *simply-typed*, otherwise they are *polymorphic*. Some chapters of this thesis will be restricted to simply-typed terms.

Definition 2.35 (Size of a Term)**.** *The size of a term size(t) is a natural number recursively defined as*

$$size(x) = 1$$

$$size(f_{\langle \tau_1, \dots, \tau_m \rangle}(t_1, \dots, t_n)) = 1 + \sum_{i=1}^{n} size(t_i)$$

Definition 2.36 (Free Variables of a Term)**.** *The set of free variables of a term t, noted freevars(t), is defined recursively as*

$$freevars(x) = \{x\}$$

$$freevars(f_{\langle \tau_1, \dots, \tau_m \rangle}(t_1, \dots, t_n)) = \bigcup_{i=1}^{m} freevars(\tau_i) \cup \bigcup_{j=1}^{n} freevars(t_j)$$

Note that in general, freevars(t) contains type variables, and term variables of various type.

Definition 2.37 (Subterm relation)**.** *If s and t are two terms, we say s is a subterm of t, or $s \trianglelefteq t$, if either $s = t$, or t is $f_{\langle \tau_1, \dots, \tau_n \rangle}(t_1, \dots, t_n)$ and $s \trianglelefteq t_i$ for some $i \in \{1, \dots, n\}$. We say s is a strict subterm of t — noted $s \lhd t$ — if $s \trianglelefteq t$ and $s \neq t$.*

Lemma 2.1 (Subterm as a Well Founded Order)**.** *The relation $\lhd$ is a well-founded partial order.*

Proof. $s \lhd t$ clearly implies $0 \leq size(s) < size(t)$ (trivial induction on t). $\qquad\square$

Definition 2.38 (Position)**.** *A position is either ϵ, or $n \cdot p$ if p is a position and $n \in \mathbb{N}^+$.*

Definition 2.39 (Subterm at Position)**.** *Let s, t be terms and p be a position. Then s is the subterm of t at position p, written $s = t|_p$, in the two following cases:*
- *if $p = \epsilon$, then $s = t = t|_\epsilon$;*
- *if $p = i \cdot p'$ and $t = f_{\langle \tau_1, \dots, \tau_m \rangle}(t_1, \dots, t_n)$ with $1 \leq i \leq n$, then $s = t_i|_{p'}$.*

In both cases, we say p is a valid position in t.

Definition 2.40 (Replacement at Position)**.** *Let s, t be terms and p be a valid position in t — that is, $t|_p$ is well defined. Then, we can replace $t|_p$ with s to obtain a new term, $t[s]_p$, defined by*

$$
\begin{aligned}
t[s]_\epsilon &= s \\
t[s]_{i \cdot p} &= f_{\langle \tau_1, \dots, \tau_m \rangle}(t_1, \dots, t_{i-1}, t_i[s]_p, t_{i+1}, \dots, t_n) \\
&\qquad if\ 1 \leq i \leq n\ and\ t = \left(f_{\langle \tau_1, \dots, \tau_m \rangle}(t_1, \dots, t_n)\right)
\end{aligned}
$$

Example 2.4 (Positions)**.** *Let* $\Sigma = \{f : \iota \to \iota, g : \iota \times \iota \to \iota, a : \iota, b : \iota\}$. *Then,*

$$f(g(a,b))|_\epsilon = f(g(a,b))$$
$$f(g(a,b))|_{1\cdot\epsilon} = g(a,b)$$
$$f(g(a,b))|_{1\cdot1\cdot\epsilon} = a$$
$$f(g(a,b))|_{1\cdot2\cdot\epsilon} = b$$
$$\overbrace{f(\ldots f(a)\ldots)}^{m+n}|_{\underbrace{1\cdot\ldots\cdot1\cdot\epsilon}_{n}} = \overbrace{f(\ldots f(a)\ldots)}^{m}$$

Definition 2.41 (Monotonicity)**.** *A relation R on terms is* monotonic *if for any terms s, t, u and position p valid in u,, sRt implies $s[u]_p \, R \, t[u]_p$.*

Definition 2.42 (Term Substitution)**.** *A term substitution is a finite injective mapping $\{x_1 \mapsto t_1 , \ldots, x_n \mapsto t_n\} \cup \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_m \mapsto \tau_m\}$ from term variables to terms and from type variables to types (it includes a type substitution, see Definition 2.31). The set of term variables mapped by σ is the* domain *of σ, denoted $dom(\sigma) \overset{def}{=} \{x_1, \ldots, x_n\}$. A substitution σ can be applied to a term t (denoted $t\sigma$) as follows:*

$$
\begin{aligned}
x_i\sigma &= t_i & \text{if } \sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \\
y\sigma &= y & \text{otherwise, if } y \notin dom(\sigma) \\
f_{\langle \tau_1,\ldots,\tau_m \rangle}(t_1, \ldots, t_n)\sigma &= f_{\langle \tau_1\sigma,\ldots,\tau_m\sigma \rangle}(t_1\sigma, \ldots, t_n\sigma)
\end{aligned}
$$

Definition 2.43 (More General Substitution)**.** *A substitution σ is* more general *than another substitution ρ, denoted $\sigma \leq \rho$, if there exists θ such that $\forall x. \; x\rho = (x\sigma)\theta$.*

Definition 2.44 (Unifier)**.** *A unifier of two terms s and t is a substitution σ such that $s\sigma = t\sigma$. If s and t have a unifier, they are* unifiable. *In general, the substitution also binds some type variables (see next example).*

Definition 2.45 (Most General Unifier (mgu))**.** *A most general unifier (or mgu) of two terms s and t is a unifier σ of s and t such that for any unifier ρ of s and t, $\sigma \leq \rho$. If s and t are unifiable, then they have a unique most general unifier, denoted $mgu(s, t)$, up to renaming [Rob71, Smo89].*

Example 2.5 (Mgu)**.** *A few examples to illustrate the concept of mgu, in the same signature as Example 2.3:*

$$mgu\big(f(f(x^\iota, a), b), f(y^\iota, b)\big) = \{y^\iota \mapsto f(x^\iota, a)\}$$
$$mgu\Big([]_{\langle\alpha\rangle}, x^{list(\iota)}\Big) = \Big\{x^{list(\iota)} \mapsto []_{\langle\alpha\rangle}, \alpha \mapsto \iota\Big\}$$
$$mgu\Big(rev_{\langle\iota\rangle}(x^\iota ::_{\langle\iota\rangle} (b ::_{\langle\iota\rangle} []_{\langle\iota\rangle})), rev_{\langle\beta\rangle}(y^\beta ::_{\langle\beta\rangle} z^{list(\beta)})\Big) = \Big\{x^\iota \mapsto y^\beta, z^{list(\beta)} \mapsto (b ::_{\langle\iota\rangle} []_{\langle\iota\rangle}), \beta \mapsto \iota\Big\}$$

but the pairs of terms $\big([]_{\langle\iota\rangle}, x^\alpha ::_{\langle\alpha\rangle} []_{\langle\alpha\rangle}\big)$, $(a, f(b))$, or $(x^\iota, f(x^\iota))$ are not unifiable. The last case is usually is ruled out under the occur-check *rule: never unify a variable x with a term containing x as a strict subterm.*

Definition 2.46 (Simplification Ordering)**.** *A* simplification ordering *is an ordering $\succ$ on terms that has the properties*

Well-Founded *there is no infinite chain $t_1 \succ t_2 \succ \ldots$;*
Subterm Property *if $s \lhd t$, then $s \prec t$;*
Monotonicity *if $t_i \prec t_i'$, then $f(t_1, \ldots, t_i, \ldots, t_n) \prec f(t_1, \ldots, t_i', \ldots, t_n)$ for any symbol f;*
Stability under Substitution *if $s \prec t$, then for any substitution σ, $s\sigma \prec t\sigma$.*

Definition 2.47 (Recursive Path Ordering (RPO)). *The* Recursive Path Ordering, *also called RPO, is a term ordering $\prec_{rpo}$ parametrized by a* precedence — *a total order $s_1 < s_2 < \ldots < s_n$ on the symbols of Σ —, a precedence $<_\tau$ on type constructors of Σ_τ — similarly, a total order —, and a* status, *either lexicographic or multiset, for each term symbol. By $=_{ms}$ we denote equality modulo permutation of arguments of function symbols that have a multiset status. It is defined as follows:*

- *$x \prec_{rpo} t$ if $x \in freevars(t)$ and $x \neq t$, or*
- *$s \overset{def}{=} f_{\langle \tau_{s,1},\ldots,\tau_{s,m} \rangle}(s_1,\ldots,s_{m'}) \prec_{rpo} t \overset{def}{=} g_{\langle \tau_{t,1},\ldots,\tau_{t,n} \rangle}(t_1,\ldots,t_{n'})$, and either:*
  - *$s \prec_{rpo} t_j$ for some j, or*
  - *$s =_{ms} t_j$ for some j, or*
  - *$f < g$ and $s_i \prec_{rpo} t$ for some i, or*
  - *$f = g$ (and $m = n$ and $m' = n'$) and $(\tau_{s,1},\ldots,\tau_{s,m}) \prec_{rpo,lex,\tau} (\tau_{t,1},\ldots,\tau_{t,m})$*
  - *$f = g$, f has multiset status, $(\tau_{s,1},\ldots,\tau_{s,m}) = (\tau_{t,1},\ldots,\tau_{t,m})$, and $\{s_1,\ldots,s_{m'}\} \ll_{rpo} \{t_1,\ldots,t_{m'}\}$, or*
  - *$f = g$, f has lexicographic status, $(\tau_{s,1},\ldots,\tau_{s,m}) = (\tau_{t,1},\ldots,\tau_{t,m})$, and $(s_1,\ldots,s_{m'}) \prec_{rpo,\,lex} (t_1,\ldots,t_{m'})$.*

where $\prec_{rpo,\tau}$ is the ordering on types defined by

- *$\alpha \prec_{rpo,\tau} \tau$ if $\alpha \in freevars(\tau)$ and $\alpha \neq \tau$, or*
- *$\tau_1 \overset{def}{=} c_1(\tau_{1,1},\ldots,\tau_{1,m}) \prec_{rpo,\tau} \tau_2 \overset{def}{=} c_2(\tau_{2,1},\ldots,\tau_{2,n})$, and either:*
  - *$\tau_1 \prec_{rpo} \tau_{2,j}$ for some j, or*
  - *$\tau_1 = \tau_{2,j}$ for some j, or*
  - *$c_1 <_\tau c_2$ and $\tau_{1,i} \prec_{rpo} \tau_2$ for some i, or*
  - *$c_1 = c_2$ (and $m = n$) and $(\tau_{1,1},\ldots,\tau_{1,m}) \prec_{rpo,lex,\tau} (\tau_{2,1},\ldots,\tau_{2,m})$.*

If all symbols have lexicographic status, then we speak of a Lexicographic Path Ordering, *or LPO. The ordering on types is, basically, similar to LPO. This definition is inspired by the typed version of KBO from [Wan14].*

Lemma 2.2 (RPO is a Simplification Ordering [NR99]). *Any RPO is a simplification ordering. In addition, LPO is a total order on ground terms. Usually those are proved on untyped terms but the generalization to typed terms is straightforward.*

There is another popular ordering in the literature on first-order terms, the *Knuth-Bendix Ordering* (or KBO), but we do not define it because RPO is sufficient for our needs. The implementation can use both, however.

2.3.3 Formulas, Literals, and Clauses

Definition 2.48 (Leibniz Equality). *Leibniz equality, denoted $\simeq: \Pi\alpha.\,(\alpha \times \alpha) \to o$, is a polymorphic equivalence (transitive, symmetric and reflexive) relation on terms satisfying the following family of axiom parametrized by function symbol $f : (\tau_1 \times \ldots \times \tau_n) \to \tau$:*

$$\forall x_1, y_1 : \tau_1 \,\ldots\, x_n, y_n : \tau_n. \left(\bigwedge_{i=1}^{n} x_i \simeq y_i \right) \Rightarrow f(x_1,\ldots,x_n) \simeq f(y_1,\ldots,y_n)$$

Remark 2.7 (Notions of Equality). *Leibniz equality $\simeq$ — also called* equality on uninterpreted terms — *is not to be confused with the syntactic equality $=$. The latter is a meta-level notion, whereas the former belongs to the logic language itself; the theorem prover will reason on equations defined using $\simeq$.*

Definition 2.49 (First-order Formula). *If Σ is a signature, the set of first-order formulas $Forms(\Sigma)$ is defined inductively by*

Predicate *if $t : o$ is a well-typed term of $Terms(\Sigma)$, then $t \in Forms(\Sigma)$;*

Negation *if $F \in Forms(\Sigma)$, then $(\neg F) \in Forms(\Sigma)$;*
Disjunction *if $F, G \in Forms(\Sigma)$, then $(F \vee G) \in Forms(\Sigma)$;*
Conjunction *if $F, G \in Forms(\Sigma)$, then $(F \wedge G) \in Forms(\Sigma)$;*
Universal Quantification *if $F \in Forms(\Sigma)$ and $x \in \mathscr{X}_\tau$, then $(\forall x : \tau. \, F) \in Forms(\Sigma)$;*
Existential Quantification *if $F \in Forms(\Sigma)$ and $x \in \mathscr{X}_\tau$, then $\exists x : \tau. \, F \in Forms(\Sigma)$.*

In general, we will omit types in quantifiers and in some terms when they are easy to infer from context. Equations $u \simeq_{\langle \tau \rangle} v$ belong to the predicate *case; we will shorten $\neg(u \simeq v)$ into $u \not\simeq v$, and $u \simeq v$ will mean either $u \simeq v$ or $u \not\simeq v$.*

Remark 2.8 (Other Formulas). *Some additional connective are often useful and can be defined from the primitive ones:*

Implication *if $F, G \in Forms(\Sigma)$, then $(F \Rightarrow G) \stackrel{def}{=} ((\neg F) \vee G) \in Forms(\Sigma)$;*
Equivalence *if $F, G \in Forms(\Sigma)$, then $(F \Longleftrightarrow G) \stackrel{def}{=} ((F \Rightarrow G) \wedge (G \Rightarrow F)) \in Forms(\Sigma)$;*
Exclusive Disjunction *if $F, G \in Forms(\Sigma)$, then $(F \oplus G) \stackrel{def}{=} ((F \wedge \neg G) \vee (G \wedge \neg F)) \in Forms(\Sigma)$.*

Example 2.6 (Socrates). *At this point, we are ready to rephrase a piece of secular wisdom: the classic syllogism "if Socrates is a man, and all men are mortal, then Socrates is mortal", as a first-order formula:*

$$[man(Socrates) \wedge (\forall x : \iota. \, man(x) \Rightarrow mortal(x))] \Rightarrow mortal(Socrates)$$

using the signature $\Sigma = \{Socrates : \iota, man : \iota \to o, mortal : \iota \to o\}$.

Definition 2.50 (Literal). *A* literal *is either an atomic formula (a predicate) or the negation of an atomic formula. In particular, equations $a \simeq b$ and inequations $a \not\simeq b$ are literals. It is sometimes simpler to consider that all literals are (in-)equations, predicates (of the form $p(t_1, \ldots, t_n)$) being encoded as o-typed equations with $\top : o$ (e.g., $p(t_1, \ldots, t_n)$ becomes $p(t_1, \ldots, t_n) \simeq \top$).*

Definition 2.51 (Clause). *A* clause *is a disjunction of literals $\bigvee_{i=1}^{n} l_i$. The empty clause $\bot$ is the empty disjunction, and as such it is always false. A formula of the form $\bigwedge_i C_i$ where each C_i is a clause is in* conjunctive normal form, *or CNF. If C is a clause with $freevars(C) = \{x_1, \ldots, x_n\}$, we write $\overline{\forall} C$ for the closed formula $\forall x_1 \ldots x_n. \, C$.*

Definition 2.52 (Skolemization). *A formula $F \stackrel{def}{=} \exists x : \tau. \, G[x]$ with $freevars(F) = \{y_1 : \tau_1, \ldots, y_n : \tau_n\}$ can be* skolemized *into $G\{x \mapsto f(y_1, \ldots, y_n)\}$, where $f : (\tau_1 \times \ldots \times \tau_n) \to \tau$ is a fresh* Skolem *symbol that acts as a witness for the existence of a $x : \tau$ such that G holds. It requires the axiom of choice.*

Definition 2.53 (CNF Procedure). *There exist some procedures that transform a formula F into an equi-satisfiable set of clauses (see for instance [NW01]). We will denote $cnf(F)$ such a set of clauses obtained from F. Skolemization is one of the steps required for computing $cnf(F)$, and the reason why $F \Longleftrightarrow cnf(F)$ is false in general.*

Definition 2.54 (Ordering on Literals). *We assume $\succ$ is a simplification ordering on terms (Definition 2.46) in which $\top$ is minimal, and define an ordering $\succ_{lit}$ on literals as follows. Let $\mathscr{M}_e(\cdot)$ be a function that maps a literal l to a multiset of terms, defined by*

$$\mathscr{M}_e(l) \stackrel{def}{=} \begin{cases} \{s, t\} & \text{if } l = (s \simeq t) \\ \{s, s, t, t\} & \text{if } l = (s \not\simeq t) \end{cases}$$

Then, we define the order: $l_1 \succ_{lit} l_2$ iff $\mathscr{M}_e(l_1) \gg \mathscr{M}_e(l_2)$. The point of $s \not\simeq t$ being larger than $s \simeq t$ in the ordering is that Superposition will tend to eliminate negative literals (inequations) first, keeping equations as rewrite rules.

Definition 2.55 (Ordering on Clauses)**.** *In the same vein, we define $\succ_c$ on clauses by $C \succ_c D$ iff $C \gg_{lit} D$, reducing clauses to the multiset of their literals. This ordering is well founded, and total on ground clauses if $\succ$ is total on ground terms.*

Definition 2.56 (A-clause)**.** *An* A-clause, *or* clause with assertions, *is a pair $C \leftarrow a_1 \sqcap \ldots \sqcap a_n$ where C is a clause and $a_1 \sqcap \ldots \sqcap a_n$ is a conjunction of boolean literals (the* trail*). Any clause C can also be seen trivially as an A-clause $C \leftarrow 1$, and we will not emphasize the difference when no ambiguity ensues.*

Definition 2.57 (Grounding of a Set)**.** *Given a set of first-order clauses N, we call the* grounding *of N the set $Gnd(N) \stackrel{def}{=} \{C\sigma \mid C \in N, freevars(C\sigma) = \emptyset\}$. $Gnd(N)$ contains all the ground instances of the clauses of N.*

In the rest of this thesis, F, G will be formulas, l will be a literal, C, D, K will be clauses or A-clauses, depending on the context.

2.3.4   Semantics: the Central Notion of Model

Logic is about building correct proofs of statements in a formal way, using precise syntactic rules. Intuitively, correctness means that only "true" formulas are provable (in particular, the falsity $\bot$ should not be provable). A possible way to define what *true* means is the notion of *model*: a model maps terms and formulas to other mathematical objects in which connectives (negation, conjunction, etc.) have a precise meaning. First-order logic enjoys good properties when it comes to models; in particular, a formula F is a theorem iff $\neg F$ has no model. Each model defines a specific way of interpreting what a formula means.

Definition 2.58 (Model)**.** *A* model $\mathcal{M}$ *is a tuple $\left((D)_\tau, (\hat{f})_{f \in \Sigma}, \hat{\top}, \hat{\bot}\right)$ where*
- *$(D)_\tau$ is a type-indexed family of domains defined on ground atomic types. For each ground atomic type $\tau \in Types(\Sigma_\tau)$, D_τ is a non-empty set of values;*
- *$(\hat{f})_{f \in \Sigma}$ is a symbol-indexed family of functions. For every $f \in \Sigma$ with $f : \Pi\alpha_1 \ldots \alpha_m. (\tau_1 \times \ldots \times \tau_n) \to \tau$, $\hat{f}$ is a family of functions parametrized by m-tuples of types, such that for all types $\tau'_1, \ldots, \tau'_m$, $\hat{f}_{\langle \tau'_1, \ldots, \tau'_m \rangle}$ is a function from $D_{\tau_1\sigma} \times \ldots \times D_{\tau_n\sigma}$ into $D_{\tau\sigma}$ where $\sigma \stackrel{def}{=} \{\alpha_1 \mapsto \tau'_1, \ldots, \alpha_m \mapsto \tau'_m\}$. Since the type of f is closed (by definition of a signature), σ is a grounding type substitution, which guarantees that each $D_{\tau_i\sigma}$ and $D_{\tau\sigma}$ are well-defined.*
- *$D_o = \{\hat{\top}, \hat{\bot}\}$ such that $\hat{\top}$ and $\hat{\bot}$ are distinct.*

Definition 2.59 (Interpretation of Terms)**.** *The* interpretation *of a ground term $t : \tau \in Terms(\Sigma)$ in a model $\mathcal{M}$ and a valuation σ (that maps variables of type α to elements of D_α), noted $[\![t]\!]^{\mathcal{M}}_\sigma$, is an element of D_τ, inductively defined by*

$$[\![x]\!]^{\mathcal{M}}_\sigma = \sigma(x)$$
$$[\![f_{\langle \tau_1, \ldots, \tau_m \rangle}(t_1, \ldots, t_n)]\!]^{\mathcal{M}}_\sigma = \hat{f}_{\langle \tau_1, \ldots, \tau_m \rangle}\left([\![t_1]\!]^{\mathcal{M}}_\sigma, \ldots, [\![t_n]\!]^{\mathcal{M}}\right)$$

Definition 2.60 (Interpretation of Formulas)**.** *The* interpretation *of a closed formula F in a*

model $\mathcal{M}$ and a valuation σ, noted $[\![F]\!]_\sigma^{\mathcal{M}}$, is inductively defined by

$$[\![F]\!]_\sigma^{\mathcal{M}} = [\![t]\!]_\sigma^{\mathcal{M}} \text{ if } F \text{ is } t : o \in \textit{Terms}(\Sigma)$$

$$[\![\neg F]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ if } [\![F]\!]_\sigma^{\mathcal{M}} = \hat{\bot}$$

$$= \hat{\bot} \text{ if } [\![F]\!]_\sigma^{\mathcal{M}} = \hat{\top}$$

$$[\![F \vee G]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ if } [\![F]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ or } [\![G]\!]_\sigma^{\mathcal{M}} = \hat{\top}$$

$$= \hat{\bot} \text{ otherwise}$$

$$[\![F \wedge G]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ if } [\![F]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ and } [\![G]\!]_\sigma^{\mathcal{M}} = \hat{\top}$$

$$= \hat{\bot} \text{ otherwise}$$

$$[\![\forall x : \tau.\ F]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ if for every } t \in D_\tau, [\![F]\!]_{\{x \mapsto t\} \cup \sigma}^{\mathcal{M}} = \hat{\top}$$

$$= \hat{\bot} \text{ otherwise}$$

$$[\![\exists x : \tau.\ F]\!]_\sigma^{\mathcal{M}} = \hat{\top} \text{ if there is some } t \in D_\tau \text{ such that } [\![F]\!]_{\{x \mapsto t\} \cup \sigma}^{\mathcal{M}} = \hat{\top}$$

$$= \hat{\bot} \text{ otherwise}$$

This definition maps trivially to literals and clauses; it suffices to remember that a literal is an atomic formula or the negation thereof, and a clause $C \overset{def}{=} \bigvee_{i=1}^n l_i$ is indeed the closed formula $\forall x_1 \ldots x_m.\ \bigvee_{i=1}^n l_i$ where $\{x_1, \ldots, x_m\} = \textit{freevars}(C)$.

We say the model $\mathcal{M}$ satisfies the formula F, noted $\mathcal{M} \models F$, iff $[\![F]\!]^{\mathcal{M}} \overset{def}{=} [\![F]\!]_\emptyset^{\mathcal{M}} = \hat{\top}$. A clause is satisfied iff at least one of its literals is — the empty clause can therefore never be satisfied.

A valid formula *is one that is satisfied in every model.*

Definition 2.61 (Equational Model)**.** *A model $\mathcal{M}$ for a signature containing Leibniz equality $\simeq$ is an* equational model *iff $\mathcal{M}$ satisfies the Leibniz axioms. More precisely, $\mathcal{M}$ must satisfy symmetry, reflexivity and transitivity for $\simeq$ on every type; moreover, for every $f \in \Sigma$ with $f : \Pi\alpha_1 \ldots \alpha_m.\ (\tau_1 \times \ldots \times \tau_n) \to \tau$, for every m−tuple of ground atomic types $(\tau_1', \ldots, \tau_m') \in \Sigma_\tau^m$, let $\sigma \overset{def}{=} \{\alpha_1 \mapsto \tau_1', \ldots, \alpha_m \mapsto \tau_m'\}$; the following* congruence *axiom must be satisfied in $\mathcal{M}$:*

$$\forall s_1, t_1 : \tau_1 \sigma \ \ldots \ s_n, t_n : \tau_n \sigma.\ \Big[(s_1 \simeq t_1 \wedge \ldots \wedge s_n \simeq t_n) \Rightarrow f_{\langle \tau_1', \ldots, \tau_m' \rangle}(s_1, \ldots, s_n) \simeq f_{\langle \tau_1', \ldots, \tau_m' \rangle}(t_1, \ldots, t_n)\Big]$$

Definition 2.62 (Herbrand Model)**.** *A* Herbrand model *is a model in which every domain D_τ is $\{t \in \textit{Terms}(\Sigma) \mid t : \tau\}$, and such that $\hat{f}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$; that is, function symbols are interpreted by themselves.*

An equational Herbrand model *is a Herbrand model such that, for each type other than o, $\simeq_{\langle\tau\rangle}$ is interpreted by a congruence — that is, a relation that is symmetric, transitive, reflexive and monotonic. Herbrand models play an important rôle in proof of completeness for Superposition.*

Definition 2.63 (Entailment)**.** *Given two formulas F and G, we say F* entails *G, or $F \vdash G$, iff for every model $\mathcal{M}$, $\mathcal{M} \models F$ implies $\mathcal{M} \models G$. The same notion extends to clauses.*

Definition 2.64 (Provability)**.** *A* proof, *informally, is a syntactic object that justifies why some formula F is a theorem. In this thesis, we do not care much about the proofs themselves — unlike, say, in intuitionistic proof assistants in which the Curry-Howard correspondence turns every proof into a function. Later, a* proof *of F will be a derivation of $\bot$ from $\neg F$ in some inference system — Superposition, AVATAR, or our own extensions of Superposition in Chapter 4 (arithmetic) and Chapter 5 (structural induction).*

We only need a provability *notion: a formula F is* provable, *or F is a* theorem *(noted $\textit{thm}(F)$) if there is such a proof of F.*

Definition 2.65 (Proof System)**.** *A* proof system *is a procedure that inputs a formula F and either diverges (never terminates) or returns one of $\{\bot, \pi\}$ where π is a proof of F. A provability relation can be naturally defined by $\textit{thm}(F)$ holding for every F on which the procedure returns a proof.*

Definition 2.66 (Soundness)**.** *A provability relation* thm$(\cdot)$ *is* sound *if all provable formulas are true in every model. In other words, given a formula F, if* thm(F) *then* $\mathcal{M} \models F$ *must hold in every model* $\mathcal{M}$. *A proof system is* sound *iff every formula on which it successfully returns a proof is true in every model.*

Definition 2.67 (Completeness of a Proof System)**.** *A proof system is* complete *if every valid formula F is provable in the system. In other words, it means that* thm(F) *holds for every formula F such that F is satisfied in every model.*

Remark 2.9 (Semi-Completeness)**.** *What we call here* completeness *is sometimes called* semi-completeness*; the proof system can fail to terminate in case the input formula is not a theorem. There exists no truly complete proof systems for first-order logic, as implied by the Halting problem. However, there are several sound and (semi-)complete proof system for first-order logic, including Sequent Calculus, Natural Deduction, and Superposition — different techniques that have different completeness proofs, going back to Gödel Completeness Theorem.*

All the previous notions are standard ones that define models and how to interpret formulas and clauses in them; now, we extend this usual notion of model into one that can interpret A-clauses (Definition 2.56). A-clauses are a recent notion and their formal semantics is a small contribution we make here.

Definition 2.68 (Combined Model)**.** *A* combined model *(shortened into* model *when there is no ambiguity) is a pair* $(\mathcal{M}, v)$ *where* $\mathcal{M}$ *is a model and v is a boolean valuation (See 2.19).*

Definition 2.69 (Interpretation in a Combined Model)**.** *An A-clause $C \leftarrow \Gamma$ has an interpretation* $[\![C \leftarrow \Gamma]\!]^{\mathcal{M},v}$ *in the combined model* $(\mathcal{M}, v)$, *defined by*

$$[\![C \leftarrow \Gamma]\!]^{\mathcal{M},v} = \hat{\top} \qquad\qquad \text{if } v(b) = 0 \text{ for some } b \in \Gamma$$

$$[\![C \leftarrow \Gamma]\!]^{\mathcal{M},v} = [\![\overline{\forall}C]\!]_{\varnothing}^{\mathcal{M}} \qquad\qquad \text{otherwise}$$

We say $C \leftarrow \Gamma$ is satisfied *in* $(\mathcal{M}, v)$, *noted* $(\mathcal{M}, v) \models C \leftarrow \Gamma$, *iff* $[\![C \leftarrow \Gamma]\!]^{\mathcal{M},v} = \hat{\top}$.

Definition 2.70 (Combined State)**.** *A* combined state *is a pair* $(\mathcal{N}, \mathscr{F}_b)$ *where* $\mathcal{N}$ *is a set of clauses and $\mathscr{F}_b$ a boolean formula. A combined model $(\mathcal{M}, v)$ satisfies a combined state* $(\mathcal{N}, \mathscr{F}_b)$ *iff* $(\mathcal{M}, v) \models \mathcal{N}$ *and $v(\mathscr{F}_b) = 1$.*

As the next section explains, the inference process consists in successive transformations from a combined state to another, where every step is satisfiability-preserving.

2.4   Superposition

Superposition is a refutationally complete deduction system for first-order equational logic — if a set of clauses is unsatisfiable then Superposition will reach the empty clause after a finite, but unbounded, amount of time. We briefly recap the standard inference system for Superposition, then expose a few simplification rules. See [Sch02] for a nice introduction to the emblematic open source prover E, its inference system and implementation; see [NR99] — from which most definitions and theorems from this section come from — for more theoretical explanations of Superposition, its principle, and completeness arguments.

Superposition only works on clauses, but any formula can be turned into an *equi-satisfiable* CNF (see for instance [NW01] for an overview of algorithms that transform formulas into sets of clauses). Equi-satisfiable means that the CNF is satisfiable (has a model) iff the formula is satisfiable. All Superposition provers there start by reducing the negation of the conjecture into CNF, then proceed to applying inference rules.

2.4.1 Inference and Simplification Rules

In both this section and the following one, we present *inference rule* and *simplification rule*. Basically, an inference rule is a recipe for deducing, from clauses $C_1,\ldots,C_n$, a new clause D such that $\bigwedge_{i=1}^{n} C_i$ logically entails D. This way, starting from a set of axioms, a theorem prover can deduce new clauses that follow from the axioms, in the hope that it eventually reaches $\bot$ (or stops because it deduced all possible conclusions without reaching a contradiction); incompleteness of first-order logic implies that the prover might also loop forever in the case where the axioms are not contradictory.

Definition 2.71 (Inference Rule). *An* inference rule *is a relation between one or several clauses $C_1,\ldots,C_n$ called the* premises, *and one or more clauses called the* conclusion D. *If $n = 1$ the rule is* unary, *if $n = 2$ it is* binary. *Premises are assumed to share no variable (possibly by renaming them). We will use the following notation throughout this thesis (possibly with an annotation* (A) *to specify which inference rule is used):*

$$\frac{C_1 \quad \ldots \quad C_n}{D} \text{ (A)}$$

Example 2.7 (Resolution Rule). *A very simple and central inference rule is* resolution *[Rob65]. We will not use it directly in this work, but Superposition is often considered a refinement of Resolution, which played an important role in Automated Deduction.*

$$\textbf{\textit{Resolution}} \text{ (Res)}$$

$$\frac{l \vee C \quad \neg l' \vee C'}{(C \vee C')\sigma} \text{ (Res)}$$

$$if \, \sigma = mgu(l, l')$$

Remark 2.10 (Inference Rule with Multiple Conclusions). *We slightly abuse the notation and allow some inference rules to return several conclusions, as a compact way of writing several rules that have the same set of premises.*

Remark 2.11 (Boolean Inference Rule). *By convention, we will use a dotted line for inference rules that operate on propositional literals and clauses (as opposed to first-order clauses). For instance, the propositional resolution rule, as used in some SAT solvers, is expressed as follows:*

$$\textbf{\textit{Boolean Resolution}}$$

$$\frac{a \sqcup C \quad \neg a \sqcup D}{C \sqcup D}$$

Definition 2.72 (Simplification Rule). *In some occasions, the conclusion D of an inference rule with premises $C_1,\ldots,C_n$ is equivalent to C_1 under assumption $C_2,\ldots,C_n$, and $D \prec_c C_1$ for some order $\prec_c$ on clauses. In such cases, it will sometimes be better (especially for performance reasons) to replace C_1 with D; we may then speak of a* simplification *rule, denoted:*

$$\frac{C_1 \quad \ldots \quad C_n}{D} \text{ (A)}$$

Example 2.8 (Deletion of Resolved Literals). *The following rule is sound, but is also a simplification:*

Deletion of Resolved Literals

$$\frac{t \not\simeq t \vee C}{C}$$

We are now ready to define the inference rules of the Superposition calculus.

2.4.2 The Calculus

The Superposition calculus, called *Sup*, is detailed in figure 2.1, in its first-order version (the ground version basically replaces $\not\prec$ with $\succ$ since the ordering is total on ground terms).

Superposition (Sup)

$$\frac{C \vee s \simeq t \qquad D \vee u \dot{\simeq} v}{(C \vee D \vee u\,[t]_p \dot{\simeq} v)\sigma}$$

where $s\sigma \not\preceq t\sigma$, $(s \simeq t)\sigma \not\preceq C\sigma$, $\sigma = \mathrm{mgu}\left(u|_p, s\right)$, $u\sigma \not\prec v\sigma$,
$(u \dot{\simeq} v)\sigma \not\prec D\sigma$.

Equality Factoring (EqFact)

$$\frac{C \vee s \simeq s' \vee t \simeq t'}{(C \vee s' \not\simeq t' \vee t \simeq t')\sigma}$$

where $\sigma = \mathrm{mgu}\,(s, t)$, $t\sigma \not\prec t'\sigma$, $s\sigma \not\prec s'\sigma$, $(s \simeq s')\sigma \not\prec C\sigma$.

Equality Resolution (EqRes)

$$\frac{C \vee s \not\simeq t}{C\sigma}$$

where $\sigma = \mathrm{mgu}\,(s, t)$, $(s \not\simeq t)\sigma \not\prec C\sigma$

Figure 2.1: Inference rules of Superposition

Let us explain the inference rules and give some intuition.

Superposition uses a positive equation $s \simeq t$ to rewrite, in an equation $u \dot{\simeq} v$, the subterm of u at position p, if s and $u|_p$ are unifiable. The reasoning is that, in any model of both clauses, if the contexts C and D are false then necessarily $s \simeq t$ and $u \dot{\simeq} v$ are both true; $u|_p\sigma$ (syntactically equal to $s\sigma$) is equal to $t\sigma$ and, by definition of $\simeq$, $u\,[t\sigma]_p\,\sigma \simeq u\sigma \simeq v\sigma$. This rule can be seen as *conditional rewriting*: $u|_p$ is rewritten by $s \simeq t$ assuming C and D are both false.

Equality Resolution is simple: if $C \vee s \not\simeq t$ is true, in any model either C is true or $s \not\simeq t$ is. If s and t are unified by σ, then it is impossible that $s\sigma \not\simeq t\sigma$ be true in any model; therefore $C\sigma$ must hold instead.

Equality Factoring starts from $C \vee s \simeq s' \vee t \simeq t'$. If $\sigma = \mathrm{mgu}\,(s, t)$, then in any model of the premise, there are three possibilities, reflected in the conclusion:
- $C\sigma$ holds;
- $s'\sigma \not\simeq t'\sigma$ holds.
- $s'\sigma \simeq t'\sigma$ holds, in which case the literals $s\sigma \simeq s'\sigma$ and $t\sigma \simeq t'\sigma$ are interpreted the same, because $t\sigma = s\sigma \simeq s'\sigma \simeq t'\sigma$ by assumption. In this case, we can *factor* the two literals: merge them into only one literal, for instance $t\sigma \simeq t'\sigma$;

The ordering conditions based on $\succ$, the simplification term ordering, restrict the cases in which rules can be applied. They matter both for the completeness proof — based on a well-founded induction on $\succ_c$ — and for the practical efficiency of the Superposition calculus —

they significantly prune the search space by allowing inferences to operate only on maximal literals and maximal terms.

Definition 2.73 (Saturation)**.** Saturating *a set of clauses N consists in repeating the following operation until a fixpoint is reached: pick clauses $C_1, \ldots, C_n \in N$ such that* $\dfrac{C_1 \quad \ldots \quad C_n}{D}$ (A) *for some rule* (A) *in Sup, and add D to N. If $\perp$ is deduced, the unsatisfiability of N has been proved and the procedure stops. This procedure can loop forever.*

Theorem 2.1 (Superposition is Complete [NR99])**.** *Superposition is* complete *for the first-order logic with equality, that is, for every unsatisfiable formula F, there is a Superposition derivation of $\perp$ from cnf(F) (see Definition 2.67). In addition, Superposition is sound.*

Since Superposition is complete, proving a theorem F under assumption Γ (a set of axioms) can be reduced to the following steps: (1) compute cnf($(\bigwedge_{G \in \Gamma} G) \wedge \neg F$); (2) try to reach $\perp$ by fair saturation using *Sup*. Many theorem provers are based on this principle.

Remark 2.12 (Resolution)**.** *Although the inference rules presented in Figure 2.1 do not contain Resolution, the rule is easy to simulate (assuming, again, that a predicate p is encoded as an equation $p \simeq \top$)*

$$\dfrac{\dfrac{C \vee p \simeq \top \quad C' \vee p' \not\simeq \top}{(C \vee C' \vee \top \not\simeq \top)\sigma} \text{ (Sup)}}{(C \vee C')\sigma} \text{ (EqRes)}$$

Recall that $\top$ is the smallest term in $\succ$, which makes p maximal in $p \simeq \top$. To keep proofs readable, we will keep the predicate notation and the resolution rule in derivation trees, even though the actual proof uses Superposition and equality resolution rule in such cases.

To help the reader forge a bit of intuition of what a Superposition proof looks like, we present a few examples.

Example 2.9 (Socrates Dies Again)**.** *First, a proof of our previous claim that Socrates is mortal (Example 2.6), by mere Resolution. The reduction to CNF of the* negation *of the formula we had yields the set of clauses*

$$\{man(Socrates), \neg mortal(Socrates), \neg man(x) \vee mortal(x)\}$$

From there we can derive false, proving that the syllogism's negation is absurd, and therefore that the syllogism is a theorem

$$\dfrac{\dfrac{\neg man(x) \vee mortal(x) \quad man(Socrates)}{mortal(Socrates)} \text{ (Sup)} \quad \neg mortal(Socrates)}{\perp} \text{ (Sup)}$$

Example 2.10 (Teaching)**.** *Excerpt from the problem* `PUZ131_1.p` *from TPTP[3]:*

> *Every student is enrolled in at least one course. Every professor teaches at least one course. Every course has at least one student enrolled. Every course has at least one professor teaching. The coordinator of a course teaches the course. If a student is enrolled in a course then the student is taught by every professor who teaches the course. Michael is enrolled in CSC410. Victor is the coordinator of CSC410. Therefore, Michael is taught by Victor.*

---

[3] A large archive of first-order problems.

The problem formulation makes use of the types course, student and prof, with the signature

$$\left\{ \begin{array}{l} v : prof, m : student, c : course, \\ teaches : prof \times course \rightarrow o, coord. : course \rightarrow prof, \\ tb. : student \times prof \rightarrow o, enr. : student \times course \rightarrow o \end{array} \right\}$$

The predicate teaches(p, c) means that p teaches the course c; coord.(c) is the coordinator of c; tb.(s, p) means the student s is taught by professor p; enr.(s, c) means that s is enrolled in course c. We can deduce that Victor (v) teaches to Michael m. In general, we would add $\neg teaches(v, m)$ to the set of clauses, but here we can even deduce it as a fact:

$$\frac{\dfrac{coord.(c) \simeq v \quad teaches(coord.(x), x)}{teaches(v, c)} \quad \dfrac{enr.(m, c) \quad \dfrac{\neg enr.(y, x) \vee \neg teaches(z, x) \vee tb.(y, z)}{\neg teaches(z, c) \vee tb.(m, z)}}{}}{tb.(m, v)}$$

Example 2.11 (Group Theory). *To illustrate equational reasoning a bit, we prove that the (untyped) axiomatization of groups*

$$0 + x \simeq x$$
$$(-x) + x \simeq 0$$
$$(x + y) + z \simeq x + (y + z)$$

in the signature $\{+ : (\iota \times \iota) \rightarrow \iota, - : \iota \rightarrow \iota, 0 : \iota\}$ implies the theorem $\forall x\ y.\ x + y \simeq 0 \Rightarrow y + x \simeq 0$. The proof (in which rule names are omitted for lack of space) starts by introducing $a, b : \iota$ after negating the goal, which becomes $\{a + b \simeq 0, b + a \not\simeq 0\}$, then applying (Sup) many times, and finally (EqRes) to conclude.

$$\frac{\dfrac{\dfrac{(-x) + x \simeq 0 \quad (x + y) + z \simeq x + (y + z)}{0 + y \simeq -(x) + (x + y) \quad 0 + x \simeq x}}{y \simeq (-x) + (x + y)} \quad \dfrac{\dfrac{\dfrac{(x + y) + z \simeq x + (y + z) \quad a + b \simeq 0}{0 + x \simeq a + (b + x) \quad 0 + x \simeq x}}{x \simeq a + (b + x)}}{b + x \simeq (-a) + x \quad (-x) + x \simeq 0}}{\dfrac{b + a \simeq 0 \quad b + a \not\simeq 0}{\bot}}$$

2.4.3 Redundancy Criteria

The rules from Section 2.4.2 are sufficient in theory; in practice, for most problems the search space is intractable. A lot of work (see again [NR99] for an overview) has been dedicated to refining the Superposition calculus to make it more efficient. The notion of *redundancy* is the workhorse of most of those refinements; intuitively, a clause is *redundant* if it brings no more knowledge to the problem than *smaller* clauses — the larger clause can therefore be removed without loss of information.

Definition 2.74 (Redundancy). *Given a total order $\succ_c$ on ground clauses, a ground clause C and a set of ground clauses N, we say that C is redundant w.r.t. N iff $N^{<C} \vdash C$. In other words, some clauses in N, that are smaller than C, entail C. This general criterion is not computable, but provides a common frame to several computable criteria (some examples are listed below).*

A first-order clause C is redundant w.r.t. a set of clauses N iff for each substitution σ such that $C\sigma$ is ground, $\text{Gnd}(N)^{<C\sigma} \vdash C\sigma$. In other words, C is redundant if all its ground instances are.

An inference (A) that deduces D from premises $C_1, \ldots, C_n$, where C_1 is maximal, is redundant w.r.t. a set of clauses N if D is redundant w.r.t. $N^{<C_1}$.

If a clause C is redundant w.r.t. N, it is useless to add C to N, and it is of no use either to perform any inference between C and N. If an inference is redundant w.r.t. N, it is not necessary to perform it. In general, the problem of computing whether C is redundant w.r.t. N is undecidable, but many sufficient criteria exist. A few useful simplification rules are presented here, most of which are already detailed in [Sch02].

Remark 2.13 (Simplification Rule). *The notion of simplification rule, as defined in Section 2.4.1, becomes clear in the light of the notion of redundancy: if $\dfrac{C_1 \quad C_2}{D}$ with $C_2 \prec_c C_1$, $D \prec_c C_1$ and $C_2 \wedge D \vdash C_1$, there is no need to keep C_1 once D is inferred, because C_1 is redundant w.r.t. $\{C_2, D\}$. Therefore, we can remove C_1 and add D in its place.*

Definition 2.75 (Saturation up to Redundancy). *A* derivation *is a possibly infinite sequence of clause sets $N_1, N_2, \ldots$, such that either*
- *$N_{i+1} = N_i \cup \{C\}$ where C is deduced from N using a non-redundant inference;*
- *$N_{i+1} = N_i \setminus \{C\}$ where $C \in N_i$ is redundant w.r.t. $N_i \setminus \{C\}$.*

Given a clause C, if there is some k such that $\forall i \geq k.\ C \in N_i$, we say C is persistent. *The set of all persistent clauses is $N_\infty \overset{def}{=} \bigcup_{i=1}^n \bigcap_{j>i} N_j$.*

Definition 2.76 (Fairness). *A derivation $N_1, N_2, \ldots$ is* fair *w.r.t. some inference system I if, for every inference with premises in N_∞ and conclusion D, D is redundant in N_∞. In other words, it means that eventually, all non-redundant inferences have been performed — in practice, no inference should be postponed forever.*

Definition 2.77 (Completeness up to Redundancy). *An inference system I is* complete up to redundancy *iff, for any fair derivation $N_1, N_2, \ldots$, either:*
- *N_1 is satisfiable, and N_∞ does not contain $\bot$, or*
- *N_1 is unsatisfiable, and there is some $i \in \mathbb{N}$ such that $\bot \in N_i$.*

Theorem 2.2 (Superposition is Complete up to Redundancy [NR99]). *Superposition, as defined in Figure 2.1, is complete up to redundancy.*

Now, the notion of redundancy makes several interesting simplification rules usable. Some of them[4] are shown in Figure 2.2.

Subsumption and Non-Strict Redundancy

A classic rule of Resolution and Superposition, crucial in practice for saturation-based provers, is *subsumption.* But first, we need to introduce a slightly more powerful notion of redundancy.

Definition 2.78 (Non-Strict Redundancy). *A first-order clause C is* non-strictly redundant *w.r.t. a set of clauses N iff, for each ground instance $C\sigma$, $Gnd(N)^{\leq C\sigma} \vdash C\sigma$. See again [NR99] for more details.*

The definitions of saturation and completeness up to non-strict redundancy are trivially adaptable from Definitions 2.75 and 2.77 — note that a clause C can be removed from N_i if it is non-strictly redundant w.r.t. $N_i \setminus \{C\}$, because if $C \in N_i$ then C is always non-strictly redundant w.r.t. N_i.

Theorem 2.3 (Superposition is Complete up to Non-Strict Redundancy [NR99]). *Superposition, as defined in Figure 2.1, is complete up to non-strict redundancy.*

Definition 2.79 (Subsumption). *Let C and D be first-order clauses. Then, C* subsumes *D iff there is a substitution σ such that $C\sigma \subseteq D$ (multiset inclusion); in this case we write $C \sqsubseteq_\sigma D$, or $C \sqsubseteq D$ if the substitution is irrelevant. We might also write $l_1 \sqsubseteq_\sigma l_2$ for literals l_1 and l_2 if $l_1\sigma \vdash l_2$ according to a given decidable criterion (syntactic equality modulo symmetry of $\simeq$ here). If C subsumes D, then D is non-strictly redundant w.r.t. any set that contains C.*

---

[4] (DER) is not exactly a simplification rule according to definition 2.72, but it plays the same role.

Demodulation (Demod)

$$\frac{u \simeq v \vee C \qquad l \simeq r}{u[r\sigma]_p \simeq v \vee C}$$

if $l\sigma = u|_p$, $l\sigma \succ r\sigma$, $u \not\succ v$

Destructive Equality Resolution (DER)

$$\frac{x \not\simeq t \vee C}{C\sigma} \text{ (DER)}$$

$x \notin \text{freevars}(t)$, $\sigma = \{x \mapsto t\}$

Deletion of Duplicate or Absurd Literals

$$\frac{s \simeq t \vee s \simeq t \vee C}{s \simeq t \vee C} \text{ and } \frac{s \not\simeq s \vee C}{C}$$

Syntactic Tautology Deletion

$$\frac{s \simeq s \vee C}{\top} \text{ and } \frac{s \simeq t \vee s \not\simeq t \vee C}{\top}$$

Figure 2.2: Some Simplification Rules

The appeal of subsumption is that it is not directly linked to inference rules; wherever two clauses come from, we can check whether one subsumes the other. When a Superposition prover processes a clause C, it will first check whether C is subsumed by other known clauses — in which case the clause is deleted immediately —; else, it will delete clauses subsumed by C from its memory. Although the subsumption test is NP-complete, this single rule is very powerful and often reduces drastically the size of the search space; besides, there are some indexing techniques that reduce the number of subsumption tests to perform. Figure 2.3 defines powerful simplification rules that build upon the subsumption relation $\sqsubseteq$; they are also used in the E [Sch02] prover. In Section 4.3.2, we will extend the relation $\sqsubseteq$ on linear integer arithmetic literals and clauses, but the inference rules of Figure 2.3 will still be valid — they only assume that $C \sqsubseteq_\sigma D$ implies $C\sigma \vdash D$.

Subsumption

$$\frac{C \qquad C\sigma \vee R}{C \qquad \top}$$

Condensation

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\sigma}$$

where $l_2 \sqsubseteq_\sigma l_1$ and $(C \vee l_1)\sigma \sqsubseteq (C \vee l_1 \vee l_2)$

Contextual Literal Cutting

$$\frac{C \qquad D \vee l}{C \qquad D}$$

where $C \sqsubseteq D \vee \neg l$

Figure 2.3: Simplification Rules using Subsumption

Theorem 2.4 (Soudness of Subsumption-Based Simplifications)**.** *The rules presented in Figure 2.3 are sound.*

Proof. **Condensation** Let $D \stackrel{\text{def}}{=} C \vee l_1 \vee l_2$ be a first-order clause such that $l_2 \sqsubseteq_\sigma l_1$ and $(C \vee l_1)\sigma \sqsubseteq C \vee l_1 \vee l_2$. Note that the latter hypothesis makes condensation a simplification rule. To prove it sound, let $\sigma \leq \rho$ be a grounding substitution of D and $\mathscr{M}$ be a model of $D\rho$. By case on which part of the disjunction $(C \vee l_1 \vee l_2)\rho$ is true in $\mathscr{M}$:

- $\mathscr{M} \models C\rho$ implies that $\mathscr{M} \models (C \vee l_1)\sigma\rho$;
- $\mathscr{M} \models l_1\rho$ implies $\mathscr{M} \models (C \vee l_1)\sigma\rho$;
- $\mathscr{M} \models l_2\rho$ means that $\mathscr{M} \models l_1\rho$, since $l_2 \sqsubseteq_\sigma l_1$ and $\sigma \leq \rho$. Therefore $\mathscr{M} \models (C \vee l_1)\rho$ and the rule is sound for every ground instance of the conclusion. The core idea of this rule lies here: whether l_1 or l_2 is the chosen literal in D, l_1 is true, so we can merge both cases into one.

Contextual Literal Cutting Let C and D be first-order clauses with $C \sqsubseteq D \vee \neg l$ and freevars$(C) \cap$ freevars$(D) = \emptyset$. Let $\mathscr{M} \models C \wedge (D \vee l)$ and ρ be a grounding substitution for C and D. Let us prove $\mathscr{M} \models D\rho$ by case on which part of $(D \vee l)\rho$ is satisfied in $\mathscr{M}$:

- if $\mathscr{M} \models D\rho$, then we are done.
- if $\mathscr{M} \models l\rho$: since $\mathscr{M} \models C\rho$ by assumption on $\mathscr{M}$, and $C \sqsubseteq D \vee \neg l$, it means $\mathscr{M} \models (D \vee \neg l)\rho$, that is, either $\mathscr{M} \models D\rho$ or $\mathscr{M} \models \neg l\rho$. Since $\mathscr{M}$ is consistent, it cannot satisfy both $l\rho$ and $\neg l\rho$, so the second case is absurd, therefore $\mathscr{M} \models D\rho$.

Also, $C \wedge D \vdash C \wedge (D \vee l)$ is trivial, which makes $(D \vee l)$ redundant w.r.t. $\{C, D\}$. $\qquad\square$

As presented above, Superposition is already a very successful calculus, implemented in many theorem provers. In the next section, we give a short presentation of AVATAR, a recent extension of Superposition [Vor14]; the purpose of AVATAR is to deal more efficiently with boolean disjunctions, by delegating boolean reasoning to a (comparatively very efficient) SAT solver.

2.5 AVATAR

AVATAR [Vor14] extends the inference rules of classic Superposition to A-clauses and adds a few specific rules. In this work, we build on AVATAR and A-clauses because trails allow us to keep track of hypothesis and inferences that lead to a particular clause. In usual inference rules, trails are inherited, in the conclusion, from all premises. The general scheme for adapting a $k-$ary deductive inference rule (A) from Superposition to AVATAR is the following:

$$\frac{C_1 \leftarrow \Gamma_1 \quad \dots \quad C_k \leftarrow \Gamma_k}{D \leftarrow \Gamma_1 \sqcap \dots \sqcap \Gamma_k} \text{ (A)} \qquad \text{assuming} \qquad \frac{C_1 \quad \dots \quad C_k}{D} \text{ (A)}$$

Example 2.12 (Superposition Rule for AVATAR)**.** *For instance, the regular Superposition rule* (Sup) *from Figure 2.1, applied to the A-clauses $f(a) \not\simeq c \leftarrow l_1 \sqcap l_3$ and $a \simeq b \leftarrow l_2 \sqcap l_3$ with $>$ being* $LPO(f > a > b > c)$, *is:*

$$\frac{f(a) \not\simeq c \leftarrow l_1 \sqcap l_3 \qquad a \simeq b \leftarrow l_2 \sqcap l_3}{f(b) \not\simeq c \leftarrow l_1 \sqcap l_2 \sqcap l_3}$$

Two additional rules required by AVATAR are defined in Figure 2.4. AVATAR maintains a global set of boolean constraints that we call $S_{\text{constraints}}$; the goal of the boolean solver is to find a solution to $S_{\text{constraints}}$, otherwise the whole problem is unsatisfiable. Before presenting the rules, we need a notion of *boxing*, that is used to embed clauses into boolean literals.

Definition 2.80 (Boxing). *The* boxing operation *is an injective mapping (modulo alpha-equivalence, AC-properties, etc.) from some object x (a clause, an A-clause, some meta-level statement about a clause, etc.) to a boolean literal $\lVert x \rVert$, such that $\lVert \neg l \rVert = \neg \lVert l \rVert$ if l is a ground literal, and $\lVert \bot \rVert = 0$. For* AVATAR *we only need to box clauses (modulo renaming of variables, AC-properties of $\vee$, and symmetry of $\simeq$), but later, in Chapter 5, we will make use of boxing for other objects.*

Avatar Splitting splits an A-clause $C_1 \vee \ldots \vee C_n \leftarrow \Gamma$ (with $n \geq 2$) into *components* C_i where each component share no variables with other components ($\forall i \; j . \; i \neq j \Rightarrow \mathrm{freevars}(C_i) \cap \mathrm{freevars}(C_j) = \emptyset$). Indeed, in this case, $\overline{\forall}\,(C_1 \vee \ldots \vee C_n)$ is the same as $\left(\overline{\forall} C_1\right) \vee \ldots \vee \left(\overline{\forall} C_n\right)$, and we use the box $\lVert C_i \rVert$ to represent the validity of $\overline{\forall} C_i$. Each C_i can be deduced under the assumption that the boolean solver makes $\lVert C_i \rVert$ true; the boolean constraint $\Gamma \rightarrowtail \bigsqcup_{i=1}^{n} \lVert C_i \rVert$ is added to $S_{\mathrm{constraints}}$ as a side effect, so that the boolean solver has to make at least one $\lVert C_i \rVert$ true whenever Γ is true. Boolean atoms of the form $\lVert C_i \rVert$ where C_i is a clause component are added to a set S_{atoms}.

Avatar Absurd forbids the boolean solver to choose an assignment that makes Γ true, if $\bot \leftarrow \Gamma$ was deduced, by adding a boolean constraint $\neg\Gamma \overset{\mathrm{def}}{=} \bigsqcup_{b \in \Gamma} \neg b$ to $S_{\mathrm{constraints}}$.

Remark 2.14 (Trail Inheritance). *In the rule* (ASplit)*, we can soundly deduce $C_i \leftarrow \lVert C_i \rVert \sqcap \Gamma$ instead of $C_i \leftarrow \lVert C_i \rVert$, or even $C_i \leftarrow \lVert C_i \rVert \sqcap \Delta$ for any $\Delta \subseteq \Gamma$. In the original* AVATAR *paper, this is useless, but in Chapter 5 on structural induction, we will actually keep a subset of Γ in clauses obtained by splitting. The subset $\Delta \subseteq \Gamma$ that is kept is inherited in the conclusion.*

Avatar Splitting (ASplit)

$$\frac{C_1 \vee \ldots \vee C_n \leftarrow \Gamma}{\bigwedge_{i=1}^{n} \left(C_i \leftarrow \lVert C_i \rVert\right) \qquad \Gamma \rightarrowtail \bigsqcup_{i=1}^{n} \lVert C_i \rVert}$$

if each C_i is a component.

Avatar Absurd (A$\bot$)

$$\frac{\bot \leftarrow \bigsqcap_{i=1}^{n} b_i}{\bigsqcup_{i=1}^{n} \neg b_i}$$

Figure 2.4: AVATAR Rules

Example 2.13. *The A-clause $p(x) \vee q(y) \vee r(y, f(z)) \vee \neg s \leftarrow l_1 \sqcap l_2$ can be split as follows, with the boolean constraint $l_1 \sqcap l_2 \rightarrowtail \lVert p(x) \rVert \sqcup \lVert q(y) \vee r(y, f(z)) \rVert \sqcup \neg \lVert s \rVert$.*

$$\frac{p(x) \vee q(y) \vee r(y, f(z)) \vee \neg s \leftarrow l_1 \sqcap l_2}{p(x) \leftarrow \lVert p(x) \rVert \qquad q(y) \vee r(y, f(z)) \leftarrow \lVert q(y) \vee r(y, f(z)) \rVert \qquad \neg s \leftarrow \neg \lVert s \rVert \\ l_1 \sqcap l_2 \rightarrowtail \lVert p(x) \rVert \sqcup \neg \lVert s \rVert \sqcup \lVert q(y) \vee r(y, f(z)) \rVert} \text{(ASplit)}$$

The prover explores only one branch at a time because only clauses whose trail is true in the current valuation of the SAT-solver can participate in inferences. We skip over the details of simplification rules that can be found in [Vor14], but the takeaway is that cross-branch simplifications are possible (depending on whether the simplifying clause's trail subsumes the simplified clause's trail). Because of this, AVATAR competes well with other splitting techniques that were proposed for Superposition [RV01a, FW09].

Remark 2.15 (Incrementality). *Most SAT solvers are able to solve efficiently a sequence of boolean formulas $(F_i)_{i \in \mathbb{N}}$ such that $F_{i+1} = F_i \wedge G_i$ (the $i-th$ iteration adds G_i to the previous one).* AVATAR *naturally leverages this ability, because $S_{constraints}$ is only modified by adding new clauses to it.*

Chapter 3

Implementing Superposition in a Modular Way and Extending It

Many successful provers are based on Superposition [Sch02, McC10] [RV01b, WSH⁺07]. However, most of them are implemented in C, and heavily optimized, which makes for large code bases that are difficult to modify. During the course of this thesis, we favored a hands-on approach, by implementing new ideas to get a feeling of how they would actually behave on problems, discovering flaws in them, coming up with new ideas and loop. This requires that each iteration is short and does not introduce too many bugs that must be fixed immediately. Because of that, we preferred to use a high-level functional language, OCaml, for its decent performance and much better expressiveness and safety (in particular w.r.t. memory management), and use it to rewrite a new prover designed for flexibility rather than performance[1]. We felt there was a need for a chapter about the issues we faced in implementing the prover (and then extending it in various ways; in every following chapters there was a lot of implementation work) and the solutions we came up with. We started from a unit Superposition prover used in Matita [AT10] and gradually replaced and extended the code to handle full Superposition. Later, a part of the code was detached and made into a logic library (called Logtk; more details in Section 3.1). We also added support for typed logic (including polymorphism *à la ML*), a feature that to our knowledge was found in no other Superposition prover at the time we implemented it. The last versions of Zipperposition possess many features, a sizable fraction of which are *extensions*; its architecture is relatively modular.

We try to adopt a well-founded presentation of our implementation work: first, the basics of any theorem prover — terms, formulas, unification, etc. —; then, the Zipperposition prover itself that builds upon those basics. The rest of the thesis will be concerned with extensions to Superposition and their respective implementation.

3.1 Logtk: A Modular Library for First-Order Logic

Writing automated reasoning tools, in particular theorem provers, is a difficult engineering task that requires solving many difficult problems in addition to the actual deduction rules. As mentioned before, efficient provers for first-order logic, such as E [Sch02], SPASS [WSH⁺07] or Vampire [RV01b] are usually developed in a low-level language, over many years with great effort, making them a bad fit for rapid prototyping. Our goal with Logtk is to make prototyping easier by providing solid foundations that most systems need, including typing (and type inference), term representations, formulas, indexing, substitutions, unification algorithms, parsers for standard formats (e.g., TPTP) and various transformations (in particular, reduction to CNF

---

[1] There is also a Superposition prover in Prolog, Saturate [GNN98], but it has been unmaintained for years and only compiles on deprecated architectures. Besides, OCaml's strong typing helps prevent many errors.

of a set of formulas).

The OCaml language is a representative of the ML family, and as such is well-suited to symbolic manipulations and theorem proving. It was therefore a natural choice for such a library, as a trade-off between safety, expressiveness and performance. Logtk is free software, available at `https://www.rocq.inria.fr/deducteam/Logtk/index.html`.

We first present the fundamental building blocks for processing symbolic first-order logic: how to represent terms, formulas, substitutions, and how to manipulate them. We target polymorphic first-order logic, as described in the TFF-1 format [BP13] and Section 2.3, because it encompasses the usual untyped logic but brings more safety and expressiveness for many problems involving data structures, arithmetic, set theory, etc. Our library can also be used, in a lesser extent, for higher-order logic, and other term representations are relatively easy to implement from the existing ones.

3.1.1   Terms, Types and Formulas in OCaml

Interactions between terms, types and formulas are non-trivial. For instance, unifying terms also requires unifying their types, and substituting a type variable deep inside a formula should deal with all formula, term and type binders in between. In general, we make a distinction between bound variables, represented as De Bruijn indices [DB72], and free variables — allowed to participate in unification, and therefore useful for resolution procedures, type inference, etc. — that have meaningless numbers as names.

Example 3.1 (Term, Type and Formula Interleaving). *Given the type constructor* list : *Type* → *Type, the list signature from Example 2.3, and* $p : \Pi\alpha.\ \alpha \to o$*, the formula*

$$\forall \alpha : \textit{Type}.\ \forall x^{\alpha} : \alpha.\ p_{\langle \textit{list}(\alpha) \rangle}(x^{\alpha} ::_{\langle \alpha \rangle} (x^{\alpha} ::_{\langle \alpha \rangle} []_{\langle \alpha \rangle}))$$

mixes terms, types, and formulas in a non-trivial way. In particular, instantiating $\{\alpha \mapsto \textit{nat}\}$ *requires substituting* α *in formulas, terms and types.*

We could represent types, terms, and formulas with different OCaml types, but that leads to some repetitions and duplicated code for dealing with substitutions, unification and bound variables (especially type variables). Instead, we take a different path and define a single underlying type, named *scoped term*, roughly as shown in Figure 3.1. More variants, including extensible records[2], are not shown here for the sake of brevity.

The type `scoped_term` can be used to represent many term-like structures, which will then define more specific *views* and constructors that use `scoped_term` underneath. The sum type `term_kind` is a dynamic tag[3] that is used to efficiently discriminate between terms, types, formulas, etc. when downcasting a value of type `scoped_term` to a more specific type such as `Type.t`. For instance, a fragment of the `Type` module, in Figure 3.2, displays a type-centric view and dedicated constructors. Other types (such as higher-order terms) can be built on top of `scoped_term`[4] by providing similar constructors and views, and adding a variant to `term_kind`[5]. Also note the field `ty`, which points to another term representing the type, (or maybe another term for dependently-typed calculi). It is wrapped in an option so that the inductive type is actually well-founded[6].

---

[2] Extensible records are an interesting case, because they can appear both in terms and in their types. Since they are useful, e.g. in the meta-prover of Chapter 6, and make unification relatively subtle, we included them.

[3] Similar tags are very common in dynamic programming languages such as Python.

[4] because the term is responsible for manipulating properly scoped De Bruijn indices.

[5] OCaml features *open types* from version 4.02 upwards. They are similar to exceptions in that an open type can be declared somewhere and extended in many other places. That would be a good fit for our tags.

[6] In TPTP, the pseudo-type $tType is used as the top of the type hierarchy, as the "type of types"; its own `ty` field is therefore left empty.

```
type scoped_term = {
    ty : scoped_term option;
    term : term_cell
    kind : term_kind
}
and term_cell =
    | At of scoped_term * scoped_term
    | App of scoped_term * scoped_term list
    | Var of int
    | BoundVar of int
    | Bind of symbol * scoped_term * scoped_term
and term_kind =
    | FOTerm
    | HOTerm
    | Type
    | Formula
```

Figure 3.1: Declaration of `scoped_term`

Let us detail more precisely the code in Figure 3.2. First, the type `Type.t` (OCaml values that represent the logic types) is defined as a *private alias* of `scoped_term`, which means every `Type.t` can be safely coerced into the generic representation (e.g. for substitutions, unifications, etc.) but not conversely; down-casting must be done by calling `Type.of_term t` that checks the dynamic tag `t.kind`. The type `Type.view` is used for pattern-matching against types, using the eponymous function. Finally, some constructors that always return valid types (without down-casting) are defined.

Unification, substitutions, equality, hashconsing[7], handling of De Bruijn indices are all defined only once to operate on `scoped_terms`. It is also easier to mix term and type arguments, to quantify over types in formula-level binders, etc. because the underlying common structure will ensure that substitutions and unification remain correct.

`FOTerm` is the module of (typed) first-order terms. All constructors for leaf terms require a type argument (variables and constants are typed); other constructors just check the types of their arguments and deduce the type of their result. Every term is annotated with its type; the reason is that unifying terms also requires unifying their types, which must be easy to obtain. As is done for the `Type` module, `FOTerm` provides a *view* of terms into the following variant:

- `Var`: free variable, whose name is an integer;
- `BVar`: bound variable (De Bruijn index);
- `TyApp`: apply a term to a type (for instance nil(int));
- `Const`: constant term, parametrized by a symbol (and its type);
- `App` apply a term to a list of other terms. The first term should be composed only of `TyApp` and `Const` so that the term remains in the first-order fragment.

Remark 3.1 (Modularity). *In retrospect, it should be possible to make Logtk even more modular by functorizing every module over its dependencies. For instance, `Unif` (responsible for unification, see below) could be functorized over the concrete term representation, rather than working over `scoped_term`. A mathematical notion of "first-order term" would be represented by any type abiding by the following signature:*

```
type α view =
    | Var of int
    | App of symbol * α list
```

---

[7] hashconsing is used both to reduce the memory footprint of terms, formulas and clauses, and to make some operations much faster — in particular, comparison of terms by their unique ID. The curious reader might refer to [FC06] for another example of hashconsing in OCaml.

```
module Type : sig
    type t = private scoped_term

    type view = private
      | Var of int              (* Type variable *)
      | BVar of int             (* Bound variable (De Bruijn index) *)
      | App of symbol * t list  (* parametrized type *)
      | Fun of t list * t       (* Function type (arg list → ret) *)
      | Forall of t             (* explicit quantification *)

    val view : t → view          (* open the type's root *)
    val of_term : scoped_term → t option (* dynamic cast *)

    val var : int → t
    val app : symbol → t list → t
    val const : symbol → t
    val arrow : t → t → t
    val forall : t list → t → t
end

module FOTerm : sig
    type t = private scoped_term

    type view = private
      | Var of int            (* Term variable *)
      | BVar of int           (* Bound variable (De Bruijn index) *)
      | Const of Symbol.t     (* Typed constant *)
      | TyApp of t * Type.t   (* Type parameter *)
      | App of t  * t list    (* List of parameters *)

    val view : t → view
    val of_term : scoped_term → t option

    val var : ty:Type.t → int → t
    val bvar : ty:Type.t → int → t
    val const : ty:Type.t → symbol → t
    val tyapp : t → Type.t → t
    val app : t → t list → t
end
```

Figure 3.2: View and Constructor for Type and FOTerm

```
module type FOTERM = sig
    type t
    val view : t → t view
    val build : t view → t
    val fold : (α view → α) → t → α
end
```

Then we could define several implementations of this signature (e.g., hashconsed terms, and non-hashconsed terms); algorithms on terms would be functorized over FOTERM.

3.1.2 Substitutions

We distinguish here *substitutions*, that is, say mapping from free variables to terms (or types), from *environments* that are used in conjunction with bound variables and the De Bruijn indexing system. Let us examine substitutions more closely. In many cases (rewriting, resolution...), unification works on free variables, but often requires renaming:

- for term rewriting, a subterm $t|_p$ is matched against the left-hand side of a rule $l \to r$ so is it necessary for t and l not to share any variable;
- for resolution (or Superposition), binary inferences such as

$$\frac{C \vee l_1 \qquad C' \vee \neg l_2}{(C \vee C')\sigma} \text{ if } l_1\sigma = l_2\sigma$$

will require the two clauses to share no variable prior to unification.

To avoid renaming, which can be costly, some techniques have been used by provers such as SPASS [WSH+07] or Otter [McC95], involving so-called *variable banks*. Assuming variables are indexed by natural numbers, a variable bank is an array that maps each index $0 \le i < \text{MAXVAR}$ (where MAXVAR is a higher bound on the total number of distinct variables) to either:

- nothing (variable not bound), or
- a tuple (term, varbank) where varbank is a variable bank (possibly the same) that provides bindings to free variables of term; if term is a variable, lookup recurses with it and the new bank. Variable banks can therefore point to one another in a cyclic way, for instance after unifying the terms $f(x, g(z))$ and $f(g(y), y)$ where x and z live in one bank and y in another one.

This technique works fine and is efficient but suffers from two limitations:

- it requires substitutions to be mutable arrays (rather than functional-friendly immutable structure that can safely be kept for generating proofs, or stored in data structures);
- it requires allocating big arrays (as big as the maximal authorized variable index), which limits the number of substitutions that are allowed to live simultaneously.

To overcome those limitations we use a persistent representation and a notion of *scope*, inspired by the code[8] of iProver [Kor08].

A *scope* is a value that represents one interpretation for free variables, which means that the same variable can have distinct bindings in distinct scopes. In our implementation a scope is simply an integer. Substitutions and unification therefore map pairs (variable, scope) to pairs (term, scope), rather than directly variable to term. A substitution is a finite mapping from pairs to pairs (currently a persistent hash table, but balanced trees or mere linked lists could do too). Figure 3.3 shows the type signatures of some operations on substitutions[9]. Note that if one does not wish to rename variables (e.g. for type inference), one can use only one scope and essentially fall back to the usual representation of substitutions. We write $(\!|t|\!)_i$ for the term t interpreted in the scope i, and trivially extend the notation to literals and clauses.

---

[8] it is, to our knowledge, the first occurrence of this technique.

[9] The type renaming is abstracted into a function for clarity.

When a substitution σ has been computed by unification or matching (see Section 3.1.3), for instance after a resolution step between two clauses $(\!|C|\!)_0$ and $(\!|C' \vee \neg l_2|\!)_1$, we need to *apply* it to build a new clause $(C \vee C')\sigma$. Here we need be careful because, in $C \vee C'$, some variables are bound in scope 0, some other in scope 1; we need to evaluate $\big((\!|C|\!)_0 \vee (\!|C'|\!)_1\big)\sigma$ instead. Now the question is: how shall we deal with free variables that are not bound in the substitution?

For instance, say we have a substitution $\sigma \stackrel{\text{def}}{=} \big\{(\!|x|\!)_0 \mapsto (\!|f(x)|\!)_1, (\!|x|\!)_1 \mapsto (\!|y|\!)_1\big\}$ (remember that $(\!|x|\!)_0$ and $(\!|x|\!)_1$ are distinct variables because they live in different scopes). To evaluate the clause $(\!|p(x, y)|\!)_0 \sigma \vee (\!|p(x, y)|\!)_1 \sigma$ we must rename one of $(\!|y|\!)_0$ and $(\!|y|\!)_1$ because *they are distinct variables*. To do so, applying a substitution requires an object called *renaming*, that builds an injection from (`variable`, `scope`) to `variable`; the result, as expected, will be alpha-equivalent to $p(f(x), y) \vee p(x, x)$ (renaming $(\!|y|\!)_1$ to x, and $(\!|y|\!)_0$ to y).

```
type scope = int
type subst = (scoped_term * scope * scoped_term * scope) list
type renaming = (variable * scope) → variable

val unify : scoped_term → scope → scoped_term → scope → subst option
val rename : renaming → variable → scope → variable
val apply : renaming → subst → scoped_term → scope → scoped_term
```

Figure 3.3: Operations on Substitutions

3.1.3 Algorithms

Many algorithms are very often useful for processing logic formulas. Some particularly useful ones — for our purposes — are implemented in Logtk.

Unification and Matching

The usual first-order unification and matching algorithms are implemented only once, on the `scoped_term` shared structure. Their type signature is shown in Figure 3.3. The algorithm can be used with any view of `scoped_term`, including `FOTerm.t` and `Type.t`. We need to recursively unify subterms pairwise, but also types. Indeed, term-level variables can have polymorphic types, as is shown in the few clauses of Example 2.3 and Figure 3.4 that declare polymorphic lists and some of their axioms. Note the presence of Skolem symbols head and tail in the inversion axiom, that encode the fact that any non-nil list is necessarily an application of (::). Which such axioms, we may need to unify both terms and types (the type variable α) when working with concrete lists such as $1 ::_{\langle \text{int} \rangle} []_{\langle \text{int} \rangle}$; if some variables are *unshielded* (i.e., they appear under some equation, but under no function symbol) then unifying types becomes crucial for soundness (see Remark 3.2). We will see other examples of theories with similar axioms in Chapter 5, about induction.

$$
\begin{aligned}
[] &: \Pi\alpha.\, \text{list}(\alpha) \\
(::) &: \Pi\alpha.\, (\alpha \times \text{list}(\alpha)) \to \text{list}(\alpha) \\
\text{head} &: \Pi\alpha.\, \text{list}(\alpha) \to \alpha \\
\text{tail} &: \Pi\alpha.\, \text{list}(\alpha) \to \text{list}(\alpha) \\
\forall x : \alpha.\, \forall l : \text{list}(\alpha).\, x ::_{\langle\alpha\rangle} l &\not\simeq []_{\langle\alpha\rangle} \quad \text{(non-overlap)} \\
\forall l : \text{list}(\alpha).\, l \simeq []_{\langle\alpha\rangle} \vee l &\simeq \text{head}_{\langle\alpha\rangle}(l) ::_{\langle\alpha\rangle} \text{tail}_{\langle\alpha\rangle}(l) \quad \text{(inversion)}
\end{aligned}
$$

Figure 3.4: A Polymorphic Theory containing Unshielded Variables

Remark 3.2 (Typing and Unsoundness). *If unification were to ignore types of variables during unification, the prover becomes unsound, as the following example demonstrates. We use the two classic types bool (the two boolean values true and false) and unit (unit type, containing exactly one value **1**). The following theory is satisfiable:*

$$true \not\simeq false$$
$$\forall x^{bool} : bool. \; x^{bool} \simeq true \vee x^{bool} \simeq false$$
$$\forall y^{unit} : unit. \; y^{unit} \simeq \mathbf{1}$$

but, if we ignore types, the following derivation of $\bot$ is possible (successively unifying y^{unit} with true, then false):

$$\cfrac{\cfrac{true \not\simeq false \quad y^{unit} \simeq \mathbf{1}}{\mathbf{1} \not\simeq false} \text{(Sup)} \quad y^{unit} \simeq \mathbf{1}}{\cfrac{\mathbf{1} \not\simeq \mathbf{1}}{\bot} \text{(EqRes)}} \text{(Sup)}$$

Reduction to Clausal Normal Form

It is often practical to transform a given problem into CNF (clausal form, see Definition 2.53). Resolution provers, for instance, require it. However, in many cases they prefer to rely on an external prover (for instance SPASS [WSH$^+$07]). Here, we can't do that, first because Logtk is intended to be self-contained, and because our terms may be more general, for they are typed and may contain additional constructs such as records or curried application. Naive CNF is quite easy to implement; however, many real problems cause naive CNF to blow up because the number of clauses is exponential in the size of the input formula; many others do suboptimal Skolemization (Definition 2.52). Therefore, we implemented CNF reduction with miniscoping and formula renaming[10], following [NW01]. This is enough to avoid all exponential blowup.

Indexing

Saturation provers rely heavily on unification. When the clause set grows, *term indices* become necessarily to keep a good inference rate. In Logtk we define several such indices for first-order (typed) terms, parametrized by the data stored at the leaves of the index. Conceptually, a term index maps each term to a set of values of some type (for instance, a pair (`clause * position`) can be used for Superposition provers), and allows to retrieve values by unification or matching with a query term. We provide several indexing schemes for theorem provers, rewriting systems, etc.

- *fingerprint indexing* [Sch12] as a general purpose index;
- *feature vector indexing* [Sch04] for subsumption checking;
- *perfect discrimination trees* [RSV01] for rewriting, and *non-perfect discrimination trees* as a general purpose index.

The index implementations are all purely functional, which is facilitated by their tree-like structure (most often a prefix tree). This can be useful in contexts where duplicating an index might be necessary, for instance in Tableaux provers or for other splitting-like inference rules.

Let us focus on the implementation of the discrimination trees. The classic way to implement them is based on the use of *flatterms*, i.e., terms represented as a flat array of symbols (including a special symbol $*$ that represents variables in imperfect discrimination trees; perfect discrimination trees also allow variables in flatterms). However this representation is not convenient for many other operations, and it is incompatible with any kind of subterm sharing.

---

[10] although the criterion for triggering the renaming of a formula is simpler than the optimal one presented in [NW01].

Conversion between tree-like terms and flatterms can be very costly. A pathologic example would be, in the context of term rewriting, the application of the rule $s(x) + y \mapsto s(x + y)$ that describes the addition in Peano arithmetic to the term $\overline{500,000} + \overline{500,000}$ (where $\overline{n}$ is the encoding of $n \in \mathbb{N}$ into the Peano term $s^n(0)$). We would build a flatterm of size $1,000,000$, match it against a shallow rule, only to obtain the term $s(\overline{499,999} + \overline{500,000})$ which would then be converted to a flatterm of the same size, matched, and so on. This series of conversions would be very expensive.

Our solution here is to perform a *lazy* conversion to flatterms, by using a specialized iterator type that provides the required next and skip operations. The type of the iterator is shown in Figure 3.5 and is discussed further. At any point in the traversal of a term (we traverse the term and the corresponding branches of the discrimination tree) we remember its siblings and the siblings of its superterms. When the term has been fully traversed, calling next or skip will return None. This iterator type is persistent, which makes backtracking (exploring several branches of a discrimination tree) trivial.

Listing 3.1: Interface of Lazy Flatterm

```
type iterator

val skip : iterator → iterator option
val next : iterator → iterator option
val flatten : FOTerm.t → iterator
```

Listing 3.2: Implementation of Lazy Flatterm

```
module T = FOTerm

type iterator = {
    cur_term : FOTerm.t;   (* current sub-term *)
    stack : FOTerm.t list list;
}

let open_term stack t = match T.view t with
  | T.Var _
  | T.BVar _
  | T.TyApp _
  | T.Const _ →
    Some {cur_term=t; stack=[]::stack;}
  | T.App (_, l) →
    Some {cur_term=t; stack=l::stack;}

let rec next_rec stack = match stack with
  | [] → None
  | []::stack' → next_rec stack'
  | (t::next')::stack' →
    open_term (next'::stack') t

let skip iter = match iter.stack with
  | [] → None
  | _::stack' → next_rec stack'

let next iter = next_rec iter.stack
let flatten t = open_term [] t
```

Figure 3.5: Lazy Conversion to Flatterms

In Figure 3.5, the function open_term is used to flatten its term argument's root (given a stack of parent terms and their siblings) into a new iterator; flatten starts the flattening of a whole term (meaning the surrounding stack is empty). The function next and skip both use

the stack; the only difference is that the latter ignores the current term's siblings (if any).

3.1.4   Architecture

Figure 3.6 contains the dependency graph of the most important modules of Logtk. We include it to give the reader an overall view of how Logtk is organized.

- `Symbol` defines the type of symbols and many operations on it;
- `Position` describes positions in types, terms, etc.;
- `ParseLocation` represents location in input files;
- `ScopedTerm`, as explained above, is the generic tree representation responsible for scoping, traversal, hashconsing, and comparisons;
- `PrologTerm` is used as a flexible AST for parsers to output; it uses strings as variable names and does not enforce hashconsing nor proper scoping;
- `Type` builds on `ScopedTerm` to represent polymorphic types;
- `FOTerm` and `HOTerm` build on `ScopedTerm` and `Type` to represent respectively first-order and higher-order typed terms;
- `Formula` represents classical formulas over an arbitrary term type (for instance `FOTerm`) using a functor;
- `Precedence` and `Ordering` are used for term orderings (RPO, KBO);
- `Signature` uses `Symbol` and `Type` to represent a signature as a finite map from symbols to types;
- `TypeInference` features Hindley-Milner style type inference for first-order and higher-order terms — it is used to convert untyped `PrologTerm.t` into `FOTerm.t` or `HOTerm.t`;
- `Skolem` deals with Skolem symbols;
- `Cnf` transforms `Formula.FO.t` (a formula whose leaves are of type `FOTerm.t`) into clauses;
- `Substs` contains the representation of substitutions, and operations to build them and apply them to types, terms and formulas;
- `Unif` contains unification algorithms;
- `Index` defines abstract types and signatures for term and clause indices;
- `FeatureVector`, `Fingerprint`, `NPDtree` and `Dtree` are implementation of term and clause indices;
- `Rewriting` implements some basic term rewriting techniques.

3.1.5   Simple Tools

The interface provided by Logtk makes it well-suited for writing tools that process (first-order) logic objects. Several such tools are provided in the library, both for their usefulness and as examples of how to use it. A quick description of those tools:

proof_check_tstp calls external provers to check traces a theorem prover can print upon success. For instance if E [Sch02] proves a theorem, it can print the DAG of the inferences it had to perform. `proof_check_tstp` can then parse this DAG (in the TSTP [Sut09] format), and check the validity of every deductive inference by calling one or more trusted provers. Steps that only preserve satisfiability, such as skolemization, are not checked;

cnf_of_tptp parses TPTP files, infers types, and prints the clausal normal form (CNF) of the parsed formulas;

type_check_tptp is a simple type-checker for TFF0 and TFF1 problems, including some type inference for wildcards $_ (type arguments omitted in terms because they can be inferred from the context);
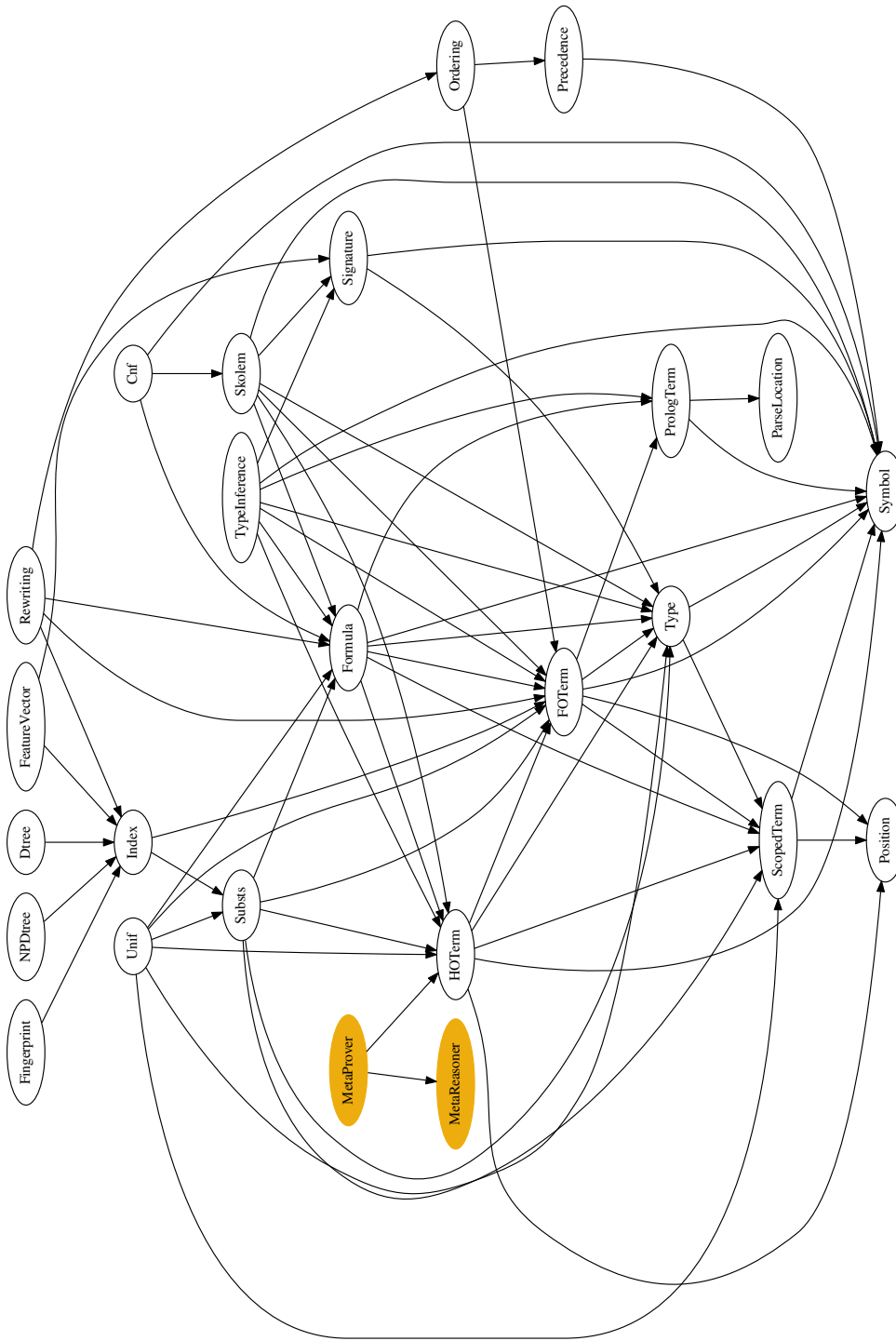
Figure 3.6: Dependency Graph of Logtk

detect_theories can use the implementation of a *meta prover* [BC13] — see also Chapter 6
— to detect instances of axiomatic theories in a problem. For instance it will detect the

presence of an abelian group in `RNG008-4.p`[11];

orient reads a term rewriting system from a file, and looks for a LPO precedence that orients all rules left-to-right (thus proving the termination of the system in this case. Our tool can then print the witness precedence if required). The part that attempts to orient rewrite rules using a LPO is one of the modules provided in Logtk;

hysteresis is a more sophisticated tool that currently serves as a pre-processor for E. It detects theories using the aforementioned meta-prover, collects associated rewrite systems (if any), attempts to orient them (see previous tool) using a LPO and sends the modified problem to E. We also add to modify E so that it could handle simply-typed logic.

3.1.6 Discussion

Many provers ship with some internal library that is designed to cope with the same problems as Logtk, for instance E [Sch02] comes with CLIB, Prover9 [McC10] with LADR, some other with the Dedam [NRV97] system, etc. However, there are several significant differences with most of those libraries, and ours.

First, Logtk is written in OCaml. While the choice of a programming language is important for such a performance-sensitive area as Automated Theorem Proving, we made this trade-off to make prototyping much faster than in all the aforementioned C libraries. OCaml, as a dialect of ML, has a long record track of usage for symbolic reasoning, including the implementation of Coq [HKPM97]. We clearly cannot hope to beat optimized C in terms of performance, but our goal with Logtk is to make prototyping and writing decent theorem provers much easier. Similarly, abstractions like iterators (on subterms, subformulas, the types in a term, etc.) are pervasively used and exposed to make the code simpler and avoid repeating the same recursive functions everywhere. This kind of abstraction again brings more expressiveness to the user (and implementer of the library)[12]. Stronger typing (absence of `NULL`, polymorphism, modules) and the presence of recursive algebraic types and pattern-matching also improve readability and safety. For instance the formula representation is an algebraic type with 14 cases; checking the exhaustiveness of pattern-matching helps ensuring every case is dealt with.

Providing functional structures for types such as substitutions, term indices, and signatures is also a significant difference. More allocations are needed (although OCaml's GC is very good at allocating short-lived structures) but reasoning about the program behavior becomes easier; again, less time spent debugging improves the programmer's productivity.

The library comes with small tools that illustrate the use of some of its core features – type-checking, reduction to CNF, etc. – but is separated from Zipperposition. We deliberately kept the superposition-specific structures outside of Logtk (in particular, the representation of clauses which is very specific) so as not to constrain users to follow the same design choices. It is possible, however, that some structures we use in Zipperposition for linear arithmetic migrate back to Logtk (e.g., linear expressions)[13].

Since Logtk is still very young, we can't evaluate yet how easy (or difficult) it is for someone to use it without any assistance for the authors. Good documentation and openness to contributions will be necessary to make it as easy as possible. The choice of the very permissive BSD2 license should make Logtk easy to use and contribute to.

---

[11] `RNG008-4.p` is a ring theory problem available in TPTP. After installing Logtk, the command
`$ detect_theories $TPTP/Problems/RNG/RNG008-4.p`
should print some detected axioms and theories, including the additive abelian group.

[12] The performance impact is hard to evaluate but shouldn't be high, especially outside of critical paths.

[13] Some changes needed for Zipperposition have been made, when useful in general. For instance, multisets in which elements can have very large multiplicities are often useful for linear arithmetic (Chapter 4): $n \cdot t$ is a shortcut for $\sum_{i=1}^{n} t$, a sum of n elements, that will then be compared using $\succ$.

3.2 Zipperposition: a Modular Theorem Prover

Logtk has been used to implement our experimental theorem prover, Zipperposition. Zipperposition is based on the Superposition calculus and has been modified, during our thesis, to include a simple implementation of AVATAR and to experiment on arithmetic, polymorphism, and other extensions. Many components of Logtk are used, including the typing system, type inference, the TPTP parser, term indexes, unification algorithms, subterm positions, reduction to CNF, etc. One benefit is that, would first-order terms in Logtk be extended with new variants (records, sum types, curried application, etc.), few changes would be required at all in Zipperposition to support the extension.

3.2.1 Architecture

Figure 3.7 shows the dependency graphs of some of the most important modules of Zipperposition. In topological order, let us explain their respective rôle in a few words.

- `Monome` helps represent integer linear expressions, as defined in Chapter 4;
- `ArithLit` defines arithmetic literals, from the same chapter (equations, comparisons, and divisibility statements on linear expressions);
- `Literal` contains the representation of literals, including arithmetic ones, and many operations on literals;
- `CompactClause` is a small modules used to represent clauses in a compact way — mainly used in proof traces;
- `Proof` represents proof traces (the inference DAG);
- `PFormula` pairs a formula (from Logtk) together with a `Proof.t`;
- `ClauseContext` is used in induction, see Chapter 5;
- `BoolSolver` is a generic interface to boolean solvers (SAT and QBF solvers) so that different solvers can be used the same way;
- `BBox` helps with boxing clauses (and other statements) into boolean literals, a requirement for AVATAR;
- `Selection` defines selection functions (a heuristic for Superposition);
- `Ctx` contains some global parameters (selection function, ordering, sets of inductive types, etc.) encapsulated into a functor;
- `Clause` defines first-order clauses and a number of combinators and tools to process clauses. It is clearly a central component of Zipperposition;
- `ClauseQueue` contains heuristics to choose the clause to process in the saturation loop (see below);
- `ProofState` holds the sets of clauses required by the saturation loop, sets of rewrite rules, etc.;
- `PEnv` defines some pre-processing operations that occur before the main saturation loop starts;
- `Env` is a crucial component, as the dependency arrows show. It stores the set of inference rules, simplification rules, an instance of `Ctx`, an instance of `ProofState`; in general it contains everything that is required for Superposition — and other calculi — to perform their inferences. More details will be given below;
- `Saturate` defines the main saturation loop, parametrized by an instance of `Env` that defines which rules and clause sets shall be used;
- `Extensions` defines a mechanism to plug *extensions* into Zipperposition— that is, modules defining new axioms, inference rules, simplification rules, and so on. The gold-colored boxes are extensions that can be disabled or enabled easily;
- `Avatar`, `ArithLit` (arithmetic), `Chaining` (a calculus that deals with total orders), `Superposition` (standard Superposition), `MetaProverState` (interface to the meta-prover, see Chapter 6),

`Avatar`, and `Induction_sat` and `Induction_qbf` (inductive reasoning, Chapter 5) are extensions defining various calculi.
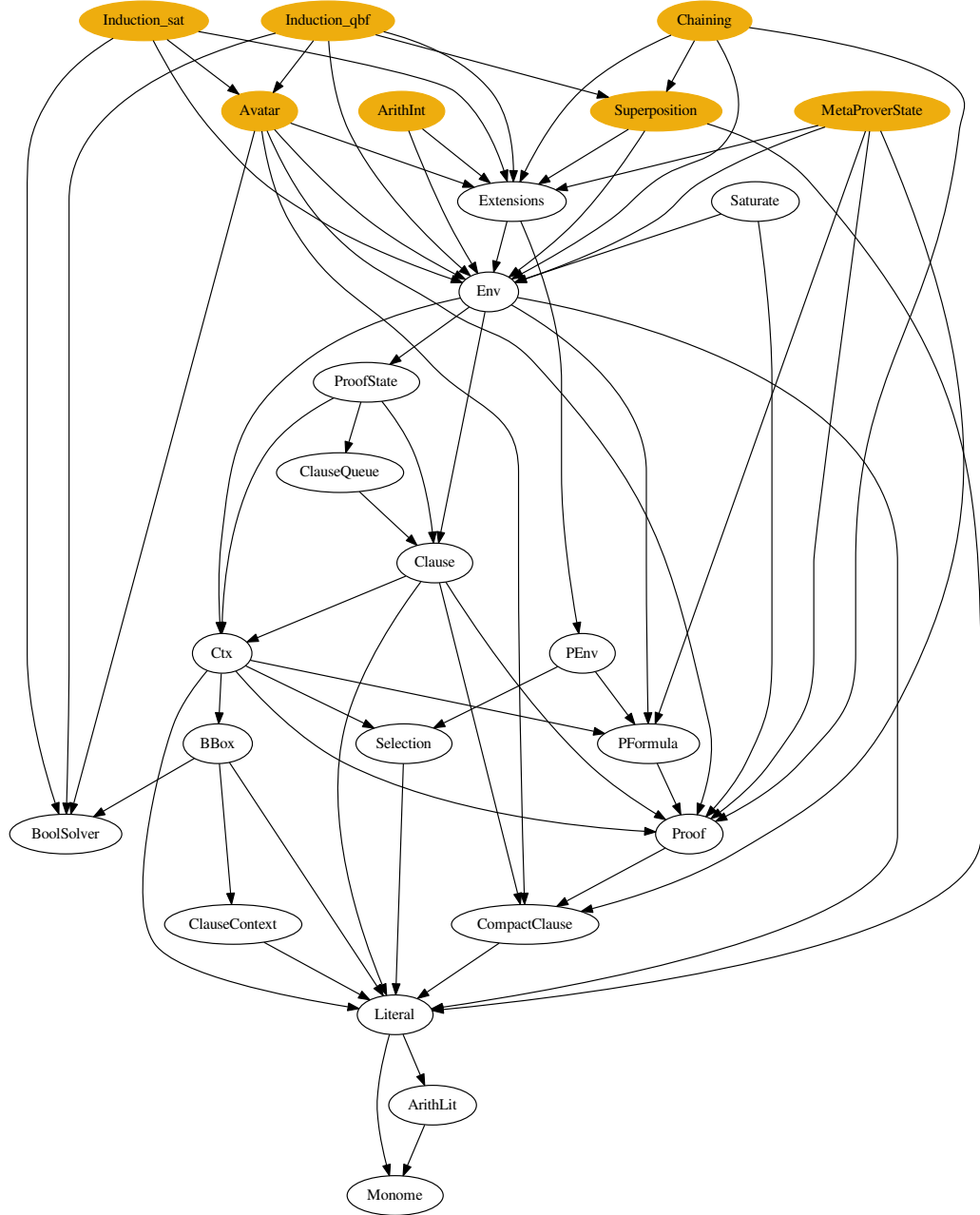

Figure 3.7: Dependency Graph in Zipperposition

The Central Rôle of `Env`

As mentioned before, `Env` plays a very important rôle in the modular architecture of Zipperposition. It stores most state required by the saturation algorithm, and also keeps track of which inference rules, simplification rules, concrete redundancy criteria, etc. have been defined so

far by extensions. Similar to what `Ctx` does, `Env` defines a functor (parametrized by `Ctx.S` here) that returns a module rich with global state. To process a new problem, those functors are instantiated, so they do not share the "global" state with previous instantiations; yet, from the point of view of functions defined within the functor, most parameters are global, which simplifies a lot the code — no need for explicitly carrying parameters around in each function call.

To further illustrate our point, we present snippets of the interface of `Env.S` (obtained by applying the functor `Env.Make`) and comment them. First, some types are defined:

```
module type S = sig
  module Ctx : Ctx.S
  module C : Clause.S with module Ctx = Ctx
  module ProofState : ProofState.S with module C = C and module Ctx = Ctx

  type binary_inf_rule = C.t → C.t list
  type unary_inf_rule = C.t → C.t list

  type simplify_rule = C.t → C.t (* Simplify clause *)
  type redundant_rule = C.t → bool (* Clause is redundant? *)
  type is_trivial_rule = C.t → bool (* Cheap test for redundancy *)

  type multi_simpl_rule = C.t → C.t list option
```

Basic operations follow, to modify the state by adding clauses to sets of clauses, or defining new rules:

```
  val add_passive : C.t list → unit (* Add passive clause *)
  val add_active : C.t list → unit  (* Add active clause *)
  val add_simpl : C.t list → unit   (* Add simplification clause *)

  val remove_passive : C.t list → unit   (* Remove passive clauses *)
  val remove_active : C.t list → unit    (* Remove active clauses *)

  val add_binary_inf : string → binary_inf_rule → unit
  val add_unary_inf : string → unary_inf_rule → unit
```

Then, higher-level operations are directly used by `Saturate` — the main saturation loop — to perform inferences, simplify clauses, etc. using the global state. The function `next_passive` picks a clause from the passive set (according to heuristics defined in `ClauseQueue`, see Figure 3.7); some functions simplify the given clause w.r.t. the active set, or simplify clauses from the active set using the given clause, or apply all the inference rules to obtain new clauses.

```
  val cnf : PFormula.Set.t → C.t list (* Reduce formulas to CNF *)

  val next_passive : unit  → C.t option (* Next Given Clause *)

  val do_binary_inferences : C.t → C.t list
  val do_unary_inferences : C.t → C.t list

  val is_trivial : C.t → bool
  val is_active : C.t → bool
  val is_passive : C.t → bool

  val simplify : C.t → C.t (* Basic, Cheap Simplifications *)

  val backward_simplify : C.t → C.t list * C.t list
  (* Perform backward simplification with the given clause. It returns the
       list of clauses that become redundant, and the list of those
       very same clauses after simplification. *)

  val forward_simplify : C.t → C.t
  (* Simplify given clause wrt active set *)

  val remove_orphans : C.t list → unit
```

```
  (* remove orphans of the (now redundant) clauses *)

  val generate : C.t → C.t list (* Perform all generating inferences *)

  val is_redundant : C.t → bool (* Is the clause redundant wrt active set? *)

  val subsumed_by : C.t → C.t list
  (* List of active clauses subsumed by the given clause *)

  val all_simplify : C.t → C.t list
  (* Use all simplification rules to convert a clause into a set
      of maximally simplified clause (or [] if they are all trivial). *)
end
```

The Saturation Loop

The `Saturate` module uses a `Env.S` instance and provides two functions implementing the saturation algorithm:

```
type result = Sat | Unknown | Timeout | Unsat of Proof.t

module Make(E : Env.S) : sig
  val given_clause_step : unit → result
  (** Perform one step of the given clause algorithm *)

  val given_clause: ?steps:int → ?timeout:float → unit → result * int
  (** Run the given clause algorithm until a timeout occurs or a result
      is found. It returns a tuple (result, number of steps done) *)
end
```

3.2.2 Extensibility

Zipperposition is designed so that additional features (typically, new inference systems that are compatible with Superposition) can be added through *extensions*. In a nutshell, an extension (the type `t` in the following listing) is a list of actions that can be performed on a `Env.S` instance — mainly, calls to `Env.add_binary_inf`, `Env.add_unary_inf`, and other functions that add simplification rules and redundancy rules. Note the use of a *first-class module* as a parameter to actions.

```
type action = Action of ((module Env.S) → unit)

type t = {
  name : string;
  actions : action list;
}

val register : t → unit (* register new extension *)
val all : unit → t list (* all registered extensions *)
val apply_env : env:(module Env.S) → t → unit (* activate extension by side-effect *)
```

In Zipperposition-0.5, as we can see in Figure 3.7, there are several extensions that implement deductive inference systems, including Superposition and AVATAR as described in Sections 2.4 and 2.5. In practice, some extensions depend on other extensions (e.g., AVATAR depends on some of the Superposition rules).

3.2.3 Lessons Learnt from Implementing Zipperposition

Implementing a theorem prover (almost) from scratch, even using a well-known calculus, is challenging. A large collections of algorithms has to be coded efficiently; some of them can be

quite sophisticated (for instance, a first-order CNF procedure that avoids exponential blowup). We also wrote new extensions to Superposition (polymorphic terms, arithmetic, induction, etc.). Implementing calculi whose exact rules are in flux can be challenging too. OCaml, being expressive and quite safe, was indeed a good language for prototyping, but it still cannot prevent all errors; finding and fixing errors in our prototype was one of the biggest difficulty in the whole implementation effort. To debug Logtk and Zipperposition, we combined several approaches:

- in Logtk, unit tests and random tests are used to check that some functions work at least on some inputs. Random testing (that checks a universal invariant of type $\alpha \rightarrow$ `bool` against a set of randomly-generated instances of the type α) proved particularly useful to test implementations of term indexing with properties such as "all terms retrieved from an index unify with the query term", or "an index returns every term that unifies with the query term".

- Zipperposition is mostly tested as a whole, against problems from TPTP or the *Pelletier problems* [Pel86]. Bugs can be categorized in three different classes:

  soundness bugs cause the prover to output "unsat" on a satisfiable problem, because it found an incorrect derivation. Those are usually relatively easy to find, by asking the prover to output a derivation and staring at it for long enough. Derivations can be printed either in text form, or using a graphical output based on graphviz[14], an example of which will be presented later, in Section 4.6.5.

  completeness bugs cause the prover to stop and output "sat" on unsatisfiable problems, failing to find a derivation even though, in theory, they should find one or diverge. Such bugs are extremely hard to find because they require to make the prover print every small step it takes and stare at it, hoping to find a point where it should have deduced a clause and failed to do so. We did not find a satisfying solution to this kind of problems.

  other bugs cause the prover to crash, or have no direct incidence on the correctness of its results. They can be debugged with assertions, print statements, etc.

Conclusion

As explained before, implementation is a crucial part of Automated Theorem Proving. A technique that works in theory but is terribly slow and redundant in practice will not be very useful.

That is why a significant portion of our PhD was dedicated to implementation. This chapter presented the most important softwares we developed, their architecture, and their specificities; Logtk is a general-purpose library for typed first-order logic, and Zipperposition is a modular Superposition prover built on top of Logtk.

Now that we presented Logtk and Zipperposition, we can present the three main contributions of this thesis, and their implementation on top of our theorem prover. Our calculus for linear integer arithmetic (Chapter 4) was quite challenging to implement (its cousin [Wal01] that deals with Superposition on rationals was never implemented, as far as we know). In induction as well (Chapter 5), search space problems and the use of AVATAR (a technique that was published at two thirds of this thesis) required some experimentation. In both cases we modified and extended Zipperposition to study the feasibility of our new techniques. Theory detection (Chapter 6) was implemented too, but as a sub-library in Logtk; it was later interfaced with the induction plugin of Zipperposition so it could suggest inductive lemmas. Consequently, each chapter will feature a section on implementation or experimental evaluation.

---

[14] see `http://graphviz.org`.

Chapter 4

Linear Integer Arithmetic

Introduction

Superposition, as presented in Section 2.4, is an efficient calculus for automated reasoning with equality on uninterpreted function symbols. However, some important theories such as Presburger Arithmetic are very difficult to deal with in a purely axiomatic framework. Many efforts have been put into developing calculi for superposition modulo $\mathcal{T}$, for theories $\mathcal{T}$ ranging from AC [BG95] (associativity-commutativity) to linear arithmetic over rationals using several distinct approaches [Wal01, AKW09, KV07]. In this chapter, we present a calculus for linear integer arithmetic that extends superposition in a framework of saturation up to redundancy, unlike SPASS+T [PW06] or hierarchic superposition [BGW94, BW13] that both rely on an external black-box solver to perform theory reasoning. Such solvers do not deal with first-order logic, and will only deal with the satisfiability of formulas over a finite number of ground terms. Our technique, on the contrary, can deduce non-ground formulas containing arithmetic terms (a simple example is deducing $f(x) \simeq 1$ from $g(x) \simeq 2$ and $f(x) + 1 \simeq g(x)$).

The extension of superposition we develop here deals with equations, comparisons and divisibility in structures that include $\mathbb{Z}$. Such structures are of great interest in fields as important as cryptography, where divisibility and modular arithmetic are pervasive, or program verification where many proof obligations include some integer arithmetic — most often using bounded representations for integers — for looping or accessing arrays elements. Our calculus, intuitively, deals with divisibility statements the same way usual superposition deals with equations, by rewriting terms that are "big" in some ordering into terms that are smaller, but with subtleties that come from interactions between equality, inequality and divisibility ($a \simeq b$ implies $n \mid a - b$ for all n; $a \leq b \wedge b \leq a$ implies $a \simeq b$) and even between divisibility statements in distinct rings ($4 \mid 2 \cdot a + b$ implies $2 \mid b$). We also try to counteract some particularly glaring sources of inefficiencies; in particular, the obligation to reason by case for literals of the form $n \nmid a$ (see Example 4.1) is mitigated by reducing the problem into the more specific case where $n = d^k$ with d prime, and then reasoning over dk cases instead of $d^k - 1$. Inequations are dealt with using ordered chaining [BG94], which drastically reduces the search space compared to naive resolution with the transitivity axioms. In particular, chaining can saturate for some problems.

Example 4.1 (Reasoning by Case). *Unlike rational arithmetic, integer arithmetic sometimes requires reasoning by case distinction. The following two simple problems should demonstrate it:*

- $p(0) \wedge p(1) \Rightarrow \forall x : int. \ (0 \leq x \leq 1 \Rightarrow p(x))$. *Clearly, $p(0)$ and $p(1)$ cover all the cases that $\forall x. \ (0 \leq x \leq 1 \Rightarrow \ldots)$ ranges over.*
- $p(a) \wedge p(a + 1) \wedge p(a + 2) \Rightarrow \exists x. \ (3 \mid x \wedge p(x))$. *Among $\{a, a + 1, a + 2\}$, 3 divides exactly one term — the question is, which one? A refutational proof will have a goal $3 \nmid x \vee \neg p(x)$,*

which leads to $3 \nmid a$, $3 \nmid a+1$ and $3 \nmid a+2$ by resolution with the hypothesis. From there, the only way forward is to reason by case on whether the remainder of a divided by 3 is 0, 1 or 2. We will see how our calculus deals with this problem in Examples 4.6 and 4.7.

In addition to the inference system (Section 4.2), we describe several useful redundancy criteria, including a subsumption relation over pairs of literals, a generalization of subsumption for (sets of) inequations, and a semantic tautology rule (Section 4.3). Those criteria have been developed to fix some inefficiencies in our experiments. They can be re-used in any clausal calculus that deals with integer linear arithmetic. In general, this work can be seen as a toolbox to reason modulo integer arithmetic in the context of clausal saturation, so that provers that use other approaches (e.g., hierarchic superposition) can still pick some parts of it. We then expose a variable elimination algorithm based on Cooper's algorithm [Coo72] — a decision procedure for Presburger arithmetic (Section 4.4). This greatly simplifies inference rules, because arithmetic variables that occur directly in arithmetic expressions can be safely ignored. A full AC1-unification algorithm is not required. Finally, the exposition heads for a prototype implementation of the full calculus, including simplification rules and redundancy criteria, in our theorem prover Zipperposition (Section 4.6), and some experimental results (Section 4.7).

4.1 Preliminaries

We start with definitions and some basic rules that reduce arithmetic literals and clauses to canonical forms. Working on canonical forms makes it possible to restrict the number of cases where rules apply. The additional assumptions also enable more succinct formulations of rules. The calculus deals with integers, living in $\mathbb{Z}$, but the canonical terms and literals will all be natural numbers — a negative number $-n + u \simeq v$ is simply put on the other side to obtain $u \simeq v + n$. A family of divisibility predicates $n \mid u$ (where n is a strictly positive natural number and u a linear expression) is part of the language; we will focus on cases $n \mid u$ where n is prime (reducing divisibility by a non-prime number to divisibility by its prime factors). The following lemma will be useful to deal with prime numbers.

Lemma 4.1 (Prime Decomposition). *Let $\{d_i\}_{i=1}^{k}$ denote a set of distinct prime numbers and $\{e_i\}_{i=1}^{k}$ be strictly positive integers. For any integer m, if $\bigwedge_{i=1}^{k} d_i^{e_i} \mid m$ then $\left(\prod_{i=1}^{k} d_i^{e_i}\right) \mid m$.*

Proof. By induction on the number of distinct prime factors k. For $k = 1$ the result is immediate. Otherwise, let us assume the result holds for $k-1$. Let $S = \{d_i^{e_i} \mid i = 1 \dots k\}$ and $m \in \mathbb{Z}$ divisible by every $d_i^{e_i}$ in S. Since, by hypothesis, $d_k^{e_k} \mid m$, there is some m' such that $m = m' \times d_k^{e_k}$. Euclid's lemma implies that for all $i < k$, since $d_i^{e_i} \mid m$, $d_i^{e_i}$ must divide m' because it's coprime with $d_k^{e_k}$ ($d_i \neq d^k$). By induction hypothesis, $\prod_{i=1}^{k-1} d_i^{e_i} \mid m'$ and therefore $\prod_{i=1}^{k} d_i^{e_i} \mid m$. $\square$

Lemma 4.2 (Bézout Identity). *The classic Bézout identity [Bé79]: given x and y non-zero integers, there exists $u, v : \mathbb{Z}$ such that $x \times u + y \times v = gcd(x, y)$.*

4.1.1 Definitions

In a nutshell, the language used throughout this chapter is typed first-order logic with a type int and a signature containing $\{\mathbf{0}, \mathbf{1} : int, + : (int \times int) \rightarrow int, \leq : (int \times int) \rightarrow o\}$ and a family of predicates $(n \mid) : int \rightarrow o$ indexed by positive numbers $n \in \mathbb{N}^+$. We introduce more specific definitions for two reasons: (i) restricting the shape of arithmetic literals (in particular, limit the presence of negation), and (ii) adding notational convenience such as the scalar product $n \cdot t$ (short for $\sum_{i=1}^{n} t$).

Definition 4.1 (Arithmetic Term). *An arithmetic term is a term of the special type int, including the special constants $\mathbf{0} : int$, $\mathbf{1} : int$ and $+ : (int \times int) \rightarrow int$. Intuitively, the type int represents $\mathbb{Z}$, the set of integers.*

Definition 4.2 (AC1)**.** *AC1 is a theory composed of the following axioms on the signature* $\{+, 0\}$ *(0 is called the* neutral element*):*

Associativity $\forall x\ y\ z.\ (x+y)+z \simeq x+(y+z)$
Commutativity $\forall x\ y.\ x+y \simeq y+x$
Identity $\forall x.\ 0+x \simeq x$

In the rest of this chapter, we assume the signature contains function symbols $+$ and $\mathbf{0}$ that, together, satisfy the properties of AC1. Note that we use a theory AC1 but not a group theory; as mentioned before, negative numbers will never occur in our canonical literals.

Definition 4.3 (Product by a Constant)**.** *If* $n \in \mathbb{N}$ *and t is a term, then* $n \cdot t$ *is a notation for the n-ary sum* $\sum_{i=1}^{n} t$. *In particular,* $0 \cdot t = \mathbf{0}$ *and* $1 \cdot t = t$. *To avoid confusion with the meta-level product, the latter will be denoted* $\times$.

Remark 4.1. *Note that in* $n \cdot t$, *n is a natural number and not a term —* $\mathbf{0} \cdot t$ *and* $\mathbf{1} \cdot t$ *are not valid expressions. To erase any trace of doubt, the following are valid terms:* $1 \cdot \mathbf{1} \stackrel{def}{=} \mathbf{1}$, $0 \cdot \mathbf{1} \stackrel{def}{=} \mathbf{0}$, $0 \cdot \mathbf{0} \stackrel{def}{=} \mathbf{0}$, *but* $\mathbf{0} \cdot \mathbf{1}$ *is not a term.*

Definition 4.4 (Linear Expression)**.** *We say an arithmetic term is* atomic *if it does not contain the symbol $+$. A* linear expression *is an integer-sorted sum of atomic terms, of the form* $\sum_{k=1}^{n} a_k \cdot t_k$ *where for each k,* $a_k \in \mathbb{N}^*$ *and t_k is an atomic term. Note that* $\mathbf{0}$ *is a valid linear expression.*

Remark 4.2. *Multiplication by a constant* $n \in \mathbb{N}$ *trivially extends to linear expressions as follows:* $n \cdot \sum_i a_i \cdot t_i = \sum_i (n \times a_i) \cdot t_i.$

Definition 4.5 (Arithmetic Literal)**.** *An* arithmetic literal *is a signed atomic formula of the form*

- $u \simeq v$ *or* $u \not\simeq v$ *when* $u, v : int$ *are linear expressions;*
- $u \leq v$ *when u and v are linear expressions (no other form of comparison is needed, because* $u < v$ *can be translated into* $u + \mathbf{1} \leq v$, $\neg(u \leq v)$ *into* $v + \mathbf{1} \leq u$, *and* $\neg(u < v)$ *into* $v \leq u$);
- $n \mid u$ *or* $n \nmid u$ *where* $n \in \mathbb{N}$, $n \geq 2$ *and a is a linear expression (the case $n = 1$ is always trivial and can be eliminated during preprocessing). This relation is to be interpreted, in models, as the statement that n divides u, for instance by* $[\![(n \mid u)]\!]^{\mathcal{M}} = [\![\exists k \in [\![int]\!]^{\mathcal{M}}.\ u \simeq (k \times n) \cdot \mathbf{1}]\!]^{\mathcal{M}}$. *If* $u = \sum_i a_i \cdot t_i$ *and* $v = \sum_j b_j \cdot t_j$, *we write* $u - v\ [n]$ *("$u - v$ modulo n") for the linear expression* $\sum_i a_i' \cdot t_i + \sum_j (-b_j)' \cdot t_j$ *where* a_i' *(resp.* $(-b_j)'$*) is the euclidian rest of a_i (resp. $-b_j$) by n, and we note* $n \mid u - v$ *the proposition* $n \mid (u - v\ [n])$.

Remark 4.3 (Sign of Literals)**.** *Arithmetic literals exist in positive and negative flavour (except for the predicate $\leq$), but negative ones are always eliminated by simplification rules. This is why most inference and simplification rules deal only with positive literals. We still need to have negative literals because some inference rules introduce them in their conclusions, and so does variable elimination (Section 4.4).*

Remark 4.4 (Translation from Integer Formulas)**.** *An input problem might contain atomic formulas that are not arithmetic literals; e.g.,* $a \simeq b - 2$ *or* $2 \cdot a - b < a$. *They can easily be translated to canonical literals by moving negated terms to the other side of the relation (and simplifying); here,* $a + 2 \simeq b$ *and* $a + 1 \leq b$.

From now on, we will write $u \sim v$ for either $u \simeq v$, $u \leq v$ or $n \mid u$ (in which case v is simply $\mathbf{0}$), $u \overset{.}{\sim} v$ for $u \sim v$, $u \not\simeq v$ or $n \nmid u$ ($v = \mathbf{0}$). If $l \stackrel{def}{=} u \simeq v$ or $l \stackrel{def}{=} m \mid u - v$, then $n \mid l$ means $n \mid u - v$. $n \overset{?}{\mid} u$ means either $n \mid u$ or $n \nmid u$. We will write $u \overset{\leq}{\gtrless} v$ for either $u \leq v$ or $v \leq u$. Literals $u \simeq v$ and $v \simeq u$ are considered the same (i.e., we work modulo commutativity of $\simeq$). Given an AC1-compatible[1] (for instance, [Wal98]) simplification term ordering $\succ$ with the multiset property ($\forall i \in I.\ s \succ t_i$ implies $s \succ \sum_{i \in I} t_i$ for any multiset I), in which $\mathbf{0}$ and $\mathbf{1}$ are the smallest integer-sorted terms and $\mathbf{1} \succ \mathbf{0}$, let $\gg$ be its multiset extension.

---

[1] AC1-compatibility is only needed at the root of a literal, not under function symbols other than $+$, because clauses will be purified, see Section 4.1.3.

Definition 4.6 (Maximal Atomic Term). *Let* $mt(l)$ *be the* maximal atomic term *of a ground literal* l *w.r.t.* $\succ$. *A positive arithmetic literal* l *can be denoted as* $l \stackrel{\text{def}}{=} a \cdot t + u \sim v$, *where* $t = mt(l)$ *if* $t \succ u$ *and* $t \succ v$.

To define inference rules, we will need an ordering $\succ_{lit}$ on literals (and, by multiset extension, on clauses; this is similar to Superposition which also has a literal ordering). The reader might skip the precise definition of $\succ_{lit}$ at first, and just think of it as a convenient way to compare literals and clauses. First, we introduce *Bézout normalization*:

Lemma 4.3 (Bézout Normalization). *Any ground literal* $d^e \mid a \cdot t + u$ *where* $t \succ u$ *and* d^e *does not divide* a *can be changed into an equivalent literal where the coefficient of* t *is minimal and has the form* d^k *with* $k < e$. *We call* $\mathscr{B}(l)$ *(standing for* Bézout normalization *of* l*) the literal obtained this way from the literal* l.

Proof. Using the Bézout identity (Lemma 4.2) on $\gcd(a, d^e) = d^k$ with $k < e$ we can obtain (minimal) $m, n \in \mathbb{N} \times \mathbb{Z}$ with $m \times a + n \times d^e = d^k$, hence by summing $d^e \mid a \cdot t + u$ with itself $m - 1$ times we get $d^e \mid (m \times a) \cdot t + m \cdot u$, then $d^e \mid (d^k - n \times d^e) \cdot t + m \cdot u$, and cancellation yields $d^e \mid d^k \cdot t + m \cdot u$. $\qquad\square$

Definition 4.7 (Arithmetic Literal Ordering). *To fullfill those requirements, we define the* arithmetic ordering $\succ_{lit}$ *on ground literals (regular literals and arithmetic literals) as the lexicographic combination of the following comparisons:*

1. *compare their maximal term* $mt(\cdot)$
2. *compare their polarity (negative* $\succ$ *positive)*
3. *compare their kind* kind *(division* $\succ$ *inequality* $\succ$ *equality)*
4. *compare the number of sides of the relation the maximal term occurs in*
5. *depending on the kind of literal:*
   - *compare* $n_1 \mid^? u_1$ *and* $n_2 \mid^? u_2$ *by* $(>_{\mathbb{N}}, \succ)_{lex}$ *on* $(n_1, \mathscr{B}(u_1))$ *and* $(n_2, \mathscr{B}(u_2))$;
   - *compare* $s_1 \simeq t_1$ *and* $s_2 \simeq t_2$ *by* $\gg$ *on multisets* $\{s_1, t_1\}$ *and* $\{s_2, t_2\}$;
   - *compare* $s_1 \leq t_1$ *and* $s_2 \leq t_2$ *by* $\gg$ *on multisets* $\{s_1, s_1, t_1\}$ *and* $\{s_2, s_2, t_2\}$.

$\succ_{lit}$ *can be extended to non-ground literals by asserting that* $l_1 \succ_{lit} l_2$ *iff* $l_1\sigma \succ_{lit} l_2\sigma$ *for every grounding substitution* σ. *We extend* $\succ_{lit}$ *to clauses by its multiset extension* $\gg_{lit}$ *(or* $\succ_c$*).*

Lemma 4.4 (Compatibility of $\succ_{lit}$). *The ordering* $\succ_{lit}$ *is an extension of (is* compatible with*) the ordering on literals used in superposition (Definition 2.54).*

Proof. The ordering on equational literals from Superposition is defined by $\gg$ on their multiset encoding $\mathscr{M}_e(\cdot)$ defined by: (i) $\mathscr{M}_e(s \neq t) = \{s, s, t, t\}$; (ii) $\mathscr{M}_e(s \simeq t) = \{s, t\}$. Given two equational literals e_1 and e_2 such that $e_1 \succ_{lit} e_2$, there are three possible cases:

1. if $mt(e_1) \succ mt(e_2)$, then $\mathscr{M}_e(e_1) \gg \mathscr{M}_e(e_2)$;
2. otherwise, if $e_1 = s \neq t_1$ and $e_2 = s \simeq t_2$ with $t_1 \not\succ s$ and $t_2 \not\succ s$ (i.e., $mt(e_1) = mt(e_2) = s$), then $\mathscr{M}_e(e_1) = \{s, s, t_1, t_1\} \gg \{s, t_2\} = \mathscr{M}_e(e_2)$;
3. if both have the same sign and $s \stackrel{\text{def}}{=} mt(e_1)$ occurs on both side of the equation, whereas s occurs only on one side of e_2, then $\mathscr{M}_e(e_1) \gg \mathscr{M}_e(e_2)$;
4. otherwise, both have the same sign, and comparing $\mathscr{M}_e(e_i)$ with $\gg$ amounts to comparing $\{s, t_i\}$ (where $e_i = s \simeq t_i$) with $\gg$.

$\qquad\square$

Example 4.2 (Comparisons of Literals). *Let* $a \succ b \succ c \succ d$.

- $a \simeq \mathbf{0} \succ_{lit} b + c \simeq d$ *by maximal terms:* $a \succ b$.
- $3 \nmid a + 2 \cdot c \succ_{lit} 5 \mid 2 \cdot a + b$ *by polarity (same maximal term).*

45

- $a + b + c \simeq d >_{lit} a + b \simeq c + d$, using the last case because $\{a, b, c\}$ dominates both $\{a, b\}$ and $\{c, d\}$.
- $5 \nmid b + d >_{lit} 3 \nmid b + d$ since $5 > 3$.

Lemma 4.5 ($>_{lit}$ is a Simplification Ordering)**.** $>_{lit}$ *is a partial ordering on literals, total on ground literals modulo AC1, well-founded, and stable by substitution.*

Definition 4.8 (Arithmetic Model)**.** *An* arithmetic model $\mathcal{M}$ *is an interpretation (see Definition 2.58) that maps terms of type int into the set of integers $\mathbb{Z}$ with the standard interpretation of* $\mathbf{0}, \mathbf{1}, +, |$ *and* $\cdot$*. We write $\mathcal{M} \models_{\text{arith}} C$ if the arithmetic model $\mathcal{M}$ satisfies the clause C (idem for set of clauses).*

Definition 4.9 (Arithmetic Entailment)**.** *A clause set S is said to* entail *a clause C w.r.t. integers arithmetic, denoted $S \vdash_{\text{arith}} C$, iff for every arithmetic model $\mathcal{M}$, $\mathcal{M} \models_{\text{arith}} S$ implies $\mathcal{M} \models_{\text{arith}} C$.*

4.1.2 Normalization of Literals and Clauses

In general, it is preferable not to have to perform explicit inference steps to reckon that two literals are equivalent. That explains why we defined canonical forms for literals in the previous section. Some additional normalizations on literals and clauses are needed, but are not easy or convenient to express as syntactic restrictions: trivially decidable literals (with only $\mathbf{0}$ and $\mathbf{1}$ as terms); literals $n \overset{?}{|} u$ where n is not prime, which are normalized into a conjunction or disjunction of several literals; literals of the form $u \neq v$, transformed into $u < v \lor u > v$. The rules are shown in Figure 4.1; only a subset is named because the other rules are so simple that their application should be obvious.

Some words of explanation for each rule are in order. We also justify briefly their soundness in arithmetic models.

Prime Case Switch is used to eliminate literals of the form $d^k \nmid u$, where d is prime and $k \geq 1$. A naive rule would directly reason by case on the remainder of u when divided by d^k (yielding the $d^k - 1$ cases $\bigvee_{i=1}^{k-1} d^k \mid u + i \cdot \mathbf{1}$). However, we might want to reason in, for instance, $\mathbb{Z}/2^{32}\mathbb{Z}$ (unsigned machine integers). A case switch over $2^{32} - 1$ cases is not reasonable. We can use the following fact: u not being divisible by d^k means that some of the k first digits of u in base d is not 0. If the least significant digit of u in base d that is not 0 is the e-th one ($e < k$), it means $u = i \cdot d^e + d^{e+1} \cdot u'$ for some $i \in \{1, \ldots, d - 1\}$. Therefore $u + (d - i) \cdot d^e = d^{e+1} \cdot u' + (d - i + i) \cdot d^e = d^{e+1} \cdot u' + d^{e+1}$ is divisible by d^{e+1}. That is, $\bigvee_{j=1}^{d-1} d^{e+1} \mid u + (j \times d^e) \cdot \mathbf{1}$ after the substitution $j \overset{\text{def}}{=} d - i$. Since $d^k \nmid u$, there is such a digit; the outer disjunction follows. We only have $(d - 1) \times k$ cases, which is much better than $d^k - 1$ when d or k grows — in the case of machine integer, only 32 cases instead of $2^{32} - 1$.

Division Simplification simplifies $d^k \mid u + d^{k+k'} \cdot t$ (since $d^{k+k'} \cdot t$ is obviously always divisible by d^k) and simplifies $d^{k+k'} \mid d^{k'} \cdot u$ into $d^k \mid u$.

Inequality Simplification exploits the properties of integers to round up or down inequalities[2]. For instance, $2 \cdot a \leq 4 \cdot b + 3$ becomes $a \leq 2 \cdot b + 1$, because $2 \cdot a \leq 4 \cdot b + 3 \iff 2 \cdot (a - 2 \cdot b) \leq 3 \iff 2 \cdot (a - 2 \cdot b) \leq 2 \iff 2 \cdot a \leq 2 \cdot (2 \cdot b + 1) \iff a \leq 2 \cdot b + 1$. Conversely, $2 \cdot a + 3 \leq 4 \cdot b$ becomes $a + 2 \leq 2 \cdot b$ by rounding $3/2$ up.

Prime Decomposition uses respectively (the contrapositive of) Lemma 4.1 and regular decomposition into prime factors.

Cancellative Equality Resolution is trivial.

Cancellative Inequality Resolution idem.

Division Elimination idem.

Total Order replaces a literal $u \neq v$ with an alternative between $u < v$ and $u > v$.

---

[2] This criterion amounts to checking whether the gcd g of all coefficients, excluding the constant if there is one, is ≥ 2, and then dividing them all by g.

Prime Case Switch (PrimeCS)

$$\frac{C \vee d^k \nmid u}{C \vee \bigvee_{e=0}^{k-1} \bigvee_{i=1}^{d-1} d^{e+1} \mid u + (i \times d^e) \cdot \mathbf{1}}$$

where d is prime, $k \geq 1$

Division Simplification

$$\frac{C \vee d^k \mid^? d^{k+k'} \cdot t + u}{C \vee d^k \mid^? u} \quad \text{and} \quad \frac{C \vee d^{k+k'} \mid d^{k'} \cdot u}{C \vee d^k \mid u}$$

where d prime, $k \geq 1$, $k' \geq 1$

Inequality Simplification

$$\frac{C \vee k \cdot u \leq k \cdot v + (k \times c + d) \cdot \mathbf{1}}{C \vee u \leq v + c \cdot \mathbf{1}} \quad \text{and} \quad \frac{C \vee k \cdot u + (k \times c + d) \cdot \mathbf{1} \leq k \cdot v}{C \vee u + (c + \delta) \cdot \mathbf{1} \leq v}$$

for $k \geq 2$ and $0 \leq d < k$, with $\delta = \begin{cases} 0 \text{ if } d = 0 \\ 1 \text{ otherwise} \end{cases}$

Prime Decomposition (PrimeDecomp)

$$\frac{C \vee n \nmid u}{C \vee \bigvee_{i=1}^{k} d_i^{e_i} \nmid u} \quad \text{and} \quad \frac{C \vee n \mid u}{\{C \vee d_i^{e_i} \mid u\}_{i=1}^{k}}$$

where $n = \prod_{i=1}^{k} d_i^{e_i}$, $k \geq 2$

Cancellative Equality Resolution

$$\frac{C \vee \mathbf{0} \not\simeq \mathbf{0}}{C} \quad \text{and} \quad \frac{C \vee i \cdot \mathbf{1} \simeq \mathbf{0}}{C}$$

where $i \geq 1$

Cancellative Inequality Resolution

$$\frac{C \vee i \cdot \mathbf{1} \leq \mathbf{0}}{C} \quad \text{and} \quad \frac{C \vee \mathbf{0} \leq j \cdot \mathbf{1}}{\top}$$

where $i \geq 1$, $j \geq 0$

Division Elimination

$$\frac{C \vee d \mid i \cdot \mathbf{1}}{C} \quad \text{and} \quad \frac{C \vee d \nmid i \cdot \mathbf{1}}{\top}$$

where $d > 1$, $1 \leq i \leq d - 1$

Total Order (TO)

$$\frac{C \vee u \not\simeq v}{C \vee u + \mathbf{1} \leq v \vee v + \mathbf{1} \leq u}$$

Figure 4.1: The Normalization Rules of $\mathfrak{I}_{\text{arith}}$

4.1.3 Purification of Clauses

The calculus we develop in this chapter cannot handle arithmetic terms that occur under function symbols. The reason is that most inference rules will require multiplication of linear expressions by a scalar constant, so as to obtain the same coefficient for the term to rewrite; under a function symbol we have no idea whether this is allowed. For instance, given a rule $2 \cdot t \simeq u$, rewriting t in $P(t)$ is impossible because $P(t)$ does not necessarily imply $P(2 \cdot t)$. On the other hand, given $P(x) \vee x \not\simeq t$, the following inference is acceptable:

$$\frac{\dfrac{P(x) \vee x \not\simeq t}{P(x) \vee 2 \cdot x \not\simeq 2 \cdot t} \quad 2 \cdot t \simeq u}{P(x) \vee 2 \cdot x \not\simeq u} \text{ (Sup)}$$

Definition 4.10 (Shielded Term). *A term t is* shielded *in a clause C if it occurs in C under a function or predicate symbol. For instance, in $p(a + \mathbf{1}) \vee f(b) \simeq b + c + \mathbf{1}$, both $a + \mathbf{1}$ and b are shielded, but c is not. A term that is not shielded is* unshielded. *Unshielded variables will be dealt with in Section 4.4.*

Definition 4.11 (Purified Clause). *A* purified clause[3] *is a clause in which all shielded terms of type int are either variables or integer constants (of the form $k \cdot \mathbf{1}$).*

Example 4.3 (Purified Clauses). *Let $x : \iota$ and $y : int$ be variables, and $a : int$, $f : \iota \to int$, $g : int \to int$, $p : o$ be function or predicate symbols.*

- *$g(a) + y \simeq 3 \vee p$ is not purified, because $a : int$ occurs under the function symbol g;*
- *$g(f(x)) \leq a$ is not purified, for the same reason;*
- *$g(y) \simeq 2 \cdot y \vee g(10)$ is purified.*

Intuitively, if all clauses are purified, any two shielded terms of type int are either distinct or easily unifiable. There is no need for unification modulo AC1 under terms.

Definition 4.12 (Purification Ritual). *To purify a clause C, it suffices to take its normal form w.r.t. the rewrite system $\longrightarrow_{pur}{}^*$. If a clause C contains the linear expression m (neither a variable nor an arithmetic constant) under uninterpreted functions at positions $\rho_1, \ldots, \rho_k$ ($k \geq 1$) — in other words, those occurrences of m make C impure —, let $x : int$ be a fresh variable, and*

$$C \longrightarrow_{pur} C[x]_{\rho_1} \cdots [x]_{\rho_k} \vee x \not\simeq m$$

Each rewrite step of $\longrightarrow_{pur}$ eliminates one linear expression occurring under a function symbol by replacing it with x. In a sense, $\longrightarrow_{pur}$ is the opposite of the (DER) *rule in Superposition (Figure 2.2). This relation terminates because the number of linear expressions occurring under a function symbol in the clause decreases strictly at each step.*

Example 4.4 (Purification). *The clause $p(f(a + \mathbf{1})) \vee q(2 \cdot b) \vee r$ is purified as follows:*

$$p(f(a + \mathbf{1})) \vee q(2 \cdot b, a + \mathbf{1}) \vee r$$
$$\longrightarrow_{pur} p(f(x)) \vee q(2 \cdot b, x) \vee r \vee x \not\simeq a + \mathbf{1}$$
$$\longrightarrow_{pur} p(f(x)) \vee q(y, x) \vee r \vee x \not\simeq a + \mathbf{1} \vee y \not\simeq 2 \cdot b$$

4.2 Inference Rules

We now present the core innovation of this chapter: the inference system that a saturation prover uses in its quest for $\bot$. This set of rules, similarly to the superposition calculus (Section 2.4), although it can be complemented by some simplification rules and other redundancy criteria to improve its efficiency (see Section 4.3), is the foundation a theorem prover can lie on. We will also demonstrate that the inference system is usable in practice with our proof of concept implementation in Zipperposition.

---

[3] also referred to as *abstracted clauses* in the literature, in particular in [BGW94, BW13].

4.2.1 Ground Version of the Rules

One of the main contributions in this chapter is a set of inference rules that complement the (typed) superposition calculus (Section 2.4). Those rules are listed in Figures 4.2 and 4.3 in their *ground* version: as explained in Section 4.2.2, lifting is straightforward but makes each rule harder to read and understand. There are 10 rules, organized along two axes: (1) the predicate symbols of the literals involved in the inference ($\simeq$, $\leq$ and $n \mid$) — $\leq$ and $n \mid$ do not interact —, and (2) the number of premises (one or two). In a given rule, multiple occurrences of notations such as $\lessgtr$ and $\sim$ denote the same concrete relation. In Section 4.6 we will give more details about a possible way to implement those rules. Let us develop the intuition beneath a few of those rules — hopefully the reader will see how the explanations carry over the other rules.

Cancellative Superposition uses an equational literal of the form $a \cdot t + u \simeq v$, where t is the maximal atomic term, to "eliminate" t within another literal $a' \cdot t + u' \sim v'$ (from some other clause) — that is, deduce a new literal in which t doesn't occur, so that the inference is decreasing. Contrary to classic superposition, we can sum a literal with itself as many times as needed; here, we sum the literals respectively φ times and φ' times where $a \cdot \varphi = a' \cdot \varphi' = \mathrm{lcm}(a, a')$, obtaining $\mathrm{lcm}(a, a') \cdot t + \varphi \cdot u \simeq \varphi \cdot v$ and $\mathrm{lcm}(a, a') \cdot t + \varphi' \cdot u' \sim \varphi' \cdot v'$. Now we can swap sides in the first literal, and sum both literals to obtain $\mathrm{lcm}(a, a') \cdot t + \varphi \cdot v + \varphi' \cdot u' \sim \mathrm{lcm}(a, a') \cdot t + \varphi \cdot u + \varphi' \cdot v'$, which simplifies to the conclusion $\varphi \cdot v + \varphi' \cdot u' \sim \varphi \cdot u + \varphi' \cdot v'$ by cancelling $\mathrm{lcm}(a, a') \cdot t$ out.

Cancellative Equality Factoring merges two equations $a \cdot t + u \simeq v$ and $a' \cdot t + u' \simeq v'$ (with maximal term t) into one single equation, under the condition that they are actually the same (that is, $\frac{v-u}{a} = \frac{v'-u'}{a'}$). This is similar to the (EqFact)rule (Equality Factoring) in Superposition, as explained in Section 2.4.2.

Cancellation comes from the reflexivity of $\simeq$ and $\leq$ (and the tautology $\forall t.\ d^k \mid d^k \cdot t$). The ground version looks trivial, but once lifted this rule allows us to unify maximal terms on both sides of an (in-)equation so they cancel out into a smaller literal, for instance inferring $u \leq v$ from $f(x) + u \leq f(a) + v$ (with $\{x \mapsto a\}$).

Cancellative Chaining expresses the transitivity of $\leq$, in a very similar way to Waldmann's work [Wal01]. Intuitively, chaining $v \leq a \cdot t + u$ and $a' \cdot t + u' \leq v'$ starts with multiplying by φ and φ' respectively to obtain the same coefficient for t, then isolating t: $\varphi \cdot v - \varphi \cdot u \leq \mathrm{lcm}(a, a') \cdot t \leq \varphi' \cdot v' - \varphi' \cdot u'$, which entails $\varphi \cdot v - \varphi \cdot u \leq \varphi' \cdot v' - \varphi' \cdot u'$. Then we normalize into $\varphi \cdot v + \varphi' \cdot u' \leq \varphi' \cdot v' + \varphi \cdot u$.

Cancellative Case Switch allows reasoning by case on a term if it belongs to a finite range. Since, here, $v \leq a \cdot t + u$ and $a' \cdot t + u' \leq v'$, it means $\varphi \cdot v - \varphi \cdot u \leq \mathrm{lcm}(a, a') \cdot t \leq \varphi' \cdot v' - \varphi' \cdot u'$ where $a \times \varphi = \mathrm{lcm}(a, a') = a' \times \varphi'$. If we assume there is a constant $k \in \mathbb{N}$ such that $\varphi' \cdot v' + \varphi \cdot u = \varphi \cdot v + \varphi' \cdot u' + k \cdot \mathbf{1}$, then the range of possible values for $\mathrm{lcm}(a, b) \cdot t$ is finite and contains $k+1$ values that are $\varphi \cdot v - \varphi \cdot v + i \cdot \mathbf{1}$ for $i \in \{0, \ldots, k\}$. We can therefore deduce $\bigvee_{i=0}^{k} \varphi \cdot v - \varphi \cdot u + i \cdot \mathbf{1} \simeq \mathrm{lcm}(a, a') t$, normalized into $\bigvee_{i=0}^{k} \mathrm{lcm}(a, 'a) \cdot t + \varphi \cdot u \simeq \varphi \cdot v + i \cdot \mathbf{1}$. This inference rule is crucial to solve the first case of Example 4.1. We don't allow k to be negative, for Cancellative Chaining already deals with this case.

Cancellative Inequality Factoring merges two literals l and l' into l', if l' is an inequality, both share the same maximal term t, and $l \vdash_{\mathrm{arith}} l'$. If $l \stackrel{\mathrm{def}}{=} a \cdot t + u \simeq v$, then $l \vdash_{\mathrm{arith}} a \cdot t + u \lessgtr v$, so we only explain this latter case; by symmetry we even assume $l \stackrel{\mathrm{def}}{=} a \cdot t + u \leq v$ and $l' \stackrel{\mathrm{def}}{=} a' \cdot t + u' \leq v'$. In this case, a sufficient condition for $l \vdash_{\mathrm{arith}} l'$ is if $\frac{v-u}{a} \leq \frac{v'-u'}{a'}$, in other words $C' \vee (\varphi \cdot v + \varphi' \cdot u' \leq \varphi \cdot u + \varphi' \cdot v') \Rightarrow (a' \cdot t + u' \leq v')$. Normalizing the first literal yields the expected conclusion, $C' \vee (\varphi \cdot u + \varphi' \cdot v' + \mathbf{1} \leq \varphi \cdot v + \varphi' \cdot u') \vee (a' \cdot t + u' \leq v')$.

Modular Chaining implements the fact that divisibility commutes with addition and subtraction. We assume $d^e \mid a \cdot t + u$ and $d^{e+k} \mid a' \cdot t + u'$. From the former we deduce $d^{e+k} \mid (a \times d^k) \cdot t + d^k \cdot u$; then, we multiply by φ, respectively φ', such that $\varphi \times (a \times d^k) = \varphi' \times a' = \mathrm{lcm}(a \times d^k, a')$ — by assumption $\mathrm{lcm}(a \times d^k, a') < d^{e+k}$ so t is not simplified away from

the literals — and subtract the two resulting literals into $d^{e+k} \mid \text{lcm}(a \times d^k, a') \cdot (t - t) + (\varphi \times d^k) \cdot u - \varphi' \cdot u'$, which simplifies into $d^{e+k} \mid (\varphi \times d^k) \cdot u - \varphi' \cdot u'$.

Modular Factoring is similar to Equality and Inequality Factoring. It merges together $l \stackrel{\text{def}}{=} d^e \mid a \cdot t + u$ and $l' \stackrel{\text{def}}{=} d^{e+k} \mid a' \cdot t + u'$ if a side-condition, $l \vdash_{\text{arith}} l'$, is solved. From l we deduce $d^{e+k} \mid (d^k \times a) \cdot t + d^k \cdot u$. The inference requires $\gcd(d^k \times a, d^{e+k}) \mid \gcd(a', d^{e+k})$ because otherwise l could not imply l': for instance if $l \stackrel{\text{def}}{=} 2 \mid t$, $l' \stackrel{\text{def}}{=} 4 \mid t$, from l we cannot deduce any information on the divisibility of t by 4, only by 2. We could say that in l, t lives in $\mathbb{Z}/2\mathbb{Z}$, whereas in l' it lives in $\mathbb{Z}/4\mathbb{Z}$. If the condition is fulfilled, it's again a matter of expressing whether $d^{e+k} \mid (\varphi \times d^k) \cdot u - \varphi' \cdot u'$ holds.

Modular Equality Factoring reduces to Modular Factoring by noticing $a \cdot t + u \simeq v$ entails $d^e \mid a \cdot t + u - v$.

Divisibility as some other rules (e.g., Modular Equality Factoring), it witnesses the fact that $u \simeq v$ implies $n \mid u - v$ for all n. Divisibility is explicitly needed because, although $a \cdot t + u \simeq v$ already implies $a \mid u - v$, the latter's maximal term (some atomic subterm of u or v) is strictly smaller in $\succ$ than the former's maximal term (t), and therefore some inferences apply to $a \mid u - v$ that wouldn't otherwise.

Example 4.5 (Simple Example). *Let us show that $\{16 \mid 2 \cdot a + b, 4 \mid c + \mathbf{1}, b \simeq c\}$ is unsatisfiable, with $a \succ b \succ c$.*

$$\frac{\dfrac{\dfrac{16 \mid 2 \cdot a + b}{2 \mid b} \text{(CDiv)} \quad b \simeq c}{2 \mid c} \text{(CSup)} \quad 4 \mid c + \mathbf{1}}{\dfrac{4 \mid 2 \cdot \mathbf{1}}{\bot}} \text{(Chain|)}$$

Example 4.6 (Modular Case Splits). *Let us show that among a, $a+1$, $a+2$, one term is a multiple of 3. The refutation, as follows, starts with $\{3 \nmid a, 3 \nmid a+1, 3 \nmid a+2\}$ and uses Modular Chaining to combine clauses and* AVATAR *splits (Section 2.5), as well as trivial normalizations from Figure 4.1.*

$$\frac{\dfrac{3 \nmid a}{3 \mid a+\mathbf{1} \vee 3 \mid a+2 \cdot \mathbf{1}}}{3 \mid a+\mathbf{1} \leftarrow \llbracket 3 \mid a+\mathbf{1} \rrbracket \quad 3 \mid a+2 \cdot \mathbf{1} \leftarrow \llbracket 3 \mid a+2 \cdot \mathbf{1} \rrbracket \quad \llbracket 3 \mid a+\mathbf{1} \rrbracket \sqcup \llbracket 3 \mid a+2 \cdot \mathbf{1} \rrbracket}$$

$$\vdots \ \pi_0$$

$$\frac{\dfrac{3 \nmid a+\mathbf{1}}{3 \mid a \vee 3 \mid a+2 \cdot \mathbf{1}}}{3 \mid a \leftarrow \llbracket 3 \mid a \rrbracket \quad 3 \mid a+2 \cdot \mathbf{1} \leftarrow \llbracket 3 \mid a+2 \cdot \mathbf{1} \rrbracket \quad \llbracket 3 \mid a \rrbracket \sqcup \llbracket 3 \mid a+2 \cdot \mathbf{1} \rrbracket}$$

$$\vdots \ \pi_1$$

$$\frac{\dfrac{3 \nmid a+2 \cdot \mathbf{1}}{3 \mid a \vee 3 \mid a+\mathbf{1}}}{3 \mid a \leftarrow \llbracket 3 \mid a \rrbracket \quad 3 \mid a+\mathbf{1} \leftarrow \llbracket 3 \mid a+\mathbf{1} \rrbracket \quad \llbracket 3 \mid a \rrbracket \sqcup \llbracket 3 \mid a+\mathbf{1} \rrbracket}$$

$$\vdots \ \pi_2$$

Cancellative Superposition (CSup)

$$\frac{C \vee a \cdot t + u \simeq v \qquad C' \vee a' \cdot t + u' \sim v'}{C \vee C' \vee \varphi' \cdot u + \varphi \cdot v' \sim \varphi \cdot u' + \varphi' \cdot v}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $\varphi \times a = \varphi' \times a' = \mathrm{lcm}(a, a')$,
$a \cdot t + u \simeq v \succ_c C$, $a' \cdot t + u' \sim v' \succ_c C'$

Cancellative Equality Factoring (CFact≃≃)

$$\frac{C \vee a \cdot t + u \simeq v \vee a' \cdot t + u' \simeq v'}{C \vee \varphi \cdot u + \varphi' \cdot v' \not\simeq \varphi' \cdot u' + \varphi \cdot v \vee a' \cdot t + u' \simeq v'}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $\varphi \times a = \varphi' \times a' = \mathrm{lcm}(a, a')$,
the last literal is maximal

Cancellation (Canc)

$$\frac{C \vee a \cdot t + u \mathrel{\dot\sim} a' \cdot t + v}{C \vee (a - a') \cdot t + u \mathrel{\dot\sim} v} \quad \text{and} \quad \frac{C \vee d^k \mid d^k \cdot t + u}{C \vee d^k \mid u}$$

where $t \succ u$, $t \succ v$, $a \geq a'$, the literal is maximal

Cancellative Chaining (Chain≤)

$$\frac{C \vee v \leq a \cdot t + u \qquad C' \vee a' \cdot t + u' \leq v'}{C \vee C' \vee \varphi \cdot v + \varphi' \cdot u' \leq \varphi' \cdot v' + \varphi \cdot u}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $a \times \varphi = a' \times \varphi' = \mathrm{lcm}(a, a')$,
the literals are maximal in their respective clause

Cancellative Case Switch (CSwitch)

$$\frac{C \vee v \leq a \cdot t + u \qquad C' \vee a' \cdot t + u' \leq v'}{C \vee C' \vee \bigvee_{i=0}^{k} (\varphi \times a) \cdot t + \varphi \cdot u \simeq \varphi \cdot v + i \cdot \mathbf{1}}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $a \times \varphi = a' \times \varphi' = \mathrm{lcm}(a, a')$,
there is a $k \in \mathbb{N}$ such that $\varphi \cdot v + \varphi' \cdot u' + k \cdot \mathbf{1} = \varphi' \cdot v' + \varphi \cdot u$,
the literals are maximal.

Cancellative Ineq. Factoring (CFact≤)

$$\frac{C \vee \left\{ \begin{array}{c} a \cdot t + u \lessgtr v \\ \text{or} \quad a \cdot t + u \simeq v \end{array} \right\} \vee a' \cdot t + u' \lessgtr v'}{C \vee \varphi \cdot u + \varphi' \cdot v' + \mathbf{1} \lessgtr \varphi \cdot v + \varphi' \cdot u' \vee a' \cdot t + u' \lessgtr v'}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $a \times \varphi = a' \times \varphi' = \mathrm{lcm}(a, a')$,
the last literal is maximal.

Figure 4.2: The Inference Rules on $\simeq$ and $\leq$ of $\mathfrak{I}_{\mathrm{arith}}$ (ground version)

$$\frac{\dfrac{\vdots \; \pi_0 \qquad\qquad\qquad \vdots \; \pi_1}{3 \mid a + 1 \leftarrow \lVert 3 \mid a + 1 \rVert \qquad 3 \mid a \leftarrow \lVert 3 \mid a \rVert}}{\dfrac{3 \mid a + 1 + 2 \cdot a \leftarrow \lVert 3 \mid a + 1 \rVert \sqcap \lVert 3 \mid a \rVert}{\dfrac{3 \mid 1 \leftarrow \lVert 3 \mid a + 1 \rVert \sqcap \lVert 3 \mid a \rVert}{\bot \leftarrow \lVert 3 \mid a + 1 \rVert \sqcap \lVert 3 \mid a \rVert}}} \; (\text{Chain}\mid)$$

Modular Chaining (Chain|)

$$\frac{C \vee d^e \mid a \cdot t + u \qquad C' \vee d^{e+k} \mid a' \cdot t + u'}{C \vee C' \vee d^{e+k} \mid (\varphi \times d^k) \cdot u - \varphi' \cdot u'}$$

where $t \succ u$, $t \succ u'$, d prime, $k \geq 0$,
$\varphi \times (a \times d^k) = \varphi' \times a' = \mathrm{lcm}(a \times d^k, a') < d^{e+k}$,
literals are maximal in their clause

Modular Factoring (CFact||)

$$\frac{C \vee d^e \mid a' \cdot t + u' \vee d^{e+k} \mid a \cdot t + u}{C \vee d^{e+k} \nmid \varphi \cdot u - (d^k \times \varphi') \cdot u' \vee d^{e+k} \mid a \cdot t + u}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $\varphi \times a = \varphi' \times a' = \mathrm{lcm}(a, a')$,
$\gcd(a', d^e) \cdot d^k \mid \gcd(a, d^{e+k})$, d prime, $k \geq 0$,
the last literal is maximal

Modular Equality Factoring (CFact|≃)

$$\frac{C \vee a \cdot t + u \simeq v \vee d^e \mid a' \cdot t + u'}{C \vee d^e \nmid \varphi \cdot v + \varphi' \cdot u' - \varphi \cdot u \vee d^e \mid a' \cdot t + u'}$$

where $t \succ u$, $t \succ v$, $t \succ u'$, $t \succ v'$, $\gcd(a, d^e) \mid \gcd(a', d^e)$,
$\varphi \cdot a = \varphi' \cdot a'$, $a \cdot t + u \simeq v \succ_c C$, d prime

Divisibility (CDiv)

$$\frac{C \vee a \cdot t + u \simeq v}{C \vee a \mid u - v} \quad \text{and} \quad \frac{C \vee d^{k+k'} \mid (b \times d^k) \cdot t + u}{C \vee d^k \mid u}$$

where $t \succ u$, $t \succ v$, d prime, $k \geq 1$, $k' \geq 1$, $a \geq 2$, $b \geq 1$,
the literal is maximal

Figure 4.3: The Inference Rules on divisibility of $\mathfrak{I}_{\mathrm{arith}}$ (ground version)

Similarly, we can obtain $\bot \leftarrow \llbracket 3 \mid a + \mathbf{1} \rrbracket \sqcap \llbracket 3 \mid a + 2 \cdot \mathbf{1} \rrbracket$ and $\bot \leftarrow \llbracket 3 \mid a \rrbracket \sqcap \llbracket 3 \mid a + 2 \cdot \mathbf{1} \rrbracket$. At least two among $\{\llbracket 3 \mid a \rrbracket, \llbracket 3 \mid a + \mathbf{1} \rrbracket, \llbracket 3 \mid a + 2 \cdot \mathbf{1} \rrbracket\}$ must be true by the splitting constraints, but since we just found they are mutually exclusive the constraints are unsatisfiable.

Example 4.7 (Case Splits on Inequalities). *Going back to Example 4.1, we prove $p(a) \wedge p(a+1) \wedge p(a+2) \Rightarrow \exists x. (3 \mid x \wedge p(x))$. The negation of the goal, after Skolemization (Definition 2.52) and purification (Definition 4.12), is the set of clauses $\{p(x) \vee x \neq a, p(x) \vee x \neq a+1, p(x) \vee x \neq a+2, 3 \nmid p(x) \vee \neg p(x)\}$. We obtain the following derivations (using a very simple case of the variable elimination algorithm presented in Section 4.4):*

$$\frac{\dfrac{p(x) \vee x \neq a \qquad 3 \nmid x \vee \neg p(x)}{x \neq a \vee 3 \nmid x} \text{ (CSup)}}{3 \nmid a} \text{ (VarElim)}$$

$$\frac{\dfrac{p(x) \vee x \neq a+1 \qquad 3 \nmid x \vee \neg p(x)}{x \neq a+1 \vee 3 \nmid x} \text{ (CSup)}}{3 \nmid a+1} \text{ (VarElim)}$$

$$\frac{\dfrac{p(x) \vee x \neq a+2 \qquad 3 \nmid x \vee \neg p(x)}{x \neq a+2 \vee 3 \nmid x} \text{ (CSup)}}{3 \nmid a+2} \text{ (VarElim)}$$

from there, we use the derivation from Example 4.6 to conclude.

Example 4.8 (Even-Odd term)**.** *We prove that $2 \cdot a_1 \simeq b \wedge 2 \cdot a_2 \simeq b + 1$ is unsatisfiable. There are several proofs, depending on the ordering of $\{a_1, a_2, b\}$.*

- *if $a_1 > a_2 > b$ or $a_2 > a_1 > b$, a_1 and a_2 are eliminated by Divisibility; then, by Modular Chaining, we obtain an absurd literal.*

$$
\cfrac{\cfrac{2 \cdot a_1 \simeq b}{2 \mid b} \text{ (CDiv)} \quad \cfrac{2 \cdot a_2 \simeq b + 1}{2 \mid b + 1} \text{ (CDiv)}}{\cfrac{2 \mid 1}{\bot}} \text{ (Chain|)}
$$

- *if $b > a_1 > a_2$ ($b > a_2 > a_1$ is symmetric), we eliminate b by superposition, then a_1 by Divisibility.*

$$
\cfrac{\cfrac{2 \cdot a_1 \simeq b \quad 2 \cdot a_2 \simeq b + 1}{2 \cdot a_1 + 1 \simeq 2 \cdot a_2} \text{ (CSup)}}{\cfrac{2 \mid 1}{\bot}} \text{ (CDiv)}
$$

- *if $a_1 > b > a_2$ (and the symmetric case), we start with Divisibility on a_1, then Superposition on b (which also eliminates a_2 since it occurs in $\mathbb{Z}/2\mathbb{Z}$).*

$$
\cfrac{\cfrac{2 \cdot a_1 \simeq b}{2 \mid b} \text{ (CDiv)} \quad 2 \cdot a_2 \simeq b + 1}{\cfrac{2 \mid 2 \cdot a_2 + 1}{\bot}} \text{ (CSup)}
$$

Example 4.9 (Divisibility and Equalities)**.** *In this example, we show how a divisibility constraint can filter the possible values for a term. From $\bigvee_{i=1}^{4} a \simeq i$ and $3 \mid a$, we prove that $a \simeq 3$ must hold. First, we show the version without* AVATAR*:*

$$
\cfrac{\cfrac{a \simeq 1 \vee a \simeq 2 \vee a \simeq 3 \vee a \simeq 4 \quad 3 \mid a}{3 \mid 1 \vee a \simeq 1 \vee a \simeq 2 \vee a \simeq 3}}{a \simeq 1 \vee a \simeq 2 \vee a \simeq 3} \text{ (CSup)}
$$
$$
\vdots \pi_2
$$

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{a \neq 3}{a \leq 2 \vee 4 \leq a} \text{ (TO)} \quad \vdots \pi_2}{a \simeq 1 \vee a \simeq 2 \vee 1 \leq 0 \vee 4 \leq a} \text{ (CSup)} \quad \vdots \pi_2}{a \simeq 1 \vee a \simeq 2 \vee 4 \leq a}}{a \simeq 1 \vee a \simeq 2 \vee 4 \leq 3} \text{ (CSup)} \quad 3 \mid a}{\cfrac{a \simeq 1 \vee 3 \mid 2}{a \simeq 1}} \text{ (CSup)} \quad 3 \mid a}{\cfrac{3 \mid 1}{\bot}} \text{ (CSup)}
$$

Now, we can also leverage AVATAR *to reason by case:*

$$
\cfrac{a \simeq 1 \vee a \simeq 2 \vee a \simeq 3 \vee a \simeq 4}{a \simeq 1 \leftarrow \lfloor\!\lfloor a \simeq 1 \rfloor\!\rfloor \quad a \simeq 2 \leftarrow \lfloor\!\lfloor a \simeq 2 \rfloor\!\rfloor \quad a \simeq 3 \leftarrow \lfloor\!\lfloor a \simeq 3 \rfloor\!\rfloor \quad a \simeq 4 \leftarrow \lfloor\!\lfloor a \simeq 4 \rfloor\!\rfloor}{\lfloor\!\lfloor a \simeq 1 \rfloor\!\rfloor \sqcup \lfloor\!\lfloor a \simeq 2 \rfloor\!\rfloor \sqcup \lfloor\!\lfloor a \simeq 3 \rfloor\!\rfloor \sqcup \lfloor\!\lfloor a \simeq 4 \rfloor\!\rfloor} \text{ (ASplit)}
$$

$$
\vdots \pi
$$

$$\frac{\vdots \pi}{\dfrac{\overline{a \simeq 1 \leftarrow \|a \simeq 1\|} \quad 3 \mid a}{\dfrac{3 \mid 1 \leftarrow \|a \simeq 1\|}{\bot \leftarrow \|a \simeq 1\|}}} \text{(CSup)}$$

and the same for $a \simeq 2$ and $a \simeq 4$. For $a \simeq 3$, we will use the negation of our goal, that is, $a \not\simeq 3$:

$$\frac{\dfrac{a \not\simeq 3}{a \leq 2 \vee 4 \leq a} \text{(TO)}}{a \leq 2 \leftarrow \|a \leq 2\| \quad 4 \leq a \leftarrow \|4 \leq a\| \quad \|a \leq 2\| \sqcup \|4 \leq a\|} \text{(ASplit)}$$

$$\vdots \pi_2$$

$$\frac{\vdots \pi_2 \qquad\qquad \vdots \pi}{\dfrac{\overline{a \leq 2 \leftarrow \|a \leq 2\|} \quad \overline{a \simeq 3 \leftarrow \|a \simeq 3\|}}{\dfrac{3 \leq 2 \leftarrow \|a \simeq 3\| \sqcap \|a \leq 2\|}{\bot \leftarrow \|a \simeq 3\| \sqcap \|a \leq 2\|}}} \text{(CSup)}$$

$$\frac{\vdots \pi_2 \qquad\qquad \vdots \pi}{\dfrac{\overline{4 \leq a \leftarrow \|4 \leq a\|} \quad \overline{a \simeq 3 \leftarrow \|a \simeq 3\|}}{\dfrac{4 \leq 3 \leftarrow \|a \simeq 3\| \sqcap \|4 \leq a\|}{\bot \leftarrow \|a \simeq 3\| \sqcap \|4 \leq a\|}}} \text{(CSup)}$$

so we obtain the clauses $\left(\neg\|a \simeq 1\|\right)$, $\left(\neg\|a \simeq 2\|\right)$, $\left(\neg\|a \simeq 4\|\right)$, $\left(\neg\|a \simeq 3\| \sqcup \neg\|a \leq 2\|\right)$, $\left(\neg\|a \simeq 3\| \sqcup \neg\|4 \leq a\|\right)$, $\left(\|a \leq 2\| \sqcup \|4 \leq a\|\right)$ and $\left(\|a \simeq 1\| \sqcup \|a \simeq 2\| \sqcup \|a \simeq 3\| \sqcup \|a \simeq 4\|\right)$, that is, an unsatisfiable boolean constraint.

Lemma 4.6 (Rules are Decreasing)**.** *The conclusion of an inference is strictly smaller (w.r.t. the $\prec_c$ ordering) than the maximal premise of the inference.*

Proof. By definition of $\prec_c$ for each case. $\qquad\square$

Remark 4.5. *It is possible that Modular Equality Factoring (CFact$|\simeq$) and the second case of Cancellative Inequality Factoring (CFact$\leq$) do not make the system more complete [Wal15]. We kept them in this presentation because their presence in the implementation might have an influence on the experimental results shown later.*

4.2.2   Lifting to First-Order

To reason over first-order clauses, including axioms on uninterpreted axioms (typically, monotonicity of a function, transitivity of a predicate, etc.), we lift this calculus to non-ground terms. Inferences will then require applying a substitution to their conclusion, and assume their premises share no variables. The first-order version of *Modular Chaining* is shown in Figure 4.4 as an illustration, the other first-order rules being similar. A restricted version of AC1-unification is used, that doesn't unify variables appearing directly under sums; the unification procedures will be explained in Section 4.6.3. We assume the clauses satisfy the following properties:

Modular Chaining

$$\frac{C \vee d^{e+k} \mid \sum_i a_i \cdot t_i + u \qquad C' \vee d^e \mid \sum_j a'_j \cdot t'_j + u'}{(C \vee C' \vee d^{e+k} \mid \varphi \cdot u - (d^k \cdot \varphi') \cdot u')\sigma}$$

where $a = \sum_i a_i \geq 1$, $a' = \sum_j a'_j \geq 1$, σ is a most general
AC1-unifier of all the t_i and t'_j, $a \cdot \varphi = a' \cdot d^k \cdot \varphi'$, literals are
maximal in their clause after applying σ, $t_1 \sigma \nprec u\sigma$, $t'_1 \sigma \nprec u'\sigma$,
$k \geq 0$

Figure 4.4: Inference Rule lifted to First-Order

- no *unshielded variable* (or *naked variable*) occurs in an arithmetic literal. We will present an algorithm to get rid of such variables in Section 4.4;
- clauses must be *purified*, as explained in Definition 4.12.

Every instance of such lifted rules correspond to some rule in Figure 4.2 or Figure 4.3 — except the ordering constraints that need be respected by at least one instance, not all of them. This makes the lifted versions sound iff the ground rules are. Of course, the actual implementation shall use the lifted rules, with all the subtleties entailed by the need of unifying multiple terms in each resolvent literal.

Remark 4.6. *Some rules, such as Cancellation of a divisibility literal, are only useful in the presence of variables, for otherwise they are subsumed by normalization rules.*

Example 4.10 (Inequality Factoring)**.** *Let us prove that the conjunction of $\forall x \, y. \, 10 \leq f(x) \vee 11 \leq f(y)$ and $f(a) \leq 5$ is unsatisfiable. The unary inference used here is Inequality factoring[4] using $\{x \mapsto y\}$, and the binary one is chaining, with $\{y \mapsto a\}$.*

$$\cfrac{\cfrac{\dfrac{10 \leq f(x) \vee 11 \leq f(y)}{11 + 1 \leq 10 \vee 10 \leq f(y)} \text{ (CFact≤)}}{10 \leq f(y)} \qquad f(a) \leq 5}{\dfrac{10 \leq 5}{\perp}} \text{ (Chain≤)}$$

4.3 Redundancy

It is well known that automated theorem provers generally need refinements that help them prune large parts of the search space and avoid wasting resources on useless clauses or formulas. After writing a simple theorem prover, one can easily see why redundancy criteria are required, in practice, for the prover not to drown in too large a search space. Happily, we can rely on the usual abstract notion of redundancy presented earlier (Section 2.4.3). The classic rules of superposition — subsumption and demodulation (rewriting with unit equations) — still apply to our arithmetic calculus, but other, more specific, rules are useful too.

Definition 4.13 (Arithmetic Redundancy)**.** *A ground clause C is $\mathfrak{I}_{\text{arith}}$-redundant w.r.t. a set of ground clauses N iff $\exists D_1 \ldots D_n \in N. \, D_1 \wedge \cdots \wedge D_n \vdash_{\text{arith}} C$ and $C \succ_c D_i$ for all D_i. A clause C is $\mathfrak{I}_{\text{arith}}$-redundant with respect to N if for each of its ground instances $C\sigma$, $N^{<C\sigma} \vdash_{\text{arith}} C\sigma$*

---

[4] In the next section we will present the Condensation simplification rule, which also applies here.

4.3.1 Simplification Rules

Let us first focus on inferences that make one of their premises redundant, commonly named *simplifications* (because the conclusion "replaces" the now obsolete premise). The usual demodulation (rewriting with unit positive equations) is easily extended to unit positive arithmetic equations and divisibility literals. A set of useful simplifications dedicated to arithmetic that are implemented in Zipperposition is listed in Figure 4.5. We see that Cancellative Demodulation is actually a specialized version of Cancellative Superposition, for cases where the active clause is unit and the rewriting strictly decreases (which is not always the case in the first-order rule; the inference is only non-increasing); similarly, Cancellative Divisibility Demodulation is a specialized version of Modular Chaining.

Cancellative Demodulation

$$\frac{\sum_i a_i \cdot t_i + u \simeq v \qquad C' \vee a' \cdot t' + u' \sim v'}{C' \vee \varphi \cdot u\sigma + \varphi' \cdot v' \sim \varphi' \cdot u' + \varphi \cdot v\sigma}$$

where $\sim \in \{\simeq, \not\simeq\}$, $\forall i.\ t' = t_i\sigma$, $\varphi \times \sum_i a_i = \varphi' \times a'$, $t' > u\sigma$, $t' > v\sigma$, $t' \not< u'$, $t' \not< v'$, all vars of the first premise are bound in σ, $(a' \cdot t + u' \sim v') \succ_c C'$

Cancellative Divisibility Demodulation

$$\frac{d^k \mid \sum_i a_i \cdot t_i + u \qquad C' \vee d^k \mid a' \cdot t' + u'}{C' \vee d^k \mid \varphi \cdot u\sigma - \varphi' \cdot u'}$$

where $\forall i.\ t' = t_i\sigma$, $\varphi \times \sum_i a_i = \varphi' \times a'$, $t' > u\sigma$, $t' \not< u'$, all vars of the first premise are bound in σ, $(d^k \mid a' \cdot t + u') \succ_c C'$

Figure 4.5: The Simplification Rules of $\mathfrak{I}_{\text{arith}}$

4.3.2 Subsumption

A clause C can also be made redundant by other clauses that aren't directly deduced from C. In particular, the notion of *subsumption* is used in almost every saturation-based theorem prover. Roughly, given clauses C and D, we say C *subsumes* D with a substitution σ if $C\sigma \subseteq D$ (where $\subseteq$ is the multiset inclusion). It means that every instance of D is implied by a smaller instance of C, and therefore D is redundant. Even though subsumption is a decidable subset of implication, it only uses syntactic equality to check whether a literal (in $C\sigma$) implies another literal (in D). We could use the same notion of implication for arithmetic clauses, but in this section we will see a much stronger notion of decidable entailment between arithmetic literal (and therefore between clauses).

The subsumption relation we use is noted $l_1 \sqsubseteq_{\text{arith},\sigma} l_2$ (l_1 subsumes l_2 with substitution σ, that is, $l_1\sigma \vdash_{\text{arith}} l_2$). We write $l_1 \sqsubseteq_{\text{arith}} l_2$ if there is a σ such that $l_1 \sqsubseteq_{\text{arith},\sigma} l_2$. This subsumption relation extends to clauses by $\bigvee_{i=1}^{n} u_i \sqsubseteq_{\text{arith},\sigma} \bigvee_{j=1}^{m} v_j$ ($m \geq n$) if there is an injection ρ with $\forall i \in \{1,\dots,n\}.\ u_i \sqsubseteq_{\text{arith},\sigma} v_{\rho(i)}$.

To define $\sqsubseteq_{\text{arith},\sigma}$, we first define *matching substitutions* on linear expressions (sums of atomic terms) u and u' as tuples $(\sigma, \varphi, \varphi')$ where σ is a substitution, $\varphi, \varphi' \in \mathbb{N}^{+2}$, and $\varphi \cdot u\sigma = \varphi' \cdot u'$. The notation $u \succ_{(\sigma,\varphi,\varphi')} u'$ means that the tuple $(\sigma, \varphi, \varphi')$ matches u with u'. This relation also extends to multisets of linear expressions by $\{u_1, u_2, \dots, u_n\} \succ_{(\sigma,\varphi,\varphi')} \{u'_1, u'_2, \dots, u'_n\}$ if there is a permutation ρ with $\forall i \in \{1,\dots,n\}.\ u_i \succ_{(\sigma,\varphi,\varphi')} u'_{\rho(i)}$, and it extends to tuples of linear expressions of equal arity by pairwise matching of the tuples' components: $(u_1, u_2, \dots, u_n) \succ_{(\sigma,\varphi,\varphi')} (u'_1, u'_2, \dots, u'_n)$ if $\forall i \in \{1,\dots,n\}.\ u_i \succ_{(\sigma,\varphi,\varphi')} u'_i$. Using this notion of matching, subsumption is

defined in Figure 4.6.

$$
\begin{array}{llll}
u \simeq v & \sqsubseteq_{\mathrm{arith},\sigma} & u' \simeq v' & \text{if } \{u,v\} \succ_{(\sigma,\varphi,1)} \{u',v'\} \\
u \simeq v & \sqsubseteq_{\mathrm{arith},\sigma} & u' \le v' + k \cdot \mathbf{1} & \text{if } (u,v) \succ_{(\sigma,\varphi,\varphi')} (u',v'), k \ge 0 \\
u \le v & \sqsubseteq_{\mathrm{arith},\sigma} & u' \not\simeq v' + k \cdot \mathbf{1} & \text{if } (u,v) \succ_{(\sigma,\varphi,\varphi')} (u',v'), k > 0 \\
& & & \text{or } (u,v) \succ_{(\sigma,\varphi,\varphi')} (v',u'), k < 0 \\
u \le v & \sqsubseteq_{\mathrm{arith},\sigma} & u' \le v' + k \cdot \mathbf{1} & \text{if } (u,v) \succ_{(\sigma,\varphi,\varphi')} (u',v'), k \ge 0 \\
u \simeq v & \sqsubseteq_{\mathrm{arith},\sigma} & d^k \mid u' & \text{if } u - v \, [d^k] \succ_{(\sigma,\varphi,1)} u' \text{ and } \mathrm{mt}(u \simeq v) = \mathrm{mt}(u - v \, [d^k]) \\
d^{k+k'} \mid u & \sqsubseteq_{\mathrm{arith},\sigma} & d^k \mid u' & \text{if } u \succ_{(\sigma,\varphi,1)} u'
\end{array}
$$

Figure 4.6: Subsumption Relation on Arithmetic Literals

Remark 4.7. *Care must be taken that the conclusion of an inference rule is not subsumed by a premise, as in Superposition. In particular, $2 \cdot t \simeq v$ cannot subsume $2 \mid v$ (the conclusion of a Divisibility inference) because if $t > v$, then some necessary inferences with $2 \mid v$ cannot be done with $2 \cdot t \simeq v$ — hence the restrictions on the corresponding subsumption rule. The attentive reader might notice that $(2 \cdot t \simeq v) \succ_{lit} (2 \mid v)$ anyway in this case, which prevents the former from subsuming the latter according to Definition 4.13.*

Example 4.11 (Subsumption). *To better grasp the meaning of those subsumption rules, let use consider a few examples:*

- $\big(f(x) + f(y) \le b\big) \sqsubseteq_{arith,\sigma} \big(2 \cdot f(a) \not\simeq b + 10\big)$, *with* $\sigma = \{x \mapsto y, y \mapsto a\}$.
- $(0 \le len(l)) \sqsubseteq_{arith,\sigma} \big(0 \le 2 \cdot len(l') + 1\big)$ *with* $\sigma = \{l \mapsto l'\}$
- $\big(f(x) \simeq x + a\big) \sqsubseteq_{arith,\sigma} \big(f(a) \simeq 2 \cdot a\big)$ *with* $\sigma = \{x \mapsto a\}$
- $(a \simeq 2 \cdot b + 4 \cdot c) \sqsubseteq_{arith,\emptyset} (4 \mid 2 \cdot b + 3 \cdot a)$

Theorem 4.1. *The subsumption relation $\sqsubseteq_{arith}$ is sound w.r.t. integer linear arithmetic, that is, $l_1 \sqsubseteq_{arith,\sigma} l_2$ implies $l_1 \sigma \vdash_{\mathrm{arith}} l_2$.*

Using this subsumption relation, we can both remove clauses that are subsumed by other clauses, and powerful simplification rules built upon subsumption such as *condensation* and *contextual literal cutting* (see Figure 2.3).

Remark 4.8 (Decidable Entailment). *In some contexts, a decidable entailment relation such as $\sqsubseteq_{arith}$ can prove very useful. For instance, the particular type of induction proposed by Kersani & Peltier [KP13] uses such a relation (typically alpha-equivalence or subsumption) to detect loops in the search space and thus reason by infinite descent. In such cases, our subsumption relation can prove useful.*

4.3.3 Inequality Demodulation

A big issue with ordering literals is that they do not have an equivalent of demodulation, because no inference we can perform on them preserves equivalence (in contrast to equality, where $a \simeq b$ makes $C[a]_p$ and $C[b]_p$ equivalent, thus allowing us to replace the former with the latter). On the other hand, experience shows that literals such as $l = \forall x : \mathsf{list(int)}. \, \mathsf{len}(x) \ge 0$ combined with other axioms tend to generate a lot of useless variants such as $\forall x_1 \, x_2 : \mathsf{list(int)}.$ $\mathsf{len}(x_1) + 2 \cdot \mathsf{len}(x_2) \ge 0$ which are not properly subsumed by the original axiom l (because the latter only has one variable and cannot match both l_1 and l_2). We need to show those conclusions are redundant, using *several* instances of l. If a set of unit $\le$-clauses can be used to rewrite a literal l to $\top$, then we know $\top \Rightarrow l$ (meaning l is redundant) — similarly, a literal l can

be shown to imply $\bot$, making it absurd. We define below such a rewrite system and the exact rules by which a trivial or absurd literal is eliminated.

We know that if we have a unit clause $C \stackrel{\text{def}}{=} t + u \le v$, and some clause $D \stackrel{\text{def}}{=} D' \vee t\sigma + u' \le v'$, then $(v - u)\sigma \le (v' - u')$ (that is, $v\sigma + u' \le u\sigma + v'$) means that $t\sigma \le v' - u'$ is true. In this case the clause D is redundant (if it's bigger than C in the ordering). We therefore define the relation $\xhookrightarrow{\Leftarrow,\le,N}$ parametrized over a set of clauses N by the rewrite system in Figure 4.7. Intuitively, $l \xhookrightarrow{\Leftarrow,\le,N} l'$ means $l' \wedge N \vdash_{\text{arith}} l$; in other words, l' is a sufficient condition for l given some already proved background assumptions N, and if we can prove that $l \xhookrightarrow{\Leftarrow,\le,N} \top$ it means that l is trivially true when N is. A literal L is tautological if $L \xhookrightarrow{\Leftarrow,\le,N}{}^{*} \top$. Note that $\xhookrightarrow{\Leftarrow,\le,N}$ does not rewrite literals in place — that would not preserve equivalence —, but instead, we compute a normal form of l using $\xhookrightarrow{\Leftarrow,\le,N}$ and compare it to $\top$.

Inequality Demodulation Left

$$a \cdot t + u \le v \xhookrightarrow{\Leftarrow,\le,N} \varphi \cdot u + \varphi' \cdot v'\sigma \le \varphi \cdot v + \varphi' \cdot u'\sigma$$

$$\text{if } (\textstyle\sum_{i=1}^{n} a_i' \cdot t_i' + u' \le v') \in N, \; a' \stackrel{\text{def}}{=} \textstyle\sum_{i=1}^{n} a_i',$$
$$\varphi \times a = \varphi' \times a' = \text{lcm}(a, a')$$
$$t \succ u, \; t \succ v, \; \forall i \in \{1, \dots, n\}. \; t_i'\sigma = t, t_i'\sigma \succ u'\sigma, t_i'\sigma \succ v'\sigma$$

Inequality Demodulation Right

$$u \le a \cdot t + v \xhookrightarrow{\Leftarrow,\le,N} \varphi \cdot u + \varphi' \cdot v'\sigma \le \varphi \cdot v + \varphi' \cdot u'\sigma$$

$$\text{if } (u' \le \textstyle\sum_{i=1}^{n} a_i' \cdot t_i' + v') \in N, \; a' \stackrel{\text{def}}{=} \textstyle\sum_{i=1}^{n} a_i',$$
$$\varphi \times a = \varphi' \times a' = \text{lcm}(a, a')$$
$$t \succ u, \; t \succ v, \; \forall i \in \{1, \dots, n\}. \; t_i'\sigma = t, t_i'\sigma \succ u'\sigma, t_i'\sigma \succ v'\sigma$$

Figure 4.7: Inequality Rewrite System

Similarly, we can specialize the regular chaining relation from Figure 4.2 into a simplification version $\xhookrightarrow{\Rightarrow,\le,N}$. $t\sigma + u \le v \xhookrightarrow{\Rightarrow,\le,N} u + v'\sigma \le v + u'\sigma$ if $v' \le u' + t \in N$ (and symmetrically). If, for some literal l, $l \xhookrightarrow{\Rightarrow,\le,N}{}^{*} \bot$, we know that l is absurd and can be removed from the clause, because $l \xhookrightarrow{\Rightarrow,\le,N} l'$ implies $l \wedge N \vdash_{\text{arith}} l'$, i.e., $N \vdash_{\text{arith}} l \Rightarrow l'$. Otherwise, l is kept intact.

Lemma 4.7 (Termination)**.** *The rewrite relations* $\xhookrightarrow{\Leftarrow,\le,N}$ *and* $\xhookrightarrow{\Rightarrow,\le,N}$ *are terminating.*

Proof. At each step the maximal term is replaced with finitely many strictly smaller terms, which makes the literal smaller w.r.t. $\succ_{\text{lit}}$. $\qquad\square$

We've seen how to tackle a problem that often occurs with inequality literals and chaining. Both those rules and the subsumption relation from Section 4.3.2 can also be used regardless of the inference system. In particular, they provide a decidable implication relation (i.e., a relation included in $\vdash_{\text{arith}}$) that might be leveraged in other calculi operating on arithmetic literals, such as hierarchic superposition [BGW94, BW13].

4.3.4 Semantic Tautologies

Tautologies are harmful to Superposition theorem provers if not eliminated, because they cannot contribute to any unsatisfiability proof but still increase the size of the search space because they can participate in inferences. Some tautologies are very easy to detect — for instance, $l \vee \neg l \vee C$ is obviously a trivial clause — but some stronger criteria exist. As a comparison point, E [Sch02] has a notion of *equational tautologies*: $\bigvee_i u_i \ne v_i \vee \bigvee_j u_j' \simeq v_j'$ is redundant if there is a

j such that $\left[\left(\bigwedge_i u_i \simeq v_i\right) \Rightarrow u'_j \simeq v'_j\right]\sigma$ with σ a substitution mapping each variable to a different opaque constant (in practice, one can use a congruence closure algorithm [NO80] to check it efficiently). In the same vein, we use the Simplex method [KS08] as follows: for a clause

$$C \overset{\text{def}}{=} \bigvee_i (u_i \leq v_i) \vee \bigvee_j \left(u'_j \neq v'_j\right) \vee D$$

we define the set of linear equations

$$S_C \overset{\text{def}}{=} \{u_i - v_i \geq 1\}_i \cup \left\{u'_j - v'_j \geq 0, u'_j - v'_j \leq 0\right\}_j \cup$$

S_C is a linear integer problem whose conjunction represents the negation of $C \setminus D$ (i.e., $\bigwedge_i (u_i > v_i) \wedge \bigwedge_j (u'_j \simeq v'_j)$). We can use the Simplex method to determine whether it is satisfiable in $\mathbb{Q}$. If S_C is not satisfiable in $\mathbb{Q}$, it doesn't admit rational solutions and therefore it doesn't admit integer solutions either. In that case, its negation $C \setminus D$ is a tautology and so is C.

Lemma 4.8 (Tautology Detection via Simplex). *If S_C is unsatisfiable in $\mathbb{Q}$, then C is a tautology and can be safely removed from the set of clauses.*

Example 4.12 (Tautological Clause). *For instance, the clause $C \overset{\text{def}}{=} p \vee 4 \mid t+1 \vee 2 \cdot t \leq 5 \vee t \simeq 3 \vee 2 \leq t$ is a tautology. By definition*

$$S_C \overset{\text{def}}{=} \{2 \cdot t - 5 \geq 1, 2 - t \geq 1\} = \{t \geq 3, t \leq 1\}$$

which makes S_C trivially unsatisfiable in $\mathbb{Q}$ and C trivial.

4.4 Variable Elimination

The lifted version of $\mathfrak{I}_{\text{arith}}$ (Section 4.2.2) only works with clauses whose variables are all shielded (Definition 4.10). The reason is that shielded variables are always smaller than their shielding term and therefore cannot be involved in inferences, sparing us from having to use AC1-unification. However, some inferences may *un-shield* variables (by eliminating the last shielding term); therefore, we need to combine $\mathfrak{I}_{\text{arith}}$ with a procedure to eliminate those un-shielded variables so we get usable clauses again. If we accept to interpret terms in int with the standard integers $\mathbb{Z}$ (and operators, including divisibility, defined the obvious way), we can use Cooper's quantifier elimination algorithm [Coo72] for Presburger arithmetic.

Let us consider a clause C in which the variable $x : \text{int}$ is unshielded. Our goal is to find a set of clauses $\text{elim}_x(C)$ such that $x \notin \text{freevars}(C)$, $C \vdash_{\text{arith}} \bigwedge_{D \in \text{elim}_x(C)} D$ and $\text{elim}_x(C) \vdash_{\text{arith}} C$. For a start, if $C \overset{\text{def}}{=} C' \vee x + u \neq v$ with x unshielded and $x \notin \text{freevars}(u)$ [5], we can eliminate x directly and simplify C into $C'\{x \mapsto v - u\}$ (for any other instance of x will trivially satisfy the clause).

Let $C \overset{\text{def}}{=} C' \vee \bigvee_{i=1}^{k} l_i[x]$ with $x \notin \text{freevars}(C')$ and $k \geq 1$. C is classically equivalent to $C' \vee \neg\left(\exists x. \bigwedge_{i=1}^{k} \neg l_i[x]\right)$; the sub-formula $F[x] = \bigwedge_{i=1}^{k} \neg l_i[x]$ is quantifier-free, in disjunctive normal form, and all its literals are by hypothesis arithmetic literals directly involving x. We can therefore apply Cooper's algorithm to $\exists x. F[x]$ to eliminate x. First, let δ be the least common multiple of all δ_i such that $\delta_i \cdot x + u_i \simeq v_i$ is a $\neg l_i[x]$ in $F[x]$. Then, multiply sides of every $\neg l_i[x]$ by δ/δ_i (an integer), replace $\delta \cdot x$ by x', thus obtaining a formula $F'[x']$ (in which all occurrences of x' appear with coefficient 1). Let $G[x'] = F'[x'] \wedge \delta \mid x'$, so that by construction $\exists x'. G[x'] \Leftrightarrow \exists x. F[x]$. We partition $G[x']$ into several "kinds" of literals (remember that literals

---

[5] $x \notin \text{freevars}(v)$ is always true if the clause is normalized, which is assumed here.

in $G[x']$ are the negation of literals of C; that explains the use of $<$ rather than $\le$):

$$a_i[x'] \overset{\text{def}}{=} x' + u_{a_i} \simeq v_{a_i}$$
$$b_j[x'] \overset{\text{def}}{=} x' + u_{b_j} \not\simeq v_{b_j}$$
$$c_k[x'] \overset{\text{def}}{=} n_{c_k} \mid x' + u_{c_k}$$
$$d_l[x'] \overset{\text{def}}{=} n_{d_l} \nmid x' + u_{d_l}$$
$$e_m[x'] \overset{\text{def}}{=} x' + u_{e_m} < v_{e_m}$$
$$f_n[x'] \overset{\text{def}}{=} u_{f_n} < x' + v_{f_n}$$

Remark 4.9. *We treat negative literals in the input, of shapes $u \not\simeq v$ and $n \nmid u$ — even though they do not appear in normalized forms —, in order to be as comprehensive as possible. If a prover was built with slightly different assumptions (for instance, if Prime Case Switch and Total Order were inference rules rather than normalization ones), it could still benefit from the algorithm presented here.*

Now, let $A \overset{\text{def}}{=} \{v_{e_m} - u_{e_m}\}_m \cup \{v_{a_i} - u_{a_i} + \mathbf{1}\}_i \cup \{v_{b_j} - u_{a_j}\}_j$ and $B \overset{\text{def}}{=} \{v_{a_i} - u_{a_i} - \mathbf{1}\}_i \cup \{v_{b_j} - u_{b_j}\}_j \cup \{u_{f_n} - v_{f_n}\}_n$ be sets of signed linear expressions and $\delta' \overset{\text{def}}{=} \text{lcm}\left(\bigcup_k \{n_{c_k}\} \cup \bigcup_l \{n_{d_l}\}\right)$. Intuitively, A is a set of potential strict upper bounds, and B a set of potential strict lower bounds for x'. We can choose between the following versions of Cooper's algorithm, depending on whether A has more elements than B[6]:

$$\exists x'.\, G[x'] \iff \bigvee_{n=1}^{\delta'} G_{-\infty}[n] \vee \bigvee_{n=1}^{\delta'} \bigvee_{j \in B} G[j + n]$$

or (if A is smaller)

$$\exists x'.\, G[x'] \iff \bigvee_{n=1}^{\delta'} G_{\infty}[-n] \vee \bigvee_{n=1}^{\delta'} \bigvee_{j \in A} G[j - n]$$

where

$$G_{-\infty}[x] = \begin{cases} \bot & \text{if } \{a_i[x']\}_i \cup \{f_n[x']\}_n \neq \varnothing \\ \bigwedge_{k,l}\left(n_{c_k} \mid u_{c_k} + x \wedge n_{d_l} \nmid u_{d_l} + x\right) & \text{otherwise} \end{cases}$$

$$G_{\infty}[x] = \begin{cases} \bot & \text{if } \{a_i[x']\}_i \cup \{e_m[x']\}_m \neq \varnothing \\ \bigwedge_{k,l}\left(n_{c_k} \mid u_{c_k} + x \wedge n_{d_l} \nmid u_{d_l} + x\right) & \text{otherwise} \end{cases}$$

Now we use the distributivity of $\wedge$ and $\vee$ to obtain the conjunctive normal form of our result. $C' \vee \neg(\exists x'.\, G[x'])$ becomes, writing $\varphi^T[x] = \bigvee_i \neg l_i[x]$ when $\varphi[x] = \bigwedge_i l_i[x]$, the following set:

$$\text{elim}_x(C) = \bigcup_{n=1}^{\delta} \{C' \vee G_{-\infty}^T[n]\} \cup \bigcup_{n=1}^{\delta} \bigcup_{j \in B} \{C' \vee G^T[j + n]\}$$

or

$$\text{elim}_x(C) = \bigcup_{n=1}^{\delta} \{C' \vee G_{\infty}^T[-n]\} \cup \bigcup_{n=1}^{\delta} \bigcup_{j \in A} \{C' \vee G^T[j - n]\}$$

Note that if $G_{-\infty}[n]$ is $\bot$, then $G_{-\infty}^T[n]$ is $\top$ and the corresponding clause is trivial, so we can ignore it.

---

[6]Both choices are always valid, the only difference is efficiency w.r.t. the number of clauses generated.

Theorem 4.2 (Variable Elimination)**.** *Let C is a clause with unshielded variables $x_1, \ldots, x_n$ and $elim_{x_1,\ldots,x_n}(C) \stackrel{def}{=} elim_{x_1}(\ldots(elim_{x_n}(C)\ldots))$. Then no clause in $elim_{x_1,\ldots,x_n}(C)$ contains any unshielded variables, $C \vdash_{\text{arith}} elim_{x_1,\ldots,x_n}(C)$, and $elim_{x_1,\ldots,x_n}(C) \vdash_{\text{arith}} C$.*

Example 4.13 (Variable Elimination)**.** *Let $C \stackrel{def}{=} p(x) \vee x \neq 3 \cdot y$ (typically obtained by purifying $\forall x.\ p(3 \cdot x)$). To eliminate y, we typically perform the renaming $\{y' \mapsto 3 \cdot y\}$ and obtain $A = B \stackrel{def}{=} \{x\}$, $C' \stackrel{def}{=} p(x)$, $G[y'] \stackrel{def}{=} y' \simeq x \wedge 3 \mid y'$, and $\delta' = 3$. Both forms will yield the same result, let us show what happens with the $G_{-\infty}$ one: $\exists y'.\ G[y'] \iff \bigvee_{n=1}^{3} G_{-\infty}[n] + \bigvee_{n=1}^{3} \{C' \vee G^T[x+n]\}$. Clearly, $G_{-\infty}[-n]$ is $\bot$ for every $n \in \{1,\ldots,3\}$, and $G^T[x+n] = x+n \not\simeq x \vee 3 \nmid x+n$, so we obtain $\bigvee_{n=1}^{3} (p(x) \vee x+n \not\simeq x \vee 3 \nmid x+n)$. The only non-trivial case is $n = 0$; the final result after simplification is $p(x) \vee 3 \nmid x$.*

4.5 Completeness

Although we strived to tackle as many cases as possible with the inference system $\mathfrak{I}_{\text{arith}}$, it is not refutationally complete in the general case. The following counter-example is due to Uwe Waldmann [Wal15]:

Example 4.14 (Counter-Example to Completeness)**.** *Assuming $a > b > c > d > e$, the clauses*

$$7 \mid a \quad 7 \mid b \quad a \leq b \quad b \leq a+c$$
$$2 \cdot c + d \simeq e \vee 2 \cdot c + d \simeq e + 4 \vee e \leq d$$
$$d + 2 \simeq e \vee d + 4 \simeq e$$

are unsatisfiable, because the two last clauses imply $\bigvee_{i=1}^{4} c \simeq i$ (by case on the last clause). Yet no equality from $\{c \simeq i\}_{i=1}^{4}$ is generated, because of the term ordering $\succ$. Without those equalities, the crucial case switch between $a \leq b$ and $b \leq a+c$ is not performed, and the contradiction with 7 dividing both a and b is not exposed.

It is not clear, as of now, how the inference system should be extended to tackle this problem. However, as we will see in the next section, the calculus can be implemented and performs well in practice.

4.6 Implementation

So far, we have defined several inference rules, simplification rules and other techniques to deal with redundancy or unshielded variables. Many of them were crafted to solve or mitigate actual problems in the implementation (in particular, the concept of Inequality Demodulation, Section 4.3.3). Implementing the inference and simplification rules is a challenge by itself: to our knowledge, the calculus from Waldmann [Wal01] was never implemented despite its completeness for arithmetic on an axiomatization of rational numbers[7]. We emphasize the importance of implementation for as complex an inference system as $\mathfrak{I}_{\text{arith}}$. It may look good on paper, but until a prototype that behaves reasonably well is built, the practical usefulness of the calculus is doubtful at best. In this section, we will address some issues we met while implementing our calculus in the experimental theorem prover Zipperposition. The total amount of code required for the arithmetic extension is around 4,000 lines of OCaml, including a module to deal with generic linear sums. We used the Zarith[8] wrapper around the GMP[9] library, to

---

[7] Unlike the present work and [KV07], the calculus from [Wal01] uses a set of axioms that have $\mathbb{Q}$ as a model, and decides (un)satisfiability w.r.t. any model of this set of axioms rather than the standard model $\mathbb{Q}$ (which is undecidable in general).

[8] https://forge.ocamlcore.org/projects/zarith

[9] https://gmplib.org/

represent arbitrary-precision integers and rational numbers. Guillaume Bury's simplex implementation[10] was also used for detecting semantic tautologies.

The lifted rules (Figure 4.4) require unifying several terms in linear expressions within literals of one or two clauses. We will see that full AC1-unification is not needed; the implementation relies on the same term index structures as standard superposition. The type `Literal.t` is enriched with new variants to represent arithmetic literals, and some simplification and inference rules to the saturation loop. The most subtle part of the implementation is related to unification and matching of *linear expressions* and literals (see in particular the matching relation used for subsumption, Section 4.3.2). We first present a brief version of the code used to represent linear expressions; then, we present unification and matching algorithms related to linear expressions and literals, but only after we explain how iterators can help deal with the inherent complexity of this kind of backtracking algorithms.

4.6.1   Representation of Linear Expressions

Linear Expressions (sums of atomic terms) are the workhorse of arithmetic literals and clauses. An integer linear expression is represented in OCaml as follows (`Z.t` is the type of arbitrary precision integers in Zarith, although we always deal only with non-negative numbers).

```
type linexp = {
    const : Z.t; (* ≥0 *)
    terms : (Z.t * term) list; (* each coeff >0 *)
}

val singleton : Z.t → term → linexp
val add : Z.t → term → linexp → linexp

val sum : linexp → linexp → linexp
val difference : linexp → linexp → linexp
(* ... *)

type focused_linexp = {
    term : term;
    coeff : Z.t; (* >0 *)
    rest : linexp;
}
val focus : term → linexp → focused_linexp option
val unfocus : focused_linexp → linexp
```

A value `m : linexp` describes a linear expression $\sum_{(a,t)\in\mathsf{m.terms}} a \cdot t + \mathsf{m.const} \cdot \mathbf{1}$, such that $a > 0$ for every $(a, t) \in \mathsf{m.terms}$. A `focused_linexp` can be obtained from a non-constant linear expression by simply extracting a term and its coefficient with `focus` (which can fail if the term is not present in the linear expression), and conversely get a linear expression back using `unfocus`. We will underline the focused term in algorithms when necessary. For instance, $\underline{2 \cdot t} + 3 \cdot u + 5 \cdot v$ is the focused linear expression m with $\mathsf{m.term} = t$, $\mathsf{m.coeff} = 2$, and $\mathsf{m.rest} = 3 \cdot u + 5 \cdot v$. We extend this notion of focusing to arithmetic literals by focusing on a term in one side of the literal (e.g., $\underline{3 \cdot t} + u \not\approx 2 \cdot v$ in which $3 \cdot t$ is focused on).

4.6.2   Monadic Iterators for Backtracking

The unification algorithms might return several values, and in general are backtracking in nature. Since we chose OCaml and not Prolog for implementation, we sought a way to write backtracking functions without tearing our hair out[11]. Our quest lead us to the type[12] α `sequence`

---

[10] https://github.com/Gbury/Ocaml-simplex

[11] The number of opportunities to lose hair during a technical career is already high enough...

[12] and then to writing a library around it: https://github.com/c-cube/sequence. The curious reader can read `sequence.ml` to get a grasp of how other combinators are implemented.

(shown below) which is a very simple and fast iterator over values of type α. In particular, back-tracking is easy to achieve using the monadic interface (`return` and `>>=`) or with an explicit continuation of type $\alpha \rightarrow$ `unit`.

```
type α sequence = (α → unit) → unit

val of_list : α list → α sequence
val empty : α sequence
val return : α → α sequence
val map : (α → β) → α sequence → β sequence
val (>>=) : α sequence → (α → β sequence) → β sequence
val (@) : α sequence → α sequence → α sequence
val head : α sequence → α option
val cons : α → α sequence → α sequence
(* ... *)
```

and the implementation

```
let empty _ κ = ()
let return x κ = κ x
let map f s κ = s (fun x → κ (f x))
let (>>=) s f κ = s (fun x → f x κ)
let (@) a b κ = a κ; b κ
let cons hd tl κ = κ hd; tl κ

let head s =
  let r = ref None in
  ( try s (fun x → r := Some x; raise Exit)
    with Exit → ()
  );
  !r

let rec of_list l κ = match l with
    | [] → ()
    | x :: tl → κ x; of_list tl κ
```

A simple example of backtracking using `sequence` is sorting a list by enumerating all its permutations[13], filtering to keep only the sorted ones, and keep only the first one. To enumerate all the permutations, we first define a way to insert an element e in a list l (iterating over all possible ways to do so), then we define `permute (e::l) = permute l >>= insert e` — permute the tail, then put the head anywhere in each resulting permutation.

```
open Sequence

(* insert [e] at every position in [l] *)
let rec insert e l = match l with
  | [] → return [e]
  | x::tail →
    cons (e::l) (insert e tail >>= fun tail2 → return (x::tail2))

let rec permute l = match l with
  | [] → return []
  | x::l → permute l >>= fun l2 → insert x l2

let rec sorted l = match l with
  | [] | [_] → true
  | x::((y::l') as l) → x ≤ y ∧ sorted l

let perm_sort l = head (filter sorted (permute l))
```

---

[13]we know it is not the most efficient way to do it.

4.6.3 Unification Algorithms

A few unification and matching algorithms are necessary to implement the inference and simplification rules if we want to avoid implementing AC1-unification (and, more critically, AC1-indexing). We present a few important functions, implemented in the continuation-passing style introduced above. The techniques presented here go a long way in making the implementation of $\mathfrak{I}_{\text{arith}}$ tractable.

```
(* on focused linear expressions *)
val unify_self_f : subst → focused_linexp →
                    (focused_linexp * subst) sequence

val unify_ff : focused_linexp → focused_linexp →
               (focused_linexp * focused_linexp * subst) sequence

val unify_mm : linexp → linexp →
               (focused_linexp * focused_linexp * subst) sequence

val unify_self_m : subst → linexp → (focused_linexp * subst) sequence

val matching : subst → linexp → linexp → subst sequence
```

Let us detail the unification algorithms[14], all of which are n-ary and therefore return iterators over solutions. We use the α sequence combinators defined above to handle backtracking.

unify_self_f takes σ:subst and m:focused_linexp and iterates over distinct pairs (m', ρ) such that $\sigma \le \rho$ and $m\rho = m'$. In other words, it can unify together several terms inside $m\sigma$. Example: it will yield $(3 \cdot f(x) + a, \{y \mapsto x\})$ and $(2 \cdot f(x) + f(y) + a, \emptyset)$ for $m = 2 \cdot f(x) + f(y) + a$ and $\sigma = \emptyset$. It is used in the implementation of some of the following functions, and in the code for the Divisibility rule.

```
let rec iter_self σ c t l m = match l with
| [] →
    return ({coeff=c; term=t; rest=m}, σ)
| (c2, t2) :: l2 →
    (* must merge, t = t2 † *)
    if tσ = t2σ then iter_self σ (c + c2) t l2 m
    else (
      (* we can choose not to unify t and t2. *)
      iter_self σ c t l2 (add c2 t2 m) @
      (try (* try to unify t and t2 *)
        let ρ = unify σ t t2 in
        let m2 = {m with terms=[]} in (* might have to merge † *)
        iter_self ρ (c + c2) t (l2 @ m.terms) m2
      with Fail → empty)
    )

let unify_self_f σ mf =
    let m = mf.rest in  (* unfocused part *)
    iter_self σ mf.coeff mf.term m.terms {m with terms=[]}
```

This code might be difficult for readers not accustomed to sequence. The function that does the work, iter_self, is given σ and $c \cdot t + l + m$ (where l is a (Z.t * term) list). It iterates on l and, for each pair $(c_2, t_2) \in l$, makes a choice between unifying $t\sigma$ and $t_2\sigma$ (obtaining $\sigma \le \rho$) or keeping them distinct — putting (c_2, t_2) in the unfocused part of the result. The lines annotated † exist because unifying $t\sigma$ with $t_2\sigma$ with $\sigma \le \rho$ might make some terms of $m\rho$ equal to $t\rho$ and thus extend the *focus* area to them. The function terminates because the pair

---

[14] in names, "f" is short for "focused" and "m" for "monome", the old designation of linear expressions in the code.

(`length l + length m.terms, length l`) decreases strictly at each recursive call. Note that `iter_self` will be re-used in the implementation of several other functions below.

unify_self_m   is similar to `unify_self_f`, but with an unfocused linear expression m as argument. If it can unify (at least) two terms t_1 and t_2 in $m \stackrel{\text{def}}{=} a_1 \cdot t_1 + a_2 \cdot t_2 + m'$ with σ, it yields $((a_1 + a_2) \cdot t_1 + m', \sigma)$ (or it can extend the substitution to other terms in m'). On $2 \cdot f(x) + f(y) + a$, for instance, it will only yield $(3 \cdot f(x) + a, \{y \mapsto x\})$. It is used, for instance, to implement Cancellation in literals $n \mid^?_u u$.

The implementation has to unify at least two terms in the linear expression (respectively chosen by `choose_first` and `choose_second`) to succeed; for any choice of (`t, t2`) unified by ρ, `iter_self` is called to enumerate the ways of extending the substitutions to other terms (and eventually call the continuation κ upon success).

```
let unify_self_m σ m =
  (* find a term to focus on *)
  let rec choose_first σ l m = match l with
  | [] → empty
  | (c,t)::l2 →
      choose_second σ c t l2 m @ (* focus on t *)
      choose_first σ l2 (add c t m) (* do not focus on t *)

  (* find a second term in l to unify with focused term t *)
  and choose_second σ c t l m = match l with
  | [] → empty
  | (c2,t2)::l2 →
      (* ignore t2 and search another partner *)
      choose_second σ c t l2 (add c2 t2 m) @
      (try (* see whether we can unify t and t2 *)
        let ρ = unify σ t t2 in
        (* extend the unifier to other terms *)
        iter_self ρ (c + c2) t l2 m
      with Fail → empty)
  in
  choose_first σ m.terms {m with terms=[]}
```

unify_ff   takes focused linear expressions $m_1 \sigma$ and $m_2 \sigma$, unifies their focused terms together (if possible) with some $\sigma \le \rho$ and then yields a set of unifiers that extend ρ. Those unifiers are triples (u_1, u_2, θ), where u_1 and u_2 are focused linear expressions and $\rho \le \theta$. The relation between m_i and u_i ($i \in \{1, 2\}$) is: let $m_1 \stackrel{\text{def}}{=} a_1 \cdot t_1 + \sum_j b_{1,j} \cdot t'_{1,j} + \sum_k c_{1,k} \cdot t''_{1,k}$ and $m_2 \stackrel{\text{def}}{=} a_2 \cdot t_2 + \sum_j b_{2,j} \cdot t'_{2,j} + \sum_k c_{2,k} \cdot t''_{2,k}$, with $\forall j.\ t'_{i,j} \theta = t_i \theta$ (first the terms made equal to t_i by θ, and second the remaining terms); then $u_i = (a_i + \sum_j b_{i,j}) \cdot t_i \theta + \sum_k c_{i,k} \cdot t''_{i,k} \theta$. The function `unify_self` is used to split the linear expressions' rests into two parts. This function is mostly used together with term indices (see Section 3.1.3, paragraph Indexing): indexing structures are used to unify two atomic terms from two linear expressions in two distinct clauses, then `unify_self` is used on both linear expressions to extend the unifier to sums of terms.

To find unifiers of the two focused linear expressions, we must first unify their focused terms (or fail), and then extend the unifier to other terms of both `mf1.rest` and `mf2.rest` using `iter_self`.

```
let unify_ff σ mf1 mf2 =
  try
    let ρ1 = unify σ mf1.term mf2.term in
    iter_self ρ1 mf1.coeff mf1.term mf1.rest.terms {mf1.rest with terms=[]}
    ≫= fun (new_mf1, ρ2) →
    iter_self ρ2 mf2.coeff mf2.term mf2.rest.terms {mf2.rest with terms=[]}
    ≫= fun (new_mf2, θ) →
    return (new_mf1, new_mf2, θ)
```

```
      with Fail → empty
```

unify_mm takes linear expressions m_1 and m_2 and tries to find all the $(a_1 \cdot t_1) \in m_1$, $(a_2 \cdot t_2) \in m_2$ and σ such that $t_1\sigma = t_2\sigma$. For any such triple, it then put the focus respectively on t_1 and t_2 and yields the control to `unify_ff`. This is useful for implementing Cancellation or factoring on (in)equations.

To unify two unfocused linear expressions, well, find all the ways to unify one term of each (obtaining focused linear expressions); for each such pair of focused linear expression and partial unifier $\sigma 1$ try to extend the unifier to other terms. This is close to what `unify_ff` does, but also enumerating all possible focusings for the linear expressions. Termination is easily proved by the strict decrease of the multiset $\{l_1, l_2\}$.

```
let unify_mm σ  m1 m2 =
  (* unify a term of l1 with a term of l2. m1 and m2 will be
     the unfocused part *)
  let rec choose_first σ l1 m1 l2 m2 = match l1, l2 with
    | [], _
    | _, [] → ()
    | (c1,t1)::tail1, (c2,t2)::tail2 →
        (* don't choose t1 *)
        choose_first σ tail1 (add c1 t1 m1) l2 m2 @
        (* don't choose t2 *)
        choose_first σ l1 m1 tail2 (add c2 t2 m2) @
        (* choose t1 and t2 if they are unifiable, and extend the unifier *)
        (try
           let ρ = unify σ t1 t2 in
           iter_self ρ c1 t1 tail1 {m1 with terms=[]}
           ≫= fun (mf1, ρ2) →
           iter_self ρ2 c2 t2 tail2 {m2 with terms=[]}
           ≫= fun (mf2, θ) →
           return (mf1, mf2, θ)
         with Fail → empty)
  in
  let m1' = {m1 with terms=[]} in
  let m2' = {m2 with terms=[]} in
  choose_first σ m1.terms m1' m2.terms m2'
```

matching matches two linear expressions $m_1\sigma$ and $m_2\sigma$ by returning substitutions ρ such that $\sigma \le \rho$ and $m_1\rho = m_2\sigma$. An important distinction here is that we match linear expressions entirely, whereas the previous functions would only unify part of a linear expression (the *focused* part) with a part of the other linear expression. The functions terminate respectively because `length l1` and `length l2` decrease at each call.

```
let matching σ m1 m2 =
  let rec start σ l1 l2 = match l1, l2 with
    | [], [] → return σ (* success *)
    | [], _ | _, [] → empty (* failure *)
    | (c1,t1)::tail1, _ → traverse_lists σ (c1,t1) tail1 [] l2
  (* must match all c1 occurrences of t1 with some (c2,t2) ∈ m2 *)
  and traverse_lists σ (c1,t1) tail1 m2 l2 = match l2 with
    | [] → empty (* failure, cannot match t1 *)
    | (c2,t2)::tail2 →
      (if c1 ≤ c2
        then try
          let ρ = match_terms σ  t1 t2 in
          if c1 = c2 (* t2 disappears from matchee *)
            then start ρ tail1 (List.append m2 tail2)
            else (* some instances of t2 remain to be matched *)
              start σ tail1 ((c2 - c1, t2) :: List.append tail2 m2)
```

```
      with Fail → empty
      else empty) @
    traverse_lists σ (c1,t1) tail1 ((c2,t2)::m2) tail2 (* skip t2 *)
  in
  if m1.const = m2.const
    then start σ m1.terms m2.terms
    else empty
```

4.6.4 Other Implementation Notes

We do not explain every detail of the implementation. The sources for Zipperposition-0.5 can be found at `https://github.com/c-cube/zipperposition/archive/0.5.tar.gz`, and the part relevant to arithmetic is in the modules `ArithLit`, `Monome` (the former name of linear expression), and `ArithInt` (in the folders `src` and `src/calculi`).

Literal Ordering   The ordering on literals (Definition 4.7) is quite complicated to decide on first-order terms, especially since some pairs of terms are not comparable. Any superset of the ordering relation preserves soundness. For instance, currently, Zipperposition does not order division literals that live in the same $\mathbb{Z}/n\mathbb{Z}$ (see the module `Literal.Comp`). We think some kind of constraint solving is necessary to compare more accurately division literals, since the number of cases to consider in the non-ground case is high — in particular, first-order literals may have several maximal terms ($>$ not being total).

Inference and Simplification Rules   are all implemented in `calculi/ArithInt`. The binary rules use term indices from Logtk to reduce the number of unification problems to solve.

Normalization Rules   are mostly dealt with directly in the `Literal.t` (and `ArithLit.t`) constructor functions, which are so-called *smart constructors* — functions that build a private datatype and enforce some invariant that will hold by construction. A few of the rules (e.g., Prime Elimination) are full-fledged simplification rules.

The subsumption relation   from Section 4.3.2 is quite subtle to implement and we needed additional n-ary unification functions similar to those in Section 4.6.3. In particular, we need take care of *scaling* literals (multiplying a literal with a constant to adjust the coefficient of some of its terms), depending on their shape. The brave reader can take a look at the module `ArithLit.Subsumption` in Zipperposition[15].

4.6.5 Graphical Output for Debugging

Figure 4.8 shows the graphical output of the theorem prover on a small geography problem (GEG022=1.p) that axiomatizes an Euclidian distance d (with $d(x, y) \simeq d(y, x)$, $d(x, x) \simeq 0$, and the triangular inequality $d(x, z) \leq d(x, y) + d(y, z)$), lists the distances[16] between a few German cities, and requires to prove the goal $d(\text{hamburg}, \text{munich}) \leq 700$ by effectively computing a short path between the two cities. The proof was edited to make it more readable, by abstracting the (large) negated goal into the bottom yellow box (the red box on top is the empty clause). The edges connecting clauses are labelled with the inference rule used (where, for instance, `canc_demod` stands for Cancellative Demodulation, etc.) Such a graphical display of proofs as DAGs was truly invaluable in our work, facilitating the understanding of proof traces — very important when debugging soundness issues or bugs in the implementation of rules.

---

[15] The module `ArithLit` is in `src/arithLit.ml` and `src/arithLit.mli`.

[16] The unit is not specified in the original problem, but let us assume distances are expressed in kilometers rather than parsecs, for the sake of tourism in Germany.
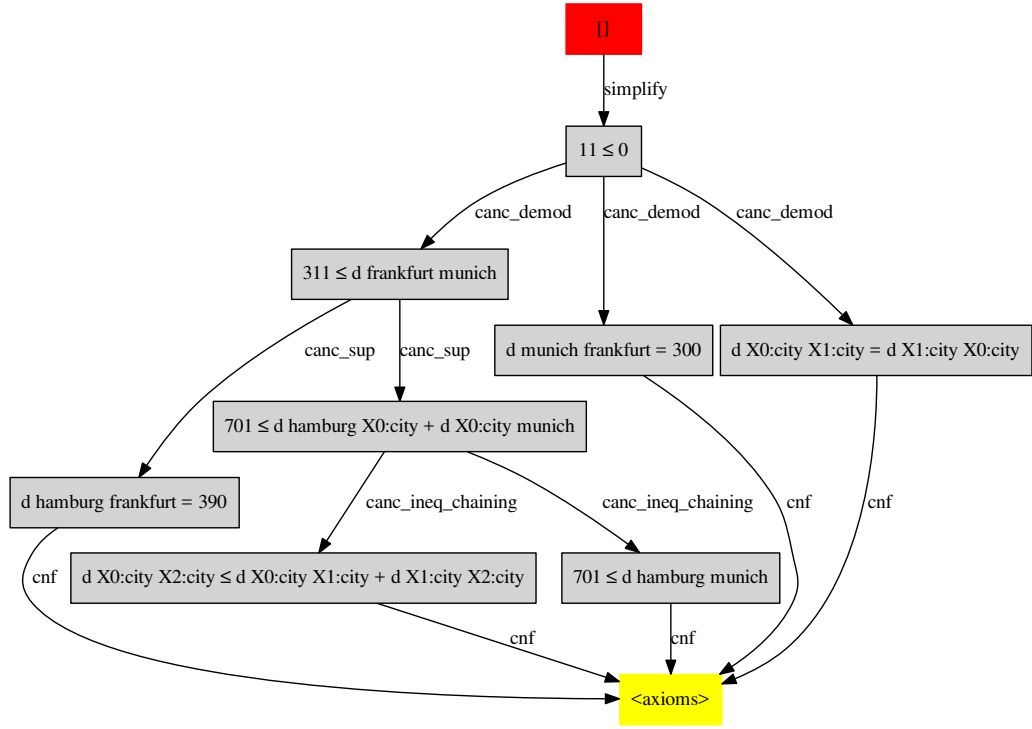
Figure 4.8: Solution for `GEG022=1.p`

4.7 Experimental Evaluation

Zipperposition 0.4 entered the TFA (arithmetic) division at CASC-J7 [SS06][17] and came close second after Princess [Rüm08] on integer problems In Figure 4.9, we show the results in the TFA division on integer arithmetic problems[18]. The columns respectively gather the total number of problems solved (all of them are theorems), the average time needed to solve a problem, the *efficiency measure* (balances the number of problems solved with the time taken[19]), the state of the art contribution ("SOTAC", sum of inverse of number of provers solving each problem, quantifying how much a prover can solve problems that are hard for other provers), and last the number of problems solved among the 50 new problems introduced in CASC-J7.

prover	solved/100	avg time (s)	μ-efficiency	SOTAC	new/50
Princess	81	20.3	291	0.22	35
Zip	80	6.5	626	0.27	44
CVC4	80	10	605	0.24	33
SPASS+T	75	6.8	314	0.18	30
Beagle	73	12.7	325	0.18	28

Figure 4.9: Results of CASC-J7

---

[17] http://www.cs.miami.edu/~tptp/CASC/J7/

[18] Solver versions: (i) Princess 140704 (ii) Zipperposition 0.4-TFF (iii) CVC4 1.4-TFA (iv) SPASS+T 2.2.20 (v) Beagle 0.9

[19] See http://cs.miami.edu/~tptp/CASC/J7/Design.html.

We also ran benchmarks on two subsets of integer arithmetic problems from TPTP-6.1 (filtering out problems containing some rational or real arithmetic). We compare Zipperposition to Princess (release 2013–09–06) and Beagle 0.9, with 300s of timeout and 1GB of memory on a 2.20GHz Intel Xeon. The whole set of problems is listed in the file `bench_arith/int_problems` in the archive downloadable at `https://who.rocq.inria.fr/Simon.Cruanes/files/bench_arith.tar.gz`. We split the results into several bags of TPTP-categories, that we describe below, and comment on:

- ARI,NUM,GEG,PUZ,SEV,SYN,SYO: basic arithmetic problems, and various arithmetic problems appearing in small quantities in other categories, of relatively low difficulty. All three provers perform very well on this category.
- DAT: data structures, on which the Superposition-based provers (here, Beagle and Zipperposition) perform better than the tableaux-based Princess.
- HWV: hardware verification, a set of large ground problems which are probably better suited to SMT solvers.
- SWV,SWW: software verification, quite large proof obligations on which Princess shines. We conjecture that this is partly linked to the fact tableaux provers do not have to reduce their input to CNF, and can ignore irrelevant axioms better.

Benchmarks from ARI,NUM,GEG,PUZ,SEV,SYN,SYO					
prover	unsat (/263)	%solved	unique	time (s)	avg time (s)
beagle	254	97	6	321	1.27
princess	251	95	0	229	0.91
zip	247	94	0	53	0.22

Benchmarks from DAT					
prover	unsat (/87)	%solved	unique	time (s)	avg time (s)
beagle	75	86	5	223	2.98
princess	60	69	1	326	5.44
zip	74	85	5	85	2.03

Benchmarks from HWV					
prover	unsat (/68)	%solved	unique	time (s)	avg time (s)
beagle	0	0	0	-	-
princess	0	0	0	-	-
zip	0	0	0	-	-

Benchmarks from SWV,SWW					
prover	unsat (/179)	%solved	unique	time (s)	avg time (s)
beagle	81	45	0	1432	17.6
princess	178	99	56	917	05.1
zip	52	29	0	1599	30.7

Figure 4.10: Benchmarks on TPTP problems

We see that both in the CASC competition, and in the benchmarks on TPTP-6.1, the prototype performs quite well. It solves a reasonable amount of problems and answers more quickly — we must note, however, that both Princess and Beagle run on the JVM which starts slowly, around 0.5s on the benchmark machine. Overall, our proof-of-concept implementation performs quite well, and solves some problems that the two other provers do not solve. Among those, for instance, we find `DAT/DAT086=1.p`: a problem on lists (DAT is about data struc-

tures) mixing symbols from the theory of lists (inRange, length, count, append) and arithmetic in a non-trivial way, since lists contain integers. Many interesting problems can be formulated in a way that mixes arithmetic and uninterpreted symbols; we saw earlier the example of GEG022=1.p, with triangular inequality on distances between cities. In formal methods — especially in the lack of verifiable certificates —, it is better to use several solvers on one single problem, for two reasons: (i) if the problem is difficult, there is more chance at least one solver will be good enough to solve it (especially when the solvers are based on distinct techniques); (ii) the probability of all solvers that answer to have a bug that triggers on a given problem is low. The benchmarks on TPTP above show indeed the value of having three solvers based on complementary techniques.

Conclusion

We presented our purely deductive system of inference that extends superposition to deal with integer linear arithmetic, along with an implementation and some experimental results that show it already behaves quite well in practice. The calculus uses the usual notion of redundancy to un-clutter its search space. We believe it is especially suitable for problems that tightly mix first-order reasoning and arithmetic, for instance when axioms involving arithmetic are involved — such as the triangular inequality on Euclidian distances or monotonicity properties.

Compared to the state of the art, our approach builds on Superposition's saturation process by adding deduction rules and powerful redundancy criteria. We extend ordered chaining and provide a variable elimination technique, so the prover actually reasons on arithmetic at the first-order level (rather than on a set of ground constraints as in, say, Hierarchic Superposition [BGW94, BW13]). On the other hand, since we propose several sets of sound, mostly independent rules, it is possible to cherry-pick some of them and include them in blackbox approaches to prune redundant clauses or deal with additional arithmetic axioms (monotonicity of a function, Euclidian distance, etc.) at the first-order level. It also builds upon AVATAR (section 2.5) to deal with ground case splits, in particular those introduced to eliminate divisibility constraints.

We tried hard to tightly interleave superposition with arithmetic reasoning, at least for the linear integer arithmetic fragment. A proof of concept implementation in Zipperposition shows that the approach is viable. This specific treatment of arithmetic was motivated by the inherent difficulty of this theory — in particular, the variable elimination procedure embeds non-trivial knowledge about integer arithmetic. It is no question that arithmetic is very useful in a large number of problems (from the industry, or other domains of formal verification such as refinement types, proof of programs, and so on). Still, many other theories are useful and deserve special treatment. In the next chapter we propose a calculus for adding structural induction to superposition; many theories fall within the range of induction, especially when one is concerned with data structures such as lists or trees. Induction can also be used to reason on natural numbers (and from then, on some encoding of integer numbers), but we will see that the current chapter is still relevant as proving inductively as simple a lemma as addition being commutative is not trivial.

Chapter 5

Structural Induction

To prove universal properties, a very common reasoning method in Mathematics is *proof by induction*. Its programming counterpart, recursion, is so important that it is the only way of iterating and looping in some languages such as Scheme. Induction's strength is its ability to use *local* reasoning — proving one step entails the next one — to prove *global* properties, that range over an infinite number of elements. Of course, the first and foremost form of inductive reasoning is *proof by recurrence*, that is, induction on natural numbers; *structural induction* is widely used in Computer Science (for instance in Coq [HKPM97]); and the more general *Noetherian induction* is a strong tool. Supporting some form of inductive reasoning in automated theorem provers has been a longstanding effort (see for instance [KB95] for a series of inductive provers dating back to the seventies), yet the gap between first-order theorem provers and provers specialized to handle induction[1] is still wide. Superposition is a very successful paradigm for automated reasoning in first-order logic, yet many problems require inductive reasoning (e.g., verifying programs that deal with lists, natural numbers; proof obligations from interactive provers such as Coq [HKPM97], etc.). Without the insight of a human, explicitly instantiating inductive schemata is doomed to fail; many techniques (e.g., [Com94]) and provers (e.g., [BKR92, Str12]) have been dedicated to mitigating this issue. On the other hand, Superposition provers such as E [Sch02] can reason over arbitrary formulas, with large equational theories, and it seems desirable to carry their capabilities into an inductive prover. First steps in this direction have been made in [KP13], but with the restriction that only induction on natural numbers is possible. We will show here how the recent technique of AVATAR (see Section 2.5) helps narrowing this gap by making our Superposition prover deal with structural induction on inductive types such as lists, natural numbers, binary trees, etc. The limitations of our approach are its inability to perform nested induction without introducing a cut (a lemma), the inability to perform induction on mutually recursive types (e.g., a tree with a list of sub-trees), and, as in many other techniques, the heuristic nature of the mechanism that introduces lemmas (Section 5.3.1).

As often within automated theorem proving, we will not focus on the direct form of induction, but rather on another formulation of the same deep concept, the existence of a minimal (Herbrand) model. If a property $P : \tau \to o$ satisfies $\neg P(t)$ for some term t of an inductive type, then there must be some term u of the same type such that $\neg P(u)$ and $\forall v.\ v \lhd t \Rightarrow P(v)$ where $\lhd$ is the subterm ordering — that is, u is a minimal counter-example to P.

Therefore, the existence of a minimal counter-example for every property P such that $F \vdash \exists t.\ \neg P(t)$ is a necessary condition for a formula F to have an inductive model. We will express the existence of a minimal counter-example for non-universal properties expressed as sets of *clause contexts*, and encode the criterion for the existence of the counter-example into a boolean formula within the AVATAR framework to make it decidable. A first version of the criterion uses a SAT formula, then a stronger version using QBF (Quantified Boolean Formula, see Def-

---

[1] Induction can be thought of as a schema of axioms for first-order logic, but as far as automated theorem proving is concerned using such an axiomatization makes managing the search space intractable.

inition 2.22) is detailed. Obviously there is no hope for a model nor a refutation to be found in general when induction is involved[2], but our decidable criterion can, in some cases, detect unsatisfiability of the initial formula F.

This chapter starts with some notations and definition, including a proper definition of what we mean by *structural induction*. Then, a semantics of inductive types is developed, we define what an inductive model and a minimal inductive model are. Some inference rules that deal with inductive constructors are presented, before we present the main contributions. Section 5.2 and Section 5.4 present two techniques for encoding the existence of a minimal counter-example to properties that are known not to hold on at least one term of an inductive type. The second technique is an extension of the first one, and it can deal with a wider range of properties. After that come some considerations about proof traces and proof certificates, followed by a presentation of our proof-of-concept implementation in Zipperposition.

5.1 Inductive Types and Models

5.1.1 Notations and Definitions

Definition 5.1 (Inductive Type). *An* inductive type τ *is a type for which a fixed set of symbols* $cstors(\tau) \subseteq \Sigma$ *exists, with* $cstors(\tau) \neq \emptyset$, *and the following axioms hold (in any inductive model, as will be defined in Definition 5.9):*

Well-Typedness $\forall c \in cstors(\tau).\ c : \Pi \alpha_1, \ldots, \alpha_m.\ (\tau_1 \times \ldots \times \tau_n) \rightarrow \tau$

Non-Overlap $\forall t_1 \ldots t_n.\ \forall t'_1 \ldots t'_m.\ c_1(t_1, \ldots, t_n) \neq c_2(t'_1, \ldots, t'_m)$ *where c_1 and c_2 are distinct constructors of the same inductive type with respective arity n and m;*

Injectivity $\forall t_1 \ldots t_n.\ \forall t'_1 \ldots t'_n.\ c(t_1, \ldots, t_n) \simeq c(t'_1, \ldots, t'_n) \Rightarrow \bigwedge_{i=1}^{n} t_i \simeq t'_i;$

Surjectivity $\forall x : \tau.\ \bigvee_{c \in cstors(\tau)} \exists t_1 \ldots t_n \in Terms(cstors(\tau)).\ x \simeq c(t_1, \ldots, t_n)$ *where each t_i is built from constructors only;*

In the following, Σ_{ind} will denote the signature composed of all inductive constructors for all inductive types. We call inductive values *the terms that are built exclusively from inductive constructors and symbols that do not have an inductive return type. We speak of* structural induction *because the induction principle is based on $\lhd$, sometimes called structural ordering.*

Because $\lhd$ is well-founded, the following family of axioms parametrized by formulas $P : \tau \rightarrow o$, called *Induction Scheme*, always holds:

$$\big(\forall t : \tau.\ \big(\forall t' : \tau.\ t' \lhd t \Rightarrow P(t')\big) \Rightarrow P(t)\big) \Rightarrow \forall t : \tau.\ P(t)$$

Definition 5.2 (Inductive Constant). *An* inductive constant *is a symbol of arity 0 that has an inductive type but is not a constructor (for instance, a Skolem symbol).*

In the following, we will denote inductive constants by $\mathfrak{i}$, or $\mathfrak{n}$, $\mathfrak{l}$, $\mathfrak{t}$, etc. depending on their type — any type for the former, nat or list for the latter. $\mathfrak{I}$ will be a set of inductive constants. We will use the *A-clauses*, or *clauses with assertions* of Definition 2.56. A possible alternative would be to use *labelled clauses* [LAWRS07].

Definition 5.3 (Clause context). *We consider a family of constants $(\diamond_\tau)_{\tau : Type}$ indexed by their type τ (exactly one constant per type). A* clause context $C[\diamond_\tau]$ *is a clause that contains one or more occurrences of $\diamond_\tau$, and $C[t]$ is the clause obtained by replacing simultaneously all occurrences of $\diamond_\tau$ in $C[\diamond]$ with the term $t : \tau$.*

Definition 5.4 (Type of a Clause context). *The type of a clause context $C[\diamond_\tau]$ is the type τ. Applying the context $C[\diamond_\tau]$ to some term t requires that $t : \tau$.*

---

[2] unlike with regular first-order logic, where a refutation will be found for any non-theorem.

In this chapter, clause contexts will have the same naming conventions as clauses, but they will always have an explicit argument. For instance, C is a clause (or more generally an A-clause), and $C[\diamond_\tau]$ is a clause context. We will generally omit the type of the context hole and write $\diamond$ instead of $\diamond_\tau$ where the type can be easily inferred by the reader.

Definition 5.5 (Coverset). *A coverset S for an inductive type τ is a set of terms composed of inductive constructors and variables $x_1, \ldots, x_n$ such that each variable x_i occurs in one position exactly, and such that $\forall t : \tau.\ \bigoplus_{u \in S} \exists x_1 \ldots x_n.\ t \simeq u$ holds in any model satisfying the axioms of the inductive type. It follows that the terms of a coverset are distinct in any such model. Coversets were first defined in [ZKK88].*

Definition 5.6 (Ground Coverset). *A ground coverset $\kappa(i)$ is a set of ground terms obtained by replacing all variables in a coverset with fresh Skolem constants (not present in the signature), such that $\bigoplus_{t \in \kappa(i)} i \simeq t$ holds in any model of the new, extended signature. The elements of $\kappa(i)$ represent all the possible "shapes" of i in any model. If $t, i : \tau$ and there is some $t' \in \kappa(i)$ such that $t \vartriangleleft t'$, we write $\mathrm{sub}(t, i)$. We define $\kappa_{\downarrow}(i) = \{t \in \kappa(i) \mid \exists t' \vartriangleleft t.\ \mathrm{sub}(t', i)\}$ (recursive cases), and $\kappa_{\perp}(i) = \kappa(i) \setminus \kappa_{\downarrow}(i)$ (base cases). Note that introducing the Skolem symbols only preserves satisfiability.*

Example 5.1 (Natural Numbers). *The type of natural numbers, nat, is a classic inductive type whose constructors are $\mathrm{cstors}(\mathrm{nat}) = \{0, s\}$. Its inductive values are all the natural numbers $\{0, s(0), \ldots, s^k(0), \ldots\}$, and ground coversets are of the form $\{0, s(0), \ldots, s^k(0), s^{k+1}(\mathfrak{n})\}$ for some $k \geq 0$ and Skolem constant $\mathfrak{n}$.*

We use clause contexts to isolate the inductive term from the clauses that contain it. For instance at the beginning we might have a Skolem symbol $\mathfrak{n}$ that occurs in two clauses, noted $C[\mathfrak{n}]$ and $D[\mathfrak{n}]$ (with $\mathfrak{n}$ not occurring in neither $C[\diamond]$ nor $D[\diamond]$). If we assert $\mathfrak{n} \simeq 0 \vee \mathfrak{n} \simeq s(\mathfrak{n}')$ (with $\mathfrak{n}'$ a new constant) then the contexts $C[0]$, $C[s(\mathfrak{n}')]$, $D[0]$ and $D[s(\mathfrak{n}')]$ will become relevant for refuting $C[\mathfrak{n}]$ and $D[\mathfrak{n}]$. Here, $\kappa(\mathfrak{n}) = \{0, s(\mathfrak{n}')\}$, $\mathrm{sub}(\mathfrak{n}', \mathfrak{n})$ holds, $\kappa_{\perp}(\mathfrak{n}) = \{0\}$, and $\kappa_{\downarrow}(\mathfrak{n}) = \{s(\mathfrak{n}')\}$.

Remark 5.1 (Peano Axioms). *Many examples will use natural numbers (type nat) to illustrate the ideas in a simple way, even though they apply to other inductive types. The following Peano axioms for addition will be used without mention: (a) $\forall n.\ 0 + n \simeq n$ (b) $\forall m\ n.\ s(m) + n \simeq s(m+n)$.*

5.1.2 Restrictions on the Term Ordering

We will also need some restrictions on the term ordering $\succ$ in the following sections.

Definition 5.7 (Admissible Ordering). *A simplification ordering on terms $\succ$ is* admissible *for induction over a given signature Σ if it satisfies the following properties:*
- *$i \succ t$ for any $t \in \kappa(i)$;*
- *$t \succ t'$ if t' is a ground pure inductive term of $\Sigma_{ind} \subseteq \Sigma$ and t is a ground impure term of the same type (i.e., $t : \tau$, $t' : \tau$, $t' \in Terms(\Sigma_{ind})$ and $t \in Terms(\Sigma) \setminus Terms(\Sigma_{ind})$).*

The purpose of those restrictions is to make literals of the form $i \simeq t$ with $t \in \kappa(i)$ into left-to-right rewrite rules, and to ensure that pure inductive terms are normal forms. Note that $\vartriangleleft$ is always included in $\prec$ for simplification orderings.

Definition 5.8 (Admissible RPO). *A RPO on terms $\succ$ is* admissible *for induction over a given signature Σ if the precedence the ordering is built on satisfies:*
- *constructor symbols are smaller than other function symbols;*
- *any inductive constant i is higher in the precedence than any Skolem constant t such that $t \vartriangleleft t'$ for some $t' \in \kappa(i)$.*

Lemma 5.1 (Admissible RPO are Admissible). *Any admissible RPO is an admissible ordering.*

Proof. The first condition ensures that impure terms, which contain at least one function symbol, are bigger than pure terms built exclusively from symbols in Σ_{ind}. Together with the second condition, any $t \in \kappa(\mathfrak{i})$ — built from inductive constructors and Skolem constants — is smaller than $\mathfrak{i}$ because t is ground and all its symbols are smaller in the precedence than $\mathfrak{i}$. $\qquad\square$

Example 5.2 (Admissible RPOs on Natural numbers). *Given the usual signature $\Sigma = \{\mathfrak{n}, \mathfrak{n}', s, 0, +, \times\}$ with the ground coverset $\kappa(\mathfrak{n}) = \{0, s(\mathfrak{n}')\}$, the RPO over the precedence $\times > + > \mathfrak{n} > \mathfrak{n}' > s > 0$ is admissible, and so is the RPO over $\mathfrak{n} > + > \mathfrak{n}' > \times > 0 > s$. Those orderings make $\mathfrak{n} \simeq s(\mathfrak{n}')$ into a rewrite rule for $\mathfrak{n}$, and ensure (together with the appropriate axioms) that $s(s(0)) + s(s(0))$ rewrites into $s(s(s(s(0))))$ rather than the opposite.*

In the rest of this chapter, we will assume that $\succ$ is an admissible ordering on terms. Our implementation uses an admissible LPO (all symbols have lexicographic status).

5.1.3 Dealing with Constructors

Inductive constructors have some properties that are best handled with dedicated inference rules that will be useful throughout the rest of this chapter. In those rules, presented in Figure 5.1, c and c' are distinct inductive constructors (the empty list [], the successor symbol, etc.)

Injectivity (Inj)
$$\frac{c(t_1, \ldots, t_n) \simeq c(t'_1, \ldots, t'_n) \vee D}{\bigwedge_{i=1}^n \left(t_i \simeq t'_i \vee D \right)}$$

Non-Overlap (NOv)
$$\frac{c(t_1, \ldots, t_n) \simeq c'(t'_1, \ldots, t'_m) \vee D}{D} \quad \text{and} \quad \frac{c(t_1, \ldots, t_n) \not\simeq c'(t'_1, \ldots, t'_m) \vee D}{\top}$$

if c and c' are distinct inductive constructors

Figure 5.1: Inference Rules to deal with Inductive Constructors

Lemma 5.2 (Soundness). *The rules from Figure 5.1 are sound w.r.t. the definition of inductive types (Definition 5.1), and they are compatible with any simplification ordering.*

5.1.4 Semantics and Minimal Models

Usually, saturation-based theorem proving is concerned with finding a model — or a sufficient criterion for the existence of a model, because we are primarily interested in the satisfiability (or unsatisfiability) of a formula — for the set of input clauses. However, in presence of inductive types, it is impossible in general to find any sufficient criterion for the existence of a standard inductive model[3]. We will instead strive to express *necessary* conditions for such a model to exist. Perhaps those conditions will be satisfied even for formulas that have no model; however, we have no choice but make a parallel to the famous quote[4] from E. Dijkstra:

> *program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.* — E. Dijkstra

---

[3] We focus on the existence or non-existence of *standard* models, that is, models in which all elements of the domain of an inductive type are built from the corresponding inductive constructors (e.g., the standard model of Peano axioms is $\mathbb{N}$)

[4] See https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html.

In a similar vein, our criteria will be able to sometimes show the absence of inductive models, but never to show their presence. More precisely, building upon a regular saturation process, we manipulate a model candidate; necessary conditions for the existence of inductive models will come in the form of additional side-conditions that express the possibility for the model candidate to be *minimal* (w.r.t. an extension of the structural ordering on terms to models, see Definition 5.10). In a further refinement, we will assert the stronger condition that every subset of the set of inductive clauses *can* have a minimal model — corresponding to choosing the (negation of the) subset as the induction hypothesis. This approach is connected to the work of A. Kersani and N. Peltier [KP13].

Definition 5.9 (Inductive Model). *An* inductive model *of a combined state* $(\mathcal{N}, \mathcal{F}_b)$ *w.r.t. a set of inductive constants* $\mathfrak{I}$ *is a combined Herbrand model* $(\mathcal{M}, v)$ *(see Section 2.3.4) that satisfies all the axioms of inductive types (Definition 5.1) and such that all inductive constants are mapped to* inductive values *(built exclusively from inductive constructors and symbols of non-inductive type).*

Example 5.3 (Inductive Model for nat). *In the case of the natural numbers* nat *equipped with the constructors* $0 : \mathsf{nat}$ *and* $s : \mathsf{nat} \to \mathsf{nat}$, *an inductive model is one that maps any term of type* nat *to some term* $s^k(0)$ *with* $k \in \mathbb{N}$ *— in other words, the standard model of arithmetic.*

We will only consider inductive models from now on.

Definition 5.10 (Minimal Inductive Model). *An inductive model* $(\mathcal{M}, v)$ *of a set* $\mathcal{N}$ *of A-clauses is* minimal *w.r.t. an inductive constant* $\mathfrak{i}$ *iff no other model* $(\mathcal{M}', v)$ *of* $\mathcal{N}$ *verifies* $[\![\mathfrak{i}]\!]^{\mathcal{M}'} \lhd [\![\mathfrak{i}]\!]^{\mathcal{M}}$.

Lemma 5.3 (Existence of a Minimal Model). *Any satisfiable set* $\mathcal{N}$ *admits a minimal model w.r.t.* $\mathfrak{i}$.

Proof. $\mathcal{N}$ has a model $S_0 \overset{\text{def}}{=} (\mathcal{M}, v)$. For any $n \in \mathbb{N}$, if S_n isn't minimal, then by definition there exists $S_{n+1} \overset{\text{def}}{=} (\mathcal{M}', v)$ with $[\![\mathfrak{i}]\!]^{S_{n+1}} \lhd [\![\mathfrak{i}]\!]^{S_n}$. Since $\lhd$ is well-founded, this sequence must be finite, and its last element is a minimal model of $\mathcal{N}$. □

Lemma 5.4. *Let* $(\mathcal{M}, v)$ *a combined model and* $(\mathcal{N}, \mathcal{F}_b)$ *a state it satisfies. Then, for any set of clauses* $\mathcal{N}' \subseteq \mathcal{N}$ *and inductive constant* $\mathfrak{i} \in \Sigma_{ind}$, *there exists a model* $(\mathcal{M}', v)$ *minimal w.r.t.* $\mathfrak{i}$ *such that* $(\mathcal{M}', v) \models (\mathcal{N}', \mathcal{F}_b)$.

Proof. Directly from Lemma 5.3 since $(\mathcal{M}, v)$ is also a model of $\mathcal{N}'$. □

Remark 5.2. *Those definitions could be generalized to any well-founded ordering, as in Noetherian induction, but we kept* $\lhd$ *for the sake of simplicity.*

Now that we have defined what inductive models and minimal inductive models are, we can start wondering about their existence in a given theory. We leave to the platonic reader any reflection about the pre-existence of the notion of inductive models themselves.

5.2 Inductive Strengthening

We now have all the tools required to extend AVATAR to (structural) inductive reasoning. The first approach only considers performing induction on formulas that are already present in the problem. This is similar to techniques used in many provers, for instance in CVC4 [RK15][5]. Putting our Superposition lens on, the induction is performed on one clause context $C[\diamond]$ such that $C[t]$ is already present in the problem for some t; $C[\diamond]$ will be the proposition for which the existence of a minimal model will be questioned. We will see in the next section that there are cases where this is not enough (for instance, Example 5.13).

---

[5] The CVC4 SMT solver, http://cvc4.cs.nyu.edu/web/

To prove a conjecture $\forall x.\ F[x]$ in the theory G (a set of formulas), we usually reduce $G \wedge \neg \forall x.\ F[x]$ to CNF, introducing a Skolem constant $\mathfrak{i}$ standing for the counter-example to $\forall x.\ F[x]$, and proceed to deduce $\bot$ from $\mathrm{cnf}(G \wedge \neg F[\mathfrak{i}])$. When x has an inductive type τ, this is not enough, as Example 5.4 shows.

Example 5.4 (0 neutral on right of +)**.** *Given the usual Peano axioms (without induction since it is an infinite schemata of axioms), definition of +, and the inference rules of Figure 5.1, let us try to prove* $\forall x.\ x + 0 \simeq x$*. Superposition starts from the negation of the goal,* $\mathfrak{n} + 0 \not\simeq \mathfrak{n}$ *and the coverset* $\kappa(\mathfrak{n}) = \{0, s(\mathfrak{n}')\}$ *— where* $\mathfrak{n}$ *and* $\mathfrak{n}'$ *are fresh Skolem constants. By case split* (ASplit) *on* $\mathfrak{n} \simeq 0 \vee \mathfrak{n} \simeq s(\mathfrak{n}')$*, we add the A-clauses* $\mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert$ *and* $\mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert$*. We show the derivation for the recursive case:*

$$\frac{\dfrac{\mathfrak{n} + 0 \not\simeq \mathfrak{n} \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{s(\mathfrak{n}') + 0 \not\simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}\ (\mathsf{Sup}) \qquad s(x) + y \simeq s(x + y)}{s(\mathfrak{n}' + 0) \not\simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}\ (\mathsf{Sup})$$

The last step of the derivation is at least as hard to solve as the first step (namely $\mathfrak{n} + 0 \not\simeq \mathfrak{n}$*). We could repeat the very same derivation any number of times without making any progress towards* $\bot$*. Unsurprisingly, induction is needed here.*

We clearly need a way to avoid this infinite derivation tree. We know that if there is a counter-example, then there must be a minimal counter-example (Lemma 5.3), and we can reason on a smallest counter-example while preserving equi-satisfiability. The whole idea of inductive strengthening, as used in other provers such as CVC4 [RK15][6], is to assert that $\mathfrak{i}$ is a minimal counter-example.

Let, again, $\forall x : \tau.\ F[x]$ be an inductive formula we want to prove by induction on x, $\mathfrak{i}$ be the related Skolem constant for which the model of $\neg F[\mathfrak{i}]$ will be minimal, and $\kappa(\mathfrak{i})$ a coverset. We *assert* the following sets of A-clauses described in Definition 5.11 as a *necessary condition* for the existence of a minimal model of $\neg F[\mathfrak{i}]$ w.r.t. $\mathfrak{i}$.

Definition 5.11 (Minimal Strenghtening Set)**.** *The* minimal strengthening set *of a formula* $F[\diamond]$ *is the union of the following sets of A-clauses:*
- $\mathrm{cnf}(\neg F[\mathfrak{i}])$
- $\{D \leftarrow \lVert \mathfrak{i} \simeq t \rVert \mid D \in \mathrm{cnf}(F[t'])\}$ *for each* $t \in \kappa(\mathfrak{i})$ *with* $t' : \tau \lhd t$ *and* $\mathrm{sub}(t', \mathfrak{i})$*.*
- $\{\mathfrak{i} \simeq t \leftarrow \lVert \mathfrak{i} \simeq t \rVert \mid t \in \kappa(\mathfrak{i})\}$

Example 5.5 (0 neutral on right of + (continued))**.** *In the case of Example 5.4, with* $F[x] \overset{\text{def}}{=} x + 0 \simeq x$*, we again introduce the Skolem constant* $\mathfrak{n}$ *and coverset* $\kappa(\mathfrak{n}) = \{0, s(\mathfrak{n}')\}$*, but this time, we add the A-clause* $\mathfrak{n}' + 0 \simeq \mathfrak{n}' \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert$ *to the goal* $\mathfrak{n} + 0 \not\simeq \mathfrak{n}$ *and the split* $\mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert$ *and* $\mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert$*. The base case is easy:*

$$\frac{\dfrac{\dfrac{\dfrac{\mathfrak{n} + 0 \not\simeq \mathfrak{n} \qquad \mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}{0 + 0 \not\simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}\ (\mathsf{Sup}) \qquad 0 + x \simeq x}{0 \not\simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}\ (\mathsf{Sup})}{\bot \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}\ (\mathsf{EqRes})}{\neg \lVert \mathfrak{n} \simeq 0 \rVert}\ (\mathsf{A}\bot)$$

and now the recursive case succeeds:

---

[6]The CVC4 SMT solver, http://cvc4.cs.nyu.edu/web/

$$\frac{\begin{array}{cc}\mathfrak{n}+0\not\simeq\mathfrak{n} & \mathfrak{n}\simeq s(\mathfrak{n}')\leftarrow\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert\end{array}}{\begin{array}{cc}\dfrac{s(\mathfrak{n}')+0\not\simeq s(\mathfrak{n}')\leftarrow\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert & s(x)+y\simeq s(x+y)}{\dfrac{s(\mathfrak{n}'+0)\not\simeq s(\mathfrak{n}')\leftarrow\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert & \mathfrak{n}'+0\simeq\mathfrak{n}'\leftarrow\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert}{\dfrac{s(\mathfrak{n}')\not\simeq s(\mathfrak{n}')\leftarrow\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert}{\dfrac{\bot\leftarrow\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert}{\neg\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert}}}\end{array}}$$

(Sup), (Sup), (Sup), (EqRes), (A⊥)

Both cases are closed in a finite number of steps, adding constraints $\neg\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert$ *and* $\neg\lVert\mathfrak{n}\simeq 0\rVert$ *to the split constraint* $\lVert\mathfrak{n}\simeq s(\mathfrak{n}')\rVert\oplus\lVert\mathfrak{n}\simeq 0\rVert$. *The result is clear: no minimal model can exist, so the goal's negation is not satisfiable. The proof attempt succeeds.*

Remark 5.3. *To prove a formula* $\forall x_1:\tau_1\ \ldots\ x_n:\tau_n.\ F[x_1,\ldots,x_n]$ *we use the same technique, but with a different CNF for each counter-example; in other words, sharing Skolem symbols or Skolem predicates (standing for intermediate formulas, as described in [NW01]) between the strengthening sets of distinct inductive constants is forbidden. Indeed, using the same Skolemized formula* $\neg F[i_1,\ldots,i_n]$ *for each induction attempt is wrong because it asserts the existence of a model that is minimal for every* x_i *simultaneously — something not necessary true.*

A stronger version of splitting rule from Figure 2.4 is used, to reason by case on $\kappa(i)$, by choosing a coverset and adding clauses $i\simeq t\leftarrow\lVert i\simeq t\rVert$ for each $t\in\kappa(i)$, and adding the boolean constraint $\bigoplus_{t\in\kappa(i)}\lVert i\simeq t\rVert$ to $S_{\text{constraints}}$. A useful optimization this $\oplus$ affords is deleting clauses of the form $C\leftarrow\lVert i=t_1\rVert\sqcap\lVert i=t_2\rVert\sqcap\Gamma$ where $t_1\neq t_2$ are distinct cases of $\kappa(i)$; such clauses are trivial, consuming memory for nothing, as their trail will never be satisfied.

Example 5.6 (+ right-commutes with $s(\cdot)$). *To prove* $\forall m\ n.\ m+s(n)\simeq s(m+n)$ *from Peano axioms, we have two choices: induction on* m *or induction on* n. *Let's describe the induction on* m *(the successful one; the case for* n *starts the same way, with different constants, but fails, which we believe makes it less interesting). We introduce new Skolem constants* $\mathfrak{n}_1$ *and* $\mathfrak{n}_2$, *a coverset* $\kappa(\mathfrak{n}_1)=\{0,s(\mathfrak{n}'_1)\}$ *(where* $\mathfrak{n}'_1$ *is another fresh constant), the clause context* $C[\diamond]\overset{def}{=}\diamond+s(\mathfrak{n}_2)\not\simeq s(\diamond+\mathfrak{n}_2)$, *and assert that* $\mathfrak{n}_1$ *is the minimal witness for* $C[\diamond]$ *with the clauses* $\{\mathfrak{n}_1+s(\mathfrak{n}_2)\not\simeq s(\mathfrak{n}_1+\mathfrak{n}_2),\mathfrak{n}'_1+s(\mathfrak{n}_2)\simeq s(\mathfrak{n}'_1+\mathfrak{n}_2)\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert,\mathfrak{n}_1\simeq 0\leftarrow\lVert\mathfrak{n}_1\simeq 0\rVert,\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert\}$ *and the boolean constraint* $\lVert\mathfrak{n}_1\simeq 0\rVert\oplus\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert$.

$$\frac{\begin{array}{cc}\mathfrak{n}_1+s(\mathfrak{n}_2)\not\simeq s(\mathfrak{n}_1+\mathfrak{n}_2) & \mathfrak{n}_1\simeq 0\leftarrow\lVert\mathfrak{n}_1\simeq 0\rVert\end{array}}{\dfrac{0+s(\mathfrak{n}_2)\not\simeq s(0+\mathfrak{n}_2)\leftarrow\lVert\mathfrak{n}_1\simeq 0\rVert\qquad 0+x\simeq x}{\dfrac{s(\mathfrak{n}_2)\not\simeq s(\mathfrak{n}_2)\leftarrow\lVert\mathfrak{n}_1\simeq 0\rVert}{\dfrac{\bot\leftarrow\lVert\mathfrak{n}_1\simeq 0\rVert}{\neg\lVert\mathfrak{n}_1\simeq 0\rVert}}}}$$

(Sup), (Sup), (EqRes), (A⊥)

$$\frac{\begin{array}{cc}\mathfrak{n}_1+s(\mathfrak{n}_2)\not\simeq s(\mathfrak{n}_1+\mathfrak{n}_2) & \mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert\end{array}}{\dfrac{s(\mathfrak{n}'_1)+s(\mathfrak{n}_2)\not\simeq s(s(\mathfrak{n}'_1)+\mathfrak{n}_2)\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert\qquad s(x)+y\simeq s(x+y)}{s(\mathfrak{n}'_1+s(\mathfrak{n}_2))\not\simeq s(s(\mathfrak{n}'_1+\mathfrak{n}_2))\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert}}$$

(Sup), (Sup)

$$\vdots\ \pi$$

$$\vdots\ \pi$$

$$\frac{\begin{array}{cc}s(\mathfrak{n}'_1+s(\mathfrak{n}_2))\not\simeq s(s(\mathfrak{n}'_1+\mathfrak{n}_2))\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert & \mathfrak{n}'_1+s(\mathfrak{n}_2)\simeq s(\mathfrak{n}'_1+\mathfrak{n}_2)\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert\end{array}}{\dfrac{s(s(\mathfrak{n}'_1+\mathfrak{n}_2))\not\simeq s(s(\mathfrak{n}'_1+\mathfrak{n}_2))\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert}{\dfrac{\bot\leftarrow\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert}{\neg\lVert\mathfrak{n}_1\simeq s(\mathfrak{n}'_1)\rVert}}}$$

(Sup), (EqRes), (A⊥)

We can also deal with more complicated structures, such as binary trees, as the following examples shows:

Example 5.7 (Simple Induction on Trees). *Let tree $\stackrel{def}{=} E \mid N(tree, \iota, tree)$ be the type of binary trees, $p : \iota \to o$ and $q : tree \to o$. We assume $\forall t : \iota. \; p(t)$, $q(E)$ and $\forall t : \iota. \; \forall l : tree. \; \forall r : tree. \; p(t) \land q(l) \land q(r) \Rightarrow q(N(l, t, r))$. Our goal is to prove $\forall t : tree. \; q(t)$. As we saw before, the proof is quite straightforward and proceeds as follows, using the Skolem constants $\mathsf{t}, \mathsf{t}_l, \mathsf{t}_r : tree$ and $a : \iota$ with $\kappa(\mathsf{t}) = \{E, N(\mathsf{t}_l, a, \mathsf{t}_r)\}$. We introduce the clause context $C[\diamond] \stackrel{def}{=} \neg q(\diamond)$ and prove the theorem as follows:*

$$
\dfrac{\dfrac{\neg q(\mathsf{t}) \qquad \mathsf{t} \simeq E \leftarrow \lVert \mathsf{t} \simeq E \rVert}{\neg q(E) \leftarrow \lVert \mathsf{t} \simeq E \rVert} \; (\text{Sup}) \qquad q(E)}{\dfrac{\bot \leftarrow \lVert \mathsf{t} \simeq E \rVert}{\neg \lVert \mathsf{t} \simeq E \rVert} \; (\text{A}\bot)} \; (\text{Sup})
$$

and, for the recursive case

$$
\dfrac{\dfrac{\dfrac{\neg q(\mathsf{t}) \qquad \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\neg q(N(\mathsf{t}_l, a, \mathsf{t}_r)) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert} \; (\text{Sup}) \qquad \neg p(x) \lor \neg q(l) \lor \neg q(r) \lor q(N(l, x, r))}{\neg p(a) \lor \neg q(\mathsf{t}_l) \lor \neg q(\mathsf{t}_r) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert} \; (\text{Sup})}{\dfrac{\neg p(a) \leftarrow \neg \lVert p(a) \rVert \qquad \neg q(\mathsf{t}_l) \leftarrow \neg \lVert q(\mathsf{t}_l) \rVert \qquad \neg q(\mathsf{t}_r) \leftarrow \neg \lVert q(\mathsf{t}_r) \rVert}{\lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \rightarrowtail \neg \lVert p(a) \rVert \sqcup \neg \lVert q(\mathsf{t}_l) \rVert \sqcup \neg \lVert q(\mathsf{t}_r) \rVert}} \; (\text{ASplit})
$$

leading to the three sub-cases (adding, by inductive strengthening, the clauses $q(\mathsf{t}_l) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert$ and $q(\mathsf{t}_r) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert$):

$$
\dfrac{\dfrac{\neg p(q) \leftarrow \neg \lVert p(a) \rVert \qquad p(x)}{\bot \leftarrow \neg \lVert p(a) \rVert} \; (\text{A}\bot)}{\lVert p(a) \rVert} \; (\text{Sup})
$$

$$
\dfrac{\dfrac{\neg q(\mathsf{t}_l) \leftarrow \neg \lVert q(\mathsf{t}_l) \rVert \qquad q(\mathsf{t}_l) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\bot \leftarrow \neg \lVert q(\mathsf{t}_l) \rVert \sqcap \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert} \; (\text{Sup})}{\neg \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \lor \lVert q(\mathsf{t}_l) \rVert} \; (\text{A}\bot)
$$

$$
\dfrac{\dfrac{\neg q(\mathsf{t}_r) \leftarrow \neg \lVert q(\mathsf{t}_r) \rVert \qquad q(\mathsf{t}_r) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\bot \leftarrow \neg \lVert q(\mathsf{t}_r) \rVert \sqcap \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert} \; (\text{Sup})}{\neg \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \sqcup \lVert q(\mathsf{t}_r) \rVert} \; (\text{A}\bot)
$$

The resulting constraint is unsatisfiable, allowing us to conclude:

$$
\bigsqcap \begin{cases} \lVert \mathsf{t} \simeq E \rVert \oplus \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \\ \neg \lVert \mathsf{t} \simeq E \rVert \\ \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \rightarrowtail \neg \lVert p(a) \rVert \sqcup \neg \lVert q(\mathsf{t}_l) \rVert \sqcup \neg \lVert q(\mathsf{t}_r) \rVert \\ \lVert p(a) \rVert \\ \neg \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \sqcup \lVert q(\mathsf{t}_l) \rVert \\ \neg \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \sqcup \lVert q(\mathsf{t}_r) \rVert \end{cases}
$$

Theorem 5.1 (Soundness of the Minimal Strenghtening Set). *If a set of clauses S is a superset of $cnf(\neg F[\mathsf{i}])$, then adding the minimal strengthening set of $F[\diamond]$ to S is sound, that is, it preserves satisfiability for inductive models.*

Proof. The existence of a model $\mathcal{M}$ for S implies the existence of a minimal model for $\mathrm{cnf}(\neg F[\mathrm{i}])$ by Lemma 5.4. In this case terms smaller than the chosen inductive constant $[\![\mathrm{i}]\!]^{\mathcal{M}}$ must verify $F[\cdot]$, which implies in particular that the conjunction $\bigwedge_{t' \lhd t, \mathrm{sub}(t', \mathrm{i}), D \in \mathrm{cnf}(F[t'])} D \leftarrow [\![\mathrm{i} \simeq t]\!]$ is satisfied in $\mathcal{M}$. Conversely, a model of the strengthened set can be trivially restricted to S by ignoring new Skolem symbols. □

Examples 5.6 and 5.7 demonstrate it's already possible to perform *some* inductive reasoning with strengthening only. However, inductive theorem proving does not have the sub-formula property — that is, a proof might require to introduce formulas that were not present in the initial problem — and it shows very quickly, as the next section will emphasize.

5.3 Proving and Using Lemmas

Inductive strengthening, as explained in Section 5.2, isn't enough to prove many interesting goals. For instance, proving the commutativity of addition on natural numbers, $\forall m\, n : \mathrm{nat}.\ m + n \simeq n + m$, requires the lemmas $\forall m\, n : \mathrm{nat}.\ m + s(n) \simeq s(m + n)$ (Example 5.6) and $\forall n : \mathrm{nat}.\ n + 0 \simeq n$ (Example 5.5). We also need lemmas to perform nested induction (see Remark 5.10 later). The full proof can be found in Example 5.11. More generally, we might want to introduce arbitrary lemmas in a proof (using a kind of "cut" rule that requires to first prove the lemma, then use it). For instance, the user could provide "hints" as intermediate lemmas she believes will be helpful; the system could then try to prove them and use them in the course of solving the main goal.

Fortunately, AVATAR makes it very easy to introduce several lemmas and interleave their proof with the main saturation process. Given a (candidate) lemma F (a first-order formula), the clauses $\{C \leftarrow [\![F]\!] \mid C \in \mathrm{cnf}(F)\} \cup \{D \leftarrow \neg[\![F]\!] \mid D \in \mathrm{cnf}(\neg F)\}$ are added to $\mathcal{N}$. This corresponds to a boolean split over $F \vee \neg F$, where the choice between F and $\neg F$ is represented by the boolean valuation of $[\![F]\!]$.

Definition 5.12 (Lemma Introduction)**.** *The* introduction rule *of a lemma F, where F is a first-order formula, is the following inference rule:*

$$
\begin{array}{c}
\textbf{Lemma Introduction } (\mathsf{Lemma}) \\[4pt]
\hline
\top \\[2pt]
\hline
\bigwedge_{C \in \mathrm{cnf}(F)} \quad C \leftarrow [\![F]\!] \\
\wedge \quad \bigwedge_{D \in \mathrm{cnf}(\neg F)} \quad D \leftarrow \neg[\![F]\!]
\end{array}
$$

Theorem 5.2. *The inference rule* (Lemma) *is sound.*

Proof. (Lemma) is similar to an AVATAR boolean split on $F \vee \neg F$ using the boolean $[\![F]\!]$ (F, being closed, is either valid or it is not). Since $[\![\neg F]\!] \overset{\mathrm{def}}{=} \neg[\![F]\!]$, we obtain the trivial constraint $[\![F]\!] \sqcup \neg[\![F]\!]$ and the "A-formulas" $F \leftarrow [\![F]\!]$ and $\neg F \leftarrow \neg[\![F]\!]$ that can then be reduced to CNF. In essence, (Lemma) is using an adaptation of (ASplit) to formulas. □

In part of the search space, inference with A-clauses of the form $C \leftarrow [\![F]\!]$ will correspond to using the lemma F, assuming it has been proved; in another part, inferences with A-clauses of the form $D \leftarrow \neg[\![F]\!]$ will possibly lead to (conditional) proofs of F by reaching clauses of the form $\bot \leftarrow \neg[\![F]\!] \sqcap \Gamma$ (proof of F under assumptions $\neg\Gamma$). Those proofs may also make use of inductive reasoning, as seen in Section 5.2, possibly requiring several instantiations of $\mathrm{cnf}(\neg F)$ depending on which variable is chosen for induction.

Remark 5.4 (Fairness and Lemmas)**.** *Using* (Lemma) *on a non-theorem formula F does not prevent an unsatisfiable combined state from being reached. Any derivation that could reach an inconsistent combined state ($S_{constraints}$ is absurd or empty clause was found) is still a valid derivation and can safely ignore the candidate lemma. The proof of each lemma is interleaved with the rest of the saturation process. Thanks to this, it is possible to introduce several (candidate) lemmas even if they are not all true or provable. However, it might take a longer time to find a solution, because of the larger search space.*

5.3.1 Guessing Lemmas

We now know how to introduce candidate lemmas, try to prove them and use them, but we don't know yet which lemmas to introduce. Of course, the real issue with cuts resides in finding the right one. The simple approach developed above is agnostic in this respect, so we can plug any black-box we like in. A lot of literature was dedicated to heuristics for finding relevant lemmas and generalizing the induction hypothesis [BSvH+93, BM14]. We present here a few (not exclusive) possible heuristics, but more research is needed in this direction.

Exhaustive Generation

Use an exhaustive generator of candidate lemmas up to a given depth, similar to what other tools such as CVC4 [RK15], Isaplanner [JDB10], and HipSpec [CJRS12] do. The basic principle is very simple: given a signature on one or more inductive types (i.e., the set of constructors for each type) and a set of function symbols working on those types, one can generate all formulas up to a given size, and try to prove all of them, in the spirit of the time-honored generate-and-test techniques. A good start, for provers that handle well equality reasoning, is to generate only equations, rather than arbitrary formulas.

Example 5.8 (Lists and Natural numbers)**.** *The classic types of natural numbers $nat \stackrel{def}{=} 0 \mid s(nat)$ and lists thereof $list \stackrel{def}{=} [] \mid nat :: list$ are pervasively used in Computer Science and Logic. There is a plethora of additional functions defined on those types, but let us focus on the following ones: $+: nat \times nat \to nat$, $rev: list \to list$, @: $list \times list \to list$ (concatenation), and $sum: list \to nat$ defined by the following axioms:*

$$
\begin{aligned}
0 + x &\simeq x \\
s(x) + y &\simeq s(x + y) \\
sum([]) &\simeq 0 \\
sum(x :: y) &\simeq x + sum(y) \\
[] @ x &\simeq x \\
(x :: y) @ z &\simeq x :: (y @ z) \\
rev([]) &\simeq [] \\
rev(x :: y) &\simeq rev(y) @ (x :: [])
\end{aligned}
$$

Then, generating all possible equalities (universally quantified) up to size 10 (or depth 3) will yield, among others, the interesting following lemmas:

$$
\begin{aligned}
x + s(y) &\simeq s(x + y) \\
x + 0 &\simeq x \\
x + y &\simeq y + x \\
x @ [] &\simeq x \\
rev(rev(x)) &\simeq x \\
sum(rev(x)) &\simeq sum(x)
\end{aligned}
$$

but also, among others, the falsities $s(x) \simeq x + x$, $rev(x) \simeq x$.

Since naively enumerating all the possible lemmas, even just positive equations, makes for too large a search space, we need additional filtering on candidate lemmas. Other tools use refinements of the generation technique; CVC4 uses its current model and other tricks   and HipSpec relies on randomized testing. With Superposition we have different weapons to filter obviously wrong candidates out, for instance the techniques presented in the two following paragraphs.

Generate and Filter by Narrowing to $\bot$

To refine the exhaustive generation of candidate lemmas, Figure 5.2 shows an inference rule that narrows a candidate lemma F (a positive equation, in practice) using a set of equations (deduced from the initial set of clauses). If this derivation yields to $\bot$ somehow, then the candidate clause cannot be a valid lemma. The rule is a mix of demodulation (it requires $l\sigma > r\sigma$) and regular Superposition (it uses unification rather than rewriting). It is not terminating, but in practice one can restrict the depth of derivations. It can for instance rule out the false lemma $\forall n.\ n + 0 \simeq s(n)$ (narrowing with $\forall m.\ 0 + m \simeq m$ and $\sigma = \{m \mapsto 0\}$).

<div align="center">

Decreasing Narrowing (DN)

$$\frac{l \simeq r \qquad F}{F[r]_p\,\sigma}$$

if $l\sigma = F|_p\sigma$, and $F[l]_p\,\sigma > F[r]_p\,\sigma$

</div>

Figure 5.2: Filtering Inference Rule

Example 5.9. *The exhaustive generation of formulas might stumble upon the false lemma $\forall x.\ s(x) \simeq x + x$. The following derivation finds a counter-example to the candidate lemma:*

$$\frac{\dfrac{s(x) \simeq x + x \qquad 0 + y \simeq y}{s(0) \simeq 0}\ (\text{DN})\,with\,\{x \mapsto 0, y \mapsto x\}}{\bot}\ (\text{NOv})$$

Generate and Filter by Demodulating to a previous Candidate Lemma

Using regular demodulation (See Figure 2.2), a candidate lemma F can be rewritten into a set of normal forms (the set of equations used for demodulation is not necessarily confluent before saturation is reached). If any of those normal forms is an already generated (smaller) candidate lemma, then F is redundant. That is, given the set of rules R, if $C \xrightarrow{R} D$ means that C is rewritten into D by one step of demodulation using a rule in R, then F is redundant if $F \xrightarrow{R}{}^+ F'$ where F' is an already generated lemma.

Example 5.10. *Let us assume the lemma $F_1 \stackrel{def}{=} x + s(y) \simeq s(x + y)$ has already been generated, and the exhaustive generation mechanism just came up with $F_2 \stackrel{def}{=} 0 + (x + s(y)) \simeq s(x) + y$. The following derivation, using only demodulation, reduces F_2 to F_1. Therefore, F_2 can be dropped safely.*

$$\frac{\dfrac{0 + (x + s(y)) \simeq s(x) + y \qquad 0 + x \simeq x}{x + s(y) \simeq s(x) + y}\ (\text{Demod}) \qquad s(u) + v \simeq s(u + v)}{x + s(y) \simeq s(x + y)}\ (\text{Demod})$$

Remark 5.5. *It is also possible to use previous lemmas as rewrite rules for demodulation — in the example above, F_2 would reduce to $\top$ — but it raises the new issue that a lemma F could be made redundant by a* conjunction *of several previous lemmas; if only a subset of those is false, F might still be relevant. On the contrary, demodulation keeps both candidate lemmas equivalent so we can keep only the smaller one.*

Detecting already known Theories

We will see, in Chapter 6, a technique to recognize already known (axiomatic) theories. Some of those theories can be inductive theories (e.g., the theory of lists). In this case, once a useful lemma is discovered — by any mean, including it being a goal provided by the user — and proved correct, then it can be saved in persistent storage and recalled later when the theory is recognized. For instance, once the (difficult) lemma $\mathrm{rev}(\mathrm{rev}(x)) \simeq x$ is proved, it is certainly worthwhile to save it and re-use it later when the prover acknowledges the presence of lists in the problem it tries to solve.

Generalizing Subgoals

For every ground A-clause $C \leftarrow \Gamma$ such that C contains constants $k_1 : \tau_1, \ldots, k_n : \tau_n$ where all τ_i are inductive types, we introduce a candidate lemma

$$\forall x_1 : \tau_1. \, \ldots \forall x_n : \tau_n. \, \neg C\,[k_1 \leftarrow x_1, \ldots, k_n \leftarrow x_n]$$

This amounts to using current inductive "sub-goals" to guess lemmas that could be used to immediately solve the sub-goal if they ever get proved. For instance, from $\mathfrak{n}' + 0 \not\simeq \mathfrak{n}'$ that occurs in the following example (5.11), we can try and prove the lemma $\forall x. \, x + 0 \simeq x$ (as in Example 5.5). Let us see in more details how this single lemma generation mechanism enables a fully automatic proof of the commutativity of addition.

Example 5.11 (Commutativity of +). *Let us prove the commutativity of $+$ on natural numbers. We start with $\mathfrak{n}_1 + \mathfrak{n}_2 \not\simeq \mathfrak{n}_2 + \mathfrak{n}_1$, where $\mathfrak{n}_1$ and $\mathfrak{n}_2$ are inductive Skolem constants, and try induction on $\mathfrak{n}_1$ (the branch on $\mathfrak{n}_2$ exists, but isn't shown here. Both can be explored in parallel). The first case split adds $\lVert \mathfrak{n}_1 \simeq 0 \rVert \oplus \lVert \mathfrak{n}_1 \simeq s(\mathfrak{n}_1') \rVert$ to boolean constraints, and deduces the clauses $\mathfrak{n}_1 \simeq 0 \leftarrow \lVert \mathfrak{n}_1 \simeq 0 \rVert$ and $\mathfrak{n}_1 \simeq s(\mathfrak{n}_1') \leftarrow \lVert \mathfrak{n}_1 \simeq s(\mathfrak{n}_1') \rVert$.*

$$
\cfrac{
  \cfrac{
    \mathfrak{n}_1 + \mathfrak{n}_2 \not\simeq \mathfrak{n}_2 + \mathfrak{n}_1 \quad \mathfrak{n}_1 \simeq 0 \leftarrow \lVert \mathfrak{n}_1 \simeq 0 \rVert
  }{
    \mathfrak{n}_2 \not\simeq \mathfrak{n}_2 + 0 \leftarrow \lVert \mathfrak{n}_1 \simeq 0 \rVert
  }\ (\mathsf{Sup})(\dagger)
  \qquad
  \cfrac{}{x \simeq x + 0 \leftarrow \lVert a_1 \rVert}\ (\mathsf{Sup})
}{
  \cfrac{
    \cfrac{
      \mathfrak{n}_2 \not\simeq \mathfrak{n}_2 \leftarrow \lVert \mathfrak{n}_1 \simeq 0 \rVert \sqcap \lVert a_1 \rVert
    }{
      \cfrac{
        \bot \leftarrow \lVert \mathfrak{n}_1 \simeq 0 \rVert \sqcap \lVert a_1 \rVert
      }{
        \neg \lVert \mathfrak{n}_1 \simeq 0 \rVert \sqcup \neg \lVert a_1 \rVert
      }\ (\mathsf{A}\bot)
    }\ (\mathsf{EqRes})
  }{}
}
$$

After the inference labelled (†) *we "guess" the lemma $a_1 \overset{def}{=} \forall n. \, n \simeq n + 0$ (note that $\lVert a_1 \rVert$ is a boolean literal, it is either true or false in the SAT solver's model) and use it to conclude. The lemma is proved as follows[7] (introducing $\mathfrak{n}_3$, $\mathfrak{n}_3'$ by splitting on $\mathfrak{n}_3$, and $\mathfrak{n}_3' \simeq \mathfrak{n}_3' + 0 \leftarrow \lVert \mathfrak{n}_3 \simeq s(\mathfrak{n}_3') \rVert$ by strengthening):*

$$
\cfrac{
  \cfrac{
    \mathfrak{n}_3 \not\simeq \mathfrak{n}_3 + 0 \leftarrow \neg \lVert a_1 \rVert \quad \mathfrak{n}_3 \simeq 0 \leftarrow \lVert \mathfrak{n}_3 \simeq 0 \rVert
  }{
    \cfrac{
      \mathfrak{n}_3 \not\simeq \mathfrak{n}_3 \leftarrow \neg \lVert a_1 \rVert \sqcap \lVert \mathfrak{n}_3 \simeq 0 \rVert
    }{
      \cfrac{
        \bot \leftarrow \neg \lVert a_1 \rVert \sqcap \lVert \mathfrak{n}_3 \simeq 0 \rVert
      }{
        \lVert a_1 \rVert \sqcup \neg \lVert \mathfrak{n}_3 \simeq 0 \rVert
      }\ (\mathsf{A}\bot)
    }\ (\mathsf{EqRes})
  }{}
}{}\ (\mathsf{Sup})
$$

---

[7] The lemma is already proved in Example 5.5, but here we show how it is proved as a sub-lemma of a more complicated proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{n_3 \neq n_3 + 0 \leftarrow \neg\lVert a_1 \rVert \qquad n_3 \simeq s(n_3') \leftarrow \lVert n_3 \simeq s(n_3') \rVert}{s(n_3') \neq s(n_3') + 0 \leftarrow \neg\lVert a_1 \rVert \sqcap \lVert n_3 \simeq s(n_3') \rVert} \text{(Sup)} \qquad s(x) + y \simeq s(x+y)}{s(n_3') \neq s(n_3' + 0) \leftarrow \neg\lVert a_1 \rVert \sqcap \lVert n_3 \simeq s(n_3') \rVert \qquad n_3' \simeq n_3' + 0 \leftarrow \lVert n_3 \simeq s(n_3') \rVert} \text{(Sup)}}{s(n_3') \neq s(n_3') \leftarrow \neg\lVert a_1 \rVert \sqcap \lVert n_3 \simeq s(n_3') \rVert} \text{(EqRes)}}{\dfrac{\bot \leftarrow \neg\lVert a_1 \rVert \sqcap \lVert n_3 \simeq s(n_3') \rVert}{\lVert a_1 \rVert \sqcup \neg\lVert n_3 \simeq s(n_3') \rVert} \text{(A}\bot\text{)}}$$

The SAT solver will be forced to conclude $\lVert a_1 \rVert$, making the first proof valid. Similarly, the recursive case $\lVert n_1 \simeq s(n_1') \rVert$, after inference (‡), suggests the lemma $a_2 \overset{def}{=} \forall m\ n.\ m + s(n) \simeq s(m+n)$ (easily proved, see Example 5.6):

$$\dfrac{\dfrac{n_1 + n_2 \neq n_2 + n_1 \qquad n_1 \simeq s(n_1') \leftarrow \lVert n_1 \simeq s(n_1') \rVert}{s(n_1') + n_2 \neq n_2 + s(n_1') \leftarrow \lVert n_1 \simeq s(n_1') \rVert} \text{(Sup)} \qquad s(x) + y \simeq s(x+y)}{s(n_1' + n_2) \neq n_2 + s(n_1') \leftarrow \lVert n_1 \simeq s(n_1') \rVert} \text{(Sup)}$$

$$\vdots\ \pi$$

$$\vdots\ \pi$$

$$\dfrac{\dfrac{\dfrac{n_1' + n_2 \simeq n_2 + n_1' \leftarrow \lVert n_1 \simeq s(n_1') \rVert}{s(n_2 + n_1') \neq n_2 + s(n_1') \leftarrow \lVert n_1 \simeq s(n_1') \rVert} \text{(Sup)(‡)} \qquad x + s(y) \simeq s(x+y) \leftarrow \lVert a_2 \rVert}{s(n_2 + n_1') \neq s(n_2 + n_1') \leftarrow \lVert n_1 \simeq s(n_1') \rVert \sqcap \lVert a_2 \rVert} \text{(Sup)}}{\dfrac{\bot \leftarrow \lVert n_1 \simeq s(n_1') \rVert \sqcap \lVert a_2 \rVert}{\neg\lVert n_1 \simeq s(n_1') \rVert \sqcup \neg\lVert a_2 \rVert} \text{(A}\bot\text{)}} \text{(EqRes)}$$

We have had to introduce two cuts, two lemmas, in this proof. There is no hope to always find appropriate lemmas in an automated fashion, but this examples shows that it is still possible in some cases.

Of course, this mechanism of generalization has its own limits — it could be combined with the (quasi-)exhaustive generation techniques presented above — as the following example illustrates.

Example 5.12 (Difficult Generalization). *Let us introduce the function $dup\colon nat \to nat$, axiomatized by $dup(0) \simeq 0$ and $dup(s(x)) \simeq s(s(dup(x)))$. We want to prove the theorem $\forall x.\ dup(x) \simeq x + x$, and to this end we start with the goal $dup(n) \neq n + n$. The base case works fine:*

$$\dfrac{\dfrac{\dfrac{dup(n) \neq n + n \qquad n \simeq 0 \leftarrow \lVert n \simeq 0 \rVert}{dup(0) \neq 0 + 0 \leftarrow \lVert n \simeq 0 \rVert} \text{(Sup)} \qquad 0 + x \simeq x}{dup(0) \neq 0 \leftarrow \lVert n \simeq 0 \rVert} \text{(Sup)} \qquad dup(0) \simeq 0}{\dfrac{\dfrac{0 \neq 0 \leftarrow \lVert n \simeq 0 \rVert}{\bot \leftarrow \lVert n \simeq 0 \rVert} \text{(EqRes)}}{\neg\lVert n \simeq 0 \rVert} \text{(A}\bot\text{)}} \text{(Sup)}$$

Now, for the recursive case, with the strengthening $dup(n') \simeq n' + n' \leftarrow \lVert n \simeq s(n') \rVert$:

$$\text{(Sup)}\ \dfrac{\text{(Sup)}\ \dfrac{\text{(Sup)}\ \dfrac{dup(n) \neq n + n \qquad n \simeq s(n') \leftarrow \lVert n \simeq s(n') \rVert}{dup(s(n')) \neq s(n') + s(n') \leftarrow \lVert n \simeq s(n') \rVert} \qquad s(x) + y \simeq s(x+y)}{s(s(dup(n'))) \neq s(n' + s(n')) \leftarrow \lVert n \simeq s(n') \rVert \qquad dup(n') \simeq n' + n' \leftarrow \lVert n \simeq s(n') \rVert}}{s(s(n' + n')) \neq s(n' + s(n')) \leftarrow \lVert n \simeq s(n') \rVert}$$

and we get stuck there. The problem here is that the missing lemma, $\forall x\ y.\ s(x+s(y)) \simeq s(s(x+y))$ (simplified, by (Inj), into $\forall x\ y.\ x+s(y) \simeq s(x+y)$ which is easy to prove, as Example 5.6 shows), requires generalizing the last goal $s(s(\mathfrak{n}' + \mathfrak{n}')) \not\simeq s(\mathfrak{n}' + s(\mathfrak{n}'))$ in such a way that some occurrences of $\mathfrak{n}'$ are abstracted by x and some are abstracted by y, where x and y are distinct variables. Finding a heuristic to properly infer the right lemma in this case seems very difficult.

Remark 5.6 (Injectivity Rule and Lemmas). *The rule (Inj) has not been used in the chapter yet, but this last example suggests that some lemmas might need it. When a negative goal $c(t_1, \ldots, t_n) \not\simeq c(t'_1, \ldots, t'_n)$ is met, in general, Superposition will try and eliminate it by making each pair $t_i \simeq t'_i$ valid; however if a lemma is proposed from this goal it will have the form $c(\ldots) \simeq c(\ldots)$ which can be simplified by injectivity.*

5.4 Inductive Strengthening using Several Clauses

The technique of inductive strengthening developed in the previous section works well when induction is performed on a property that is explicitly present in the set of clauses. Ignoring the heuristics that introduce lemmas, because they are mostly orthogonal to the point, this technique is still too weak in cases where the formula to perform induction on is stronger than the final goal. The following very simple example 5.13 illustrates where it fails.

We take some inspiration from the extension of Superposition to induction on natural numbers [KP13] that can use several clauses (more precisely, some equivalent of our notion of clause context) at once, and present a novel way to tackle the issue of dealing with conjunctive inductive formulas.

Example 5.13 (Non-clausal Induction Formula). *Let us assume $\forall n.\ (p(n) \vee q(n)) \Rightarrow (p(s(n)) \vee q(s(n)))$ and $p(0) \vee q(0)$. Assume we already have the clauses $\neg p(\mathfrak{n})$ and $\neg q(\mathfrak{n})$ (to prove the theorem $\forall n.\ p(n) \vee q(n)$); it is impossible to guess the relevant clause context. Proving the base case works well:*

$$\frac{p(0) \vee q(0)}{p(0) \leftarrow \lVert p(0) \rVert \qquad q(0) \leftarrow \lVert q(0) \rVert \qquad \lVert p(0) \rVert \sqcup \lVert q(0) \rVert}\ \text{(ASplit)}$$

$$\vdots\ \pi$$

and the two cases

$$\frac{\dfrac{\neg p(\mathfrak{n}) \qquad \mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}{\neg p(0) \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}\ \text{(Sup)} \qquad \dfrac{\vdots\ \pi \\ p(0) \leftarrow \lVert p(0) \rVert}{}\ \text{(Sup)}}{\dfrac{\bot \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap \lVert p(0) \rVert}{\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert p(0) \rVert}\ \text{(A}\bot\text{)}}$$

and symmetrically to obtain $\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert q(0) \rVert$. So far everything is fine. For the recursive case, we have to choose to strengthen one clause context among $\{\neg p(\diamond), \neg q(\diamond)\}$. What happens in both case is very exactly the same; to make our point, we pick $\neg p(\diamond)$, which adds the clause $p(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert$:

$$\frac{\dfrac{\neg p(\mathfrak{n}) \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{\neg p(s(\mathfrak{n}')) \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}\ \text{(Sup)} \qquad \neg p(x) \vee p(s(x)) \vee q(s(x))}{\dfrac{\neg p(\mathfrak{n}') \vee q(s(\mathfrak{n}')) \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{\neg p(\mathfrak{n}') \leftarrow \neg \lVert p(\mathfrak{n}') \rVert \qquad q(s(\mathfrak{n}')) \leftarrow \lVert q(s(\mathfrak{n}')) \rVert}\ \text{(ASplit)(†)} \\ \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \rightarrowtail \neg \lVert p(\mathfrak{n}') \rVert \sqcup \lVert q(s(\mathfrak{n}')) \rVert}\ \text{(Sup)}}$$

$$\frac{\neg p(\mathfrak{n}') \leftarrow \neg \lVert p(\mathfrak{n}') \rVert \qquad p(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{\underline{\bot \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap \neg \lVert p(\mathfrak{n}') \rVert}} \text{(Sup)}}{\neg \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcup \lVert p(\mathfrak{n}') \rVert} \text{(A}\bot\text{)}$$

So the case $\neg p(\mathfrak{n}')$ is solved under the assumption $\lVert \neg p(\mathfrak{n}') \rVert$, after the AVATAR *split at the inference annotated* (†). *However, the other case, $q(s(\mathfrak{n}'))$, cannot benefit from any strengthening hypothesis, and its branch fails to close:*

$$\frac{\neg q(\mathfrak{n}) \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{\neg q(s(\mathfrak{n}')) \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert} \text{(Sup)}$$

We need to assume that $\mathfrak{n}$ is actually minimal for both $\neg p[\diamond]$ and $\neg q[\diamond]$. *Note that in general we might need an arbitrary number of contexts, not just two, for instance k clauses if the inductive property to prove was $\forall n. \bigvee_{i=1}^{k} p_i(n) \Rightarrow \bigvee_{i=1}^{k} p_i(s(n))$.*

In the next few sections, we address this problem using a more sophisticated flavor of strengthening, in which several clause contexts — among a finite pool — represent the property for which a minimal model must exist.

5.4.1 Existence of an Inductive Model for a Subset of Clauses

Given a state $(\mathcal{N}, \mathcal{F}_b)$ and an inductive constant $\mathfrak{i}$, we've seen that the existence of an inductive model implies the existence of a minimal model for every *subset* of $\mathcal{N}$. It suffices to find a subset of $\mathcal{N}$ that provably doesn't admit a minimal inductive model w.r.t. some inductive constant, to prove that the whole state doesn't admit a model either (and neither does the initial problem). In the following, $S_{\text{cand}}(\mathfrak{i})$ (read: "set of candidate contexts for $\mathfrak{i}$") will be a finite set of clause contexts where $\{C[\mathfrak{i}] \mid C[\diamond] \in S_{\text{cand}}(\mathfrak{i})\} \subseteq \mathcal{N}$. We will shorten $\{C[\mathfrak{i}] \mid C[\diamond] \in S_{\text{cand}}(\mathfrak{i})\}$ as $S_{\text{cand}}(\mathfrak{i})[\mathfrak{i}]$. The proof search keeps the following sets of A-clauses separate (using boolean trails):

$S_{\textbf{input}}$: initially the input problem (with minor modifications), it is used to discover new salient clause contexts and to prove $C[\mathfrak{i}]$ for new contexts $C[\diamond]$ — useful because we need to ensure that the clause contexts form a subset of the initial problem when applied to $\mathfrak{i}$. All clauses in S_{input} are deductively provable from the initial state, using the inference rules of Superposition and AVATAR (see Section 2.4 and 2.5). The clauses are not used for induction proper, although they contain inductive constants.

$S_{\textbf{min}}(\mathfrak{i})$: this set, initially empty, contains induction hypothesis for $\mathfrak{i}$ (clauses of the form $C[\mathfrak{i}] \leftarrow \Gamma$ for some Γ); it is chosen dynamically as a subset of $S_{\text{cand}}(\mathfrak{i})$ (the set of all clause contexts for $\mathfrak{i}$). The proof procedure attempts to refute the minimality of some $S_{\text{min}}(\mathfrak{i}) \subseteq S_{\text{cand}}(\mathfrak{i})$. Clauses in $S_{\text{min}}(\mathfrak{i})$ do not interact at all with clauses from S_{input}.

T : The *theory* is composed of all clauses that are deducible from the input problem and do not contain any inductive constants (they might contain inductive variables). In other words, those clauses are not concerned with minimality of a model w.r.t. $\mathfrak{i}$. They can be used in inferences both with clauses from S_{input} and $S_{\text{min}}(\mathfrak{i})$ without restrictions.

Remark 5.7 (Interactions S_{input} to $S_{\text{min}}(\mathfrak{i})$). *Although S_{input} and $S_{min}(\mathfrak{i})$ are kept separate by the prover, they still interact in some way. In particular, clause contexts can be extracted from clauses in S_{input}, and a context $C[\diamond]$ can really be used for induction only after $C[\mathfrak{i}]$ has been proved in S_{input} (see Section 5.4.3). Apart from that, the system behaves as if two distinct Superposition provers were working on S_{input} and $S_{min}(\mathfrak{i})$ separately.*

The proof process performs inferences as usual on those two sets of clauses (separately), and gathers constraints in $S_{\text{constraints}}$ as usual. It succeeds when a subset of $S_{\text{cand}}(\mathfrak{i})[\mathfrak{i}]$ is found to have no minimal model, which amounts to $S_{\text{constraints}}$ being unsatisfiable.

Definition 5.13 (Minimality Witness)**.** *Given a constant* i*, a coverset* $\kappa(i)$*, a set of clause contexts* U *and* $t \in \kappa(i)$*, we call* minimality witness *for* U *if* $i \simeq t$ *the formula* $\bigwedge_{j \triangleleft t, sub(j,i)} \bigvee_{C[\diamond] \in U} \neg C[j]$*, or alternatively,* $\neg \bigvee_{j \triangleleft t, sub(j,i)} \bigwedge_{C[\diamond] \in U} C[j]$*.*

Assuming $U[i]$ and $i \simeq t$ are true, the minimality witness formula means that for every term j structurally smaller than t, some clause context $C[\diamond] \in U$ is provably false on j. Therefore, if we can derive $\bot$ from the minimality witness and $U[i]$, there cannot be a minimal model for $U[i]$ that also satisfies $i \simeq t$. In the case $t \in \kappa_\bot(i)$, the minimality witness is trivially true (degenerate case), so we only have to prove $U[i] \wedge i \simeq t \vdash \bot$.

Definition 5.14 (Criterion for the Absence of a Minimal Model)**.** *Let* $U \subseteq S_{cand}(i)$ *be a set of clause contexts. To check that the set of clauses* $U[i]$ *has no minimal model w.r.t.* i *in the theory* T*, we must find a proof of* $\bot$ *from*

$$U[i] \wedge i \simeq t \wedge T \wedge \left(\bigwedge_{j \triangleleft t, sub(j,i)} \bigvee_{C[\diamond] \in U} \neg C[j] \right)$$

for each $t \in \kappa(i)$*.*

In practice, the task of finding the proofs of $\bot$ will be divided up betwen the Superposition prover and a QBF solver, as explained in Section 5.4.2 and 5.4.5. If the criterion is met, any possible choice of an inductive value for i leads to $\bot$ or to the non-minimality of the model. By Lemma 5.4 that means that $\mathcal{N} \supseteq U[i]$ has no model. Any procedure that checks the two properties above is therefore a sound unsatisfiability condition for inductive problems. Now, we need some computable way to check whether the criterion applies; where Kersani and Peltier [KP13] propose two ad-hoc fixpoint algorithms (respectively, greater and smaller fixpoint computations), we build on the AVATAR architecture and let a boolean solver do the job — with a twist, because we need more than a SAT solver.

Theorem 5.3 (Soundness of the Criterion)**.** *Given* $U \subseteq S_{cand}(i)$*, if the criterion of Definition 5.14 is met — that is, if for each* $t \in \kappa(i)$ *the formula*

$$T \wedge i \simeq t \wedge U[i] \wedge \bigwedge_{j \triangleleft t, sub(j,i)} \bigvee_{C[\diamond] \in U} \neg C[j]$$

leads to a proof of $\bot$ *— then* $T \wedge U[i]$ *has no minimal inductive model.*

Proof. Starting from $T \wedge i \simeq t \wedge U[i] \wedge \bigwedge_{j \triangleleft t, \mathrm{sub}(j,i)} \bigvee_{C[\diamond] \in U} \neg C[j] \vdash \bot$, we obtain $T \wedge i \simeq t \wedge U[i] \vdash \bigvee_{j \triangleleft t, \mathrm{sub}(j,i)} \bigwedge_{C[\diamond] \in U} C[j]$. Let us assume there is a minimal inductive model $\mathcal{M}$ of $T \wedge U[i] \wedge i \simeq t$. Then $\mathcal{M} \models \bigvee_{j \triangleleft t, \mathrm{sub}(j,i)} \bigwedge_{C[\diamond] \in U} C[j]$. Let j be a member of $\{j \triangleleft t, \mathrm{sub}(j,i)\}$ such that $\mathcal{M} \models T \wedge U[i] \wedge \bigwedge_{C[\diamond] \in U} C[j]$, i.e., $\mathcal{M} \models T \wedge U[i] \wedge U[j]$, with $[\![j]\!]^{\mathcal{M}} \triangleleft [\![i]\!]^{\mathcal{M}}$. Then, the model $\mathcal{M}'$ obtained from $\mathcal{M}$ by mapping i to $[\![j]\!]^{\mathcal{M}}$ is a model of $T \wedge U[i]$, because i and j are both Skolem constants without any axiom on them (apart from the splitting rule). Because $[\![i]\!]^{\mathcal{M}'} = [\![j]\!]^{\mathcal{M}} \triangleleft [\![i]\!]^{\mathcal{M}}$, $\mathcal{M}$ is not minimal for i, contradicting our hypothesis. $\square$

5.4.2 Encoding to QBF

We now present an encoding of the formula from Definition 5.14 in QBF (for a reminder of what QBF is, see Section 2.22). Why QBF? First, we favored a boolean solver because the interface between Superposition and the propositional solver used the *clause trails* extensively, and it fit well within the AVATAR framework [Vor14]. With QBF we can express exponentially many formulas in a linear-sized formula, a gain in expressiveness that we will need to quantify over all (non-empty) subsets of the clause contexts. Besides, efficient solvers exist for QBF, some of which are free software. Usual boolean formulas are QBF where all variables are (implicitly) existentially quantified. Therefore, a QBF solver is also a SAT solver[8]; given a true QBF

---

[8]It is proved that QBF-solving is PSPACE-complete.

$\exists b_1 \ldots b_n. F'$ it will be able to assign values (a boolean model) to $\{b_1, \ldots, b_n\}$. This allows us to extend the AVATAR framework smoothly, quantifying split variables existentially before the rest of the QBF. The notion of *combined model* also extends trivially to set of clauses paired with quantified boolean formulas (the boolean valuation being defined only for the variables in the prenex existential fragment of the formula).

5.4.3   Inference Rules and Dependency Tracking

We will need to track the set of clauses on which induction is performed and which ones are used for each proof of false: proving heredity, in an inductive proof, requires using only the theory T and induction hypothesis. As we are going to show, A-clauses and their trails are perfect tools for this.

Keep S_{input} and $S_{\text{min}}(i)$ separate

As described in Section 5.4.1, we need to keep track of several sets of A-clauses. In the first set, S_{input}, every input clause C that contains at least one inductive constant is marked with the special boolean constant input and becomes $C \leftarrow \text{input}$.

On the other hand, clauses in $S_{\text{min}}(i)$ are deduced from induction hypothesis (and minimality witnesses) for i, all of which contain a boolean literal $\lVert C[\diamond] \in S_{\text{min}}(i) \rVert$ in their trail. Intuitively, if $\lVert C[\diamond] \in S_{\text{min}}(i) \rVert$ is true, it means that $C[\diamond]$ is a member of $S_{\text{cand}}(i)$. Redundancy rules (exposed in Figure 5.3) can be added to remove clauses that have been deduced from both S_{input} and $S_{\text{min}}(i)$, or from two incompatible $S_{\text{min}}(\cdot)$, effectively preventing those sets from interacting. Note that the SAT-solver will never make any of those trails true, but it will not *prove* that they are absurd; hence, without the simplification rules, clauses that should have been simplified would just stay frozen and consume memory forever.

$$\textbf{Redundancy } S_{\textbf{min}}(i) \textbf{ / } S_{\textbf{input}}$$
$$\frac{C \leftarrow \text{input} \sqcap \lVert D[\diamond] \in S_{\text{min}}(i) \rVert \sqcap \Gamma}{\top}$$

$$\textbf{Redundancy } S_{\textbf{min}}(i_1) \textbf{/} S_{\textbf{min}}(i_2)$$
$$\frac{C \leftarrow \lVert D_1[\diamond] \in S_{\text{min}}(i_1) \rVert \sqcap \lVert D_2[\diamond] \in S_{\text{min}}(i_2) \rVert \sqcap \Gamma}{\top}$$

Figure 5.3: Redundancy Rules keeping $S_{\text{min}}(i)$ and S_{input} separate

Initialization

A successful subset of $S_{\text{min}}(i)$ needs to be *initialized*, that is, implied by the initial problem, otherwise its satisfiability is irrelevant to the satisfiability of S_{input}. Boolean guards of the form $\lVert \text{init}(C[\diamond], i) \rVert$ are used to keep track of which clause contexts are initialized. As long as the boolean solver does not have to valuate $\lVert \text{init}(C[\diamond], i) \rVert$ to 1, $C[\diamond]$ cannot be reliably used for inductive reasoning. Given a clause context $C[\diamond]$, the set of clauses used in the Superposition prover is watched for clauses $D \leftarrow \Gamma \sqcap \text{input}$ such that D subsumes $C[i]$. In this case, we add the constraint $\Gamma \rightarrowtail \lVert \text{init}(C[\diamond], i) \rVert$ to $S_{\text{constraints}}$. Note that a given context can be initialized in more than one way, with distinct boolean trails[9].

---

[9] The boolean atom input is ignored specifically because this operation transfers constraints from S_{input} into $S_{\text{min}}(\cdot)$

Finding new Clause Contexts

We did not define how the set $S_{\text{cand}}(\mathfrak{i})$ was defined. In fact, this part is heuristic: any method is admissible for proposing clause contexts as long as they are well-typed and belong to the signature — similar to the candidates lemmas in Section 5.3.1. However, we did use a reasonable heuristic in the implementation. When a clause C occurs (possibly with a trail) in $S_{\text{min}}(\mathfrak{i}) \uplus S_{\text{input}}$ such that C is ground and contains some terms $t_1, \ldots, t_n$ of inductive types, with some restrictions on $\{t_i\}_{i=1\ldots n}$, then every one of $C[t_i \leftarrow \diamond]$ for $i \in \{1, \ldots, n\}$ is a new candidate context. The restriction on the terms is the following: C should not contain both an inductive constant $\mathfrak{i}$ and some term t such that $\text{sub}(t, \mathfrak{i})$.

Example 5.14 (Clause Context Extraction)**.** *(i) From the clause $\mathfrak{n}_1 + \mathfrak{n}_2 \not\simeq \mathfrak{n}_2 + \mathfrak{n}_1 \leftarrow \Gamma$, we can extract the two contexts $\diamond + \mathfrak{n}_2 \not\simeq \mathfrak{n}_2 + \diamond$ and $\mathfrak{n}_1 + \diamond \not\simeq \diamond + \mathfrak{n}_1$. (ii) No context can be extracted from $\mathfrak{n} + s(\mathfrak{n}') \simeq \mathfrak{n}'$ if $\kappa(\mathfrak{n}) = \{0, s(\mathfrak{n}')\}$.*

Managing Induction Hypothesis

Our "induction hypothesis"[10] on some $\mathfrak{i}$ will be a conjunction of clause contexts $\bigwedge_{k=1}^{n} C_k[\diamond]$ (where each $C_k[\diamond] \in S_{\text{cand}}(\mathfrak{i})$). We need to assess, for each such conjunction, the following:

- whether the conjunction is proved for $\mathfrak{i}$, that is, *initialization*: is $\bigwedge_{k=1}^{n} C_k[\mathfrak{i}]$ provable from S_{input}, possibly under some boolean trail?
- whether $\bigwedge_{k=1}^{n} C_k[t]$ for every $t \in \kappa_{\downarrow}(\mathfrak{i})$ is inconsistent with the minimality witness $\bigvee_{k=1}^{n} \neg C_k[t']$ for some $t' \lhd t$ where $\text{sub}(t', \mathfrak{i})$. In other words, if those two formulas are inconsistent — if $\bot$ can be deduced from their conjunction — no minimal model can exist for $t \simeq \mathfrak{i}$, as explained in Section 5.2 and Theorem 5.3.
- whether $\bigwedge_{k=1}^{n} C_k[t]$ can prove $\bot$ for every $t \in \kappa_{\bot}(\mathfrak{i})$.

The management of the induction hypothesis is done jointly between the Superposition prover's clause trails and the QBF solver's constraints.

Definition 5.15 (Inductive Strengthening)**.** *For every known[11] clause context $C[\diamond]$ and inductive constant $\mathfrak{i}$ of a compatible type, the* inductive strengthening *of the context is the set of clauses*

$$C[\mathfrak{i}] \leftarrow \lfloor\!\lfloor C[\diamond] \in S_{min}(\mathfrak{i})\rfloor\!\rfloor \sqcap \lfloor\!\lfloor init(C[\diamond], \mathfrak{i})\rfloor\!\rfloor$$

and

$$\neg L_k \sigma[t'] \leftarrow \lfloor\!\lfloor C[\diamond] \in S_{min}(\mathfrak{i})\rfloor\!\rfloor \sqcap \lfloor\!\lfloor minimal(C[\diamond], \mathfrak{i}, t')\rfloor\!\rfloor \sqcap \lfloor\!\lfloor \mathfrak{i} \simeq t\rfloor\!\rfloor$$

for each $t \in \kappa_{\downarrow}(\mathfrak{i})$ and $t' \lhd t$ where $C[\diamond] = \bigvee_{k=1}^{n} L_k$, $\text{sub}(t', \mathfrak{i})$ and σ is a grounding substitution that maps $\text{freevars}(C[\diamond])$ to fresh Skolem symbols[12].

Those clauses become candidate for inferences. The former clause, $C[\mathfrak{i}] \leftarrow \lfloor\!\lfloor C[\diamond] \in S_{\text{min}}(\mathfrak{i})\rfloor\!\rfloor \sqcap \lfloor\!\lfloor init(C[\diamond], \mathfrak{i})\rfloor\!\rfloor$, can play the role of an induction hypothesis (if the boolean $\lfloor\!\lfloor C[\diamond] \in S_{\text{min}}(\mathfrak{i})\rfloor\!\rfloor$ is true, meaning $C[\diamond] \in S_{\text{cand}}(\mathfrak{i})$, and $\lfloor\!\lfloor init(C[\diamond], \mathfrak{i})\rfloor\!\rfloor$ is also true, meaning $C[\mathfrak{i}]$ is provable from $\mathcal{N}$); the latter ones express the potential minimality of $C[\diamond]$ w.r.t. $\mathfrak{i}$ (in particular, $\lfloor\!\lfloor minimal(C[\diamond], \mathfrak{i}, t')\rfloor\!\rfloor$ expresses the falsity of $C[\diamond]$ on $t' \lhd t$). Typically, from the completion of a successful non-empty subset $U \overset{\text{def}}{=} \{C_1[\diamond], \ldots, C_n[\diamond]\}$ of $S_{\text{cand}}(\mathfrak{i})$ — one that cannot have a minimal inductive model — there would be derivations of clauses of the form $\bot \leftarrow \lfloor\!\lfloor minimal(C_j[\diamond], \mathfrak{i}, t')\rfloor\!\rfloor \sqcap \lfloor\!\lfloor \mathfrak{i} \simeq t\rfloor\!\rfloor \sqcap \bigsqcap_{k=1}^{n} (\lfloor\!\lfloor C_k[\diamond] \in S_{\text{min}}(\mathfrak{i})\rfloor\!\rfloor \sqcap \lfloor\!\lfloor init(C_k[\diamond], \mathfrak{i})\rfloor\!\rfloor)$ for each $j \in \{1, \ldots, n\}$, $t \in \kappa_{\downarrow}(\mathfrak{i})$, $t' \lhd t$ and $\text{sub}(t', \mathfrak{i})$. Each empty clause prevents any inductive model $\mathcal{M}$ to be minimal for $C_j[\diamond]$ (the model must

---

[10]More accurately, the set of contexts that we show cannot have a minimal model.

[11]Clause contexts can be "extracted" from clauses from both $S_{\text{min}}(\mathfrak{i})$ and S_{input} heuristically, by replacing an inductive term with a hole. In any state of the theorem prover, only a finite number of contexts are known.

[12]Only one substitution σ per context is needed, even if the inductive strengthening contains several clauses.

satisfy $C_j[t']$ for some $t' \lhd t \simeq \mathfrak{i}$), meaning that if $\mathcal{M}$ is minimal for $U[\diamond]$ it has to be another context $C_k[\diamond] \in U, k \neq j$ that isn't satisfied on any $t' \lhd t \simeq \mathfrak{i}$. In Sections 5.4.5 and 5.4.6 we will see how those special trail literals are used in the QBF solver.

Remark 5.8. *To reason over whether a model is minimal for $C[\diamond]$ when $\mathfrak{i} = t$ ($t \in \kappa(\mathfrak{i})$), we use the boolean literal $\| minimal(C[\diamond], \mathfrak{i}, t') \|$. Note that this literal contains $t' \lhd t$ rather than just t. For instance, in the case of binary trees (as defined in Example 5.7), if $\| \mathfrak{i} \simeq N(l, \_, r) \|$ is true, there is still a difference between $\| minimal(C[\diamond], \mathfrak{i}, l) \|$ (meaning $C[l]$ is false because the model is minimal for $C[\diamond]$) and $\| minimal(C[\diamond], \mathfrak{i}, r) \|$ (same but for the right child). It is possible to refute that the model is minimal for $C[\diamond]$ by refuting any of those two cases.*

This concludes the encoding of the criterion of Definition 5.14 into Superposition and the management of inductive properties using AVATAR. The inductive property is built gradually from several clause contexts, using the strengthening technique exposed in Definition 5.15. In the next section, we develop a boolean constraint that complements strengthening, the same way AVATAR uses a SAT solver to complement its Superposition inference rules.

5.4.4 Summary of Special Boolean Literals

In the previous sections, we have introduced several kinds of propositional literals to be added to clause trails. We review them briefly before presenting the main propositional constraint that enforces the existence of a minimal inductive model, in the next two sections.

- input is added to clauses that follow directly from the problem axiom, to distinguish them from inductive properties.
- $\| C[\diamond] \in S_{\min}(\mathfrak{i}) \|$ is true iff the context $C[\diamond]$ is part of the conjunction of inductive properties for which $\mathfrak{i}$ should have a minimal model. The valuation of all $\| C[\diamond] \in S_{\min}(\mathfrak{i}) \|$ for each $C[\diamond]$ is what determines the current "induction hypothesis" for $\mathfrak{i}$.
- $\| init(C[\diamond], \mathfrak{i}) \|$ must be true if there is some proof of $C[\mathfrak{i}]$ under the problem axioms — that is, it corresponds to the "initialization" step for proving $C[\diamond]$ inductively.
- $\| minimal(C[\diamond], \mathfrak{i}, t) \|$, if true in a model in which $\| C[\diamond] \in S_{\min}(\mathfrak{i}) \|$ holds too, forces $\neg C[t]$ to hold. Such literals enforce that in a candidate combined model $(\mathcal{M}, v)$, $\mathfrak{i}$ is a minimum value for which $\left[\!\!\left[\bigwedge_{C[\diamond] \in S_{\text{cand}}(\mathfrak{i})} \left(C[\mathfrak{i}] \leftarrow \| C[\diamond] \in S_{\min}(\mathfrak{i}) \| \right) \right]\!\!\right]^{\mathcal{M}, v} = \hat{\top}$, implying that either all $\| C[\diamond] \in S_{\min}(\mathfrak{i}) \|$ are false, or there is at least one $C[\diamond]$ such that $[\![C[t]]\!]^{\mathcal{M}} = \hat{\bot}$ and $v\big(\| C[\diamond] \in S_{\min}(\mathfrak{i}) \| \big) = \hat{\top}$.

Remark 5.9 (Trail Inheritance)**.** *All those special literals are inherited in AVATAR splitting (see Remark 2.14). This makes it possible to track the* history *of an inductive clause (i.e., the series of inferences that lead to that clause), and in particular which induction hypothesis have been used to deduce it.*

5.4.5 Induction on One Constant

For the sake of simplicity, let us start by assuming exactly one inductive constant $\mathfrak{i}$ is present in S_{input} (e.g. after Skolemization). We need to encode constraints in QBF so that the criterion from Section 5.4.1 can be checked by a QBF solver. This formula, $F_{\mathfrak{i}}$, is presented and decomposed in Figure 5.4. We briefly recap the various sets involved in the formula:

$S_{\textbf{atoms}}$ contains atoms of the form $\| C \|$, generated by the regular splitting inference in Figure 2.4, but excluding splits on the shape of inductive constants (i.e., no literal resembling $\| \mathfrak{i} \simeq t \|$ with $t \in \kappa(\mathfrak{i})$);

$S_{\textbf{constraints}}$ contains boolean constraints generated by inference rules: Splitting, introduction of induction hypothesis and minimality witnesses — i.e., inductive strengthenings.

$$F_{\mathrm{i}} \quad \overset{\text{def}}{=} \quad \exists_{a \in S_{\text{atoms}}} a$$
$$\forall_{C[\diamond] \in S_{\text{cand}}(\mathrm{i})} \big\| C[\diamond] \in S_{\min}(\mathrm{i}) \big\|$$
$$\exists_{t \in \kappa(\mathrm{i})} \big\| \mathrm{i} \simeq t \big\|$$
$$\exists_{C[\diamond] \in S_{\text{cand}}(\mathrm{i})} \big\| \text{init}(C[\diamond], \mathrm{i}) \big\|$$
$$\exists_{t', \text{sub}(t', \mathrm{i}), C[\diamond] \in S_{\text{cand}}(\mathrm{i})} \big\| \text{minimal}(C[\diamond], \mathrm{i}, t') \big\|$$
$$\left(\bigsqcap_{x \in S_{\text{constraints}}} x \right) \sqcap \left(\text{empty} \sqcup \bigsqcup_{t \in \kappa(\mathrm{i})} \big\| \mathrm{i} \simeq t \big\| \sqcap \text{minimal}(t) \right)$$
$$\text{empty} \quad \overset{\text{def}}{=} \quad \bigsqcap_{C[\diamond] \in S_{\text{cand}}(\mathrm{i})} \neg \big\| C[\diamond] \in S_{\min}(\mathrm{i}) \big\|$$
$$\text{minimal}(t) \quad \overset{\text{def}}{=} \quad \bigsqcap_{t' \lhd t, \text{sub}(t', \mathrm{i})} \bigsqcup_{C[\diamond] \in S_{\text{cand}}(\mathrm{i})} \left(\begin{array}{c} \big\| C[\diamond] \in S_{\min}(\mathrm{i}) \big\| \sqcap \\ \big\| \text{minimal}(C[\diamond], \mathrm{i}, t') \big\| \end{array} \right)$$

Figure 5.4: QBF for Induction on One Constant

We see here that the QBF is stratified into 3 levels of quantification:

1. The outermost (existential) level contains AVATAR-like splitting atoms from S_{atoms} (therefore excluding splitting on cover sets — they can change[13] depending on the set $S_{\min}(\mathrm{i})$). Atoms at this level have a valuation (model) whenever the QBF is satisfiable, same as the regular AVATAR calculus.
2. The middle level is universally quantified. It allows us to enumerate all subsets of $S_{\text{cand}}(\mathrm{i})$ by quantifying on characteristic functions $S_{\min}(\mathrm{i})$. $S_{\min}(\mathrm{i})$ is the current subset of $S_{\text{cand}}(\mathrm{i})$ for which the existence of a minimal model is challenged. This is where using QBF-solving is justified: making it possible to check the criterion on 2^n subsets of $S_{\text{cand}}(\mathrm{i})$ easily, where n is the cardinal of $S_{\text{cand}}(\mathrm{i})$.
3. The innermost level contains literals $\mathrm{i} \simeq t$ and helper predicates that depend on the value of $S_{\min}(\mathrm{i})$ — along with fresh predicates introduced by reduction to CNF, see Section 5.6.2. In addition to the choice of the shape of i (literals $\big\| \mathrm{i} \simeq t \big\|$), this last layer assesses, for each $C[\diamond] \in S_{\text{cand}}(\mathrm{i})$ and term $t \in \kappa(\mathrm{i})$, whether $C[\diamond]$ is the witness for the existence of a minimal model of $S_{\min}(\mathrm{i})$ that also satisfies $\mathrm{i} \simeq t$ (atoms $\big\| \text{minimal}(C[\diamond], \mathrm{i}, t') \big\|$ for each t' where sub(t', i)) and whether $C[\mathrm{i}]$ has been proved from the initial problem or not (atoms $\big\| \text{init}(C[\diamond], \mathrm{i}) \big\|$).

The body of the QBF enforces the constraints accumulated in $S_{\text{constraints}}$ so as to prune boolean valuations that are inconsistent with AVATAR inferences and the choice of $S_{\min}(\mathrm{i})$. The disjunction $(\text{empty} \sqcup \text{minimal})$ forces all choices of $S_{\min}(\mathrm{i})$ to be either the empty set — irrelevant, as it makes $\bigwedge_{C[\diamond] \in S_{\min}(\mathrm{i})} C[t]$ trivially true for any term t in any model — or a set that can have a minimal model by choosing $\mathrm{i} \simeq t$ with $t \in \kappa(\mathrm{i})$ and asserting that one of the contexts $C[\diamond] \in S_{\min}(\mathrm{i})$ is false for smaller values $t' \lhd t$. In the latter case, assuming a model is minimal for some $C[\diamond]$ from $S_{\min}(\mathrm{i})$ can be refuted by deducing clauses resembling $\bot \leftarrow \big\| \text{minimal}(C[\diamond], \mathrm{i}, t') \big\| \sqcap \Gamma$ (as described in Section 5.4.3).

5.4.6 Induction on Several Constants

In general, there are several distinct inductive constants in a problem: goals could have several universal variables, or lemmas could be introduced that require being inductively proved separately (as already seen in Example 5.11). To handle several constants with the QBF-based technique, we keep the same building blocks but state a stronger requirement: there should be a minimal model for each inductive constant separately[14]. Inductive constants are in finite number for a given problem, so we name the set of inductive constants $\mathfrak{I}$ (note that this set can grow during the saturation process, due to the introduction of new Skolem constants). We

---

[13] We are not trying to find a model of the input, but to assess the satisfiability of $S_{\min}(\mathrm{i})$.

[14] We lack a notion of a model that would be minimal for several constants at one.

don't consider constants occurring in coversets (i.e., terms t such that $\mathrm{sub}(t, \mathfrak{i})$ for some $\mathfrak{i} \in \mathfrak{J}$) of other constants as proper inductive constants; no induction is therefore performed on them.

Remark 5.10. *Nested induction is not dealt with directly with our approach, only by introducing cuts with the lemma mechanism (Section 5.3). We found tracking the dependencies on the outer induction from within the inner induction was far too complicated even with our approach; we would need a QBF with quantifier alternation of depth $2n$ to perform nested induction of depth n, because each nested inductive proof would depend on choices made in outer inductions, including the choice of coverset members.*

Example 5.15 (Nested Induction through Lemmas)**.** *let* $list \stackrel{\mathrm{def}}{=} nat :: list \mid []$ *be the type of lists of natural numbers,* $\mathfrak{l} : list$, *and* $p : nat \to o$, $q : list \to o$. *Assume* $p(0)$, $\forall n : nat.\ p(n) \Rightarrow p(s(n))$, $q([])$ *and* $\forall n\ nat : l.\ listp(n) \wedge q(l) \Rightarrow q(n :: l)$. *To prove* $\forall l.\ q(l)$, *we introduce* $\mathfrak{l} : list$, *with* $\kappa(\mathfrak{l}) = \{[], \mathfrak{n} :: \mathfrak{l}'\}$, *and perform the following derivation for the recursive case.*

$$
\cfrac{
  \cfrac{
    \cfrac{\neg q(\mathfrak{l}) \qquad \mathfrak{l} \simeq \mathfrak{n} :: \mathfrak{l}' \leftarrow \lVert \mathfrak{l} \simeq \mathfrak{n} :: \mathfrak{l}' \rVert}{\neg q(\mathfrak{n} :: \mathfrak{l}') \leftarrow \lVert \mathfrak{l} \simeq \mathfrak{n} :: \mathfrak{l}' \rVert}\ (\mathsf{Sup}) \qquad \neg p(x) \vee \neg q(y) \vee q(x :: y)
  }{\neg p(\mathfrak{n}) \vee \neg q(\mathfrak{l}') \leftarrow \lVert \mathfrak{l} \simeq \mathfrak{n} :: \mathfrak{l}' \rVert}\ (\mathsf{Sup})
}{
  \cfrac{\neg p(\mathfrak{n}) \leftarrow \neg \lVert p(\mathfrak{n}) \rVert \qquad \neg q(\mathfrak{l}) \leftarrow \neg \lVert q(\mathfrak{l}) \rVert}{\lVert \mathfrak{l} \simeq \mathfrak{n} :: \mathfrak{l}' \rVert \rightarrowtail \neg \lVert p(\mathfrak{n}) \rVert \sqcup \neg \lVert q(\mathfrak{l}) \rVert}
}\ (\mathsf{ASplit})
$$

As already explained, $\mathfrak{n} : nat$ *is not candidate for induction — the problem is that induction on* $\mathfrak{n}$ *should only be performed in models where* $\lVert \mathfrak{l} \simeq \mathfrak{n} :: \mathfrak{l}' \rVert$ *is valued to 1 — so we seem to be unable to solve the case* $\neg p(\mathfrak{n}) \leftarrow \lVert p(\mathfrak{n}) \rVert$. *However, we can "guess" the lemma* $\forall n.\ p(n)$ *and prove it by introducing a fresh constant* $\mathfrak{m}$ *and the clauses* $p(x) \leftarrow \lVert \forall n.\ p(n) \rVert$ *(trivially closes the branch) and* $\neg p(\mathfrak{m}) \leftarrow \neg \lVert \forall n.\ p(n) \rVert$.

The formula for induction on multiple constants $\mathfrak{i}_1, \ldots, \mathfrak{i}_n$ is basically a conjunction of the individual formulas for $\mathfrak{i}_k$, $k = 1 \ldots n$. The formula F is detailed in Figure 5.5. We will discuss its transformation to quantified CNF in Section 5.6.2, and the possibility of using incremental solving many QBF solvers provide to avoid re-checking the whole formula every time it changes. At this point, we have seen new mechanisms to deal with inductive reasoning, first with one clause context only, then with any finite subset of $S_{\mathrm{cand}}(\mathfrak{i})$ with potentially several inductive constants.

$$
\begin{aligned}
F &\stackrel{\mathrm{def}}{=} \exists_{a \in S_{\mathrm{atoms}}} a \prod_{\mathfrak{i} \in \mathfrak{J}} F_{\mathfrak{i}} \\
F_{\mathfrak{i}} &\stackrel{\mathrm{def}}{=} \forall_{C[\diamond] \in S_{\mathrm{cand}}(\mathfrak{i})} \lVert C[\diamond] \in S_{\min}(\mathfrak{i}) \rVert \\
&\quad \exists_{t \in \kappa(\mathfrak{i})} \lVert \mathfrak{i} \simeq t \rVert \\
&\quad \exists_{C[\diamond] \in S_{\mathrm{cand}}(\mathfrak{i})} \lVert \mathrm{init}(C[\diamond], \mathfrak{i}) \rVert \\
&\quad \exists_{t', \mathrm{sub}(t', \mathfrak{i}), C[\diamond] \in S_{\mathrm{cand}}(\mathfrak{i})} \lVert \mathrm{minimal}(C[\diamond], \mathfrak{i}, t') \rVert \\
&\quad \left(\prod_{x \in S_{\mathrm{constraints}}} x \right) \sqcap \left(\mathrm{empty}(\mathfrak{i}) \sqcup \bigsqcup_{t \in \kappa(\mathfrak{i})} \lVert \mathfrak{i} \simeq t \rVert \sqcap \mathrm{minimal}(\mathfrak{i}, t) \right) \\
\mathrm{empty}(\mathfrak{i}) &\stackrel{\mathrm{def}}{=} \prod_{C[\diamond] \in S_{\mathrm{cand}}(\mathfrak{i})} \neg \lVert C[\diamond] \in S_{\min}(\mathfrak{i}) \rVert \\
\mathrm{minimal}(\mathfrak{i}, t) &\stackrel{\mathrm{def}}{=} \prod_{t' \lhd t, \mathrm{sub}(t', \mathfrak{i})} \bigsqcup_{C[\diamond] \in S_{\mathrm{cand}}(\mathfrak{i})} \left(\begin{array}{c} \lVert C[\diamond] \in S_{\min}(\mathfrak{i}) \rVert \sqcap \\ \lVert \mathrm{minimal}(C[\diamond], \mathfrak{i}, t') \rVert \end{array} \right)
\end{aligned}
$$

Figure 5.5: QBF for Induction on Multiple Constants

5.4.7 Examples and Further Discussion

To help the reader acquire more intuition about how the different mechanisms described above combine into one procedure, we present an example on natural numbers that requires a conjunctive induction hypothesis.

Example 5.16. *We define $p, q : \text{nat} \to o$ and assume $p(0) \vee q(0)$ and $\forall n : \text{nat}. \ (p(n) \vee q(n)) \Rightarrow (p(s(n)) \vee q(s(n)))$. To prove $\forall n : \text{nat}. \ (p(n) \vee q(n))$, Superposition starts with the clauses[15] $\neg p(\mathfrak{n}) \leftarrow \text{input}$ and $\neg q(\mathfrak{n}) \leftarrow \text{input}$. A natural cover set for $\mathfrak{n}$ is $\{0, s(\mathfrak{n}')\}$. The clause contexts $C_p[\diamond] \overset{\text{def}}{=} \neg p(\diamond)$ and $C_q[\diamond] \overset{\text{def}}{=} \neg q(\diamond)$ are extracted from the initial clauses, then boolean literals $\lVert \text{init}(C_p[\diamond], \mathfrak{n}) \rVert$ and $\lVert \text{init}(C_q[\diamond], \mathfrak{n}) \rVert$ are added to $S_{\text{constraints}}$. We then define the boolean atoms*

$$
\begin{aligned}
hyp(C_p) \quad &\overset{\text{def}}{=} \quad \lVert \text{init}(C_p[\diamond], \mathfrak{n}) \rVert \sqcap \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \\
min(C_p) \quad &\overset{\text{def}}{=} \quad \lVert \text{minimal}(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \sqcap \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert
\end{aligned}
$$

to keep the proof readable (same for C_q). Of course, $\neg hyp(C_p)$ is short for $\neg \lVert \text{init}(C_p[\diamond], \mathfrak{n}) \rVert \sqcup \neg \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert$, etc.

$$
\frac{p(0) \vee q(0)}{p(0) \leftarrow \lVert p(0) \rVert \qquad q(0) \leftarrow \lVert q(0) \rVert \qquad \lVert p(0) \rVert \sqcup \lVert q(0) \rVert} \ (\mathsf{ASplit})
$$

$$
\frac{\dfrac{\neg p(\mathfrak{n}) \leftarrow hyp(C_p) \qquad \mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}{\neg p(0) \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap hyp(C_p)} \ (\mathsf{Sup}) \qquad p(0) \leftarrow \lVert p(0) \rVert}{\dfrac{\bot \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap \lVert p(0) \rVert \sqcap hyp(C_p)}{\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert p(0) \rVert \sqcup \neg hyp(C_p)} \ (\mathsf{A}\bot)} \ (\mathsf{Sup})
$$

$$
\frac{\dfrac{\neg q(\mathfrak{n}) \leftarrow hyp(C_q) \qquad \mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}{\neg q(0) \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap hyp(C_q)} \ (\mathsf{Sup}) \qquad q(0) \leftarrow \lVert q(0) \rVert}{\dfrac{\bot \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap \lVert q(0) \rVert \sqcap hyp(C_q)}{\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert q(0) \rVert \sqcup \neg hyp(C_q)} \ (\mathsf{A}\bot)} \ (\mathsf{Sup})
$$

Now, proceeding on to the recursive case, assuming that $C_p[\diamond]$ is within $S_{min}(\mathfrak{n})$ and that it's the minimality witness:

$$
\frac{\dfrac{\neg p(\mathfrak{n}) \leftarrow hyp(C_p) \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{\neg p(s(\mathfrak{n}')) \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_p)} \ (\mathsf{Sup}) \qquad \neg p(n) \vee p(s(n)) \vee q(s(n))}{\dfrac{\neg p(\mathfrak{n}') \vee q(s(\mathfrak{n}')) \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_p)}{\dfrac{\neg p(\mathfrak{n}') \leftarrow \neg \lVert p(\mathfrak{n}') \rVert \qquad q(s(\mathfrak{n}')) \leftarrow \lVert q(s(\mathfrak{n}')) \rVert}{\lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_p) \rightarrowtail \neg \lVert p(\mathfrak{n}') \rVert \sqcup \lVert q(s(\mathfrak{n}')) \rVert} \ (\mathsf{ASplit})}} \ (\mathsf{Sup})
$$

The first case is easy:

$$
\frac{\neg p(\mathfrak{n}') \leftarrow \neg \lVert p(\mathfrak{n}') \rVert \qquad p(\mathfrak{n}') \leftarrow min(C_p)}{\dfrac{\bot \leftarrow \neg \lVert p(\mathfrak{n}') \rVert \sqcap min(C_p)}{\lVert p(\mathfrak{n}') \rVert \sqcup \neg min(C_p)} \ (\mathsf{A}\bot)} \ (\mathsf{Sup})
$$

The second case, $q(s(\mathfrak{n}'))$, works assuming that $C_q[\diamond]$ is also part of $S_{min}(\mathfrak{n})$:

$$
\frac{\dfrac{\neg q(\mathfrak{n}) \leftarrow hyp(C_q) \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert}{\neg q(s(\mathfrak{n}')) \leftarrow \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_q)} \ (\mathsf{Sup}) \qquad q(s(\mathfrak{n}')) \leftarrow \lVert q(s(\mathfrak{n}')) \rVert}{\dfrac{\bot \leftarrow hyp(C_q) \sqcap \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap \lVert q(s(\mathfrak{n}')) \rVert}{\neg hyp(C_q) \sqcup \neg \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcup \neg \lVert q(s(\mathfrak{n}')) \rVert} \ (\mathsf{A}\bot)} \ (\mathsf{Sup})
$$

---

[15]With hindsight, a non-clausal prover could see that the subformula $p(\diamond) \vee q(\diamond)$ is the right induction hypothesis. But that's counting on luck a bit too much.

The proof when $min(C_q)$ is assumed is exactly the same, albeit symmetrical. The QBF after adding all constraints attached to $\bot$ follows. It is unsatisfiable in the case where $\lVert minimal(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert$ and $\lVert minimal(C_q[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert$ are both valued to 1 (thus making $hyp(C_p)$, $hyp(C_q)$, $min(C_p)$ and $min(C_q)$ true).

$$
\begin{aligned}
F_{\mathfrak{n}} \;\overset{def}{=}\;\; & \forall \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \; \lVert C_q[\diamond] \in S_{min}(\mathfrak{n}) \rVert \\
& \exists \lVert \mathfrak{n} \simeq 0 \rVert \; \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \\
& \exists \lVert init(C_p[\diamond], \mathfrak{n}) \rVert \; \lVert init(C_q[\diamond], \mathfrak{n}) \rVert \\
& \exists \lVert minimal(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \; \lVert minimal(C_q[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \\
& \exists hyp(C_p) \; min(C_p) \; hyp(C_q) \; min(C_q) \\
& constraints \sqcap \big(empty \sqcup \lVert \mathfrak{n} \simeq 0 \rVert \sqcup (\lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap minimal\,(s(\mathfrak{n}')))\big)
\end{aligned}
$$

$$
constraints \;\overset{def}{=}\;
\left\{
\begin{aligned}
& \lVert \mathfrak{n} \simeq 0 \rVert \oplus \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \\
& \lVert init(C_p[\diamond], \mathfrak{n}) \rVert \\
& \lVert init(C_q[\diamond], \mathfrak{n}) \rVert \\
& \lVert p(0) \rVert \sqcup \lVert q(0) \rVert \\
& \neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert p(0) \rVert \sqcup \neg hyp(C_p) \\
& \neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert q(0) \rVert \sqcup \neg hyp(C_q) \\
& \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_p) \dashrightarrow \neg \lVert p(\mathfrak{n}') \rVert \sqcup \lVert q(s(\mathfrak{n}')) \rVert \\
& \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_q) \dashrightarrow \neg \lVert q(\mathfrak{n}') \rVert \sqcup \lVert p(s(\mathfrak{n}')) \rVert \\
& \neg hyp(C_p) \sqcup \neg \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcup \neg \lVert p(s(\mathfrak{n}')) \rVert \\
& \neg hyp(C_q) \sqcup \neg \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcup \neg \lVert q(s(\mathfrak{n}')) \rVert \\
& \lVert p(\mathfrak{n}') \rVert \sqcup \neg min(C_p) \\
& \lVert q(\mathfrak{n}') \rVert \sqcup \neg min(C_q)
\end{aligned}
\right.
$$

$$
empty \;\overset{def}{=}\; \neg \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \neg \lVert C_q[\diamond] \in S_{min}(\mathfrak{n}) \rVert
$$

$$
minimal\,(s(\mathfrak{n}')) \;\overset{def}{=}\;
\left(\begin{aligned} & \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \\ & \lVert minimal(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \end{aligned} \right)
\sqcup
\left(\begin{aligned} & \lVert C_q[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \\ & \lVert minimal(C_q[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \end{aligned} \right)
$$

Now, we come back to the counter-example 5.13. We will see that what makes this case solvable is not the ability of our second encoding to use a conjunction of clause contexts as the inductive formula, but its ability to try several clause contexts without committing to a specific one.

Example 5.17 (Parallel Induction). *We define $p, q : nat \to o$ and assume $p(0)$, $q(0)$ and $\forall n : nat.\ (p(n) \wedge q(n)) \Rightarrow (p(s(n)) \wedge q(s(n)))$. To prove $\forall n : nat.\ p(n)$, Superposition starts with the clause $\neg p(\mathfrak{n}) \leftarrow input$. We use the same classic cover set for $\mathfrak{n}$, that is, $\{0, s(\mathfrak{n}')\}$. The clause context $C_p[\diamond] \overset{def}{=} \neg p(\diamond)$ is extracted from the initial clauses, then the boolean literal $\lVert init(C_p[\diamond], \mathfrak{n}) \rVert$ is added to $S_{constraints}$. We define*

$$
hyp(C_p) \overset{def}{=} \lVert init(C_p[\diamond], \mathfrak{n}) \rVert \sqcap \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert
$$

and

$$
min(C_p) \overset{def}{=} \lVert minimal(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \sqcap \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert
$$

to keep the proof readable. First, the base case is easy:

$$
\dfrac{\dfrac{\neg p(\mathfrak{n}) \leftarrow hyp(C_p) \qquad \mathfrak{n} \simeq 0 \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert}{\neg p(0) \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap hyp(C_p)}\text{(Sup)} \qquad p(0)}{\dfrac{\bot \leftarrow \lVert \mathfrak{n} \simeq 0 \rVert \sqcap hyp(C_p)}{\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg hyp(C_p)}\text{(A}\bot\text{)}}\text{(Sup)}
$$

Now, proceeding on to the recursive case, assuming that $C_p[\diamond]$ is within $S_{min}(\mathfrak{n})$ and that it's the minimality witness:

$$\dfrac{\dfrac{\neg p(\mathfrak{n}) \leftarrow hyp(C_p) \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \|\mathfrak{n} \simeq s(\mathfrak{n}')\|}{\neg p(s(\mathfrak{n}')) \leftarrow \|\mathfrak{n} \simeq s(\mathfrak{n}')\| \sqcap hyp(C_p)} \text{(Sup)} \qquad \neg p(n) \vee \neg q(n) \vee p(s(n))}{\dfrac{\dfrac{\neg p(\mathfrak{n}') \vee \neg q(\mathfrak{n}') \leftarrow \|\mathfrak{n} \simeq s(\mathfrak{n}')\| \sqcap hyp(C_p)}{\neg p(\mathfrak{n}') \leftarrow \neg\|p(\mathfrak{n}')\| \qquad \neg q(\mathfrak{n}') \leftarrow \neg\|q(\mathfrak{n}')\|}}{\|\mathfrak{n} \simeq s(\mathfrak{n}')\| \sqcap hyp(C_p) \rightarrow \neg\|p(\mathfrak{n}')\| \sqcup \neg\|q(\mathfrak{n}')\|} \text{(ASplit)}} \text{(Sup)}$$

The occurrence of the clause $\neg p(\mathfrak{n}') \vee \neg q(\mathfrak{n}')$, before it is simplified by an AVATAR split, suggests to add the context $C_{pq}[\diamond] \overset{def}{=} \neg p(\diamond) \vee \neg q(\diamond)$. This context is initialized (subsumed) by $\neg p(\mathfrak{n})$, and we will proceed with it from now on, forgetting about C_p (which is not strong enough an induction hypothesis to prove itself).

$$\dfrac{\neg p(\mathfrak{n}) \vee \neg q(\mathfrak{n}) \leftarrow hyp(C_{pq})}{\neg p(\mathfrak{n}) \leftarrow \neg\|p(\mathfrak{n})\| \qquad \neg q(\mathfrak{n}) \leftarrow \neg\|q(\mathfrak{n})\| \qquad hyp(C_{pq}) \rightarrow \neg\|p(\mathfrak{n})\| \sqcup \neg\|q(\mathfrak{n})\|} \text{(ASplit)}$$

Base case *Let us go back to initialization, for $C_{pq}[\diamond]$ this time:*

$$\dfrac{p(0) \qquad \dfrac{\dfrac{\neg p(\mathfrak{n}) \leftarrow \neg\|p(\mathfrak{n})\| \qquad \mathfrak{n} \simeq 0 \leftarrow \|\mathfrak{n} \simeq 0\|}{\neg p(0) \leftarrow \|\mathfrak{n} \simeq 0\| \sqcap \neg\|p(\mathfrak{n})\|} \text{(Sup)}}{}}{\dfrac{\bot \leftarrow \|\mathfrak{n} \simeq 0\| \sqcap \neg\|p(\mathfrak{n})\|}{\neg\|\mathfrak{n} \simeq 0\| \sqcup \|p(\mathfrak{n})\|} \text{(A}\bot\text{)}} \text{(Sup)}$$

and

$$\dfrac{q(0) \qquad \dfrac{\dfrac{\neg q(\mathfrak{n}) \leftarrow \neg\|q(\mathfrak{n})\| \qquad \mathfrak{n} \simeq 0 \leftarrow \|\mathfrak{n} \simeq 0\|}{\neg q(0) \leftarrow \|\mathfrak{n} \simeq 0\| \sqcap \neg\|q(\mathfrak{n})\|} \text{(Sup)}}{}}{\dfrac{\bot \leftarrow \|\mathfrak{n} \simeq 0\| \sqcap \neg\|q(\mathfrak{n})\|}{\neg\|\mathfrak{n} \simeq 0\| \sqcup \|q(\mathfrak{n})\|} \text{(ASplit)}} \text{(Sup)}$$

Recursive Case *Very similar, but we can use the strengthening of $C_{pq}[\diamond]$; namely, $p(\mathfrak{n}') \leftarrow min(C_{pq})$ and $q(\mathfrak{n}') \leftarrow min(C_{pq})$.*

$$\dfrac{\dfrac{\neg p(\mathfrak{n}) \leftarrow \neg\|p(\mathfrak{n})\| \qquad \mathfrak{n} \simeq s(\mathfrak{n}') \leftarrow \|\mathfrak{n} \simeq s(\mathfrak{n}')\|}{\neg p(s(\mathfrak{n}')) \leftarrow \|\mathfrak{n} \simeq s(\mathfrak{n}')\| \sqcap \neg\|p(\mathfrak{n})\|} \text{(Sup)} \qquad \neg p(n) \vee \neg q(n) \vee p(s(n))}{\dfrac{\dfrac{\neg p(\mathfrak{n}') \vee \neg q(\mathfrak{n}') \leftarrow \|\mathfrak{n} \simeq s(\mathfrak{n}')\| \sqcap \neg\|p(\mathfrak{n})\|}{\neg p(\mathfrak{n}') \leftarrow \neg\|p(\mathfrak{n}')\| \qquad \neg q(\mathfrak{n}') \leftarrow \neg\|q(\mathfrak{n}')\|}}{\|\mathfrak{n} \simeq s(\mathfrak{n}')\| \sqcap \neg\|p(\mathfrak{n})\| \rightarrow \neg\|p(\mathfrak{n}')\| \sqcup \neg\|q(\mathfrak{n}')\|} \text{(ASplit)}} \text{(Sup)}$$

and its symmetric starting with $\neg q(\mathfrak{n}) \leftarrow \neg\|q(\mathfrak{n})\|$. From then, we perform the two split cases

$$\dfrac{\dfrac{\neg p(\mathfrak{n}') \leftarrow \neg\|p(\mathfrak{n}')\| \qquad p(\mathfrak{n}') \leftarrow min(C_{pq})}{\bot \leftarrow \neg\|p(\mathfrak{n}')\| \sqcap min(C_{pq})} \text{(Sup)}}{\|p(\mathfrak{n}')\| \sqcap \neg min(C_{pq})} \text{(A}\bot\text{)}$$

and

$$\dfrac{\dfrac{\neg q(\mathfrak{n}') \leftarrow \neg\|q(\mathfrak{n}')\| \qquad q(\mathfrak{n}') \leftarrow min(C_{pq})}{\bot \leftarrow \neg\|q(\mathfrak{n}')\| \sqcap min(C_{pq})} \text{(Sup)}}{\|q(\mathfrak{n}')\| \sqcap \neg min(C_{pq})} \text{(A}\bot\text{)}$$

The QBF follows. It is quite rich because of the numerous case splits performed in the proof. The formula is unsatisfiable in the case where $S_{min}(\mathfrak{n}) = \{C_{pq}[\diamond]\}$ because of the subset of the constraints named unsat-core *(also see Section 5.5.2).*

$$
\begin{aligned}
F_{\mathfrak{n}} \;\overset{def}{=}\; & \exists \lVert p(\mathfrak{n}) \rVert \, \lVert q(\mathfrak{n}) \rVert \, \lVert p(\mathfrak{n}') \rVert \, \lVert q(\mathfrak{n}') \rVert \\
& \forall \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \, \lVert Q_{pq}[\diamond] \in S_{min}(\mathfrak{n}) \rVert \\
& \exists \lVert \mathfrak{n} \simeq 0 \rVert \, \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \\
& \exists \lVert init(C_p[\diamond], \mathfrak{n}) \rVert \, \lVert init(C_{pq}[\diamond], \mathfrak{n}) \rVert \\
& \exists \lVert minimal(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \, \lVert minimal(C_{pq}[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \\
& \exists hyp(C_p) \; min(C_{pp}) \; hyp(C_q) \; min(C_{pq}) \\
& constraints \sqcap \big(empty \sqcup \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap minimal\,(s(\mathfrak{n}')) \big)
\end{aligned}
$$

$$
constraints \;\overset{def}{=}\;
\left\{
\begin{array}{l}
\lVert init(C_p[\diamond], \mathfrak{n}) \rVert \\
\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg hyp(C_p) \\
\lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap hyp(C_p) \rightarrow \neg \lVert p(\mathfrak{n}') \rVert \sqcup \neg \lVert q(\mathfrak{n}') \rVert \\
\lVert \mathfrak{n} \simeq 0 \rVert \oplus \lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \\
\lVert init(C_{pq}[\diamond], \mathfrak{n}) \rVert \\
hyp(C_{pq}) \rightarrow \neg \lVert p(\mathfrak{n}) \rVert \sqcup \neg \lVert q(\mathfrak{n}) \rVert \\
\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \lVert p(\mathfrak{n}) \rVert \\
\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \lVert q(\mathfrak{n}) \rVert \\
\lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap \neg \lVert p(\mathfrak{n}) \rVert \rightarrow \neg \lVert p(\mathfrak{n}') \rVert \sqcup \neg \lVert q(\mathfrak{n}') \rVert \\
\lVert \mathfrak{n} \simeq s(\mathfrak{n}') \rVert \sqcap \neg \lVert q(\mathfrak{n}) \rVert \rightarrow \neg \lVert p(\mathfrak{n}') \rVert \sqcup \neg \lVert q(\mathfrak{n}') \rVert \\
\lVert p(\mathfrak{n}') \rVert \sqcup \neg min(C_{pq}) \\
\lVert q(\mathfrak{n}') \rVert \sqcup \neg min(C_{pq})
\end{array}
\right\} unsat core
$$

$$
empty \;\overset{def}{=}\; \neg \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \neg \lVert C_{pq}[\diamond] \in S_{min}(\mathfrak{n}) \rVert
$$

$$
minimal\,(s(\mathfrak{n}')) \;\overset{def}{=}\;
\left(\begin{array}{l} \lVert C_p[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \\ \lVert minimal(C_p[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \end{array} \right)
\sqcup
\left(\begin{array}{l} \lVert C_{pq}[\diamond] \in S_{min}(\mathfrak{n}) \rVert \sqcap \\ \lVert minimal(C_{pq}[\diamond], \mathfrak{n}, \mathfrak{n}') \rVert \end{array} \right)
$$

Search Space

The boolean solver, as discussed in the Section 2.5 on AVATAR, acts as an explorer of the global search space. Whenever a toplevel choice has to be made — be it a regular boolean split on a clause, or whether a lemma introduced as in Section 5.3 is valid — the solver takes an arbitrary decision, to be corrected only in the case it leads to a contradiction. For instance, deciding that the lemma $\lVert \forall n.\ n + 0 \simeq n \rVert$ should be valued to 0 will yield a conflict once the lemma is inductively proved; from then on, the solver will have to value it to 1 and the lemma will be usable.

We see here that several sub-parts of the search space can communicate through boolean constraints: in Example 5.11, the following proofs are carried separately: (i) the clause $n + 0 \simeq n \leftarrow \lVert \forall n.\ n + 0 \simeq n \rVert$ is used to disprove the case $\mathfrak{n} \simeq 0$, eventually constraining $\neg \lVert \mathfrak{n} \simeq 0 \rVert \sqcup \neg \lVert \forall n.\ n + 0 \simeq n \rVert$; (ii) the proof of $\lVert \forall n.\ n + 0 \simeq n \rVert$ eventually succeeds and adds $\lVert \forall n.\ n + 0 \simeq n \rVert$ to the set of boolean constraints. Although those two subproofs don't interact directly, together they prune the branch of the search space in which $\lVert \mathfrak{n} \simeq 0 \rVert$ is valued to 1.

Limitations

Although the QBF encoding is strictly more powerful than the direct encoding in AVATAR from Section 5.2, it also has some limitations. First, lemmas are still necessary, which makes Example 5.12 still relevant. Then, inductive properties that are not a conjunction of clause contexts that can be extracted from $\mathcal{N}$ cannot be solved. Last, our framework only deals with structural induction, not well-founded induction in general.

5.5 Reconstructing Proofs

We have seen two extensions of Superposition that can handle some inductive reasoning. However, those extensions are somewhat subtle and their implementation, as Section 5.6 can show, is not trivial. An interesting way to increase the trust humans can have in such proofs is to have the prover output, not a single "yes/no" answer, but a detailed trace of its reasoning; what we called earlier a *trace* of the proof. Such traces, depending on their level of detail, can be read by a human, or checked by a dedicated tool (possibly after some encoding).

5.5.1 SAT resolutions proofs for Inductive Strengthening

Let us first focus on the inductive strengthening technique described in Section 5.2; the one that deals with one clause context at a time. Since it uses a regular SAT solver, like AVATAR, and succeeds when the solver proves the unsatisfiability of the set of constraints, a boolean resolution proof can be obtained[16]. This proof is a DAG of boolean clauses whose leaves can have the following forms:

- $\neg l_1 \sqcup \ldots \sqcup \neg l_n$ where the constraint comes from a clause $\bot \leftarrow l_1 \sqcap \ldots \sqcap l_n$ and each l_i is either $\lfloor\! C_i \!\rfloor$ (a clause component) or $\lfloor\! i \simeq t \!\rfloor$ for some $t \in \kappa(i)$;
- $\Gamma \twoheadrightarrow \lfloor\! C_1 \!\rfloor \sqcup \ldots \sqcup \lfloor\! C_n \!\rfloor$ where the constraint comes from the splitting of the clause $C_1 \vee \ldots \vee C_n \leftarrow \Gamma$;
- $\bigoplus_{t \in \kappa(i)} \lfloor\! i \simeq t \!\rfloor$ for some inductive constant i.

In any case, we can rebuild a regular Superposition proof (along with some additional axioms that are specific to inductive reasoning).

Example 5.18 (Simple Boolean Resolution Proof). *Let us build the resolution/Superposition proof for the simple problem in Example 5.7. We glue together the Superposition proof to a resolution proof of* 0 *obtain from the (unsatisfiable) boolean constrains:*

$$
\cfrac{
\lfloor\! t \simeq E \!\rfloor \oplus \lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor
}{
\cfrac{\lfloor\! t \simeq E \!\rfloor \sqcup \lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor}{
\lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor
}
\quad
\cfrac{
\cfrac{
\cfrac{\neg q(t) \quad t \simeq E \leftarrow \lfloor\! t \simeq E \!\rfloor}{\neg q(E) \leftarrow \lfloor\! t \simeq E \!\rfloor} \text{(Sup)} \quad q(E)
}{
\cfrac{\bot \leftarrow \lfloor\! t \simeq E \!\rfloor}{\neg \lfloor\! t \simeq E \!\rfloor} \text{(A}\bot\text{)}
} \text{(Sup)}
}{}
}
$$

$$\vdots\ \pi$$

$$
\cfrac{
\cfrac{
\cfrac{\neg q(t) \quad t \simeq N(t_l, a, t_r) \leftarrow \lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor}{\neg q(N(t_l, a, t_r)) \leftarrow \lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor} \text{(Sup)} \quad \neg p(x) \vee \neg q(l) \vee \neg q(r) \vee q(N(l, x, r))
}{
\cfrac{\neg p(a) \vee \neg q(t_l) \vee \neg q(t_r) \leftarrow \lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor}{\neg p(a) \leftarrow \neg \lfloor\! p(a) \!\rfloor \quad \neg q(t_l) \leftarrow \neg \lfloor\! q(t_l) \!\rfloor \quad \neg q(t_r) \leftarrow \neg \lfloor\! q(t_r) \!\rfloor} \text{(ASplit)}
} \text{(Sup)}
}{
\lfloor\! t \simeq N(t_l, a, t_r) \!\rfloor \twoheadrightarrow \neg \lfloor\! p(a) \!\rfloor \sqcup \neg \lfloor\! q(t_l) \!\rfloor \sqcup \neg \lfloor\! q(t_r) \!\rfloor
}
$$

$$\vdots\ \pi_m$$

---

[16] Not all SAT solvers actually give access to a resolution proof, but at least it is theoretically possible.

$$\vdots \pi_m$$

$$
\frac{\neg q(\mathsf{t}_l) \leftarrow \neg \lVert q(\mathsf{t}_l) \rVert \qquad q(\mathsf{t}_l) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\dfrac{\perp \leftarrow \neg \lVert q(\mathsf{t}_l) \rVert \sqcap \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\neg \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \sqcup \lVert q(\mathsf{t}_l) \rVert}\ \text{(A}\perp\text{)}}\ \text{(Sup)} \qquad \vdots \pi \quad \overline{\lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}
$$

$$\lVert q(\mathsf{t}_l) \rVert$$

$$\vdots \pi_l$$

$$\vdots \pi_m$$

$$
\frac{\neg q(\mathsf{t}_r) \leftarrow \neg \lVert q(\mathsf{t}_r) \rVert \qquad q(\mathsf{t}_r) \leftarrow \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\dfrac{\perp \leftarrow \neg \lVert q(\mathsf{t}_r) \rVert \sqcap \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}{\neg \lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert \sqcup \lVert q(\mathsf{t}_r) \rVert}\ \text{(A}\perp\text{)}}\ \text{(Sup)} \qquad \vdots \pi \quad \overline{\lVert \mathsf{t} \simeq N(\mathsf{t}_l, a, \mathsf{t}_r) \rVert}
$$

$$\lVert q(\mathsf{t}_r) \rVert$$

$$\vdots \pi_r$$

$$\vdots \pi_m$$

$$
\vdots \pi \quad \vdots \pi_m \qquad \frac{\neg p(a) \leftarrow \neg \lVert p(a) \rVert \qquad p(x)}{\dfrac{\perp \leftarrow \neg \lVert p(a) \rVert}{\lVert p(a) \rVert}\ \text{(A}\perp\text{)}}\ \text{(Sup)} \quad \vdots \pi_l \quad \vdots \pi_r
$$

$$\neg \lVert p(a) \rVert \sqcup \neg \lVert q(\mathsf{t}_l) \rVert \sqcup \neg \lVert q(\mathsf{t}_r) \rVert$$

$$0$$

Remark 5.11 (Q-Resolution). *Where SAT problems can always be solved using boolean resolution, QBF problems can be solved using* Q-resolution *[KKF95]. The technique developed above could be adapted to Q-resolution to glue together proofs operating on portions of the search space.*

5.5.2   QBF resolution proofs using UNSAT-cores

Some QBF solvers, such as Depqbf [LB10], provide mechanisms that allow to extract, from a formula known to be unsatisfiable, a subset of the clauses known as *UNSAT-core*. This subset is unsatisfiable by itself, and none of its own strict subsets is unsatisfiable. Such an UNSAT-core filters out clause contexts that played no role in the proof. Since the creation of clause contexts is heuristic, in practice, many useless or irrelevant contexts are created and do not participate in the proof.

Given an UNSAT-core, i.e. a set of boolean clauses, we compute the set L of boolean literals of the form $\lVert C[\diamond] \in S_{\min}(\mathsf{i}) \rVert$ involved in the set. Then, we can do a regular inductive proof (or a SAT-solver based proof) by instantiating the induction principle on the formula $F[x] \overset{\text{def}}{=} \bigwedge_{\lVert C[\diamond] \in S_{\min}(\mathsf{i}) \rVert \in L} C[x]$. The QBF-based induction would act as a (semi-)procedure that finds the appropriate inductive formula before the real proof proceeds.

5.6   Implementation in Zipperposition

We developed a basic implementation in Zipperposition. It can solve some problems (including the commutativity of +, see Example 5.11) using the SAT solver MSat[17], or the QBF solvers Depqbf [LB10] or Quantor [Bie04][18]. Implementing the successive versions of inductive reasoning in Zipperposition played a central role in designing the calculus as presented here through numerous design steps[19].

5.6.1   Interfacing to Boolean Solvers

Zipperposition communicates with boolean solvers through an abstract interface detailed in Figure 5.6. This interface contains two module type, `SAT` and `QBF`, that respectively wrap a SAT-solver and a QBF-solver. By virtues of subtyping, a QBF-solver can also be used as a SAT-solver.

Both kinds of solver provide a function `add_clause` to add clauses with an optional integer tag that is used for reporting the UNSAT-core, and a helper function `add_form` that converts its argument to CNF before calling `add_clause` on every resulting clause. Once every clause of $S_{constraints}$ has been added to the solver, a call to `check` returns either `Sat` or `Unsat`. Depending on this value, `valuation` can be called to obtain the valuation of a literal in the model (in the case of QBF, only variables that belong to the outer, existentially quantified, scope have a valuation), or `unsat_core` can be called — if the solver provides it. We did not exploit resolution proofs from solvers that could provide it. Incremental checking, an important technique we will discuss in Section 5.6.2, is made possible through `save` and `restore`. The function `save` pushes the current state of the solver (i.e., the set of clauses, roughly) onto a stack, and returns the stack height; `restore` pops states from the stack down to the given height and copies the corresponding saved state back into the solver[20]. QBF solvers expose additional functions to quantify literals and create new scopes (from outermost to innermost, starting with `level0` which is the prenex existential scope, for which `valuation` is defined).

First-class modules are used to choose among several candidate solvers at runtime. Each solver is annotated with its "strength" (see the type α `solver`) — a heuristic value indicated how powerful the solver is — so that the stronger available solver is selected. This way, if a particularly strong solver is added using the plugin system, it will be used over weaker ones.

5.6.2   Reducing the QBF to CNF

We do not expand on the subject of implementing strengthening any further, as every constraint is already a boolean clause. However, the QBF in Figure 5.5 is a different story.

Definition 5.16 (Incremental Solving). *A boolean solver is* incremental *if it can solve a series of (conjunctive) formula sets $F_1, F_2, \ldots, F_n$ where $F_i \subseteq F_{i+1}$ for $1 \le i < n$ more efficiently than by solving each F_i independently.*

The interface in Figure 5.6 exposes a type `save_level` and two functions, `save` and `restore`. Given this interface to an incremental solver, the series of formulas $F \overset{\text{def}}{=} (F_1, F_2, \ldots F_n)$ can be solved by the following piece of pseudo-code. The function `solve` is given a list $[F_1, F_2 \setminus F_1, \ldots, F_n \setminus F_{n-1}]$ and outputs, for each F_i, `Sat` or `Unsat` depending on whether F_i is satisfiable or not.

---

[17] A small SAT-solver in OCaml that can output resolution proofs, see `https://github.com/Gbury/mSAT`.

[18] See `http://fmv.jku.at/quantor/`.

[19] Starting by trying to adapt directly the work from Kersani and Peltier [KP13], then trying to use cyclic terms to represent fixpoints, then several versions based on the QBF solver where each iteration would delegate more work to the solver than the previous one...

[20] Of course, for solvers that natively handle incrementality, this is much more efficient than a naive copy of the state. The solver itself provides a stack API.

```
type result = Sat | Unsat

(** One instance of boolean solver. *)
module type SAT = sig
  val add_clause : ?tag:int → lit list → unit
  val add_form : ?tag:int → formula → unit (* will be reduced to CNF *)
  val check : unit → result (* current state satisfiable? *)
  val valuation : lit → bool  (* if satisfiable, access model *)
  val unsat_core : (unit → int list) option
  type save_level  (* for incrementality *)
  val root_save_level : save_level
  val save : unit → save_level (* save current state *)
  val restore : save_level → unit (* restore to given state *)
end

type quantifier = Forall | Exists

module type QBF = sig
  include SAT (* Can use check, save, valuation, etc. *)
  type quant_level = private int (* Quantification depth *)
  val level0 : quant_level (* outermost ∃ level *)
  val push : quantifier → lit list → quant_level (* new innermost scope *)
  val quantify_lit : quant_level → lit → unit
end

type α solver = {
  create: unit → α; (** build a new instance *)
  strength : int; (** used to favor better solvers *)
}

type sat_solver = (module SAT) solver
type qbf_solver = (module QBF) solver

val sat_of_qbf : qbf_solver → sat_solver
```

Figure 5.6: Abstract Interface for Boolean Solvers

```ocaml
module Solver : SAT  (* a solver instance *)

let rec solve = function
| [] → []
| diff :: tail →
    List.iter Solver.add_clause diff; (* add Fᵢ\Fᵢ₋₁ *)
    let res = Solver.check () in
    res :: solve tail
```

In practice, as we will see soon, the real list of formulas has the more general shape $F_1 \uplus G_1, F_2 \uplus G_2, \ldots, F_n \uplus G_n$ where $F_i \subseteq F_{i+1}$ for $1 \le i < n$. Pure incremental solving does not work, because the sets G_i are arbitrary; however, the `restore` function is designed expressly for this case, as the following function `solve'` shows. This time the function is given a list $[(F_1, G_1), (F_2 \setminus F_1, G_2) \ldots, (F_n \setminus F_{n-1}), G_n]$ and maps each tuple $(F_i \setminus F_{i-1}, G_i)$ into a value of type `result` depending on the satisfiability of $F_i \uplus G_i$.

```ocaml
let rec solve' = function
| [] → []
| (diff_f, g) :: tail →
    List.iter Solver.add_clause diff_f; (* add Fᵢ\Fᵢ₋₁ *)
    let level = Solver.save () in (* add Gᵢ *)
    List.iter Solver.add_clause g;
    let res = Solver.check () in
    Solver.restore level; (* forget about Gᵢ *)
    res :: solve' tail
```

Incremental Reduction to CNF   The proof procedure revolves around the saturation loop (the "given clause algorithm" described in Section 2.4) and generates a series of QBF $Q_1, Q_2, \ldots Q_n$. Once reduced to CNF, each Q_k can be decomposed into $F_k \uplus G_k$ as explained above. For the sake of efficiency, we should strive to make G_k as small as possible. The formula Q_k after k steps of saturation has the form[21]:

$$
\begin{aligned}
Q_k &\overset{\text{def}}{=} \exists_{a \in S_{\text{atoms}(k)}} a \\
&\quad \forall_{i \in \mathfrak{I}, C[\diamond] \in S_{\text{cand}(i)_{(k)}}} \lfloor C[\diamond] \in S_{\min}(i) \rfloor \\
&\quad \exists_{i \in \mathfrak{I}, t \in \kappa(i)} \lfloor i \simeq t \rfloor \\
&\quad \exists_{i \in \mathfrak{I}, C[\diamond] \in S_{\text{cand}(i)_{(k)}}} \lfloor \text{init}(C[\diamond], i) \rfloor \\
&\quad \exists_{i \in \mathfrak{I}, t', \text{sub}(t', i), C[\diamond] \in S_{\text{cand}(i)_{(k)}}} \lfloor \text{minimal}(C[\diamond], i, t') \rfloor \\
&\quad \left(\textstyle\prod_{x \in S_{\text{constraints}(k)}} x \right) \sqcap \textstyle\prod_{i \in \mathfrak{I}} \left(\text{empty}(i) \sqcup \bigsqcup_{t \in \kappa(i)} \lfloor i \simeq t \rfloor \sqcap \text{minimal}(i, t) \right) \\[4pt]
\text{empty}(i) &\overset{\text{def}}{=} \textstyle\prod_{C[\diamond] \in S_{\text{cand}(i)_{(k)}}} \neg \lfloor C[\diamond] \in S_{\min}(i) \rfloor \\[4pt]
\text{minimal}(i, t) &\overset{\text{def}}{=} \textstyle\prod_{t' \lhd t, \text{sub}(t', i)} \bigsqcup_{C[\diamond] \in S_{\text{cand}(i)_{(k)}}} \left(\begin{array}{l} \lfloor C[\diamond] \in S_{\min}(i) \rfloor \sqcap \\ \lfloor \text{minimal}(C[\diamond], i, t') \rfloor \end{array} \right)
\end{aligned}
$$

where $S_{\text{atoms}(k)}, S_{\text{cand}(i)_{(k)}}$ and $S_{\text{constraints}(k)}$ depend on k.

Local Constraints   First, boolean formulas in $S_{\text{constraints}}$ are already in clausal form: they are either boolean splits (including splits on $\kappa(i)$ for some constant i) or come from $\bot \leftarrow \prod_{k=1}^{n} a_k$, which yields a clause $\bigsqcup_{k=1}^{n} \neg a_k$.

Global Constraints   For the rest of the formula inside the quantifiers, we use the well known Tseitin transformation [Tse83] and the polarity of sub-formula to avoid getting a set of clauses of exponential size — an area in which growth is usually frowned upon. Every boolean atom

---

[21] We obtained this formula from Figure 5.5 by merging the quantified formulas without renaming, because they share no variable, and thus quantification commutes with connectives.

that is introduced to stand for a sub-formula F is named A_F. We obtain the following conjunction of clauses:

$$\sqcap \begin{cases} \prod_{i\in\mathfrak{J}} A_{empty(i)} \sqcup \bigsqcup_{t\in\kappa(i)} A_{minimal'(i,t)} \\[4pt] \prod_{i\in\mathfrak{J},C[\diamond]\in S_{cand}(i)} \neg A_{empty(i)} \sqcup \neg \lVert C[\diamond] \in S_{min}(i) \rVert \\[4pt] \prod_{i\in\mathfrak{J},t\in\kappa(i)} \left(\neg A_{minimal'(i,t)} \sqcup \lVert i \simeq t \rVert\right) \sqcap \left(\neg A_{minimal'(i,t)} \sqcup A_{minimal(i,t)}\right) \\[4pt] \prod_{i\in\mathfrak{J},t\in\kappa(i),t'\triangleleft t,sub(t',i)} \neg A_{minimal(i,t)} \sqcup \bigsqcup_{C[\diamond]\in S_{cand}(i)} A_{minimal\_by(i,t,t',C[\diamond])} \\[4pt] \prod_{i\in\mathfrak{J},t\in\kappa(i),t'\triangleleft t,sub(t',i)C[\diamond]\in S_{cand}(i)} \sqcap \begin{cases} \left(\neg A_{minimal\_by(i,t,t',C[\diamond])} \sqcup \lVert C[\diamond] \in S_{min}(i) \rVert\right) \\ \left(\neg A_{minimal\_by(i,t,t',C[\diamond])} \sqcup \lVert minimal(C[\diamond],i,t') \rVert\right) \end{cases} \end{cases}$$

in which the following Tseitin atoms have been introduced to prevent large disjunctions of conjunctions to exert their harmful multiplication of the number of clauses:

$A_{empty(i)}$ stands for the definition of empty(i), which is a conjunction.
$A_{minimal'(i,t)}$ stands for minimal$(i,t) \sqcap \lVert i \simeq t \rVert$.
$A_{minimal(i,t)}$ stands for the definition of minimal(i,t), likely to yield a large CNF.
$A_{minimal\_by(i,t,t',C[\diamond])}$ stands for the sub-formula $\lVert C[\diamond] \in S_{min}(i) \rVert \sqcap \lVert minimal(C[\diamond],i,t') \rVert$.

The clause conjunctions in gray are added at the beginning of the saturation process (or whenever a new inductive constant and its coverset are added). Conversely, the formulas that are not colored should be added, then removed, every time the boolean solver is called (since they are part of G_k).

Summary   As we see, only a small part of the formula does not lend itself to incremental solving, mandating the usage of push and pop. It makes it possible to run a boolean satisfiability check at every iteration of the saturation loop efficiently — a crucial requirement for a prover that has to deal with real problems.

5.6.3   Experimental Evaluation of Zipperposition+Induction

Zipperposition-0.5[22] includes the SAT-based encoding of induction as described in Section 5.2, with a simplified implementation of AVATAR that does not prune inactive clauses (clauses whose trail is false in the current boolean interpretation). Still, it manages to solve some inductive problems, as Figure 5.7 shows. The problems listed there are all successfully solved using only the simple strengthening technique. The *file name* column refers to the name of the problem in the directory examples/ind/ in the repository of Zipperposition. We make the meaning of some symbols precise:
- $x @ y$ concatenates the lists x and y;
- count(x,l) is the number of times x occurs in the list l;
- mem(x,l) is true if x occurs in the list l;
- t_rev(t) reverses the tree t (so that prefix traversal becomes postfix traversal).

In some problems, in particular related to trees, a huge number of lemmas are generated. That explains why the prover takes more time on those problems. Some problems also have an "easy" version in which only the relevant axioms are kept — for instance, tree2_easy.p (same as tree2.p but with fewer axioms) is solved in 105 steps after 0.266s. Keep in mind that the implementation of inductive reasoning in Zipperposition is only a proof of concept and has not been optimized in any way; we did not compare to other systems for this reason.

---

[22] See https://github.com/c-cube/zipperposition/archive/0.5.tar.gz.

problem	file	steps	time (s)
+ is associative	`nat1.p`	28	0.027
+ is commutative	`nat2.p`	66	0.064
$x + 0 \simeq x$	`nat3.p`	15	0.012
$x + s(y) \simeq s(x + y)$	`nat5.p`	20	0.032
$(x + y) - x \simeq y$	`nat9.p`	16	0.014
$x - x \simeq 0$	`nat10.p`	13	0.013
$x \leq y \Rightarrow z + x \leq z + y$	`nat18.p`	94	0.066
$\mathrm{count}(x, l_1 \mathbin{@} l_2) \simeq \mathrm{count}(x, l_1) + \mathrm{count}(x, l_2)$	`list4.p`	86	0.131
$x \mathbin{@} [\,] \simeq x$	`list8.p`	31	0.037
@ is associative	`list9.p`	49	0.059
$\mathrm{mem}(x, l \mathbin{@} (x :: [\,]))$	`list12.p`	148	0.188
$\mathrm{mem}(x, l) \Rightarrow \mathrm{mem}(x, l' \mathbin{@} l)$	`list14.p`	157	0.218
$\mathrm{t\_rev}(\mathrm{t\_rev}(t)) \simeq t$	`tree2.p`	427	14.350

Figure 5.7: Time Needed by Zipperposition on Some Problems

Conclusion

We have shown another extension of Superposition (with AVATAR) that lends itself well to structural induction. We also demonstrated its feasability in a proof of concept implementation that can already solve non-trivial problems without a mechanism to generate lemmas from the signature — and is still compatible with such a mechanism in a very simple way. Our work extends the previous extension of Superposition to induction by Kersani and Peltier [KP13] to structural types in general. It leverages the natural ability of the AVATAR calculus to reason by case, and, using QBF constraints instead of SAT ones, it deals with exponentially many cases at the same time. It naturally composes with other parts from this thesis, including arithmetic, thanks to the uniform treatment of arithmetic clauses with deduction rules that carry boolean trails over into the conclusion. However, the calculus using QBF (Section 5.4) is mostly theoretical at this point: our prototype implements it but sorely lacks optimizations that would prune the search space.

To be integrated into a competitive prover such as E [Sch02] or Vampire [RV01b], the prover needs to deal with typed logic and to perform AVATAR splitting. The prover would also have to guess lemmas from the signature (in some more-or-less heuristic fashion) in order to solve more complex problems without human guidance. Better lemmas generalization techniques should be adapted to our framework. Since lemmas play such an important role in inductive theorem proving, we believe it would also be very useful to remember useful lemmas in a proof so that, later, when a similar inductive theory is recognized, they can be recalled immediately in the hope they will prove useful again. The next chapter presents a technique to recognize axiomatic theories — a finite set of axioms — in a signature-agnostic fashion, and take actions when a known theory is recognized in a problem (e.g., add lemmas that are valid in the theory).

Chapter 6

Theory Detection

6.1 Introduction

As already mentioned before, Superposition [NR99] appeared to handle the difficult issue of equality reasoning, that would otherwise drown most provers in a huge search space (in particular, resolution-based provers). Still, many other theories tend to generate a large number of clauses when present in the axioms, even when they are not used to prove the goal. A classic illustration of that phenomenon is the theory of Associative Commutative symbols (usually called *AC*); it has been known for a long time to slow down provers. It is so critical in some domains that a large body of research has been dedicated to its integration in proof procedures (see for instance [BG95]). Many theorem provers for first-order logic with equality contain an ad-hoc engine to recognize instances of AC symbols, composed of the two following axioms (here, for the symbol +):

Associativity: $\forall x\ y\ z.\ x + (y + z) \simeq (x + y) + z$;
Commutativity: $\forall x\ y.\ x + y \simeq y + x$.

Once the automated prover has recognized that some symbol has the AC property, it can use some specialized technique to deal with it efficiently, because this theory is very common but is known to generate a large amount of redundant clauses that bloat the search space. However, if similar techniques can be applied to other axiomatic theories — theories that can be defined in terms of a finite set of axioms — code would need to be written for those provers to handle each new theory. We propose here a system that can recognize the presence of theories in a generic and incremental way. The system is based on the use of a second theorem prover, based on Horn clauses, that reasons about the *meta-level properties* that the problem exhibits, rather than trying to solve the problem itself. In some limited sense, this is similar to what a human mathematician does: she would try to use equations and hypotheses on the problem itself, but at the same time she would recognize already known patterns and specific structures (for instance, a group structure, a linear field, or an isomorphism to some other part of Mathematics) and use this higher-level knowledge to apply theorems and lemmas she knows. Many useful theories can be finitely axiomatized, even outside of algebra; many set operators (e.g., the powerset) are defined by a set of axioms, the theory of functional arrays is widely used in program verification, etc.

We implemented this technique in the logic library Logtk and in the experimental theorem prover Zipperposition (described respectively in Section 3.1 and Section 3.2). A small deduction engine for higher-order Horn clauses is used to reason on properties of the problems, including the set of theories and axioms that we know are present. The prover and the meta-level reasoner interact by exchanging clauses on the one hand, deduced properties on the other hand. The Superposition prover can use the additional information to infer new clauses thanks to *lemmas* (using AVATAR as explained in Section 5.3) or to activate theory-specific *redundancy criteria* [BC13] or *decision procedures*.

We also expose several applications for the detection of axioms and theories. The first one is a powerful lemma that allows theorem provers that deal well with equality to discover that some relations represent the graph of a function, and to replace instances of the relation by equations. For instance, in the TPTP archive [Sut09], many algebraic problems on groups (or extensions thereof) are encoded using $sum(x, y, z)$ instead of $z \simeq add(x, y)$. This complicates the axiomatization (many more axioms, that are big Horn clauses, etc.) compared to an equational view of the problem. Our lemma, fed to the prover in a simple declarative language (using the same conventions as TPTP: `!` is universal quantification, `~` is negation, capital `X, Y` are variables, `<-` is the Horn clause implication) as:

```
axiom (functional2 P) <-
  holds (![X,Y,Z]: [~ (P X Y Z), ~ (P X Y Z2), Z = Z2]).

axiom (total2 {pred=P, fun=F}) <-
  holds (![X,Y]: [P X Y (F X Y)]).

lemma (![X,Y,Z]: [P X Y Z --> (Z = F X Y)]) <-
  axiom (functional2 P),
  axiom (total2 {pred=P, fun=F}).
```

allows to recover an equational (boolean) definition from this encoding, which can then be unfolded to simplify clauses.

Another application is the per-theory activation of an equational redundancy criterion. If we know a saturated, ground convergent system of equations for some theory [AHL03], literals that are tautological or absurd in this theory can be removed while retaining completeness. Our framework allows us to know when such a theory occurs in a problem, so we can use the corresponding redundancy criterion.

Similar work comprises a mechanism in Waldmeister that helps the prover select a term ordering (an important heuristic in any flavor of Superposition) based on a pre-computed table of algebraic theories that are detected in the prover's input. Some other provers, such as E, feature some basic built-in detection for some theories. A previous version of our work, using a different technique, was also published [BC13].

We first expose some basic definitions and notations, then successively expose techniques for recognizing individual axioms and whole theories. Then, after some examples of how to use knowledge gained about what axiomatic theories are present, we present some experimental results and conclude.

6.2   Higher-Order Reasoner

The meta-prover works on properties of functions and predicates. Therefore, functions and predicates must be first-class objects in the language of the meta-prover, which makes it intrinsically higher-order. For that reason, we made the obvious choice of using higher-order terms (and formulas). As we will directly use higher-order terms and formulas (more precisely, Horn clauses) to describe axioms, theories and other relevant properties, the term language should be expressive and designed for human readability. The language incorporates some additional constructs such as *records* and *multisets* to make it easier to use and more readable; records are convenient (and extensible) representations of tuples where elements are accessed through their label (a name) rather than an arbitrary position, and multisets are bags of unordered elements that can be used to represent arguments of associative-commutative (or merely commutative) symbols such as ∨.

6.2.1   Definitions

Definition 6.1 (Label)**.** *A* label *is a string that can be used to name the fields of a* record. *Labels are not first-class, they can only occur in records and will never be bound in substitutions.*

Definition 6.2 (Types for Higher-Order Terms)**.** *Types are inductively defined from the following grammar:*

$$\tau \stackrel{def}{=} \alpha \mid c \mid \tau \to \tau \mid \Pi\alpha.\, \tau \mid [\tau] \mid \{\!| l_1 : \tau_1, \ldots, l_n : \tau_n |\!\} \mid \{\!| l_1 : \tau_1, \ldots, l_n : \tau_n \mid \alpha |\!\}$$

in which α denotes a type variable, c a type constant, $\Pi\alpha.\, \tau$ a polymorphic type, $[\tau]$ the type of multisets of elements of type τ, and records denote the type of term records.

Definition 6.3 (Higher-Order Term)**.** *A higher-order term over a signature Σ is defined by the rule t in the following grammar, where x and ρ denote variables, c denotes a constant from Σ, τ denotes a type (see Definition 6.2), $l_1, \ldots, l_n$ are pairwise distinct* labels*, m describes term multisets, and r denotes (extensible) records with a row variable ρ [Wan87].*

$$\begin{aligned} t &\stackrel{def}{=} x \mid c \mid t_{\langle \tau \rangle} \mid \forall x.\, t \mid \exists x.\, t \mid t\, t \mid m \mid r \\ m &\stackrel{def}{=} [t, \ldots, t] \\ r &\stackrel{def}{=} \{\!| l_1 = t, \ldots, l_n = t |\!\} \mid \{\!| l_1 = t, \ldots, l_n = t \mid \rho |\!\} \end{aligned}$$

The empty record is $\{\!| |\!\}$, it has no labels. A variable occurring in the row *part of a record is usually named ρ rather than $x, y,$ or z, as in $\{\!| l_1 = t_1, \ldots, l_n = t_n \mid \rho |\!\}$. It is called a* row variable *and can only be substituted with a (possibly empty) record. Rows are subject to* flattening*: a record $\{\!| l_1 = t_1, \ldots, l_n = t_n \mid \{\!| l_{n+1} = t_{n+1}, \ldots, l_m = t_m \mid \rho |\!\} |\!\}$ is only valid if all the $(l_j)_{1 \le j \le m}$ are distinct; in this case the record is equal to $\{\!| l_1 = t_1, \ldots, l_m = t_m \mid \rho |\!\}$.*

$t_{\langle \tau \rangle}$ denotes the application of a term t to a type parameter τ, and will sometimes be noted t_τ for brevity (on infix operators, in particular).

Term application is left-associative, that is, $f\, t_1 \, \ldots \, t_n$ means $(\ldots((f\, t_1)\, t_2)\, \ldots)\, t_n$. We will work modulo alpha-equivalence of variables bound by quantifiers $\exists$ and $\forall$, in order to prevent variable captures. We may sometimes represent $\{\!| l_1 = t_1, \ldots, l_n = t_n |\!\}$ as the equivalent $\{\!| l_1 = t_1, \ldots, l_n = t_n \mid \{\!| |\!\} |\!\}$ for uniformity reasons. Conversely, sometimes we will index terms and labels with generic sets I, as in $[(t_i)_{i \in I}]\, \{\!| (l = t_l)_{l \in I} \mid \rho |\!\}$.

Remark 6.1 (Lambda Abstractions)**.** *We did not introduce lambda-abstractions in the higher-order terms, because we will need a decidable notion of unification in the rest of the chapter, which is incompatible with beta-reduction. Instead, we represent quantifiers $\forall$ and $\exists$ explicitly.*

Definition 6.4 (Variables of a Term)**.** *The set of variables of a term t, $vars(t)$, is defined by*

$$vars(x) = \{x\} \text{ if } x \in X$$
$$vars(c) = \emptyset \text{ if } c \in \Sigma$$
$$vars(s\, t) = vars(s) \cup vars(t)$$
$$vars(s_{\langle \tau \rangle}) = vars(s)$$
$$vars(\forall x.\, t) = \{x\} \cup vars(t)$$
$$vars(\exists x.\, t) = \{x\} \cup vars(t)$$
$$vars([t_1, \ldots, t_n]) = \bigcup_{i=1}^{n} vars(t_i)$$
$$vars(\{\!| l_1 = t_1, \ldots, l_n = t_n \mid \rho |\!\}) = \bigcup_{i=1}^{n} vars(t_i)$$

Definition 6.5 (Substitutions in Higher-Order Terms)**.** *Substitutions are mappings from term variables to terms (as in first-order substitutions, Definition 2.42) but additional care has to be taken in two cases:*

- *Substitution under a quantifier $\forall x.\, t$ or $\exists x.\, t$ is done modulo alpha-equivalence, so that no capture happens;*

- *row variables in records must* flatten, *that is, if* $\rho\sigma = \{\!|\, l'_1 = t'_1, \ldots, l'_m = t'_m \mid \rho'\,|\!\}$, *assuming* $\forall i \ j. \ l_i \neq l'_j$, *then*

$$\{\!|\, l_1 = t_1, \ldots, l_n = t_n \mid \rho\,|\!\}\sigma = \{\!|\, l_1 = t_1, \ldots, l_n = t_n, l'_1 = t'_1, \ldots, l'_m = t'_m \mid \rho'\,|\!\}$$

Substituting a row variable ρ with a record containing duplicate labels is undefined. Unification between terms will never introduce such substitutions.

Remark 6.2 (Records and Readability)**.** *Records are, strictly speaking, redundant. A record with n fields can be replaced with a fresh function with n arguments. However, we believe describing theories with named labels — rather than arbitrary positions in a function application — makes for more readable a formula.*

Example 6.1 (Group Theory)**.** *A very simple algebraic theory is the* group structure*. In addition to the carrier type τ, defining a group means producing the neutral element $E : \tau$, the group operation $M : \tau \to \tau \to \tau$ and the group inverse $I : \tau \to \tau$. A record $\{\!|\,neutral = E, inverse = I, op = M\,|\!\}$ makes clear which variable plays which role. For instance, integer addition as used, pervasively, in Chapter 4, forms a group that can be described as $\{\!|\,neutral = 0, inverse = -, op = +\,|\!\}$ where $\{0, +, -\}$ are regular symbols.*

Definition 6.6 (Typing Rules of Higher-Order Terms)**.** *Higher-order terms, as defined in 6.3, are typed as follows. A higher-order formula is simply a term of type o. Checking that a term t has the type τ starts a derivation with a sequent $\Sigma \vdash t : \tau$ where Σ is the global signature.*

$$\overline{\Gamma, t : \tau \vdash t : \tau}$$

$$\frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash t : \tau}{\Gamma \vdash f \ t : \tau'}$$

$$\frac{\Gamma \vdash t : \Pi\alpha.\ \tau}{\Gamma \vdash t_{\langle \tau' \rangle} : \tau\,\{\alpha \mapsto \tau'\}}$$

$$\frac{\Gamma, x : \tau \vdash F[x] : o}{\Gamma \vdash \forall x : \tau.\ F[x] : o} \qquad \frac{\Gamma, x : \tau \vdash F[x] : o}{\Gamma \vdash \exists x : \tau.\ F[x] : o}$$

$$\frac{\forall i.\ \Gamma \vdash t_i : \tau}{\Gamma \vdash [t_1, \ldots, t_n] : [\tau]}$$

$$\frac{\forall i.\ \Gamma \vdash t_i : \tau_i}{\Gamma \vdash \{\!|\, l_1 = t_1, \ldots, l_n = t_n\,|\!\} : \{\!|\, l_1 : \tau_1, \ldots, l_n : \tau_n\,|\!\}}$$

$$\frac{\forall i.\ \Gamma \vdash t_i : \tau_i \qquad \rho : \tau_\rho}{\Gamma \vdash \{\!|\, l_1 = t_1, \ldots, l_n = t_n \mid \rho\,|\!\} : \{\!|\, l_1 : \tau_1, \ldots, l_n : \tau_n \mid \tau_\rho\,|\!\}}$$

where τ_ρ is a type variable or a record type.

Definition 6.7 (Type, Term, Formula and Clause Encoding)**.** *First-order clauses and formulas will be encoded into higher-order terms using the encoding $enc(\cdot)$, roughly equivalent to currying for terms; types are encoded using $enc_{ty}(\cdot)$. The encoding uses predefined symbols $\dot\simeq : \Pi\alpha.\ [\alpha] \to o$, $\dot\neg : o \to o$, and $\dot\vee, \dot\wedge : [o] \to o$.*

$$
\begin{aligned}
enc_{ty}(\alpha) &\stackrel{def}{=} \alpha && \text{if } \alpha \in \mathcal{A} \\
enc_{ty}(c) &\stackrel{def}{=} c && \text{if } c \in \Sigma_\tau \\
enc_{ty}((\tau_1 \times \ldots \times \tau_n) \to \tau) &\stackrel{def}{=} enc_{ty}(\tau_1) \to \ldots \to enc_{ty}(\tau_n) \to enc_{ty}(\tau) \\
enc(x) &\stackrel{def}{=} x && \text{if } x \in X \\
enc(f(t_1, \ldots, t_n)) &\stackrel{def}{=} f\ enc(t_1) \ldots enc(t_n) \\
enc(\forall x : \tau.\ F[x]) &\stackrel{def}{=} \forall x : enc_{ty}(\tau).\ enc(F[x]) \\
enc(\exists x : \tau.\ F[x]) &\stackrel{def}{=} \exists x : enc_{ty}(\tau).\ enc(F[x]) \\
enc(\neg F) &\stackrel{def}{=} \dot{\neg} enc(F) \\
enc(F_1 \vee \ldots \vee F_n) &\stackrel{def}{=} \dot{\vee}\ [enc(F_1), \ldots, enc(F_n)] && \text{if } n \geq 2 \\
enc(F_1 \wedge \ldots \wedge F_n) &\stackrel{def}{=} \dot{\wedge}\ [enc(F_1), \ldots, enc(F_n)] && \text{if } n \geq 2 \\
enc(s \simeq t) &\stackrel{def}{=} \dot{\simeq}_{\langle enc_{ty}(\tau) \rangle}\ [enc(s), enc(t)] && \text{if } s, t : \tau
\end{aligned}
$$

A clause $C \stackrel{def}{=} \bigvee_{i=1}^{n} l_i$ *with free variables freevars$(C) = \{x_1, \ldots, x_k\}$ is encoded as the closed formula* $\forall x_1 \ldots x_k.\ \bigvee_{i=1}^{n} l_i$. *Note that the disjunction is encoded as a $\dot{\vee}$-prefixed multiset.*

Example 6.2 (Polymorphic Lists). *The theory of polymorphic lists already presented in Example 2.3 can be represented with higher-order terms using the following curried signature:*

$$
\Sigma = \big\{ (::) : \Pi\alpha.\ \alpha \to list(\alpha) \to list(\alpha), [\,] : \Pi\alpha.\ list(\alpha) \big\}
$$

We can define a few functions that will look familiar to functional programmers using an encoding to higher-order terms:

informal definition	encoding
$[\,] @ l \simeq l$	$\forall \alpha.\ \forall l : list(\alpha).\ \dot{\simeq}_\alpha\ \big[[\,]_{\langle \alpha \rangle} @_{\langle \alpha \rangle} l, [\,]_{\langle \alpha \rangle} \big]$
$(x :: l_1) @ l_2 \simeq x :: (l_1 @ l_2)$	$\forall \alpha.\ \forall x : \alpha.\ \forall l_1\ l_2 : list(\alpha).$
	$\quad \dot{\simeq}_\alpha\ \big[(x ::_{\langle \alpha \rangle} l_1) @_{\langle \alpha \rangle} l_2, x ::_{\langle \alpha \rangle} (l_1 @_{\langle \alpha \rangle} l_2) \big]$
$rev([\,]) \simeq [\,]$	$\forall \alpha.\ \dot{\simeq}_\alpha\ \big[rev_{\langle \alpha \rangle}([\,]_{\langle \alpha \rangle}), [\,]_{\langle \alpha \rangle} \big]$
$rev(x :: l) \simeq rev(l) @ (x :: [\,])$	$\forall \alpha.\ \forall x : \alpha.\ \forall l : list(\alpha).\ \dot{\simeq}_\alpha$
	$\quad \big[rev_{\langle \alpha \rangle}\ (x ::_{\langle \alpha \rangle} l), (rev_{\langle \alpha \rangle}\ l) @_{\langle \alpha \rangle} (x ::_{\langle \alpha \rangle} [\,]_{\langle \alpha \rangle}) \big]$

Lemma 6.1 (Encoding Preserves Well-typedness). *If F is a well-typed formula in the signature Σ, then enc(F) is a well-typed higher-order term in the signature*

$$
enc_{ty}(\Sigma) \uplus \big\{ \dot{\simeq} : \Pi\alpha.\ [\alpha] \to o, \dot{\neg} : o \to o, \dot{\vee} : [o] \to o, \dot{\wedge} : [o] \to o \big\}
$$

Proof. First, the same property holds for first-order terms, by trivial induction on the structure of the term; then, by induction on the structure of F. $\qquad\square$

Definition 6.8 (Type, Term, Formula and Clause Decoding). *A reverse decoding operation dec$(\cdot)$ can be defined in the straightforward way. It is defined only for terms in which $\dot{\simeq}$ is applied to a multiset of cardinality 2, and every term application starts with a constant (rather than a variable).*

$$
\begin{array}{rcll}
dec_{ty}(\alpha) & \overset{def}{=} & \alpha & \text{if } \alpha \in \mathscr{A} \\[4pt]
dec_{ty}(c) & \overset{def}{=} & c & \text{if } c \in \Sigma_\tau \\[4pt]
dec_{ty}(\tau_1 \to \ldots \to \tau_n \to \tau) & \overset{def}{=} & (dec_{ty}(\tau_1) \times \ldots \times dec_{ty}(\tau_n)) \to dec_{ty}(\tau) & \\[4pt]
dec(x) & \overset{def}{=} & x & \text{if } x \in X \\[4pt]
dec(f\ t_1\ \ldots\ t_n) & \overset{def}{=} & f(dec(t_1), \ldots, dec(t_n)) & \text{if } f \in \Sigma \\[4pt]
dec(\forall x : \tau.\ F[x]) & \overset{def}{=} & \forall x : dec_{ty}(\tau).\ dec(F[x]) & \\[4pt]
dec(\exists x : \tau.\ F[x]) & \overset{def}{=} & \exists x : dec_{ty}(\tau).\ dec(F[x]) & \\[4pt]
dec(\dot\neg\ F) & \overset{def}{=} & \neg dec(F) & \\[4pt]
dec(\dot\vee\ [F_1, \ldots, F_n]) & \overset{def}{=} & dec(F_1) \vee \ldots \vee dec(F_n) & \text{if } n \geq 2 \\[4pt]
dec(\dot\wedge\ [F_1, \ldots, F_n]) & \overset{def}{=} & dec(F_1) \wedge \ldots \wedge dec(F_n) & \text{if } n \geq 2 \\[4pt]
dec(\dot\simeq_{\langle\tau\rangle}\ [s, t]) & \overset{def}{=} & dec(s) \simeq dec(t) & \text{if } s, t : \tau \\[4pt]
& dec(t) & \text{undefined otherwise} &
\end{array}
$$

It is easy to see that for any formula F, $dec(enc(F))$ and F are equivalent.

Definition 6.9 (Horn Clause). *A higher-order Horn Clause is defined by a non-empty conclusion A and a possibly empty set of premises $B_1, \ldots, B_n$. Both A and all the B_i are higher-order terms of type o_{meta} such that $freevars(A) \subseteq \bigcup_{i=1}^n freevars(B_i)$. Such a Horn Clause is denoted $A \leftarrow B_1, \ldots, B_n$; if $n = 0$ it is shortened into A.*

Example 6.3 (Group Theory (full encoding)). *The full description of what a group is, including the definition of individual axioms, can be summarized in a few Horn Clauses. As anticipated in Example 6.1, the group structure itself is described using a record. The symbols $axiom : axiom \to o_{meta}$, $theory : theory \to o_{meta}$ and $holds : o \to o_{meta}$ are already defined, as we will see later. Capitalized identifiers are — possibly higher-order — variables.*

```
axiom (associative F) <-
  holds (![X,Y,Z]: [F X (F Y Z) = F (F X Y) Z]).

axiom (left_identity {op=Mult, elem=E}) <-
  holds (![X]: [Mult E X = X]).

axiom (left_inverse {op=Mult, inverse=I, elem=E}) <-
  holds (![X]: [Mult (I X) X = E]).

theory (group {op=Mult, neutral=E, inverse=I}) <-
  axiom (associative Mult),
  axiom (left_inverse {op=Mult, inverse=I, elem=E}),
  axiom (left_identity {op=Mult, elem=E}).
```

This snippet[1] reads:

- *F is associative if it holds that $\forall x\ y\ z.\ F(x, F(y, z)) \simeq F(F(x, y), z)$;*
- *E is the left-neutral element of M if it holds that $\forall x.\ M(E, x) \simeq x$;*
- *(similar definition for left-inverse)*
- *(M, E, I) forms a group if M is associative, E is the left-neutral element of M, and I is the left-inverse of M. This defines what a group structure is. Note the use of a record to be able to name the components of the group, rather than refer to them by their position in a tuple.*

We will see later that theories and axioms, such as associative or group, are higher-order constants whose type is inferred.

---

[1] The file containing some definitions, including the group theory, can be found in Zipperposition's repository under the path `src/builtin.theory`.

Remark 6.3 (Type Inference). *Logtk contains some facilities for type inference, inspired by Hindley-Milner algorithm [Mil78]. This explains the absence of type annotations in snippets from this chapter. Variables are initially typed with a type variable that is generalized if no constraint specializes it. This way, some Horn clauses are also polymorphic. For instance, from the following definitions*

```
axiom (associative F) <-
  holds (![X,Y,Z]: [F X (F Y Z) = F (F X Y) Z]).

axiom (left_inverse {op=Mult, inverse=I, elem=E}) <-
  holds (![X]: [Mult (I X) X = E]).
```

the following types are inferred:
- *associative* $: \Pi\alpha. (\alpha \to \alpha \to \alpha) \to axiom;$
- *left_inverse* $: \Pi\alpha. \{\!|op: \alpha \to \alpha \to \alpha, inverse: \alpha \to \alpha, elem: \alpha|\!\} \to axiom;$
- $F: \Pi\alpha. \alpha \to \alpha \to \alpha.$

6.2.2 Unification

The previous section underlined the need for higher-order terms and gave basic definitions, including an encoding from first-order formulas to higher-order terms. Now, if we want to *recognize* the presence of some axiom, such as associativity, in a particular Superposition derivation, we need to match the abstract definition of associativity with clauses that occur in the derivation. If some function symbol f has been defined as associative, then we want $\forall x\, y\, z: \alpha. \; \dot{\simeq}_\alpha$ $\left[F\, (F\, x\, y)\, z, F\, x\, (F\, y\, z)\right]$ and the (encoded) associativity axiom for f to unify with $\{F \mapsto f\}$ (note that α is a free variable). Unification is also needed to reason at the meta-level using resolution between Horn clauses, as we will see in Section 6.2.3.

Definition 6.10 (Equivalence). *Some terms that are not structurally equivalent should still be considered the same. We define the binary relation $=_{HO}$ as a more flexible notion of equality between terms:*

$$
\begin{array}{rcll}
x & =_{HO} & x & \textit{if } x \in X \\
c & =_{HO} & c & \textit{if } c \in \Sigma \\
s\, t & =_{HO} & s'\, t' & \textit{if } s =_{HO} s' \textit{ and } t =_{HO} t' \\
[s_1, \ldots, s_n] & =_{HO} & [t_1, \ldots, t_n] & \textit{if } \exists \theta. \; \forall i \in \{1, \ldots, n\}.\; s_i =_{HO} t_{\theta(i)} \\
s_{\langle \tau \rangle} & =_{HO} & t_{\langle \tau \rangle} & \textit{if } s =_{HO} t \\
\forall x_1 \ldots x_n.\, s & =_{HO} & \forall y_1 \ldots y_n.\, t & \textit{if } \exists \theta. \; \left(s =_{HO} t\{y_1 \mapsto x_{\theta(1)},\, \ldots, y_n \mapsto x_{\theta(n)}\}\right) \\
\exists x_1 \ldots x_n.\, s & =_{HO} & \exists y_1 \ldots y_n.\, t & \textit{(idem)} \\
\{\!|(l_i = s_i)_{i=1}^n|\!\} & =_{HO} & \{\!|(l_i = t_i)_{i=1}^n|\!\} & \textit{if } \forall i \in \{1, \ldots, n\}.\; s_i =_{HO} t_i \\
\{\!|(l_i = s_i)_{i=1}^n \mid \rho|\!\} & =_{HO} & \{\!|(l_i = t_i)_{i=1}^n \mid \rho'|\!\} & \textit{if } \forall i \in \{1, \ldots, n\}.\; s_i =_{HO} t_i \textit{ and } \rho = \rho'
\end{array}
$$

where θ is a permutation of $\{1, \ldots, n\}$.

Lemma 6.2 (Equivalence and Encoding). *If F and G are first-order formulas such that $enc(F) =_{HO} enc(G)$, then $F \Longleftrightarrow G$ is a tautology.*

Proof. By induction on the structure of the formulas, thanks to the associativity-commutativity of $\vee, \wedge$ and the symmetry of $\simeq$. $\qquad\square$

Definition 6.11 (Higher-Order Unifier). *A higher-order unifier for two terms s and t is a substitution σ such that $s\sigma =_{HO} t\sigma$.*

Remark 6.4 (*n*-ary unification). *Two unifiable higher-order terms might have several most general unifiers. For instance, $\left[f\, x, f\, y\right]$ and $\left[f\, a, f\, b\right]$ admit both $\{x \mapsto a, y \mapsto b\}$ and $\{x \mapsto b, y \mapsto a\}$ as most general unifiers.*

Definition 6.12 (Unification Algorithm)**.** *As usual in the literature [BS01], we present our uni-fication algorithm for higher-order terms as set of reduction rules over sets of unification prob-lems, where each unification problem has the form $M;\sigma$ where M is a multiset of unordered pairs $s =^?_{HO} t$ (meaning s has to be unified with t), and σ is a substitution. A solved system has the form $\emptyset;\sigma$ and contains a solution σ. Note that $\rightsquigarrow$ is not confluent and can lead to several solutions.*

$$\{x =^?_{HO} t\} \cup M;\sigma \quad \rightsquigarrow \quad M\{x \mapsto t\};\sigma \circ \{x \mapsto t\}$$
$$\text{if } canBind(x,t)$$

$$\{s_{\langle\tau\rangle} =^?_{HO} t_{\langle\tau'\rangle}\} \cup M;\sigma \quad \rightsquigarrow \quad \{s\sigma' =^?_{HO} t\sigma'\} \cup M\sigma';\sigma \circ \sigma'$$
$$\text{if } \sigma' = mgu(\tau,\tau')$$

$$\{s\ t =^?_{HO} s'\ t'\} \cup M;\sigma \quad \rightsquigarrow \quad \{s =^?_{HO} s', t =^?_{HO} t'\} \cup M;\sigma$$

$$\{c =^?_{HO} c\} \cup M;\sigma \quad \rightsquigarrow \quad M;\sigma \text{ if } c \in \Sigma \cup \bigcup_i\{c_i\}$$

$$\{[] =^?_{HO} []\} \cup M;\sigma \quad \rightsquigarrow \quad M;\sigma$$

$$\{[(s_i)_{i\in I}] =^?_{HO} [(t_i)_{i\in I}]\} \cup M;\sigma \quad \rightsquigarrow \quad \left\{ \begin{array}{l} s_1 =^?_{HO} t_j, \\ [(s_i)_{i\in I\setminus\{1\}}] =^?_{HO} [(t_i)_{i\in I\setminus\{j\}}] \end{array} \right\} \cup M;\sigma$$
$$\text{for each } j \in I$$

$$\{\{\!|(l = s_l)_{l\in I}|\!\} =^?_{HO} \{\!|(l = t_l)_{l\in I}|\!\}\} \cup M;\sigma$$
$$\{\{\!|(l = s_l)_{l\in I} \mid \rho|\!\} =^?_{HO} \{\!|(l = t_l)_{l\in I} \mid \rho|\!\}\} \cup M;\sigma \quad \rightsquigarrow \quad \bigcup_{l\in I}\{s_l =^?_{HO} t_l\} \cup M;\sigma$$

$$\{\{\!|(l = s_l)_{l\in I} \mid \rho|\!\} =^?_{HO} \{\!|(l = t_l)_{l\in I'}|\!\}\} \cup M;\sigma \quad \rightsquigarrow \quad \bigcup_{l\in I\cap I'}\{s_l\sigma' =^?_{HO} t_l\sigma'\} \cup M\sigma';\sigma \circ \sigma'$$
$$\text{where } \sigma' = \{\rho \mapsto \{\!|(l = t_l)_{l\in I'\setminus I}|\!\}\}$$
$$\text{if } canBind(\rho, \{\!|(l = t_l)_{l\in I'\setminus I}|\!\}) \text{ and } I \subseteq I'$$

$$\{\{\!|(l = s_l)_{l\in I} \mid \rho|\!\} =^?_{HO} \{\!|(l = t_l)_{l\in I'} \mid \rho'|\!\}\} \cup M;\sigma \quad \rightsquigarrow \quad \bigcup_{l\in I\cap I'}\{s_l\sigma' =^?_{HO} t_l\sigma'\} \cup M\sigma';\sigma \circ \sigma'$$
$$\text{where } \sigma' = \left\{ \begin{array}{l} \rho \mapsto \{\!|(l = t_l)_{l\in I'\setminus I} \mid \rho''|\!\} \\ \rho' \mapsto \{\!|(l = s_l)_{l\in I\setminus I'} \mid \rho''|\!\} \end{array} \right\}$$
$$\text{if } \rho \neq \rho', canBind(\rho, \{\!|(l = t_l)_{l\in I'\setminus I} \mid \rho''|\!\})$$
$$\text{and } canBind(\rho', \{\!|(l = s_l)_{l\in I\setminus I'} \mid \rho''|\!\})$$

$$\{(\exists x_1 \ldots x_n.\ s) =^?_{HO} (\exists y_1 \ldots y_n.\ t)\} \cup M;\sigma$$
$$\{(\forall x_1 \ldots x_n.\ s) =^?_{HO} (\forall y_1 \ldots y_n.\ t)\} \cup M;\sigma \quad \rightsquigarrow \quad \{s\sigma_s =^?_{HO} t\sigma_t\} \cup M;\sigma$$
$$\text{where } \sigma_s = \{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}$$
$$\text{and } \sigma_t = \{y_1 \mapsto c_{\theta(1)}, \ldots, y_n \mapsto c_{\theta(n)}\}$$

where $c_1,\ldots,c_n$ are fresh constants (not in Σ); $canBind(x,t)$ is true iff x is a variable, $x \notin freevars(t)$ and t is closed (that is, it contains no c_i); ρ'' is a fresh row variable; θ is a permutation of $\{1,\ldots,n\}$.

A few comments on the unification rules may help the reader forge some intuition.

- binding a variable $x \mapsto t$ requires a regular "occur-check" $x \notin freevars(t)$, but also that t is closed — that is, t must not contain any constant c_i introduced when quantifiers are opened;
- the rule for multisets attempts to unify multisets that have the same number of elements, by picking one element in each multiset, unifying them, and then trying to unify the remaining elements. In this way, it enumerates lazily the permutations of one of the multisets (here, the right-hand-side one);
- unification of extensible records first unifies pairwise the values under labels that are shared by both records, and then it proceeds to unify the other labels (occurring in only one of the terms) with the row of the other term. For instance, the unification problem between records

$$\{\!|l_1 = 2, l_2 = (f\ x) \mid \rho|\!\} =^?_{HO} \{\!|l_1 = y, l_2 = (f\ a), l_3 = b|\!\}$$

admits the following derivation (where ρ' is fresh):

$$\{\{\!| l_1 = 2, l_2 = (f\ x) \mid \rho |\!\} =^?_{\text{HO}} \{\!| l_1 = y, l_2 = (f\ a), l_3 = b |\!\}\} ; \emptyset$$
$$\leadsto \{2 =^?_{\text{HO}} y, (f\ x) =^?_{\text{HO}} (f\ a), \rho =^?_{\text{HO}} \{\!| l_3 = b \mid \rho' |\!\}, \{\!| |\!\} =^?_{\text{HO}} \rho'\} ; \emptyset$$
$$\leadsto \{(f\ x) =^?_{\text{HO}} (f\ a), \rho =^?_{\text{HO}} \{\!| l_3 = b \mid \rho' |\!\}, \{\!| |\!\} =^?_{\text{HO}} \rho'\} ; \{y \mapsto 2\}$$
$$\leadsto \{f =^?_{\text{HO}} f, x =^?_{\text{HO}} a, \rho =^?_{\text{HO}} \{\!| l_3 = b \mid \rho' |\!\}, \{\!| |\!\} =^?_{\text{HO}} \rho'\} ; \{y \mapsto 2\}$$
$$\leadsto \{x =^?_{\text{HO}} a, \rho =^?_{\text{HO}} \{\!| l_3 = b \mid \rho' |\!\}, \{\!| |\!\} =^?_{\text{HO}} \rho'\} ; \{y \mapsto 2\}$$
$$\leadsto \{\rho =^?_{\text{HO}} \{\!| l_3 = b \mid \rho' |\!\}, \{\!| |\!\} =^?_{\text{HO}} \rho'\} ; \{y \mapsto 2, x \mapsto a\}$$
$$\leadsto \{\rho =^?_{\text{HO}} \{\!| l_3 = b |\!\}, \} ; \{y \mapsto 2, x \mapsto a, \rho' \mapsto \{\!| |\!\}\}$$
$$\leadsto \{\} ; \{y \mapsto 2, x \mapsto a, \rho' \mapsto \{\!| |\!\}, \rho \mapsto \{\!| l_3 = b |\!\}\}$$

which yields the solution $\{y \mapsto 2, x \mapsto a, \rho \mapsto \{\!| l_3 = b |\!\}\}$ to the original problem.

- quantifier rules have to deal with swapping consecutive quantifiers, since we defined $=_{\text{HO}}$ so that $\forall x\ y.\ F[x, y] =_{\text{HO}} \forall y\ x.\ F[x, y]$. The abstract algorithm presented here simply enumerates all possible permutations of variables in one of the quantifier, we will see a more efficient implementation in Section 6.2.4

Remark 6.5 (Iterators). *In practice, our implementation uses iterators of type* `Sequence.t` *(see Section 4.6.2) to backtrack over possible choices, as we will explain in Section 6.2.4.*

Lemma 6.3 (Termination). *The rewrite system $\leadsto$ is terminating, that is, it admits no infinite derivations.*

Proof. First, we need a few intermediate orderings:

- Let $\text{n}_{\text{vars}}(M) \stackrel{\text{def}}{=} \#\text{vars}(M)$ be the number of distinct variables, bound or free, that occur in M, and $M \prec_{\text{nvars}} M'$ iff $\text{n}_{\text{vars}}(M) < \text{n}_{\text{vars}}(M')$. It decreases on the variable binding case and on the record cases where there is at least one row variable ρ or ρ', and remains constant otherwise;
- Let $\text{n}_{\text{elem}} : \text{Terms}(\Sigma) \to \mathbb{N}$ be defined as follows:

$$\text{n}_{\text{elem}}([t_1, \ldots, t_n]) = \sum_{i=1} \text{n}_{\text{elem}}(t_i)$$
$$\text{n}_{\text{elem}}(\{\!| l_1 = t_1, \ldots, l_n = t_n \mid \rho |\!\}) = 1 + \sum_{i=1}^{n} \text{n}_{\text{elem}}(t_i) + \text{n}_{\text{elem}}(\rho)$$
$$\text{n}_{\text{elem}}(s\ t) = \text{n}_{\text{elem}}(s) + \text{n}_{\text{elem}}(t)$$
$$\text{n}_{\text{elem}}(t_{\langle \tau \rangle}) = 1 + \text{n}_{\text{elem}}(t)$$
$$\text{n}_{\text{elem}}(c) = 1 \quad \text{if } c \in \Sigma$$
$$\text{n}_{\text{elem}}(x) = 1$$
$$\text{n}_{\text{elem}}(\forall x.\ t) = \text{n}_{\text{elem}}(\exists x.\ t) = \text{n}_{\text{elem}}(t)$$

Let then $s \prec_{\text{elem}} t$ iff $\text{n}_{\text{elem}}(s) < \text{n}_{\text{elem}}(t)$ and $\ll_{\text{elem}}$ be the multiset extension of $\prec_{\text{elem}}$. $\ll_{\text{elem}}$ decreases on the multiset rule, on the record rule where there are no row variables, and on the application and type application rules;
- Let $\prec_{\text{HO}} \stackrel{\text{def}}{=} (\prec_{\text{nvars}}, \ll_{\text{elem}})_{\text{lex}}$. It is well-founded because all its components are.

Every rule $M; \sigma \leadsto M'; \sigma'$ is such that $M' \prec_{\text{HO}} M$, which makes $\leadsto$ well-founded. $\square$

Theorem 6.1 (Correctness). *The unification function $\leadsto$ is correct; that is, if $\{s =^?_{HO} t\}; \emptyset \leadsto^* \emptyset; \sigma$, then $s\sigma =_{HO} t\sigma$.*

Proof. It suffices to notice that each reduction rule $M;\sigma \rightsquigarrow M';\sigma'$ satisfies the following properties: (i) $\sigma \leq \sigma'$; (ii) for any σ'' with $\sigma' \leq \sigma''$, if σ'' solves M' then σ'' solves M. . Therefore, any solution of $M';\sigma'$ is a specialization of σ and solves M. We conclude by induction on the derivation. $\qquad\square$

Theorem 6.2 (Completeness)**.** *The rewrite system $\rightsquigarrow$ is complete. In other words, for any two terms s and t and most general unifier σ of s and t, there is a derivation $\{s =^?_{HO} t\}; \emptyset \rightsquigarrow^+ \emptyset; \sigma$.*

Proof. Assume σ is a mgu of s and t. By induction on why $s\sigma =_{HO} t\sigma$:
- if $s\sigma = x \in X$, s and t must be variables and $\rightsquigarrow$ succeeds in unifying them.
- if $s\sigma = c \in \Sigma$, s and t can be either variables or c and are easily unified;
- if $s\sigma = t_1\ t_2$, then $t\sigma = t_1\ t_2$. Either $s = s_1\ s_2$ and $t = t_1\ t_2$ with $s_1\sigma =_{HO} t_1\sigma$ and $s_2\sigma =_{HO} t_2\sigma$ (in which case the induction hypothesis finds them a unifier), or one of them is a variable which is easy to unify.
- if $s\sigma = \forall x_1 \ldots x_n.\ u$ then either one of s, t is a variable, or $s = \forall x_1.\ x_n s'$ and $t = \forall y_1.\ y_n t'$ (with the same number of quantified variables). In this case, there must be a permutation θ such that $s'\sigma =_{HO} t'\{y_1 \mapsto x_{\theta(1)}, \ldots, y_n \mapsto x_{\theta(n)}\}\sigma$; since $\rightsquigarrow$ enumerates all such permutations it must find the correct θ, and then σ, by induction hypothesis.
- if $s\sigma = [u_1, \ldots, u_n]$, unless one of s, t is a variable, necessarily $s = [s_1, \ldots, s_n]$ and $t = [t_1, \ldots, t_n]$ such that there is a permutation θ with $\forall i.\ s_i\sigma =_{HO} t_{\theta(i)}\sigma$; $\rightsquigarrow$ enumerates all such θ (lazily) and must find the solution by induction.
- otherwise $s\sigma$ is a record that, if s and t are both records, has three sets of labels:
  1. labels common to s and t (unified pairwise in s and t);
  2. labels present in s and not t (unified to the possibly empty row of t);
  3. labels present in t and not s (unified to the row of s).

  Terms whose labels belong to those three sets of labels are unified properly by induction hypothesis. Note that row variables are unified too.

  $\qquad\square$

6.2.3 Calculus for the Reasoner

The purpose of having a meta-prover is to deduce high-level properties about types, functions and predicates in a particular proof attempt. Since those properties might be useful to the Superposition prover, they should be found as eagerly as possible. However, the meta-level calculus must no be too complicated; it will not be responsible for success directly, and thus should not be computationally expensive. We took inspiration from the bottom-up family of algorithms for Datalog [AHV95] — roughly, unit resolution for Horn clauses.

In Figure 6.1 we present the set of rules that allow the meta-prover to deal with meta-level properties. The rules for the meta-prover are applied until a fixpoint for $\mathcal{M}$ is reached. We assume the first-order prover is saturating a set of first-order clauses $\mathcal{N}$, and the meta-prover has a set of higher-order Horn clauses $\mathcal{M}$. Properties in the meta-prover all have type o_{meta}. The following types and constants are available in the meta-prover signature:

- holds : $o \rightarrow o_{meta}$, to state a first-order formula has been deduced in $\mathcal{N}$;
- axiom : axiom $\rightarrow o_{meta}$, to represent some axioms such as associativity;
- theory : theory $\rightarrow o_{meta}$, to represent theories such as AC;
- lemma : $o \rightarrow o_{meta}$, to deduce lemmas from the presence of theories;
- other predicates can be added, see Section 6.3.

Unit Resolution performs all meta-level reasoning, in a bottom-up fashion.
Clause Encoding informs the meta-prover of new facts deduced at the first-order level. In particular, it adds all the initial clauses, after encoding, to the meta-prover so that axioms and theories can be detected from the beginning.

Unit Resolution

$$\frac{(A \leftarrow B_1, \ldots, B_n) \in \mathcal{M} \qquad (B) \in \mathcal{M}}{(A \leftarrow B_2, \ldots, B_n)\, \sigma \in \mathcal{M}}$$

$$\text{if } \{B_1 =^?_{\text{HO}} B\}; \emptyset \rightsquigarrow^+ \emptyset; \sigma$$

Clause Encoding

$$\frac{C \in \mathcal{N}}{(\text{holds } D) \in \mathcal{M}}$$

$$\text{where } D \overset{\text{def}}{=} \text{enc}(C)$$

Lemma Decoding

$$\frac{(\text{lemma } F) \in \mathcal{M}}{\big\{C \leftarrow \|G\| \mid C \in \text{cnf}(G)\big\} \cup \big\{C \leftarrow \neg\|G\| \mid C \in \text{cnf}(\neg G)\big\} \subseteq \mathcal{N}}$$

$$\text{if } G \overset{\text{def}}{=} \text{dec}(F) \text{ is a well-formed first-order formula}$$

Figure 6.1: Inference Rules for the Meta-level Reasoner

Lemma Decoding is triggered when a lemma is found to hold at the meta-level — most likely because it was bound to some theory. For instance, if

$$\text{theory } \big(\text{group } \{\!|\text{op} = m, \text{inverse} = i, \text{neutral} = e |\!\}\big)$$

is deduced for some symbols (e, i, m), the lemma $\forall x.\ i(i(x)) \simeq x$ might prove useful. We use the technique from Section 5.3 to introduce lemmas using Avatar, in order to enforce the lemma be proved if it helps solving the problem.

Remark 6.6 (Proving Lemmas Again)**.** *An alternative to the Lemma Decoding rule above would be to save a proof of every lemma, and reuse it, instead of simply introducing a cut.*

Remark 6.7 (Matching)**.** *Since Horn Clauses are safe, unit resolution is always performed between a clause and a ground fact. Therefore, we could use matching rather than full unification.*

6.2.4 Implementation

A meta-prover implementation is provided in Logtk, and interfaced with Zipperposition. It can only recognizes clausal axioms, but otherwise is on par with the work from previous sections. Still, we should clarify how unification is really implemented. Some points in the algorithm from Definition 6.12 may lead to exponential branching — namely, permutations of quantifiers and multiset elements.

For unify multisets $[s_1, \ldots, s_n]$ and $[t_1, \ldots, t_n]$, we do not really enumerate all permutations of $[t_1, \ldots, t_n]$; instead, we reduce both multisets' size by picking s_1 and, for every t_i, if s_1 and t_i unify, recursing on remaining terms. This allows to prune all the permutations that map s_1 to t_i if s_1 and t_i do not unify. Obviously the order in which terms are chosen matters, but only heuristics (ordering terms $s_1, \ldots, s_n$ by decreasing size, for instance) can help. Of course, if the two multisets contain n distinct variables, we must enumerate all permutations because they will all yield a distinct mgu. In addition, since we use the α sequence type (Section 4.6.2), we can enumerate unifiers lazily and compute only a subset of them (for instance using Sequence.head).

For quantifiers, the problem is more subtle. To solve $\forall x_1 \ldots x_n.\ s =^?_{\text{HO}} \forall y_1 \ldots y_n.\ t$ (the case for existential quantifiers is similar), we could just enumerate all permutations of $\{y_1, \ldots, y_n\}$,

but that would not be efficient — some efforts might be wasted in unifying similar instances of s and t only to be discarded because the permutation is not correct. Instead, we build a *lazy permutation* of $\{1,\dots,n\}$, called θ, and map, in a sense, x_i to $y_{\theta(i)}$. Such a lazy permutation θ is a partial map from $\{1,\dots,n\}$ to $\{1,\dots,n\}$ that is injective; if $n-1$ integers are bound then the last one is automatically mapped to the only choice that keeps the map injective. We also have to remember, before unifying s and t, that $(y_i)_{i=1}^{n}$ are bound variables and cannot be unified with anything but themselves. Inside of $s =_{\text{HO}}^{?} t$, we use unification problem to constrain θ: if $\{s =_{\text{HO}}^{?} t\} \cup M; \sigma \rightsquigarrow^{+} x_i =_{\text{HO}}^{?} y_j \cup M'; \sigma'$ then $\theta(i) = j$ must hold. If $\theta(i)$ is already set to another index, or there is some $i' \neq i$ such that $\theta(i') = j$, unification fails; else, unification proceeds with M and $\theta' \overset{\text{def}}{=} \{i \mapsto j\} \cup \theta$.

Example 6.4 (Associativity)**.** *As an example of unification with binders, let* $s \overset{def}{=} \forall x_1\ x_2\ x_3.\ x_1 + (x_2 + x_3) \simeq (x_1 + x_2) + x_3$ *and* $t \overset{def}{=} \forall y_1\ y_2\ y_3.\ (y_2 + y_1) + y_3 \simeq y_2 + (y_1 + y_3)$ *(we use an infix notation for readability)*. *s and t are unifiable as follows:*

$$
\begin{aligned}
& \{s =_{HO}^{?} t\}; \sigma = \emptyset; \theta = \{\} \\
\rightsquigarrow\ & \{\dot{\simeq}\ [x_1 + (x_2 + x_3), (x_1 + x_2) + x_3] =_{HO}^{?} \dot{\simeq}\ [(y_2 + y_1) + y_3, y_2 + (y_1 + y_3)]\}; \sigma = \emptyset; \theta = \{\} \\
\rightsquigarrow\ & \{x_1 + (x_2 + x_3) =_{HO}^{?} y_2 + (y_1 + y_3), (x_1 + x_2) + x_3 =_{HO}^{?} (y_2 + y_1) + y_3\}; \sigma = \emptyset; \theta = \{\} \\
\rightsquigarrow\ & \{x_1 =_{HO}^{?} y_2, (x_2 + x_3) =_{HO}^{?} (y_1 + y_3), (x_1 + x_2) + x_3 =_{HO}^{?} (y_2 + y_1) + y_3\}; \sigma = \emptyset; \theta = \{\} \\
\rightsquigarrow\ & \{(x_2 + x_3) =_{HO}^{?} (y_1 + y_3), (x_1 + x_2) + x_3 =_{HO}^{?} (y_2 + y_1) + y_3\}; \sigma = \emptyset; \theta = \{1 \mapsto 2\} \\
\rightsquigarrow\ & \{x_2 =_{HO}^{?} y_1, x_3 =_{HO}^{?} y_3, (x_1 + x_2) + x_3 =_{HO}^{?} (y_2 + y_1) + y_3\}; \sigma = \emptyset; \theta = \{1 \mapsto 2\} \\
\rightsquigarrow\ & \{x_3 =_{HO}^{?} y_3, (x_1 + x_2) + x_3 =_{HO}^{?} (y_2 + y_1) + y_3\}; \sigma = \emptyset; \theta = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\} \\
\rightsquigarrow^{+}\ & \emptyset; \sigma = \emptyset; \theta = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\}
\end{aligned}
$$

Once we know $\theta(1) = 2$ and $\theta(2) = 1$, $\theta(3) = 3$ becomes necessary (because θ must be a permutation of $\{1,\dots,3\}$), and unification becomes trivial by just destructuring sums. Note that θ is not part of the solution: the formulas are actually equivalent and the mgu is $\sigma = \emptyset$.

Remark 6.8 (Term Indexing)**.** *An efficient implementation of the meta-level reasoner will need some form of term indexing. We conjecture that the Fingerprint indexing technique [Sch12] could be adapted to higher-order terms — considering multisets and quantifiers as special opaque constants.*

6.3 Applications

The meta-prover in itself is *extensible*: new meta-level predicates can be added as long as their return type is o_{meta}. We present a few possible applications; some are also presented in [BC13]. Zipperposition, as a proof-of-concept, implements some of them (Lemmas, Inductive Lemmas, and some support for AC).

Special Support for Theories Dealing with associativity-commutativity (AC) in equational theorem provers has been an important topic of research (see for instance [BG95]) and the resolution of Robbins Problem by Mccune [McC97]. However, unification modulo AC is very complex to implement, and term indexing modulo AC even more so; we take inspiration from the E prover [Sch02] again. E has some support for reasoning modulo AC using the two following simplification rules (where $\overline{AC}$ is a small set of clauses containing the AC axioms for some symbols):

AC tautology deletion

$$\frac{C \vee s \simeq t}{\top}$$

$$\text{if } s \simeq_{AC} t \text{ and } C \vee s \simeq t \notin \overline{AC}$$

AC simplification
$$\frac{C \vee s \not\simeq t}{C}$$

$$\text{if } s \simeq_{AC} t$$

This rule can be activated during the proof as soon as AC axioms are detected. More generally, provers implementing some flavour of *deduction modulo* [DHK03] could use an adequate rewriting system when they detect some particular theory.

Lemmas   The rule **Lemma Decoding** from Figure 6.1 allows the meta-prover to suggest new lemmas to the Superposition prover. As it is possible to describe axioms and theories over any signature, fundamental lemmas (e.g., group inverse is involutive) can be associated with such theories and apply to any problem in which those theories appear.

Inductive Lemmas   In Section 5.3 of our chapter about Structural Induction, we mentioned that lemmas discovered in the course of an inductive proof should be saved and re-used in other proofs. The meta-prover is an adequate mechanism to do so. In Zipperposition, we introduce a new meta-level predicate, inductive, a new type, constructor, and a new function symbol, cstor:

$$\text{cstor} \quad : \quad \Pi\alpha.\ \alpha \to \text{constructor}$$
$$\text{inductive} \quad : \quad \Pi\alpha.\ \{\!| \text{ty}:\alpha, \text{cstors}:[\text{constructor}] |\!\} \to o_{\text{meta}}$$

When an inductive type is defined, the corresponding meta-level property is asserted. For instance, if nat is inductive and has constructors s and 0, the corresponding meta-level property is:

$$\text{inductive}_{\langle\text{nat}\rangle}\ \{\!|\text{ty} = \text{nat}, \text{cstors} = \big[(\text{cstor}_{\langle\text{nat}\to\text{nat}\rangle}\ s), (\text{cstor}_{\langle\text{nat}\rangle}\ 0)\big]|\!\}$$

This inductive specification can then be used, in combination with axioms and theories (e.g., Peano axioms) to suggest lemmas to the inductive prover. For instance, the theory

```
theory (peano_add {succ=S, zero=Z, plus=P}) <-
  inductive @N {ty=@N, cstors=[(cstor _ S), (cstor _ Z)]}.
  holds (![X:N,Y:N]: [P (S X) Y = S (P X Y)]),
  holds (![X:N]: [P Z X = X]).

lemma (![X:N,Y:N]: [P X (S Y) = S (P X Y)]) <-
  theory (peano_add {succ=S, zero=_, plus=P}).
```

allows Zipperposition to prove the theorem $\forall x.\ \text{dup}(x) \simeq x + x$ from Example 5.12.

Pre-processing   In [BC13] we describe a lemma called *un-mangling of function relations*. We define the theory of unary and binary functions encoded as relations:

```
axiom (functional1 P) <-
  holds (![X,Y,Z]: [~ (P X Y), ~ (P X Z), Y = Z]).

axiom (total1 {pred=P, fun=F}) <-
  holds (![X]: [P X (F X)]).

axiom (functional2 P) <-
```

116

```
    holds (![X,Y,Z]: [~ (P X Y Z), ~ (P X Y Z2), Z = Z2]).

axiom (total2 {pred=P, fun=F}) <-
  holds (![X,Y]: [P X Y (F X Y)]).
```

A binary relation P is functional if there is a unary function F satisfying both $\forall x\, y\, z.\, P(x,y) \wedge$ $P(x,z) \Rightarrow y = z$ and $\forall x.\, P(x, F(x))$. Similarly, ternary relations can be functional w.r.t. a binary function — the relation actually represents the *graph* of the function. Some TPTP problems, designed for theorem provers with poor support for equational reasoning, use this representation of functions. However, to leverage Superposition and its good support for equality reasoning (including demodulation, that is, rewriting with unit equations), we can replace the relation with functions, using `pre_rewrite` pre-processing rules. For instance, the rule

```
pre_rewrite [![X,Y,Z]: (P X Y Z --> (Z = F X Y))] <-
  axiom (functional2 P),
  axiom (total2 {pred=P, fun=F}).
```

was used, in a previous version of Zipperposition, to eliminate any occurrence of functional relations. If the meta-prover deduced `axiom (functional2 p)` and `axiom (total2 {pred=p, fun=f})` for some symbols f and p, the commutativity axiom for p can be simplified into a unit equation:

$$\frac{\dfrac{\neg p(x,y,z) \vee p(y,x,z) \qquad \forall x\, y\, z.\, (p\ x\ y\ z) \longrightarrow (z \simeq f\ x\ y)}{z \not\simeq f(x,y) \vee z = f(y,x)}}{f(x,y) = f(y,x)}\ \text{(DER)}$$

Choosing Heuristics   The theorem prover Waldmeister uses a simplified form of theory detection to choose its term ordering — a heuristic choice that has a huge impact on the behavior of the prover — depending on the problem at hand [HJL99]. In fact, our initial design of the meta-prover as described in [BC13] owes much to Waldmeister.

Rewrite Systems   If a given equational theory is oriented by some instance of RPO (Definition 2.47), then choosing this instance of RPO enables Superposition to reason "modulo" the theory — as in *Deduction Modulo* [DHK03]. Roughly, Superposition becomes narrowing, and Demodulation (Figure 2.2) normalizes terms w.r.t. the theory rewrite system.

To study this restricted version of Deduction Modulo, we developed a small tool called *Hysteresis*, which is shipped with Logtk. The tool uses a meta-prover to detect the presence of some theories such as groups, and chooses a LPO precedence based on rewriting systems known to orient the theories present in the problem. Then, it calls a version of E [Sch02] that we modified so it could handle simple types and feeds it the problem and the LPO precedence computed above. However, an important part of Deduction Modulo — rewriting on atomic formulas — has no equivalent; it would require Superposition to be either polarized [Dow10] or non-clausal [GS03][2].

The tool was fed a rewrite system for integer arithmetic. The rewrite rules below (in a TPTP-like language) define arithmetic operations on a ternary representation of integers (borrowed from [CMR97], Section 4.2) in which positive and negative numbers possess symmetric representations. There are four constructors for integers. Let $\langle t \rangle$ be the natural number corresponding to the term t built from the four constructors:

z_0 represents 0, that is, $\langle z_0 \rangle = 0$;
z_3 multiplies by three its argument, meaning $\langle z_3(x) \rangle = 3 \times \langle x \rangle$;
z_3p1 is defined by $\langle z_{3p1}(x) \rangle = 3 \times \langle x \rangle + 1$;

---

[2] an old version of Zipperposition used this calculus.

z_3m1 is defined by $\langle z_{3m1}(x)\rangle = 3 \times \langle x\rangle - 1$.

If we enforce that $z_3(z_0)$ always reduces to z_0, each integer has a unique representation using those constructors, and $-x$ is obtained from x by swapping the rôles of z_{3p1} and z_{3m1}. The representation of a number n is a term of size $O(\log_3(n))$, making this representation more efficient than (unary) Peano numbers.

```
val z_0 : $int.
val z_3 : $int → $int.
val z_3p1 : $int → $int.
val z_3m1 : $int → $int.

z_3 z_0 ⟶ z_0.

z_plus z_0 X ⟶ X.
z_plus X z_0 ⟶ X.
z_plus (z_3 X) (z_3 Y) ⟶ z_3 (z_plus X Y).
z_plus (z_3 X) (z_3p1 Y) ⟶ z_3p1 (z_plus X Y).
z_plus (z_3 X) (z_3m1 Y) ⟶ z_3m1 (z_plus X Y).
z_plus (z_3p1 X) (z_3m1 Y) ⟶ z_3 (z_plus X Y).
z_plus (z_3m1 X) (z_3p1 Y) ⟶ z_3 (z_plus X Y).
z_plus (z_3p1 X) (z_3p1 Y) ⟶ z_3m1 (z_plus X (z_plus Y (z_3p1 z_0))).
z_plus (z_3m1 X) (z_3m1 Y) ⟶ z_3p1 (z_plus X (z_plus Y (z_3m1 z_0))).

z_opp z_0 ⟶ z_0.
z_opp (z_3 X) ⟶ z_3 (z_opp X).
z_opp (z_3p1 X) ⟶ z_3m1 (z_opp X).
z_opp (z_3m1 X) ⟶ z_3p1 (z_opp X).

z_minus X Y ⟶ z_plus X (z_opp Y).

z_mult X z_0 ⟶ z_0.
z_mult X (z_3 Y) ⟶ z_3 (z_mult X Y).
z_mult X (z_3p1 Y) ⟶ z_plus X (z_3 (z_mult X Y)).
z_mult X (z_3m1 Y) ⟶ z_plus (z_3 (z_mult X Y)) (z_opp X).
```

The rules are defined in the file `data/rewriting/balanced_int.p` in Logtk. To try it, type (assuming $TPTP points to a TPTP archive):

```
$ ./hysteresis.native –balanced–arith $TPTP/Problems/ARI/ARI017=1.p
```

The tool could solve simple arithmetic problems in the category `ARI` of TPTP, but showed its limitations quickly. Later, we developed the much more ambitious arithmetic calculus from Chapter 4, which performed considerably better on arithmetic problems that required *solving* (in)-equations, in addition to merely *computing* the normal form of expressions. We believe that solving, not only computing, is an important issue in some theories (including arithmetic), and Deduction Modulo falls short for such theories. Other techniques for proving modulo theories have acknowledged the need for both: for instance, Shostak's Decision Procedure [RS01], as used in some SMT solvers [BCC+13], requires a theory-specific decision procedure to provide both a *canonizer* (to reduce a term to a normal form) and a *solver* (eliminate a variable to solve an equation). Deduction Modulo can emulate solving through narrowing, but it might be very inefficient — possibly as inefficient as simply using the theory axioms.

6.4 Experimental Results

We compared version 0.2 of Zipperposition[3] with SPASS [WSH+07] and E [Sch02] on categories RNG and GRP of the TPTP [Sut09] base of problems. In the benchmarks, Zipperposition-meta

---

[3] We point out that our implementation of Superposition is not nearly as good as SPASS or E, which are the result of years of work.

features a meta-prover with theory detection, the relational un-mangling lemma, and redundancy criteria (for AC, commutative monoids and abelian groups, as described in [BC13]) — and Zipperposition, in which all theory handling is disabled. The results are exposed in Figure 6.2. Overall, on 1434 problems, Zipperposition-meta proves 7 problems that are not proven by SPASS nor E within the 120s timeout. Zipperposition is able to detect at least one theory in 594 problems out of 1434, and triggers the lemma in 68 problems. Among the 594 problems with theories, 31 are solved by Zipperposition-meta but not by Zipperposition, and 7 are solved by the latter but not by the former (because the prover was slower or it pruned the wrong part of the search space). This ratio becomes 7 to 2 on the problems in which the lemma is applied.

Prover	Proved	%	/594	%	/68	%
E	1047	73.0	430	72.4	59	86
SPASS	863	60.1	376	63.3	50	73
zipperposition-meta	531	37.0	202	34.0	56	82
zipperposition	504	35.1	191	32.1	52	76

Figure 6.2: Benchmark Results for Zipperposition with Meta Prover

We can already see that the redundancy criterion from [BC13], with its quite naive implementation, already brings benefits. The un-mangling lemma also makes a significant difference on the set of problems in which it applies. On individual problems, the difference can be striking: some problems that would not terminate within 2 minutes become trivial enough to get solved in 0.5s when lemma detection is enabled. Those results are encouraging, and we believe that using a meta-prover may find more uses in automated theorem proving. Profiling shows that the meta-level reasoner represents a negligible fraction of the run-time (less than 1%). On the other hand, our implementation of Superposition is more naive and less efficient than SPASS or E (which also benefit from having respectively a more powerful calculus and better heuristics), which can explain why they still solve more problems. Our technique could be integrated in other theorem provers to discover lemmas or usable redundancy criteria — especially for scheduling provers (like iProver [Kor08]) because meta-level facts that are discovered during a time slice can be used for the next ones (using a suitable term ordering, etc.).

Three problems are solved only by Zipperposition-meta (a version of Zipperposition that uses lemma detection): GRP392-1.p, GRP393-1.p and GRP394-1.p. Interestingly, all three are satisfiable problems in relational form where the un-mangling lemma transforms into easily saturated sets of equations[4]. This result means the un-mangling of functional relations might be useful in other Superposition provers.

Conclusion and Possible Extensions

We have shown a generic and flexible way to detect instances of axioms and theories during a first-order saturation process. The use of a bottom-up resolution system makes the meta-level reasoning flexible, modular and extensible by declaring one's own meta-level properties, such as the presence of inductive types. This technique already shows promising results, and can be improved further with more sophisticated uses of the detected theories. We believe that this kind of combination, although still quite simple, bears some resemblance with the way real mathematicians solve problems. Using several levels of description and proof may also help making automated proofs more understandable, saturation proofs being often blamed for being very unintuitive to human users. Further development may include:

---

[4] Zipperposition 0.2 used the *Superposition with lazy CNF* calculus, from [GS03], which turns some equivalences into rewrite rules, allowing the un-mangling technique to be expressed as a mere lemma, rather than a preprocessing rule.

- making the reasoner more proactive by having it spawning subprocesses to try to prove missing axioms;
- automatically extract lemma from successful proofs in order to help solving similar problems, in particular for inductive proofs;
- implementing this technique in a state of the art prover.

Chapter 7

Conclusion

Overview

As this thesis nears its end, we review its chapters and the contributions we made. Overall, all our work gravitates around Superposition, how to implement it (Chapter 3), and various ways of extending it, be it by adding "regular" deduction rules (e.g., to deal with linear arithmetic, in Chapter 4), by having it support structural induction (Chapter 5), or by introducing lemmas based on the presence of known theories (Chapter 6). Most techniques introduced in this document were implemented in Zipperposition, our experimental theorem prover written in OCaml (Chapter 3); this implementation provided empirical evidence that each such technique works at least on simple examples in practice. More generally, as we explored the design space of each feature, extending Zipperposition was extremely useful to find shortcomings in our approach and help solve them.

We proposed, in Chapter 4, an extension of Superposition that reasons modulo linear integer arithmetic, a fundamental theory for many use-cases, including program verification. Our contribution is a purely deductive set of rules (along with a canonical representation of arithmetic literals), which contrasts with blackbox-based techniques such as Hierarchic Superposition [BGW94] [BW13] or more ad-hoc combinations [PW06]. A nice property of this system is that it can be combined with other rule systems as long as both handle disjoint sets of literals (i.e., no literal is both an arithmetic literal and a literal from the other theory). The system should also combine well with the inductive reasoning from Chapter 5. Support for arithmetic also justifies the need for typing in a theorem prover, as one often wishes to mix arithmetic terms with uninterpreted ones (or terms of another theory, such as arrays or lists, very common in program verification). The implementation in Zipperposition was found to perform competitively with other first-order arithmetic provers at CASC in 2014; it also demonstrates how proper abstractions (iterators, mainly) help implementing such complicated and subtle inference rules: the Superposition modulo rational arithmetic [Wal01] was, to our knowledge, never implemented because of its complexity (roughly equivalent to the complexity of the rules of Chapter 4).

Chapter 5 extends Superposition along a totally different axis, pointing toward higher-order logics. First-order logic with induction is strictly more expressive than first-order logic alone; yet, few general purpose theorem provers support it. Usually, automated inductive provers such as Spike [BKR92] [Str12] build on (conditional) rewriting rather than the most successful deduction paradigms for first-order logic such as Superposition. Considering that remark, our approach is complementary to usual inductive proving as it tries to bridge the gap between general-purpose theorem proving and inductive proving by starting from regular Superposition provers instead of making inductive provers better at first-order reasoning. In achieving this goal, the recent technique AVATAR [Vor14] was very useful for encoding case-analysis problems (in particular, the various cases of coversets for the inductive step). AVATAR, too, can be

extended with more powerful boolean solving techniques — here, QBF, so that it could handle exponentially many cases in a compact way, to enumerate a powerset. Since AVATAR was published at the beginning of the last year of the thesis, it might have a positive impact on other parts of our work that might involve case analysis (in particular, arithmetic features case split in several rules). Again, using a typed logic helps a lot, since a realistic use of structural induction will have several inductive types in addition to o and ι; without them, guards must be introduced to ensure induction only applies to some elements. For instance, $\forall x : \mathrm{bool}.\ x \simeq 1 \vee x \simeq 0$ would become $\forall x.\ \mathrm{is\_bool}(x) \Rightarrow x \simeq 1 \vee x \simeq 0$ using a classic encoding; such guards are known to make reasoning heavier (especially in the case of unit clauses such as $\forall x : \alpha.\ \neg\mathrm{mem}_{\langle\alpha\rangle}(x, []_{\langle\alpha\rangle})$). Other encodings of types exist but tend to make terms heavier [BBPS13].

Finally, Chapter 6 contains a system for detecting axiomatic theories, and a modular interface between it and the Superposition prover. It can be used to introduce lemmas depending on which theories are present in a problem, for instance in inductive theories, or to preprocess the problem into a variant more suited to Superposition (e.g., transforming functional relations into functions).

Perspectives

Some systems make good use of *Unit Superposition* (for instance, iProver-Eq [KS10]) as a way to graft equational reasoning into a calculus that is not Resolution. Adopting a unit version of our arithmetic calculus, in the same context, could help implement arithmetic support in other proof techniques. Combining the normalization, inference and simplification rules from Chapter 4 to Hierarchic Superposition with Weak Abstraction [BW13] might also lead to a combined system that would behave better than each technique alone — both have quite different strengths. Similarly, combinations of Superposition in SMT [dMB08] could use (part of) our arithmetic techniques to manage and instantiate their non-ground clauses.

Another perspective is extending Arithmetic Superposition so that it is complete on ground formulas w.r.t. to an axiomatization of $\mathbb{Z}$, and then trying to make it complete on wider fragments than ground formulas.

Our work on induction opens a lot of perspectives and possible improvements. First, a strong implementation of the multi-clause inductive strengthening (Section 5.4) would permit to validate the effectiveness of the technique and compare it to [KP13]. Then, better lemmas generation and pruning heuristics are needed.

In the long term, the idea of a *meta-prover*, developed in Chapter 6, might prove useful in various areas of computer-assisted theorem proving. Not only might an automated theorem prover use such a system to detect theories it knows so it can enable theory-specific mechanisms and rules, but we can imagine a proof assistant such as Coq [HKPM97] or Isabelle [NPW02] could detect that the user is defining a well-known structure, so as to propose lemmas and theorems automatically.

As stated before, Superposition-based theorem provers such as E [Sch02], Vampire [RV01b] or SPASS [WSH⁺07] are very successful in untyped first-order logic, but fall short when it comes to theory support (although SPASS has had some extensions in that direction [PW06]). We believe the future of such automated theorem provers (or their successors) lies in dealing with more powerful and expressive logics (typed logic, arithmetic, various theories, or induction), so they can be used in the many application domains that require this increase in power. Integration of automated provers in interactive proof assistants is also a promising area of research (for instance [BBN11]). Implementing some of the techniques from this thesis in strong theorem provers would therefore be very interesting, both to evaluate their usefulness in a more practical setting, and to improve the application range of the tools.

Bibliography

[AHL03]   J. Avenhaus, Th. Hillenbrand, and B. Löchner. On using ground joinable equations in equational theorem proving. *Journal of Symbolic Computation*, 36(1–2):217 – 233, 2003.

[AHV95]   Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKW09]   Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. Superposition Modulo Linear Arithmetic SUP(LA). In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems*, volume 5749 of *LNCS*, pages 84–99. Springer, 2009.

[AT10]   Andrea Asperti and Enrico Tassi. Smart matching. In *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin Heidelberg, 2010.

[BBN11]   Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In *Frontiers of Combining Systems*, pages 12–27. Springer, 2011.

[BBPS13]   Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding Monomorphic and Polymorphic Types. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 493–507. Springer Berlin Heidelberg, 2013.

[BC13]   Guillaume Burel and Simon Cruanes. Detection of First Order Axiomatic Theories. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pages 229–244. Springer Berlin Heidelberg, 2013.

[BCC+13]   François Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008, 2013.

[BG90]   Leo Bachmair and Harald Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, volume 449 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1990.

[BG94]   Leo Bachmair and Harald Ganzinger. Rewrite Techniques for Transitive Relations. In *In Proc., 9th IEEE Symposium on Logic in Computer Science*, pages 384–393. IEEE Computer Society Press, 1994.

[BG95]   Leo Bachmair and Harald Ganzinger. Associative-Commutative Superposition. In Nachum Dershowitz and Naomi Lindenstrauss, editors, *Conditional and Typed Rewriting Systems*, volume 968 of *LNCS*. Springer, 1995.

[BGW94]   Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3-4):193–212, 1994.

[Bie04]   Armin Biere. Resolve and Expand. In *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2004.

[BKR92]   Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch. SPIKE, an automatic theorem prover. In *Logic Programming and Automated Reasoning*. Springer, 1992.

[BM14]   Robert S Boyer and J Strother Moore. *A Computational Logic Handbook: Formerly Notes and Reports in Computer Science and Applied Mathematics*. Elsevier, 2014.

[BP13]   Jasmin Christian Blanchette and Andrei Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. In *Automated Deduction–CADE-24*, pages 414–420. Springer, 2013.

[BS01]   Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [RV01c], pages 445–532.

[BSvH+93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185 – 253, 1993.

[BTPF08]   Christoph Benzmüller, Frank Theiss, Larry Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

[BW13]   Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. In *Automated Deduction–CADE-24*. Springer, 2013.

[Bé79]   Étienne Bézout. *Théorie générale des équations algébriques*. De l'imprimerie de Ph. D. Pierres, 1779.

[CDT]   The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr/`.

[Chu36]   Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[CJRS12]   Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Hipspec: Automating inductive proofs of program properties. In *ATx/WInG@ IJCAR*. Citeseer, 2012.

[CMR97]   Evelyne Contejean, Claude Marché, and Landy Rabehasaina. Rewrite systems for natural, integral, and rational arithmetic. In Hubert Comon, editor, *Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 98–112. Springer Berlin Heidelberg, 1997.

[Com94]   Hubert Comon. Inductionless induction, 1994.

[Coo71]   Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[Coo72]   David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.

[DB72]   Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[DHK03]   Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 2003.

[DM79]   Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[dMB08]   Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In *Proc. 4TH IJCAR*, 2008.

[Dow10]   Gilles Dowek. Polarized resolution modulo. In CristianS. Calude and Vladimiro Sassone, editors, *Theoretical Computer Science*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 182–196. Springer Berlin Heidelberg, 2010.

[FC06]   Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *In: Proceedings of the 2006 workshop on ML*, pages 12–19. ACM Press, 2006.

[FW09]   Arnaud Fietzke and Christoph Weidenbach. Labelled splitting. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):3–34, 2009.

[GNN98]   Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. The Saturate system. 1998.

[GS03]   Harald Ganzinger and Jürgen Stuber. Superposition with equivalence reasoning and delayed clause normal form transformation. In Franz Baader, editor, *Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 335–349. Springer, 2003.

[HJL99]   Thomas Hillenbrand, Andreas Jaeger, and Bernd Löchner. System Description: Waldmeister — Improvements in Performance and Ease of Use. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 232–236. Springer, 1999.

[HKPM97]   Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant. *A Tutorial. Version*, 5, 1997.

[JDB10]   M. Johansson, L. Dixon, and A. Bundy. Conjecture Synthesis for Inductive Theories. In *Journal of Automated Reasoning*, 2010.

[KB95]   M. Kaufmann and R. S. Boyer. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. In *Computers and Mathematics with Applications*. 1995.

[KKF95]   Hans Kleine Büning, M. Karpinski, and A. Flogel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12 – 18, 1995.

[Kor08]   Konstantin Korovin. iProver — An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 292–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[KP13]   Abdelkader Kersani and Nicolas Peltier. Combining superposition and induction: A practical realization. In Pascal Fontaine, Christophe Ringeissen, and RenateA. Schmidt, editors, *Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin Heidelberg, 2013.

[KS08]   Daniel Kroening and Ofer Strichman. *Decision Procedures, An Algorithmic Point of View*. Springer, 2008.

[KS10]   Konstantin Korovin and Christoph Sticksel. Labelled unit superposition calculi for instantiation-based reasoning. In ChristianG. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 459–473. Springer Berlin Heidelberg, 2010.

[KV07]   Konstantin Korovin and Andrei Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic*, pages 223–237. Springer, 2007.

[LAWRS07] Tal Lev-Ami, Christoph Weidenbach, Thomas Reps, and Mooly Sagiv. Labelled clauses. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 311–327. Springer Berlin Heidelberg, 2007.

[LB10]   Florian Lonsing and Armin Biere. DepQBF: A Dependency-Aware QBF Solver. In *Journal on Satisfiability, Boolean Modeling and Computation*, 2010.

[McC95]   William McCune. OTTER 3.0 Reference Manual and Guide, 1995.

[McC97]   William McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McC10]   William McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9` , 2005–2010.

[Mil78]   Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[NO80]   Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980.

[NPW02]   Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[NR99]   Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Robinson and Voronkov [RV01c].

[NRV97]   Robert Nieuwenhuis, José Rivero, and Miguel Vallejo. Dedam: A kernel of data structures and algorithms for automated deduction with equality clauses. In William McCune, editor, *Automated Deduction–CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 49–52. Springer, 1997. 10.1007/3-540-63104-6_5.

[NW01]   Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [RV01c], pages 335–367.

[Pel86]   Francis Jeffry Pelletier. Seventy-Five Problems for Testing Automatic Theorem Provers. *J. Autom. Reasoning*, 2(2):191–216, 1986.

[PW06]   Virgile Prevosto and Uwe Waldmann. SPASS + T. *Proceedings ESCoR: FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, pages 18 – 33, 2006.

[RK15]   Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2015.

[Rob65]   J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[Rob71]   J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.

[RS01]   Harald Rueß and Natarajan Shankar. Deconstructing Shostak. To be presented at LICS'2001, 2001.

[RSV01]   I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In Robinson and Voronkov [RV01c], pages 1853–1964.

[Rüm08]   Philipp Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 274–289. Springer Berlin Heidelberg, 2008.

[RV01a]   Alexandre Riazanov and Andrei Voronkov. Splitting Without Backtracking, 2001.

[RV01b]   Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, pages 376–380, London, UK, UK, 2001. Springer-Verlag.

[RV01c]   John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[Sch02]   Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[Sch04]   Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving*. Elsevier Science, 2004.

[Sch12]   Stephan Schulz. Fingerprint indexing for paramodulation and rewriting. In *Automated Reasoning*, pages 477–483. Springer, 2012.

[Smo89]   Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Universität Kaiserslautern, 1989.

[SS06]   G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[Str12]   Sorin Stratulat. A unified view of induction reasoning for first-order logic. In *Turing-100, The Alan Turing Centenary Conference*, 2012.

[Sut09]   G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Tse83]   G.S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[Vor14]   Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 696–710, 2014.

[Wal98]   Uwe Waldmann. Extending reduction orderings to ACU-compatible reduction orderings. *Information processing letters*, 67(1):43–49, 1998.

[Wal01]   Uwe Waldmann. Superposition and Chaining for Totally Ordered Divisible Abelian Groups. In *IJCAR*, Lecture Notes in Computer Science. Springer, 2001.

[Wal15]   Uwe Waldmann. personal communication, 2015.

[Wan87]   Mitchell Wand. Complete type inference for simple objects. In *LICS*, volume 87, pages 37–44, 1987.

[Wan14]   Daniel Wand. Polymorphic+Typeclass Superposition. In Boris Konev, Leonardo de Moura, and Stephan Schulz, editors, *4th Workshop on Practical Aspects of Automated Reasoning (PAAR 2014)*, page 15, Vienna, Austria, July 2014.

[WSH+07]   Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Rostislav Rusev, and Dalibor Topic. System Description: SPASS Version 3.0. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520. Springer, 2007.

[ZKK88]   Hantao Zhang, Deepak Kapur, and Mukkai S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 1988.

List of Figures

Index

Abstract

The central concept of theorem designates a claim backed by an irrefutable argument that follows formal rules, called a proof. Proving theorems is very useful in both Computer Science and Mathematics. However, many theorems are too boring and tedious for human experts (for instance, theorems generated to ensure that software abides by some specification); hence the decades-long effort in automated theorem proving, the field dedicated to writing programs that find proofs. Superposition is a very competitive technique for proving theorems in the language of first-order logic with equality over uninterpreted functions (in a nutshell, being able to replace equals by equals in any expression). Even then, Superposition falls short for many problems that require theory-specific reasoning or inductive proofs. In this thesis, we aim at developing new extensions to Superposition. Our claim is that Superposition lends itself very well to being grafted additional inference rules and reasoning mechanisms. First, we develop a Superposition-based calculus for integer linear arithmetic. Linear Integer Arithmetic is a widely studied and used theory in other areas of automated deduction, in particular SMT (Satisfiability Modulo Theory). This theory might also prove useful for problems that have a discrete, totally ordered structure, such as temporal logic, and that might be encoded efficiently into first-order logic with arithmetic. Then, we define an extension of Superposition that is able to reason by structural induction (natural numbers, lists, binary trees, etc.) Inductive reasoning is pervasive in Mathematics and Computer Science but its integration into general purpose first-order provers has not been studied much. Last, we present a theory detection system that, given a signature-agnostic description of algebraic theories, detects their presence in sets of formulas. This system is akin to the way a mathematician who studies a new object discovers that this object belong to some known structure, such as groups, allowing her to leverage the large body of knowledge on this specific theory. A large implementation effort was also carried out in this thesis; all the contributions presented above have been implemented in a library and a theorem prover, Zipperposition, both written in OCaml and released under a free software license.

Résumé

Le concept central de théorème désigne une assertion justifiée par un argument irréfutable agencé selon des règles formelles, qu'on appelle une preuve. Prouver des théorèmes est utile à la fois en Informatique et en Mathématiques. Cependant, beaucoup de théorèmes utiles, tels que ceux engendrés par la vérification formelle qu'un programme respecte une spécification, sont trop pénibles et inintéressants pour mériter l'attention d'experts humains; plusieurs décennies de recherches ont donc été consacrées au domaine de la démonstration automatique. La Superposition est une technique efficace permettant de prouver les théorèmes exprimés en logique du premier ordre avec égalité (en bref, la capacité de remplacer mutuellement deux objets égaux dans n'importe quelle expression). Pourtant, la Superposition montre ses limites dans beaucoup de cas où des théories spécifiques ou du raisonnement par récurrence sont nécessaires. Dans cette thèse, nous développons de nouvelles extensions à la Superposition; nous soutenons que cette dernière se prête bien à l'ajout de règles d'inférence et de mécanismes de raisonnement supplémentaires. Tout d'abord, nous développons un système d'inférence qui donne à la Superposition les moyens de raisonner dans l'arithmétique linéaire entière, une théorie activement utilisée et étudiée dans d'autres domaines de la preuve automatique tels que SMT (Satisfiabilité Modulo Théories). L'arithmétique peut également permettre des encodages vers la logique du premier ordre plus efficaces pour les structures discrètes totalement ordonnées — par exemple, la logique temporelle. Nous définissons ensuite un mécanisme permettant aux prouveurs fondés sur la Superposition de raisonner par récurrence sur des types algébriques (naturels, listes, arbres binaires, etc.) Le raisonnement par récurrence est très courant en Mathématiques et en Informatique, mais son intégration dans les systèmes de preuve dédiés à la logique du premier ordre a été peu étudiée. Enfin, nous présentons un système de détections de théories axiomatiques capable de déceler la présence de structures algébriques connues dans un ensemble de formules. Ce système rappelle la manière dont un mathématicien qui étudie un nouvel objet peut découvrir que ce dernier relève d'une structure connue — comme les groupes — ce qui lui permet de mobiliser ses connaissances sur cette théorie. Cette thèse comprend également une part importante de travail d'implémentation : toutes les contributions listées ci-dessus ont été incorporées dans une bibliothèque et un prouveur automatique, Zipperposition; tous deux sont écrits en OCaml et publiés sous licence libre.